Beej's Guide to C Programming

Brian "Beej Jorgensen" Hall

v0.7.28, Copyright © June 6, 2022

www.dbooks.org

Contents

1 Foreword **2**
- 1.1 Audience . . . 2
- 1.2 How to Read This Book . . . 3
- 1.3 Platform and Compiler . . . 3
- 1.4 Official Homepage . . . 3
- 1.5 Email Policy . . . 3
- 1.6 Mirroring . . . 4
- 1.7 Note for Translators . . . 4
- 1.8 Copyright and Distribution . . . 4
- 1.9 Dedication . . . 4

2 Hello, World! **6**
- 2.1 What to Expect from C . . . 6
- 2.2 Hello, World! . . . 7
- 2.3 Compilation Details . . . 9
- 2.4 Building with `gcc` . . . 9
- 2.5 Building with `clang` . . . 9
- 2.6 Building from IDEs . . . 10
- 2.7 C Versions . . . 10

3 Variables and Statements **12**
- 3.1 Variables . . . 12
  - 3.1.1 Variable Names . . . 13
  - 3.1.2 Variable Types . . . 13
  - 3.1.3 Boolean Types . . . 14
- 3.2 Operators and Expressions . . . 15
  - 3.2.1 Arithmetic . . . 15
  - 3.2.2 Ternary Operator . . . 15
  - 3.2.3 Pre-and-Post Increment-and-Decrement . . . 16
  - 3.2.4 The Comma Operator . . . 17
  - 3.2.5 Conditional Operators . . . 17
  - 3.2.6 Boolean Operators . . . 18
  - 3.2.7 The `sizeof` Operator . . . 18
- 3.3 Flow Control . . . 19
  - 3.3.1 The `if-else` statement . . . 20
  - 3.3.2 The `while` statement . . . 21
  - 3.3.3 The `do-while` statement . . . 21
  - 3.3.4 The `for` statement . . . 22
  - 3.3.5 The `switch` Statement . . . 23

4 Functions **26**
- 4.1 Passing by Value . . . 27
- 4.2 Function Prototypes . . . 28
- 4.3 Empty Parameter Lists . . . 29

5 Pointers—Cower In Fear! 30
5.1 Memory and Variables . . . 30
5.2 Pointer Types . . . 32
5.3 Dereferencing . . . 33
5.4 Passing Pointers as Arguments . . . 34
5.5 The `NULL` Pointer . . . 35
5.6 A Note on Declaring Pointers . . . 36
5.7 `sizeof` and Pointers . . . 36

6 Arrays 37
6.1 Easy Example . . . 37
6.2 Getting the Length of an Array . . . 38
6.3 Array Initializers . . . 38
6.4 Out of Bounds! . . . 40
6.5 Multidimensional Arrays . . . 41
6.6 Arrays and Pointers . . . 42
6.6.1 Getting a Pointer to an Array . . . 42
6.6.2 Passing Single Dimensional Arrays to Functions . . . 42
6.6.3 Changing Arrays in Functions . . . 43
6.6.4 Passing Multidimensional Arrays to Functions . . . 44

7 Strings 46
7.1 String Literals . . . 46
7.2 String Variables . . . 46
7.3 String Variables as Arrays . . . 47
7.4 String Initializers . . . 47
7.5 Getting String Length . . . 48
7.6 String Termination . . . 48
7.7 Copying a String . . . 49

8 Structs 51
8.1 Declaring a Struct . . . 51
8.2 Struct Initializers . . . 52
8.3 Passing Structs to Functions . . . 52
8.4 The Arrow Operator . . . 54
8.5 Copying and Returning `structs` . . . 54
8.6 Comparing `structs` . . . 54

9 File Input/Output 55
9.1 The `FILE*` Data Type . . . 55
9.2 Reading Text Files . . . 56
9.3 End of File: `EOF` . . . 56
9.3.1 Reading a Line at a Time . . . 57
9.4 Formatted Input . . . 58
9.5 Writing Text Files . . . 59
9.6 Binary File I/O . . . 59
9.6.1 `struct` and Number Caveats . . . 61

10 `typedef`: Making New Types 63
10.1 `typedef` in Theory . . . 63
10.1.1 Scoping . . . 63
10.2 `typedef` in Practice . . . 63
10.2.1 `typedef` and `structs` . . . 63
10.2.2 `typedef` and Other Types . . . 65
10.2.3 `typedef` and Pointers . . . 65

www.dbooks.org

10.2.4 `typedef` and Capitalization . . . 65
10.3 Arrays and `typedef` . . . 66

11 Pointers II: Arithmetic 67
11.1 Pointer Arithmetic . . . 67
11.1.1 Adding to Pointers . . . 67
11.1.2 Changing Pointers . . . 68
11.1.3 Subtracting Pointers . . . 69
11.2 Array/Pointer Equivalence . . . 70
11.2.1 Array/Pointer Equivalence in Function Calls . . . 71
11.3 `void` Pointers . . . 71

12 Manual Memory Allocation 76
12.1 Allocating and Deallocating, `malloc()` and `free()` . . . 76
12.2 Error Checking . . . 77
12.3 Allocating Space for an Array . . . 78
12.4 An Alternative: `calloc()` . . . 78
12.5 Changing Allocated Size with `realloc()` . . . 79
12.5.1 Reading in Lines of Arbitrary Length . . . 80
12.5.2 `realloc()` with `NULL` . . . 82
12.6 Aligned Allocations . . . 82

13 Scope 85
13.1 Block Scope . . . 85
13.1.1 Where To Define Variables . . . 86
13.1.2 Variable Hiding . . . 86
13.2 File Scope . . . 86
13.3 `for`-loop Scope . . . 87
13.4 A Note on Function Scope . . . 88

14 Types II: Way More Types! 89
14.1 Signed and Unsigned Integers . . . 89
14.2 Character Types . . . 90
14.3 More Integer Types: `short`, `long`, `long long` . . . 91
14.4 More Float: `double` and `long double` . . . 93
14.4.1 How Many Decimal Digits? . . . 94
14.4.2 Converting to Decimal and Back . . . 96
14.5 Constant Numeric Types . . . 97
14.5.1 Hexadecimal and Octal . . . 97
14.5.2 Integer Constants . . . 97
14.5.3 Floating Point Constants . . . 99

15 Types III: Conversions 101
15.1 String Conversions . . . 101
15.1.1 Numeric Value to String . . . 101
15.1.2 String to Numeric Value . . . 102
15.2 `char` Conversions . . . 104
15.3 Numeric Conversions . . . 105
15.3.1 Boolean . . . 105
15.3.2 Integer to Integer Conversions . . . 105
15.3.3 Integer and Floating Point Conversions . . . 106
15.4 Implicit Conversions . . . 106
15.4.1 The Integer Promotions . . . 106
15.4.2 The Usual Arithmetic Conversions . . . 106
15.4.3 `void*` . . . 107

15.5 Explicit Conversions . . . 107
15.5.1 Casting . . . 107

16 Types IV: Qualifiers and Specifiers 109
16.1 Type Qualifiers . . . 109
16.1.1 `const` . . . 109
16.1.2 `restrict` . . . 111
16.1.3 `volatile` . . . 112
16.1.4 `_Atomic` . . . 112
16.2 Storage-Class Specifiers . . . 113
16.2.1 `auto` . . . 113
16.2.2 `static` . . . 113
16.2.3 `extern` . . . 114
16.2.4 `register` . . . 115
16.2.5 `_Thread_local` . . . 116

17 Multifile Projects 117
17.1 Includes and Function Prototypes . . . 117
17.2 Dealing with Repeated Includes . . . 119
17.3 `static` and `extern` . . . 120
17.4 Compiling with Object Files . . . 120

18 The Outside Environment 121
18.1 Command Line Arguments . . . 121
18.1.1 The Last `argv` is `NULL` . . . 123
18.1.2 The Alternate: `char **argv` . . . 123
18.1.3 Fun Facts . . . 124
18.2 Exit Status . . . 125
18.2.1 Other Exit Status Values . . . 126
18.3 Environment Variables . . . 127
18.3.1 Setting Environment Variables . . . 128
18.3.2 Unix-like Alternative Environment Variables . . . 128

19 The C Preprocessor 130
19.1 `#include` . . . 130
19.2 Simple Macros . . . 131
19.3 Conditional Compilation . . . 132
19.3.1 If Defined, `#ifdef` and `#endif` . . . 132
19.3.2 If Not Defined, `#ifndef` . . . 132
19.3.3 `#else` . . . 133
19.3.4 General Conditional: `#if`, `#elif` . . . 133
19.3.5 Losing a Macro: `#undef` . . . 135
19.4 Built-in Macros . . . 135
19.4.1 Mandatory Macros . . . 135
19.4.2 Optional Macros . . . 136
19.5 Macros with Arguments . . . 137
19.5.1 Macros with One Argument . . . 137
19.5.2 Macros with More than One Argument . . . 138
19.5.3 Macros with Variable Arguments . . . 139
19.5.4 Stringification . . . 140
19.5.5 Concatenation . . . 140
19.6 Multiline Macros . . . 140
19.7 Example: An Assert Macro . . . 142
19.8 The `#error` Directive . . . 143
19.9 The `#pragma` Directive . . . 143

www.dbooks.org

19.9.1 Non-Standard Pragmas . . . 143
19.9.2 Standard Pragmas . . . 144
19.9.3 `_Pragma` Operator . . . 144
19.10 The `#line` Directive . . . 144
19.11 The Null Directive . . . 145

20 `structs` II: More Fun with `structs` 146
20.1 Initializers of Nested `structs` and Arrays . . . 146
20.2 Anonymous `structs` . . . 148
20.3 Self-Referential `structs` . . . 149
20.4 Flexible Array Members . . . 149
20.5 Padding Bytes . . . 151
20.6 `offsetof` . . . 151
20.7 Bit-Fields . . . 152
20.7.1 Non-Adjacent Bit-Fields . . . 153
20.7.2 Signed or Unsigned `ints` . . . 153
20.7.3 Unnamed Bit-Fields . . . 154
20.7.4 Zero-Width Unnamed Bit-Fields . . . 154
20.8 Unions . . . 155
20.8.1 Unions and Type Punning . . . 155
20.8.2 Pointers to `unions` . . . 156
20.8.3 Common Initial Sequences in Unions . . . 156
20.9 Unions and Unnamed Structs . . . 159
20.10 Passing and Returning `structs` and `unions` . . . 160

21 Characters and Strings II 161
21.1 Escape Sequences . . . 161
21.1.1 Frequently-used Escapes . . . 161
21.1.2 Rarely-used Escapes . . . 162
21.1.3 Numeric Escapes . . . 164

22 Enumerated Types: `enum` 165
22.1 Behavior of `enum` . . . 165
22.1.1 Numbering . . . 165
22.1.2 Trailing Commas . . . 166
22.1.3 Scope . . . 166
22.1.4 Style . . . 166
22.2 Your `enum` is a Type . . . 166

23 Pointers III: Pointers to Pointers and More 169
23.1 Pointers to Pointers . . . 169
23.1.1 Pointer Pointers and `const` . . . 172
23.2 Multibyte Values . . . 173
23.3 The `NULL` Pointer and Zero . . . 174
23.4 Pointers as Integers . . . 174
23.5 Casting Pointers to other Pointers . . . 175
23.6 Pointer Differences . . . 177
23.7 Pointers to Functions . . . 178

24 Bitwise Operations 181
24.1 Bitwise AND, OR, XOR, and NOT . . . 181
24.2 Bitwise Shift . . . 181

25 Variadic Functions 183
25.1 Ellipses in Function Signatures . . . 183

25.2 Getting the Extra Arguments . . . 184
25.3 va_list Functionality . . . 185
25.4 Library Functions That Use va_lists . . . 186

26 Locale and Internationalization 187
26.1 Setting the Localization, Quick and Dirty . . . 187
26.2 Getting the Monetary Locale Settings . . . 188
26.2.1 Monetary Digit Grouping . . . 189
26.2.2 Separators and Sign Position . . . 190
26.2.3 Example Values . . . 190
26.3 Localization Specifics . . . 190

27 Unicode, Wide Characters, and All That 192
27.1 What is Unicode? . . . 192
27.2 Code Points . . . 192
27.3 Encoding . . . 193
27.4 Source and Execution Character Sets . . . 194
27.5 Unicode in C . . . 195
27.6 A Quick Note on UTF-8 Before We Swerve into the Weeds . . . 196
27.7 Different Character Types . . . 196
27.7.1 Multibyte Characters . . . 196
27.7.2 Wide Characters . . . 197
27.8 Using Wide Characters and wchar_t . . . 197
27.8.1 Multibyte to wchar_t Conversions . . . 198
27.9 Wide Character Functionality . . . 199
27.9.1 wint_t . . . 199
27.9.2 I/O Stream Orientation . . . 200
27.9.3 I/O Functions . . . 200
27.9.4 Type Conversion Functions . . . 200
27.9.5 String and Memory Copying Functions . . . 201
27.9.6 String and Memory Comparing Functions . . . 201
27.9.7 String Searching Functions . . . 201
27.9.8 Length/Miscellaneous Functions . . . 202
27.9.9 Character Classification Functions . . . 202
27.10 Parse State, Restartable Functions . . . 202
27.11 Unicode Encodings and C . . . 204
27.11.1 UTF-8 . . . 204
27.11.2 UTF-16, UTF-32, char16_t, and char32_t . . . 205
27.11.3 Multibyte Conversions . . . 205
27.11.4 Third-Party Libraries . . . 206

28 Exiting a Program 207
28.1 Normal Exits . . . 207
28.1.1 Returning From main() . . . 207
28.1.2 exit() . . . 207
28.1.3 Setting Up Exit Handlers with atexit() . . . 208
28.2 Quicker Exits with quick_exit() . . . 208
28.3 Nuke it from Orbit: _Exit() . . . 209
28.4 Exiting Sometimes: assert() . . . 209
28.5 Abnormal Exit: abort() . . . 210

29 Signal Handling 211
29.1 What Are Signals? . . . 211
29.2 Handling Signals with signal() . . . 211
29.3 Writing Signal Handlers . . . 212

www.dbooks.org

29.4 What Can We Actually Do? . . . 214
29.5 Friends Don't Let Friends `signal()` . . . 216

30 Variable-Length Arrays (VLAs) 217
30.1 The Basics . . . 217
30.2 `sizeof` and VLAs . . . 218
30.3 Multidimensional VLAs . . . 219
30.4 Passing One-Dimensional VLAs to Functions . . . 219
30.5 Passing Multi-Dimensional VLAs to Functions . . . 220
30.5.1 Partial Multidimensional VLAs . . . 221
30.6 Compatibility with Regular Arrays . . . 221
30.7 `typedef` and VLAs . . . 222
30.8 Jumping Pitfalls . . . 222
30.9 General Issues . . . 222

31 `goto` 224
31.1 A Simple Example . . . 224
31.2 Labeled `continue` . . . 225
31.3 Bailing Out . . . 226
31.4 Labeled `break` . . . 227
31.5 Multi-level Cleanup . . . 227
31.6 Tail Call Optimization . . . 228
31.7 Restarting Interrupted System Calls . . . 229
31.8 `goto` and Variable Scope . . . 230
31.9 `goto` and Variable-Length Arrays . . . 231

32 Types Part V: Compound Literals and Generic Selections 232
32.1 Compound Literals . . . 232
32.1.1 Passing Unnamed Objects to Functions . . . 233
32.1.2 Unnamed `structs` . . . 233
32.1.3 Pointers to Unnamed Objects . . . 234
32.1.4 Unnamed Objects and Scope . . . 234
32.1.5 Silly Unnamed Object Example . . . 235
32.2 Generic Selections . . . 235

33 Arrays Part II 239
33.1 Type Qualifiers for Arrays in Parameter Lists . . . 239
33.2 `static` for Arrays in Parameter Lists . . . 239
33.3 Equivalent Initializers . . . 240

34 Long Jumps with `setjmp`, `longjmp` 243
34.1 Using `setjmp` and `longjmp` . . . 243
34.2 Pitfalls . . . 244
34.2.1 The Values of Local Variables . . . 244
34.2.2 How Much State is Saved? . . . 245
34.2.3 You Can't Name Anything `setjmp` . . . 245
34.2.4 You Can't `setjmp()` in a Larger Expression . . . 245
34.2.5 When Can't You `longjmp()`? . . . 246
34.2.6 You Can't Pass `0` to `longjmp()` . . . 246
34.2.7 `longjmp()` and Variable Length Arrays . . . 246

35 Incomplete Types 247
35.1 Use Case: Self-Referential Structures . . . 247
35.2 Incomplete Type Error Messages . . . 248
35.3 Other Incomplete Types . . . 248

35.4 Use Case: Arrays in Header Files . . . 249
35.5 Completing Incomplete Types . . . 249

36 Complex Numbers 251
36.1 Complex Types . . . 251
36.2 Assigning Complex Numbers . . . 252
36.3 Constructing, Deconstructing, and Printing . . . 252
36.4 Complex Arithmetic and Comparisons . . . 253
36.5 Complex Math . . . 254
36.5.1 Trigonometry Functions . . . 255
36.5.2 Exponential and Logarithmic Functions . . . 255
36.5.3 Power and Absolute Value Functions . . . 255
36.5.4 Manipulation Functions . . . 255

37 Fixed Width Integer Types 256
37.1 The Bit-Sized Types . . . 256
37.2 Maximum Integer Size Type . . . 257
37.3 Using Fixed Size Constants . . . 257
37.4 Limits of Fixed Size Integers . . . 258
37.5 Format Specifiers . . . 258

38 Date and Time Functionality 260
38.1 Quick Terminology and Information . . . 260
38.2 Date Types . . . 261
38.3 Initialization and Conversion Between Types . . . 261
38.3.1 Converting `time_t` to `struct tm` . . . 262
38.3.2 Converting `struct tm` to `time_t` . . . 262
38.4 Formatted Date Output . . . 263
38.5 More Resolution with `timespec_get()` . . . 264
38.6 Differences Between Times . . . 264

39 Multithreading 266
39.1 Background . . . 266
39.2 Things You Can Do . . . 267
39.3 Data Races and the Standard Library . . . 267
39.4 Creating and Waiting for Threads . . . 267
39.5 Detaching Threads . . . 271
39.6 Thread Local Data . . . 272
39.6.1 `_Thread_local` Storage-Class . . . 274
39.6.2 Another Option: Thread-Specific Storage . . . 274
39.7 Mutexes . . . 276
39.7.1 Different Mutex Types . . . 279
39.8 Condition Variables . . . 280
39.8.1 Timed Condition Wait . . . 283
39.8.2 Broadcast: Wake Up All Waiting Threads . . . 284
39.9 Running a Function One Time . . . 284

40 Atomics 285
40.1 Testing for Atomic Support . . . 285
40.2 Atomic Variables . . . 285
40.3 Synchronization . . . 287
40.4 Acquire and Release . . . 289
40.5 Sequential Consistency . . . 290
40.6 Atomic Assignments and Operators . . . 291
40.7 Library Functions that Automatically Synchronize . . . 292

www.dbooks.org

40.8 Atomic Type Specifier, Qualifier . . . 292
40.9 Lock-Free Atomic Variables . . . 294
40.9.1 Signal Handlers and Lock-Free Atomics . . . 295
40.10 Atomic Flags . . . 295
40.11 Atomic `structs` and `unions` . . . 295
40.12 Atomic Pointers . . . 296
40.13 Memory Order . . . 297
40.13.1 Sequential Consistency . . . 298
40.13.2 Acquire . . . 298
40.13.3 Release . . . 298
40.13.4 Consume . . . 298
40.13.5 Acquire/Release . . . 298
40.13.6 Relaxed . . . 298
40.14 Fences . . . 299
40.15 References . . . 299

41 Function Specifiers, Alignment Specifiers/Operators 301
41.1 Function Specifiers . . . 301
41.1.1 `inline` for Speed—Maybe . . . 301
41.1.2 `noreturn` and `_Noreturn` . . . 304
41.2 Alignment Specifiers and Operators . . . 304
41.2.1 `alignas` and `_Alignas` . . . 305
41.2.2 `alignof` and `_Alignof` . . . 305

42 `<assert.h>` Runtime and Compile-time Diagnostics 307
42.1 Macros . . . 307
42.2 `assert()` . . . 307
42.3 `static_assert()` . . . 309

43 `<complex.h>` Complex Number Functionality 311
43.1 `cacos()` `cacosf()` `cacosl()` . . . 313
43.2 `casin()` `casinf()` `casinl()` . . . 314
43.3 `catan()` `catanf()` `catanl()` . . . 315
43.4 `ccos()` `ccosf()` `ccosl()` . . . 316
43.5 `csin()` `csinf()` `csinl()` . . . 317
43.6 `ctan()` `ctanf()` `ctanl()` . . . 317
43.7 `cacosh()` `cacoshf()` `cacoshl()` . . . 318
43.8 `casinh()` `casinhf()` `casinhl()` . . . 319
43.9 `catanh()` `catanhf()` `catanhl()` . . . 320
43.10 `ccosh()` `ccoshf()` `ccoshl()` . . . 321
43.11 `csinh()` `csinhf()` `csinhl()` . . . 322
43.12 `ctanh()` `ctanhf()` `ctanhl()` . . . 323
43.13 `cexp()` `cexpf()` `cexpl()` . . . 324
43.14 `clog()` `clogf()` `clogl()` . . . 325
43.15 `cabs()` `cabsf()` `cabsl()` . . . 325
43.16 `cpow()` `cpowf()` `cpowl()` . . . 326
43.17 `csqrt()` `csqrtf()` `csqrtl()` . . . 327
43.18 `carg()` `cargf()` `cargl()` . . . 328
43.19 `cimag()` `cimagf()` `cimagl()` . . . 329
43.20 `CMPLX()` `CMPLXF()` `CMPLXL()` . . . 330
43.21 `conj()` `conjf()` `conjl()` . . . 331
43.22 `cproj()` `cproj()` `cproj()` . . . 332
43.23 `creal()` `crealf()` `creall()` . . . 333

44 `<ctype.h>` Character Classification and Conversion 335

44.1 isalnum() . . . 336
44.2 isalpha() . . . 336
44.3 isblank() . . . 337
44.4 iscntrl() . . . 338
44.5 isdigit() . . . 339
44.6 isgraph() . . . 340
44.7 islower() . . . 341
44.8 isprint() . . . 341
44.9 ispunct() . . . 342
44.10 isspace() . . . 343
44.11 isupper() . . . 344
44.12 isxdigit() . . . 345
44.13 tolower() . . . 346
44.14 toupper() . . . 346

45 <errno.h> Error Information 348
45.1 errno . . . 348

46 <fenv.h> Floating Point Exceptions and Environment 351
46.1 Types and Macros . . . 351
46.2 Pragmas . . . 352
46.3 feclearexcept() . . . 352
46.4 fegetexceptflag() fesetexceptflag() . . . 353
46.5 feraiseexcept() . . . 354
46.6 fetestexcept() . . . 355
46.7 fegetround() fesetround() . . . 356
46.8 fegetenv() fesetenv() . . . 358
46.9 feholdexcept() . . . 359
46.10 feupdateenv() . . . 360

47 <float.h> Floating Point Limits 363
47.1 Background . . . 364
47.2 FLT_ROUNDS Details . . . 365
47.3 FLT_EVAL_METHOD Details . . . 365
47.4 Subnormal Numbers . . . 365
47.5 How Many Decimal Places Can I Use? . . . 366
47.6 Comprehensive Example . . . 367

48 <inttypes.h> More Integer Conversions 370
48.1 Macros . . . 370
48.2 imaxabs() . . . 371
48.3 imaxdiv() . . . 372
48.4 strtoimax() strtoumax() . . . 373
48.5 wcstoimax() wcstoumax() . . . 374

49 <iso646.h> Alternative Operator Spellings 376

50 <limits.h> Numeric Limits 377
50.1 CHAR_MIN and CHAR_MAX . . . 377
50.2 Choosing the Correct Type . . . 378
50.3 Whither Two's Complement? . . . 378
50.4 Demo Program . . . 378

51 <locale.h> locale handling 380
51.1 setlocale() . . . 380

www.dbooks.org

51.2 localeconv() 382

52 <math.h> Mathematics 386
52.1 Math Function Idioms 387
52.2 Math Types 388
52.3 Math Macros 388
52.4 Math Errors 388
52.5 Math Pragmas 389
52.6 fpclassify() 389
52.7 isfinite(), isinf(), isnan(), isnormal() 391
52.8 signbit() 392
52.9 acos(), acosf(), acosl() 393
52.10 asin(), asinf(), asinl() 394
52.11 atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() 395
52.12 cos(), cosf(), cosl() 396
52.13 sin(), sinf(), sinl() 397
52.14 tan(), tanf(), tanl() 398
52.15 acosh(), acoshf(), acoshl() 399
52.16 asinh(), asinhf(), asinhl() 399
52.17 atanh(), atanhf(), atanhl() 400
52.18 cosh(), coshf(), coshl() 401
52.19 sinh(), sinhf(), sinhl() 402
52.20 tanh(), tanhf(), tanhl() 402
52.21 exp(), expf(), expl() 403
52.22 exp2(), exp2f(), exp2l() 404
52.23 expm1(), expm1f(), expm1l() 405
52.24 frexp(), frexpf(), frexpl() 406
52.25 ilogb(), ilogbf(), ilogbl() 407
52.26 ldexp(), ldexpf(), ldexpl() 408
52.27 log(), logf(), logl() 409
52.28 log10(), log10f(), log10l() 410
52.29 log1p(), log1pf(), log1pl() 410
52.30 log2(), log2f(), log2l() 411
52.31 logb(), logbf(), logbl() 412
52.32 modf(), modff(), modfl() 413
52.33 scalbn(), scalbnf(), scalbnl() scalbln(), scalblnf(), scalblnl() 415
52.34 cbrt(), cbrtf(), cbrtl() 416
52.35 fabs(), fabsf(), fabsl() 417
52.36 hypot(), hypotf(), hypotl() 418
52.37 pow(), powf(), powl() 418
52.38 sqrt() 419
52.39 erf(), erff(), erfl() 420
52.40 erfc(), erfcf(), erfcl() 421
52.41 lgamma(), lgammaf(), lgammal() 422
52.42 tgamma(), tgammaf(), tgammal() 423
52.43 ceil(), ceilf(), ceill() 424
52.44 floor(), floorf(), floorl() 425
52.45 nearbyint(), nearbyintf(), nearbyintl() 426
52.46 rint(), rintf(), rintl() 427
52.47 lrint(), lrintf(), lrintl(), llrint(), llrintf(), llrintl() 427
52.48 round(), roundf(), roundl() 429
52.49 lround(), lroundf(), lroundl() llround(), llroundf(), llroundl() 429
52.50 trunc(), truncf(), truncl() 430
52.51 fmod(), fmodf(), fmodl() 431

52.52 remainder(), remainderf(), remainderl() . . . 432
52.53 remquo(), remquof(), remquol() . . . 433
52.54 copysign(), copysignf(), copysignl() . . . 435
52.55 nan(), nanf(), nanl() . . . 436
52.56 nextafter(), nextafterf(), nextafterl() . . . 437
52.57 nexttoward(), nexttowardf(), nexttowardl() . . . 438
52.58 fdim(), fdimf(), fdiml() . . . 439
52.59 fmax(), fmaxf(), fmaxl(), fmin(), fminf(), fminl() . . . 439
52.60 fma(), fmaf(), fmal() . . . 440
52.61 isgreater(), isgreaterequal(), isless(), islessequal() . . . 441
52.62 islessgreater() . . . 442
52.63 isunordered() . . . 443

53 <setjmp.h> Non-local Goto 444
53.1 setjmp() . . . 444
53.2 longjmp() . . . 446

54 <signal.h> signal handling 449
54.1 signal() . . . 449
54.2 raise() . . . 453

55 <stdalign.h> Macros for Alignment 455
55.1 alignas() _Alignas() . . . 455
55.2 alignof() _Alignof() . . . 457

56 <stdarg.h> Variable Arguments 459
56.1 va_arg() . . . 459
56.2 va_copy() . . . 460
56.3 va_end() . . . 462
56.4 va_start() . . . 463

57 <stdatomic.h> Atomic-Related Functions 466
57.1 Atomic Types . . . 467
57.2 Lock-free Macros . . . 467
57.3 Atomic Flag . . . 468
57.4 Memory Order . . . 468
57.5 ATOMIC_VAR_INIT() . . . 469
57.6 atomic_init() . . . 469
57.7 kill_dependency() . . . 470
57.8 atomic_thread_fence() . . . 471
57.9 atomic_signal_fence() . . . 473
57.10 atomic_is_lock_free() . . . 474
57.11 atomic_store() . . . 475
57.12 atomic_load() . . . 476
57.13 atomic_exchange() . . . 477
57.14 atomic_compare_exchange_*() . . . 478
57.15 atomic_fetch_*() . . . 480
57.16 atomic_flag_test_and_set() . . . 482
57.17 atomic_flag_clear() . . . 484

58 <stdbool.h> Boolean Types 486
58.1 Example . . . 486
58.2 _Bool? . . . 487

59 <stddef.h> A Few Standard Definitions 488

www.dbooks.org

59.1 ptrdiff_t 488
59.2 size_t 489
59.3 max_align_t 489
59.4 wchar_t 490
59.5 offsetof 490

60 <stdint.h> More Integer Types 491
60.1 Specific-Width Integers 491
60.2 Other Integer Types 492
60.3 Macros 492
60.4 Other Limits 493
60.5 Macros for Declaring Constants 493

61 <stdio.h> Standard I/O Library 494
61.1 remove() 496
61.2 rename() 497
61.3 tmpfile() 498
61.4 tmpnam() 499
61.5 fclose() 500
61.6 fflush() 501
61.7 fopen() 503
61.8 freopen() 504
61.9 setbuf(), setvbuf() 505
61.10 printf(), fprintf(), sprintf(), snprintf() 507
61.11 scanf(), fscanf(), sscanf() 513
61.12 vprintf(), vfprintf(), vsprintf(), vsnprintf() 519
61.13 vscanf(), vfscanf(), vsscanf() 521
61.14 getc(), fgetc(), getchar() 522
61.15 gets(), fgets() 523
61.16 putc(), fputc(), putchar() 525
61.17 puts(), fputs() 526
61.18 ungetc() 527
61.19 fread() 528
61.20 fwrite() 529
61.21 fgetpos(), fsetpos() 530
61.22 fseek(), rewind() 532
61.23 ftell() 533
61.24 feof(), ferror(), clearerr() 534
61.25 perror() 536

62 <stdlib.h> Standard Library Functions 538
62.1 <stdlib.h> Types and Macros 539
62.2 atof() 539
62.3 atoi(), atol(), atoll() 540
62.4 strtod(), strtof(), strtold() 541
62.5 strtol(), strtoll(), strtoul(), strtoull() 543
62.6 rand() 545
62.7 srand() 547
62.8 aligned_alloc() 548
62.9 calloc(), malloc() 550
62.10 free() 551
62.11 realloc() 552
62.12 abort() 553
62.13 atexit(), at_quick_exit() 554

62.14 exit(), quick_exit(), _Exit() . . . 556
62.15 getenv() . . . 557
62.16 system() . . . 558
62.17 bsearch() . . . 559
62.18 qsort() . . . 561
62.19 abs(), labs(), llabs() . . . 562
62.20 div(), ldiv(), lldiv() . . . 563
62.21 mblen() . . . 565
62.22 mbtowc() . . . 566
62.23 wctomb() . . . 567
62.24 mbstowcs() . . . 568
62.25 wcstombs() . . . 570

63 <stdnoreturn.h> Macros for Non-Returning Functions 572

64 <string.h> String Manipulation 573
64.1 memcpy(), memmove() . . . 574
64.2 strcpy(), strncpy() . . . 574
64.3 strcat(), strncat() . . . 576
64.4 strcmp(), strncmp(), memcmp() . . . 577
64.5 strcoll() . . . 578
64.6 strxfrm() . . . 579
64.7 strchr(), strrchr(), memchr() . . . 582
64.8 strspn(), strcspn() . . . 583
64.9 strpbrk() . . . 584
64.10 strstr() . . . 585
64.11 strtok() . . . 586
64.12 memset() . . . 587
64.13 strerror() . . . 588
64.14 strlen() . . . 589

65 <tgmath.h> Type-Generic Math Functions 591
65.1 Example . . . 592

66 <threads.h> Multithreading Functions 594
66.1 call_once() . . . 595
66.2 cnd_broadcast() . . . 596
66.3 cnd_destroy() . . . 599
66.4 cnd_init() . . . 600
66.5 cnd_signal() . . . 602
66.6 cnd_timedwait() . . . 603
66.7 cnd_wait() . . . 606
66.8 mtx_destroy() . . . 607
66.9 mtx_init() . . . 608
66.10 mtx_lock() . . . 610
66.11 mtx_timedlock() . . . 612
66.12 mtx_trylock() . . . 614
66.13 mtx_unlock() . . . 615
66.14 thrd_create() . . . 617
66.15 thrd_current() . . . 619
66.16 thrd_detach() . . . 620
66.17 thrd_equal() . . . 621
66.18 thrd_exit() . . . 623
66.19 thrd_join() . . . 624
66.20 thrd_sleep() . . . 625

www.dbooks.org

66.21 thrd_yield() . . . 626
66.22 tss_create() . . . 628
66.23 tss_delete() . . . 630
66.24 tss_get() . . . 632
66.25 tss_set() . . . 634

67 <time.h> Date and Time Functions 637
67.1 Thread Safety Warning . . . 638
67.2 clock() . . . 638
67.3 difftime() . . . 639
67.4 mktime() . . . 640
67.5 time() . . . 642
67.6 timespec_get() . . . 643
67.7 asctime() . . . 645
67.8 ctime() . . . 646
67.9 gmtime() . . . 646
67.10 localtime() . . . 647
67.11 strftime() . . . 648

68 <uchar.h> Unicode utility functions 653
68.1 Types . . . 653
68.2 OS X issue . . . 654
68.3 mbrtoc16() mbrtoc32() . . . 654
68.4 c16rtomb() c32rtomb() . . . 657

69 <wchar.h> Wide Character Handling 660
69.1 Restartable Functions . . . 661
69.2 wprintf(), fwprintf(), swprintf() . . . 662
69.3 wscanf() fwscanf() swscanf() . . . 663
69.4 vwprintf() vfwprintf() vswprintf() . . . 664
69.5 vwscanf(), vfwscanf(), vswscanf() . . . 665
69.6 getwc() fgetwc() getwchar() . . . 666
69.7 fgetws() . . . 667
69.8 putwchar() putwc() fputwc() . . . 668
69.9 fputws() . . . 670
69.10 fwide() . . . 670
69.11 ungetwc() . . . 672
69.12 wcstod() wcstof() wcstold() . . . 673
69.13 wcstol() wcstoll() wcstoul() wcstoull() . . . 675
69.14 wcscpy() wcsncpy() . . . 676
69.15 wmemcpy() wmemmove() . . . 677
69.16 wcscat() wcsncat() . . . 678
69.17 wcscmp(), wcsncmp(), wmemcmp() . . . 679
69.18 wcscoll() . . . 680
69.19 wcsxfrm() . . . 681
69.20 wcschr() wcsrchr() . . . 683
69.21 wcsspn() wcscspn() . . . 684
69.22 wcspbrk() . . . 685
69.23 wcsstr() . . . 686
69.24 wcstok() . . . 687
69.25 wcslen() . . . 688
69.26 wcsftime() . . . 689
69.27 btowc() wctob() . . . 690
69.28 mbsinit() . . . 691

69.29 mbrlen() 692
69.30 mbrtowc() 693
69.31 wcrtomb() 695
69.32 mbsrtowcs() 696
69.33 wcsrtombs() 698

70 <wctype.h> Wide Character Classification and Transformation 701
70.1 iswalnum() 701
70.2 iswalpha() 702
70.3 iswblank() 703
70.4 iswcntrl() 704
70.5 iswdigit() 705
70.6 iswgraph() 706
70.7 iswlower() 706
70.8 iswprint() 707
70.9 iswpunct() 708
70.10 iswspace() 709
70.11 iswupper() 710
70.12 iswxdigit() 711
70.13 iswctype() 711
70.14 wctype() 713
70.15 towlower() 715
70.16 towupper() 716
70.17 towctrans() 716
70.18 wctrans() 718

71 Exercises 720
71.1 Intro 720
71.2 Variables and Statements 720
71.3 Functions 721
71.4 Pointers 721
71.5 Arrays 722
71.6 Strings 722
71.7 Structs 722
71.8 I/O 722
71.9 Typedef 723
71.10 Pointers II 723
71.11 Manual Memory Allocation 724
71.12 Scope 724
71.13 Types II 725
71.14 Types III 725
71.15 Types IV 726
71.16 Multifile Projects 726
71.17 The Outside Environment 727
71.18 The C Preprocessor 727

www.dbooks.org

\index{Modulus|see {\texttt{%} modulus operator}}

Chapter 1

Foreword

No point in wasting words here, folks, let's jump straight into the C code:

```
E((ck?main((z?(stat(M,&t)?P+=a+'{'?0:3:
execv(M,k),a=G,i=P,y=G&255,
sprintf(Q,y/'@'-3?A(*L(V(%d+%d)+%d,0)
```

And they lived happily ever after. The End.

What's this? You say something's still not clear about this whole C programming language thing?

Well, to be quite honest, I'm not even sure what the above code does. It's a snippet from one of the entries in the 2001 International Obfuscated C Code Contest[1], a wonderful competition wherein the entrants attempt to write the most unreadable C code possible, with often surprising results.

The bad news is that if you're a beginner in this whole thing, all C code you see probably looks obfuscated! The good news is, it's not going to be that way for long.

What we'll try to do over the course of this guide is lead you from complete and utter sheer lost confusion on to the sort of enlightened bliss that can only be obtained though pure C programming. Right on.

1.1 Audience

This guide assumes that you've already got some programming knowledge under your belt from another language, such as Python[2], JavaScript[3], Java[4], Rust[5], Go[6], Swift[7], etc. (Objective-C[8] devs will have a particularly easy time of it!)

We're going to assume you know what variables are, what loops do, how functions work, and so on.

If that's not you for whatever reason the best I can hope to provide is some honest entertainment for your reading pleasure. The only thing I can reasonably promise is that this guide won't end on a cliffhanger… or *will* it?

[1] https://www.ioccc.org/
[2] https://en.wikipedia.org/wiki/Python_(programming_language)
[3] https://en.wikipedia.org/wiki/JavaScript
[4] https://en.wikipedia.org/wiki/Java_(programming_language)
[5] https://en.wikipedia.org/wiki/Rust_(programming_language)
[6] https://en.wikipedia.org/wiki/Go_(programming_language)
[7] https://en.wikipedia.org/wiki/Swift_(programming_language)
[8] https://en.wikipedia.org/wiki/Objective-C

www.dbooks.org

1.2 How to Read This Book

The book is in two parts: the tutorial part and the reference part.

If you're new, go through the tutorial part in order, generally. The higher you get in chapters, the less important it is to go in order.

And no matter your skill level, the reference part is there with complete examples of the standard library function calls to help refresh your memory whenever needed. Good for reading over a bowl of cereal or other time.

1.3 Platform and Compiler

I'll try to stick to Plain Ol'-Fashioned ISO-standard C[9]. Well, for the most part. Here and there I might go crazy and start talking about POSIX[10] or something, but we'll see.

Unix users (e.g. Linux, BSD, etc.) try running `cc` or `gcc` from the command line–you might already have a compiler installed. If you don't, search your distribution for installing `gcc` or `clang`.

Windows users should check out Visual Studio Community[11]. Or, if you're looking for a more Unix-like experience (recommended!), install WSL[12] and `gcc`.

Mac users will want to install XCode[13], and in particular the command line tools.

There are a lot of compilers out there, and virtually all of them will work for this book. And a C++ compiler will compile a lot of (but not all!) C code. Best use a proper C compiler if you can.

1.4 Official Homepage

This official location of this document is https://beej.us/guide/bgc/[14]. Maybe this'll change in the future, but it's more likely that all the other guides are migrated off Chico State computers.

1.5 Email Policy

I'm generally available to help out with email questions so feel free to write in, but I can't guarantee a response. I lead a pretty busy life and there are times when I just can't answer a question you have. When that's the case, I usually just delete the message. It's nothing personal; I just won't ever have the time to give the detailed answer you require.

As a rule, the more complex the question, the less likely I am to respond. If you can narrow down your question before mailing it and be sure to include any pertinent information (like platform, compiler, error messages you're getting, and anything else you think might help me troubleshoot), you're much more likely to get a response.

If you don't get a response, hack on it some more, try to find the answer, and if it's still elusive, then write me again with the information you've found and hopefully it will be enough for me to help out.

Now that I've badgered you about how to write and not write me, I'd just like to let you know that I *fully* appreciate all the praise the guide has received over the years. It's a real morale boost, and it gladdens me to hear that it is being used for good! `:-)` Thank you!

[9] https://en.wikipedia.org/wiki/ANSI_C
[10] https://en.wikipedia.org/wiki/POSIX
[11] https://visualstudio.microsoft.com/vs/community/
[12] https://docs.microsoft.com/en-us/windows/wsl/install-win10
[13] https://developer.apple.com/xcode/
[14] https://beej.us/guide/bgc/

1.6 Mirroring

You are more than welcome to mirror this site, whether publicly or privately. If you publicly mirror the site and want me to link to it from the main page, drop me a line at `beej@beej.us`.

1.7 Note for Translators

If you want to translate the guide into another language, write me at `beej@beej.us` and I'll link to your translation from the main page. Feel free to add your name and contact info to the translation.

Please note the license restrictions in the Copyright and Distribution section, below.

1.8 Copyright and Distribution

Beej's Guide to C is Copyright © 2021 Brian "Beej Jorgensen" Hall.

With specific exceptions for source code and translations, below, this work is licensed under the Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 License. To view a copy of this license, visit `https://creativecommons.org/licenses/by-nc-nd/3.0/` or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

One specific exception to the "No Derivative Works" portion of the license is as follows: this guide may be freely translated into any language, provided the translation is accurate, and the guide is reprinted in its entirety. The same license restrictions apply to the translation as to the original guide. The translation may also include the name and contact information for the translator.

The C source code presented in this document is hereby granted to the public domain, and is completely free of any license restriction.

Educators are freely encouraged to recommend or supply copies of this guide to their students.

Contact `beej@beej.us` for more information.

1.9 Dedication

The hardest things about writing these guides are:

- Learning the material in enough detail to be able to explain it
- Figuring out the best way to explain it clearly, a seemingly-endless iterative process
- Putting myself out there as a so-called *authority*, when really I'm just a regular human trying to make sense of it all, just like everyone else
- Keeping at it when so many other things draw my attention

A lot of people have helped me through this process, and I want to acknowledge those who have made this book possible.

- Everyone on the Internet who decided to help share their knowledge in one form or another. The free sharing of instructive information is what makes the Internet the great place that it is.
- The volunteers at cppreference.com[15] who provide the bridge that leads from the spec to the real world.
- The helpful and knowledgeable folks on comp.lang.c[16] and r/C_Programming[17] who got me through the tougher parts of the language.
- Everyone who submitted corrections and pull-requests on everything from misleading instructions to typos.

[15] https://en.cppreference.com/
[16] https://groups.google.com/g/comp.lang.c
[17] https://www.reddit.com/r/C_Programming/

www.dbooks.org

Thank you! ♥

Chapter 2

Hello, World!

2.1 What to Expect from C

> *"Where do these stairs go?"*
> *"They go up."*
>
> —Ray Stantz and Peter Venkman, Ghostbusters

C is a low-level language.

It didn't used to be. Back in the day when people carved punch cards out of granite, C was an incredible way to be free of the drudgery of lower-level languages like assembly[1].

But now in these modern times, current-generation languages offer all kinds of features that didn't exist in 1972 when C was invented. This means C is a pretty basic language with not a lot of features. It can do *anything*, but it can make you work for it.

So why would we even use it today?

- As a learning tool: not only is C a venerable piece of computing history, but it is connected to the bare metal[2] in a way that present-day languages are not. When you learn C, you learn about how software interfaces with computer memory at a low level. There are no seatbelts. You'll write software that crashes, I assure you. And that's all part of the fun!
- As a useful tool: C still is used for certain applications, such as building operating systems[3] or in embedded systems[4]. (Though the Rust[5] programming language is eyeing both these fields!)

If you're familiar with another language, a lot of things about C are easy. C inspired many other languages, and you'll see bits of it in Go, Rust, Swift, Python, JavaScript, Java, and all kinds of other languages. Those parts will be familiar.

The one thing about C that hangs people up is *pointers*. Virtually everything else is familiar, but pointers are the weird one. The concept behind pointers is likely one you already know, but C forces you to be explicit about it, using operators you've likely never seen before.

It's especially insidious because once you *grok*[6] pointers, they're suddenly easy. But up until that moment, they're slippery eels.

[1] https://en.wikipedia.org/wiki/Assembly_language
[2] https://en.wikipedia.org/wiki/Bare_machine
[3] https://en.wikipedia.org/wiki/Operating_system
[4] https://en.wikipedia.org/wiki/Embedded_system
[5] https://en.wikipedia.org/wiki/Rust_(programming_language)
[6] https://en.wikipedia.org/wiki/Grok

www.dbooks.org

Everything else in C is just memorizing another way (or sometimes the same way!) of doing something you've done already. Pointers are the weird bit. And, arguably, even pointers are variations on a theme you're probably familiar with.

So get ready for a rollicking adventure as close to the core of the computer as you can get without assembly, in the most influential computer language of all time[7]. Hang on!

2.2 Hello, World!

This is the canonical example of a C program. Everyone uses it. (Note that the numbers to the left are for reader reference only, and are not part of the source code.)

```
1  /* Hello world program */
2
3  #include <stdio.h>
4
5  int main(void)
6  {
7      printf("Hello, World!\n");  // Actually do the work here
8  }
```

We're going to don our long-sleeved heavy-duty rubber gloves, grab a scalpel, and rip into this thing to see what makes it tick. So, scrub up, because here we go. Cutting very gently…

Let's get the easy thing out of the way: anything between the digraphs `/*` and `*/` is a comment and will be completely ignored by the compiler. Same goes for anything on a line after a `//`. This allows you to leave messages to yourself and others, so that when you come back and read your code in the distant future, you'll know what the heck it was you were trying to do. Believe me, you will forget; it happens.

Now, what is this `#include`? GROSS! Well, it tells the C Preprocessor to pull the contents of another file and insert it into the code right *there*.

Wait—what's a C Preprocessor? Good question. There are two stages[8] to compilation: the preprocessor and the compiler. Anything that starts with pound sign, or "octothorpe", (#) is something the preprocessor operates on before the compiler even gets started. Common *preprocessor directives*, as they're called, are `#include` and `#define`. More on that later.

Before we go on, why would I even begin to bother pointing out that a pound sign is called an octothorpe? The answer is simple: I think the word octothorpe is so excellently funny, I have to gratuitously spread its name around whenever I get the opportunity. Octothorpe. Octothorpe, octothorpe, octothorpe.

So *anyway*. After the C preprocessor has finished preprocessing everything, the results are ready for the compiler to take them and produce assembly code[9], machine code[10], or whatever it's about to do. Machine code is the "language" the CPU understands, and it can understand it *very rapidly*. This is one of the reasons C programs tend to be quick.

Don't worry about the technical details of compilation for now; just know that your source runs through the preprocessor, then the output of that runs through the compiler, then that produces an executable for you to run.

What about the rest of the line? What's `<stdio.h>`? That is what is known as a *header file*. It's the dot-h at the end that gives it away. In fact it's the "Standard I/O" (`stdio`) header file that you will grow to know and love. It gives us access to a bunch of I/O functionality[11]. For our demo program, we're outputting the string

[7] I know someone will fight me on that, but it's gotta be at least in the top three, right?
[8] Well, technically there are more than two, but hey, let's pretend there are two—ignorance is bliss, right?
[9] https://en.wikipedia.org/wiki/Assembly_language
[10] https://en.wikipedia.org/wiki/Machine_code
[11] Technically, it contains preprocessor directives and function prototypes (more on that later) for common input and output needs.

"Hello, World!", so we in particular need access to the `printf()` function to do this. The `<stdio.h>` file gives us this access. Basically, if we tried to use without `#include <stdio.h>`, the compiler would have complained to us about it.

How did I know I needed to `#include <stdio.h>` for `printf()`? Answer: it's in the documentation. If you're on a Unix system, `man 3 printf` and it'll tell you right at the top of the man page what header files are required. Or see the reference section in this book. `:-)`

Holy moly. That was all to cover the first line! But, let's face it, it has been completely dissected. No mystery shall remain!

So take a breather…look back over the sample code. Only a couple easy lines to go.

Welcome back from your break! I know you didn't really take a break; I was just humoring you.

The next line is `main()`. This is the definition of the function `main()`; everything between the squirrelly braces (`{` and `}`) is part of the function definition.

(How do you *call* a different function, anyway? The answer lies in the `printf()` line, but we'll get to that in a minute.)

Now, the main function is a special one in many ways, but one way stands above the rest: it is the function that will be called automatically when your program starts executing. Nothing of yours gets called before `main()`. In the case of our example, this works fine since all we want to do is print a line and exit.

Oh, that's another thing: once the program executes past the end of `main()`, down there at the closing squirrelly brace, the program will exit, and you'll be back at your command prompt.

So now we know that that program has brought in a header file, `stdio.h`, and declared a `main()` function that will execute when the program is started. What are the goodies in `main()`?

I am so happy you asked. Really! We only have the one goodie: a call to the function `printf()`. You can tell this is a function call and not a function definition in a number of ways, but one indicator is the lack of squirrelly braces after it. And you end the function call with a semicolon so the compiler knows it's the end of the expression. You'll be putting semicolons after almost everything, as you'll see.

You're passing one argument to the function `printf()`: a string to be printed when you call it. Oh, yeah—we're calling a function! We rock! Wait, wait—don't get cocky. What's that crazy `\n` at the end of the string? Well, most characters in the string will print out just like they are stored. But there are certain characters that you can't print on screen well that are embedded as two-character backslash codes. One of the most popular is `\n` (read "backslash-N") that corresponds to the *newline* character. This is the character that causes further printing to continue at the beginning of the next line instead of the current. It's like hitting return at the end of the line.

So copy that code into a file called `hello.c` and build it. On a Unix-like platform (e.g. Linux, BSD, Mac, or WSL), from the command line you'll build with a command like so:

```
gcc -o hello hello.c
```

(This means "compile `hello.c`, and output an executable called `hello`".)

After that's done, you should have a file called `hello` that you can run with this command:

```
./hello
```

(The leading `./` tells the shell to "run from the current directory".)

And see what happens:

```
Hello, World!
```

It's done and tested! Ship it!

www.dbooks.org

2.3 Compilation Details

Let's talk a bit more about how to build C programs, and what happens behind the scenes there.

Like other languages, C has *source code*. But, depending on what language you're coming from, you might never have had to *compile* your source code into an *executable*.

Compilation is the process of taking your C source code and turning it into a program that your operating system can execute.

JavaScript and Python devs aren't used to a separate compilation step at all–though behind the scenes it's happening! Python compiles your source code into something called *bytecode* that the Python virtual machine can execute. Java devs are used to compilation, but that produces bytecode for the Java Virtual Machine.

When compiling C, *machine code* is generated. This is the 1s and 0s that can be executed directly and speedily by the CPU.

> Languages that typically aren't compiled are called *interpreted* languages. But as we mentioned with Java and Python, they also have a compilation step. And there's no rule saying that C can't be interpreted. (There are C interpreters out there!) In short, it's a bunch of gray areas. Compilation in general is just taking source code and turning it into another, more easily-executed form.

The C compiler is the program that does the compilation.

As we've already said, `gcc` is a compiler that's installed on a lot of Unix-like operating systems[12]. And it's commonly run from the command line in a terminal, but not always. You can run it from your IDE, as well.

So how do we do command line builds?

2.4 Building with `gcc`

If you have a source file called `hello.c` in the current directory, you can build that into a program called `hello` with this command typed in a terminal:

```
gcc -o hello hello.c
```

The `-o` means "output to this file"[13]. And there's `hello.c` at the end, the name of the file we want to compile.

If your source is broken up into multiple files, you can compile them all together (almost as if they were one file, but the rules are actually more complex than that) by putting all the `.c` files on the command line:

```
gcc -o awesomegame ui.c characters.c npc.c items.c
```

and they'll all get built together into a big executable.

That's enough to get started—later we'll talk details about multiple source files, object files, and all kinds of fun stuff.

2.5 Building with `clang`

On Macs, the compiler isn't `gcc`—it's `clang`[t[clang]T]. But a wrapper is also installed so you can run `gcc` and have it still work.

[12] https://en.wikipedia.org/wiki/Unix

[13] If you don't give it an output filename, it will export to a file called `a.out` by default—this filename has its roots deep in Unix history.

2.6 Building from IDEs

If you're using an *Integrated Development Environment* (IDE), you probably don't have to build from the command line.

With Visual Studio, `CTRL-F7` will build, and `CTRL-F5` will run.

With VS Code, things are more complex, but you can hit `F5` to run via the debugger. (You'll have to install the C/C++ Extension.)

With XCode, you can build with `COMMAND-B` and run with `COMMAND-R`. To get the command line tools, Google for "XCode command line tools" and you'll find instructions for installing them.

For getting started, I encourage you to also try to build from the command line—it's history!

2.7 C Versions

C has come a long way over the years, and it had many named version numbers to describe which dialect of the language you're using.

These generally refer to the year of the specification.

The most famous are C89, C99, C11, and C2x. We'll focus on the latter in this book.

But here's a more complete table:

Version	Description
K&R C	1978, the original. Named after Brian Kernighan and Dennis Ritchie. Ritchie designed and coded the language, and Kernighan co-authored the book on it. You rarely see original K&R code today. If you do, it'll look odd, like Middle English looks odd to modern English readers.
C89, ANSI C, C90	In 1989, the American National Standards Institute (ANSI) produced a C language specification that set the tone for C that persists to this day. A year later, the reins were handed to the International Organization for Standardization (ISO) that produced the identical C90.
C95	A rarely-mentioned addition to C89 that included wide character support.
C99	The first big overhaul with lots of language additions. The thing most people remember is the addition of `//`-style comments. This is the most popular version of C in use as of this writing.
C11	This major version update includes Unicode support and multi-threading. Be advised that if you start using these language features, you might be sacrificing portability with places that are stuck in C99 land. But, honestly, 1999 is getting to be a while back now.
C17, C18	Bugfix update to C11. C17 seems to be the official name, but the publication was delayed until 2018. As far as I can tell, these two are interchangeable, with C17 being preferred.
C2x	What's coming next! Expected to eventually become C21.

You can force GCC to use one of these standards with the `-std=` command line argument. If you want it to be picky about the standard, add `-pedantic`.

For example:

```
gcc -std=c11 -pedantic foo.c
```

For this book, I compile programs for C2x with all warnings set:

www.dbooks.org

```
gcc -Wall -Wextra -std=c2x -pedantic foo.c
```

Chapter 3

Variables and Statements

"It takes all kinds to make a world, does it not, Padre?"
"So it does, my son, so it does."

—Pirate Captain Thomas Bartholomew Red to the Padre, Pirates

There sure can be lotsa stuff in a C program.

Yup.

And for various reasons, it'll be easier for all of us if we classify some of the types of things you can find in a program, so we can be clear what we're talking about.

3.1 Variables

It's said that "variables hold values". But another way to think about it is that a variable is a human-readable name that refers to some data in memory.

We're going to take a second here and take a peek down the rabbit hole that is pointers. Don't worry about it.

You can think of memory as a big array of bytes[1]. Data is stored in this "array"[2]. If a number is larger than a single byte, it is stored in multiple bytes. Because memory is like an array, each byte of memory can be referred to by its index. This index into memory is also called an *address*, or a *location*, or a *pointer*.

When you have a variable in C, the value of that variable is in memory *somewhere*, at some address. Of course. After all, where else would it be? But it's a pain to refer to a value by its numeric address, so we make a name for it instead, and that's what the variable is.

The reason I'm bringing all this up is twofold:

1. It's going to make it easier to understand pointer variables later—they're variables that hold the address of other variables!
2. Also, it's going to make it easier to understand pointers later.

So a variable is a name for some data that's stored in memory at some address.

[1] A "byte" is typically an 8-bit binary number. Think of it as an integer that can only hold the values from 0 to 255, inclusive. Technically, C allows bytes to be any number of bits and if you want to unambiguously refer to an 8-bit number, you should use the term *octet*. But programmers are going assume you mean 8-bits when you say "byte" unless you specify otherwise.

[2] I'm seriously oversimplifying how modern memory works, here. But the mental model works, so please forgive me.

www.dbooks.org

3.1.1 Variable Names

You can use any characters in the range 0-9, A-Z, a-z, and underscore for variable names, with the following rules:

- You can't start a variable with a digit 0-9.
- You can't start a variable name with two underscores.
- You can't start a variable name with an underscore followed by a capital A-Z.

For Unicode, just try it. There are some rules in the spec in §D.2 that talk about which Unicode codepoint ranges are allowed in which parts of identifiers, but that's too much to write about here and is probably something you'll never have to think about anyway.

3.1.2 Variable Types

Depending on which languages you already have in your toolkit, you might or might not be familiar with the idea of types. But C's kinda picky about them, so we should do a refresher.

Some example types, some of the most basic:

Type	Example	C Type
Integer	`3490`	`int`
Floating point	`3.14159`	`float`
Character (single)	`'c'`	`char`
String	`"Hello, world!"`	`char *`[3]

C makes an effort to convert automatically between most numeric types when you ask it to. But other than that, all conversions are manual, notably between string and numeric.

Almost all of the types in C are variants on these types.

Before you can use a variable, you have to *declare* that variable and tell C what type the variable holds. Once declared, the type of variable cannot be changed later at runtime. What you set it to is what it is until it falls out of scope and is reabsorbed into the universe.

Let's take our previous "Hello, world" code and add a couple variables to it:

```c
#include <stdio.h>

int main(void)
{
    int i;    // Holds signed integers, e.g. -3, -2, 0, 1, 10
    float f;  // Holds signed floating point numbers, e.g. -3.1416

    printf("Hello, World!\n");  // Ah, blessed familiarity
}
```

There! We've declared a couple of variables. We haven't used them yet, and they're both uninitialized. One holds an integer number, and the other holds a floating point number (a real number, basically, if you have a math background).

Uninitialized variables have indeterminate value[4]. They have to be initialized or else you must assume they contain some nonsense number.

[3] Read this as "pointer to a char" or "char pointer". "Char" for character. Though I can't find a study, it seems anecdotally most people pronounce this as "char", a minority say "car", and a handful say "care". We'll talk more about pointers later.

[4] Colloquially, we say they have "random" values, but they aren't truly—or even pseudo-truly—random numbers.

> This is one of the places C can "get you". Much of the time, in my experience, the indeterminate value is zero… but it can vary from run to run! Never assume the value will be zero, even if you see it is. *Always* explicitly initialize variables to some value before you use them[5].

What's this? You want to store some numbers in those variables? Insanity!

Let's go ahead and do that:

```
int main(void)
{
    int i;

    i = 2; // Assign the value 2 into the variable i

    printf("Hello, World!\n");
}
```

Killer. We've stored a value. Let's print it.

We're going to do that by passing *two* amazing arguments to the `printf()` function. The first argument is a string that describes what to print and how to print it (called the *format string*), and the second is the value to print, namely whatever is in the variable `i`.

`printf()` hunts through the format string for a variety of special sequences which start with a percent sign (`%`) that tell it what to print. For example, if it finds a `%d`, it looks to the next parameter that was passed, and prints it out as an integer. If it finds a `%f`, it prints the value out as a float. If it finds a `%s`, it prints a string.

As such, we can print out the value of various types like so:

```
#include <stdio.h>

int main(void)
{
    int i = 2;
    float f = 3.14;
    char *s = "Hello, world!";  // char * ("char pointer") is the string type

    printf("%s  i = %d and f = %f!\n", s, i, f);
}
```

And the output will be:

```
Hello, world!  i = 2 and f = 3.14!
```

In this way, `printf()` might be similar to various types of format strings or parameterized strings in other languages you're familiar with.

3.1.3 Boolean Types

C has Boolean types, true or false?

`1`!

Historically, C didn't have a Boolean type, and some might argue it still doesn't.

In C, `0` means "false", and non-zero means "true".

So `1` is true. And `-37` is true. And `0` is false.

[5] This isn't strictly 100% true. When we get to learning about static storage duration, you'll find the some variables are initialized to zero automatically. But the safe thing to do is always initialize them.

www.dbooks.org

You can just declare Boolean types as `int`s:

```
int x = 1;

if (x) {
    printf("x is true!\n");
}
```

If you `#include <stdbool.h>`, you also get access to some symbolic names that might make things look more familiar, namely a `bool` type and `true` and `false` values:

```
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    bool x = true;

    if (x) {
        printf("x is true!\n");
    }
}
```

But these are identical to using integer values for true and false. They're just a facade to make things look nice.

3.2 Operators and Expressions

C operators should be familiar to you from other languages. Let's blast through some of them here.

(There are a bunch more details than this, but we're going to do enough in this section to get started.)

3.2.1 Arithmetic

Hopefully these are familiar:

```
i = i + 3;  // Addition (+) and assignment (=) operators, add 3 to i
i = i - 8;  // Subtraction, subtract 8 from i
i = i * 9;  // Multiplication
i = i / 2;  // Division
i = i % 5;  // Modulo (division remainder)
```

There are shorthand variants for all of the above. Each of those lines could more tersely be written as:

```
i += 3;  // Same as "i = i + 3", add 3 to i
i -= 8;  // Same as "i = i - 8"
i *= 9;  // Same as "i = i * 9"
i /= 2;  // Same as "i = i / 2"
i %= 5;  // Same as "i = i % 5"
```

There is no exponentiation. You'll have to use one of the `pow()` function variants from `math.h`.

Let's get into some of the weirder stuff you might not have in your other languages!

3.2.2 Ternary Operator

C also includes the *ternary operator*. This is an expression whose value depends on the result of a conditional embedded in it.

```
// If x > 10, add 17 to y. Otherwise add 37 to y.

y += x > 10? 17: 37;
```

What a mess! You'll get used to it the more you read it. To help out a bit, I'll rewrite the above expression using `if` statements:

```
// This expression:

y += x > 10? 17: 37;

// is equivalent to this non-expression:

if (x > 10)
    y += 17;
else
    y += 37;
```

Compare those two until you see each of the components of the ternary operator.

Or, another example that prints if a number stored in `x` is odd or even:

```
printf("The number %d is %s.\n", x, x % 2 == 0? "even": "odd")
```

The `%s` format specifier in `printf()` means print a string. If the expression `x % 2` evaluates to `0`, the value of the entire ternary expression evaluates to the string `"even"`. Otherwise it evaluates to the string `"odd"`. Pretty cool!

It's important to note that the ternary operator isn't flow control like the `if` statement is. It's just an expression that evaluates to a value.

3.2.3 Pre-and-Post Increment-and-Decrement

Now, let's mess with another thing that you might not have seen.

These are the legendary post-increment and post-decrement operators:

```
i++;        // Add one to i (post-increment)
i--;        // Subtract one from i (post-decrement)
```

Very commonly, these are just used as shorter versions of:

```
i += 1;        // Add one to i
i -= 1;        // Subtract one from i
```

but they're more subtly different than that, the clever scoundrels.

Let's take a look at this variant, pre-increment and pre-decrement:

```
++i;        // Add one to i (pre-increment)
--i;        // Subtract one from i (pre-decrement)
```

With pre-increment and pre-decrement, the value of the variable is incremented or decremented *before* the expression is evaluated. Then the expression is evaluated with the new value.

With post-increment and post-decrement, the value of the expression is first computed with the value as-is, and *then* the value is incremented or decremented after the value of the expression has been determined.

You can actually embed them in expressions, like this:

```
i = 10;
j = 5 + i++;  // Compute 5 + i, _then_ increment i

printf("%d, %d\n", i, j);  // Prints 11, 15
```

Let's compare this to the pre-increment operator:

```
i = 10;
j = 5 + ++i;  // Increment i, _then_ compute 5 + i

printf("%d, %d\n", i, j);  // Prints 11, 16
```

This technique is used frequently with array and pointer access and manipulation. It gives you a way to use the value in a variable, and also increment or decrement that value before or after it is used.

But by far the most common place you'll see this is in a `for` loop:

```
for (i = 0; i < 10; i++)
    printf("i is %d\n", i);
```

But more on that later.

3.2.4 The Comma Operator

This is an uncommonly-used way to separate expressions that will run left to right:

```
x = 10, y = 20;  // First assign 10 to x, then 20 to y
```

Seems a bit silly, since you could just replace the comma with a semicolon, right?

```
x = 10; y = 20;  // First assign 10 to x, then 20 to y
```

But that's a little different. The latter is two separate expressions, while the former is a single expression!

With the comma operator, the value of the comma expression is the value of the rightmost expression:

```
x = 1, 2, 3;

printf("x is %d\n", x);  // Prints 3, because 3 is rightmost in the comma list
```

But even that's pretty contrived. One common place the comma operator is used is in `for` loops to do multiple things in each section of the statement:

```
for (i = 0, j = 10; i < 100; i++, j++)
    printf("%d, %d\n", i, j);
```

We'll revisit that later.

3.2.5 Conditional Operators

For Boolean values, we have a raft of standard operators:

```
a == b;  // True if a is equivalent to b
a != b;  // True if a is not equivalent to b
a < b;   // True if a is less than b
a > b;   // True if a is greater than b
a <= b;  // True if a is less than or equal to b
a >= b;  // True if a is greater than or equal to b
```

Don't mix up assignment = with comparison ==! Use two equals to compare, one to assign.

We can use the comparison expressions with `if` statements:

```
if (a <= 10)
    printf("Success!\n");
```

3.2.6 Boolean Operators

We can chain together or alter conditional expressions with Boolean operators for *and*, *or*, and *not*.

Operator	Boolean meaning
&&	and
||	or
!	not

An example of Boolean "and":

```
// Do something if x less than 10 and y greater than 20:

if (x < 10 && y > 20)
    printf("Doing something!\n");
```

An example of Boolean "not":

```
if (!(x < 12))
    printf("x is not less than 12\n");
```

`!` has higher precedence than the other Boolean operators, so we have to use parentheses in that case.

Of course, that's just the same as:

```
if (x >= 12)
    printf("x is not less than 12\n");
```

but I needed the example!

3.2.7 The `sizeof` Operator

This operator tells you the size (in bytes) that a particular variable or data type uses in memory.

More particularly, it tells you the size (in bytes) that the *type of a particular expression* (which might be just a single variable) uses in memory.

This can be different on different systems, except for `char` and its variants (which are always 1 byte).

And this might not seem very useful now, but we'll be making references to it here and there, so it's worth covering.

Since this computes the number of bytes needed to store a type, you might think it would return an `int`. Or… since the size can't be negative, maybe an `unsigned`?

But it turns out C has a special type to represent the return value from `sizeof`. It's `size_t`, pronounced "*size tee*"[6]. All we know is that it's an unsigned integer type that can hold the size in bytes of anything you can give to `sizeof`.

`size_t` shows up a lot of different places where counts of things are passed or returned. Think of it as a value that represents a count.

You can take the `sizeof` a variable or expression:

[6] The `_t` is short for `type`.

```
int a = 999;

// %zu is the format specifier for type size_t

printf("%zu\n", sizeof a);      // Prints 4 on my system
printf("%zu\n", sizeof(2 + 7)); // Prints 4 on my system
printf("%zu\n", sizeof 3.14);   // Prints 8 on my system

// If you need to print out negative size_t values, use %zd
```

Remember: it's the size in bytes of the *type* of the expression, not the size of the expression itself. That's why the size of `2+7` is the same as the size of `a`—they're both type `int`. We'll revisit this number `4` in the very next block of code…

…Where we'll see you can take the `sizeof` a type (note the parentheses are required around a type name, unlike an expression):

```
printf("%zu\n", sizeof(int));   // Prints 4 on my system
printf("%zu\n", sizeof(char));  // Prints 1 on all systems
```

It's important to note that `sizeof` is a *compile-time* operation[7]. The result of the expression is determined entirely at compile-time, not at runtime.

We'll make use of this later on.

3.3 Flow Control

Booleans are all good, but of course we're nowhere if we can't control program flow. Let's take a look at a number of constructs: `if`, `for`, `while`, and `do-while`.

First, a general forward-looking note about statements and blocks of statements brought to you by your local friendly C developer:

After something like an `if` or `while` statement, you can either put a single statement to be executed, or a block of statements to all be executed in sequence.

Let's start with a single statement:

```
if (x == 10) printf("x is 10\n");
```

This is also sometimes written on a separate line. (Whitespace is largely irrelevant in C—it's not like Python.)

```
if (x == 10)
    printf("x is 10\n");
```

But what if you want multiple things to happen due to the conditional? You can use squirrelly braces to mark a *block* or *compound statement*.

```
if (x == 10) {
    printf("x is 10\n");
    printf("And also this happens when x is 10\n");
}
```

It's a really common style to *always* use squirrelly braces even if they aren't necessary:

```
if (x == 10) {
    printf("x is 10\n");
}
```

[7] Except for with variable length arrays—but that's a story for another time.

Some devs feel the code is easier to read and avoids errors like this where things visually look like they're in the `if` block, but actually they aren't.

```
// BAD ERROR EXAMPLE

if (x == 10)
    printf("This happens if x is 10\n");
    printf("This happens ALWAYS\n");  // Surprise!! Unconditional!
```

`while` and `for` and the other looping constructs work the same way as the examples above. If you want to do multiple things in a loop or after an `if`, wrap them up in squirrelly braces.

In other words, the `if` is going to run the one thing after the `if`. And that one thing can be a single statement or a block of statements.

3.3.1 The `if-else` statement

We've already been using `if` for multiple examples, since it's likely you've seen it in a language before, but here's another:

```
int i = 10;

if (i > 10) {
    printf("Yes, i is greater than 10.\n");
    printf("And this will also print if i is greater than 10.\n");
}

if (i <= 10) printf("i is less than or equal to 10.\n");
```

In the example code, the message will print if `i` is greater than 10, otherwise execution continues to the next line. Notice the squirrley braces after the `if` statement; if the condition is true, either the first statement or expression right after the if will be executed, or else the collection of code in the squirlley braces after the `if` will be executed. This sort of *code block* behavior is common to all statements.

Of course, because C is fun this way, you can also do something if the condition is false with an `else` clause on your `if`:

```
int i = 99;

if (i == 10)
    printf("i is 10!\n");
else {
    printf("i is decidedly not 10.\n");
    printf("Which irritates me a little, frankly.\n");
}
```

And you can even cascade these to test a variety of conditions, like this:

```
int i = 99;

if (i == 10)
    printf("i is 10!\n");

else if (i == 20)
    printf("i is 20!\n");

else if (i == 99) {
    printf("i is 99! My favorite\n");
```

```
    printf("I can't tell you how happy I am.\n");
    printf("Really.\n");
}

else
    printf("i is some crazy number I've never heard of.\n");
```

Though if you're going that route, be sure to check out the `switch` statement for a potentially better solution. The catch is `switch` only works with equality comparisons with constant numbers. The above `if-else` cascade could check inequality, ranges, variables, or anything else you can craft in a conditional expression.

3.3.2 The `while` statement

`while` is your average run-of-the-mill looping construct. Do a thing while a condition expression is true.

Let's do one!

```
// Print the following output:
//
//   i is now 0!
//   i is now 1!
//   [ more of the same between 2 and 7 ]
//   i is now 8!
//   i is now 9!

i = 0;

while (i < 10) {
    printf("i is now %d!\n", i);
    i++;
}

printf("All done!\n");
```

That gets you a basic loop. C also has a `for` loop which would have been cleaner for that example.

A not-uncommon use of `while` is for infinite loops where you repeat while true:

```
while (1) {
    printf("1 is always true, so this repeats forever.\n");
}
```

3.3.3 The `do-while` statement

So now that we've gotten the `while` statement under control, let's take a look at its closely related cousin, `do-while`.

They are basically the same, except if the loop condition is false on the first pass, `do-while` will execute once, but `while` won't execute at all. In other words, the test to see whether or not to execute the block happens at the *end* of the block with `do-while`. It happens at the *beginning* of the block with `while`.

Let's see by example:

```
// Using a while statement:

i = 10;
```

```
// this is not executed because i is not less than 10:
while(i < 10) {
    printf("while: i is %d\n", i);
    i++;
}

// Using a do-while statement:

i = 10;

// this is executed once, because the loop condition is not checked until
// after the body of the loop runs:

do {
    printf("do-while: i is %d\n", i);
    i++;
} while (i < 10);

printf("All done!\n");
```

Notice that in both cases, the loop condition is false right away. So in the `while`, the loop fails, and the following block of code is never executed. With the `do-while`, however, the condition is checked *after* the block of code executes, so it always executes at least once. In this case, it prints the message, increments `i`, then fails the condition, and continues to the "All done!" output.

The moral of the story is this: if you want the loop to execute at least once, no matter what the loop condition, use `do-while`.

All these examples might have been better done with a `for` loop. Let's do something less deterministic—repeat until a certain random number comes up!

```
#include <stdio.h>   // For printf
#include <stdlib.h>  // For rand

int main(void)
{
    int r;

    do {
        r = rand() % 100; // Get a random number between 0 and 99
        printf("%d\n", r);
    } while (r != 37);    // Repeat until 37 comes up
}
```

Side note: did you run that more than once? If you did, did you notice the same sequence of numbers came up again. And again. And again? This is because `rand()` is a pseudorandom number generator that must be *seeded* with a different number in order to generate a different sequence. Look up the `srand()` function for more details.

3.3.4 The `for` statement

Welcome to one of the most popular loops in the world! The `for` loop!

This is a great loop if you know the number of times you want to loop in advance.

You could do the same thing using just a `while` loop, but the `for` loop can help keep the code cleaner.

Here are two pieces of equivalent code—note how the `for` loop is just a more compact representation:

```
// Print numbers between 0 and 9, inclusive...

// Using a while statement:

i = 0;
while (i < 10) {
    printf("i is %d\n", i);
    i++;
}

// Do the exact same thing with a for-loop:

for (i = 0; i < 10; i++) {
    printf("i is %d\n", i);
}
```

That's right, folks—they do exactly the same thing. But you can see how the `for` statement is a little more compact and easy on the eyes. (JavaScript users will fully appreciate its C origins at this point.)

It's split into three parts, separated by semicolons. The first is the initialization, the second is the loop condition, and the third is what should happen at the end of the block if the loop condition is true. All three of these parts are optional.

```
for (initialize things; loop if this is true; do this after each loop)
```

Note that the loop will not execute even a single time if the loop condition starts off false.

> **`for`-loop fun fact!**
>
> You can use the comma operator to do multiple things in each clause of the `for` loop!
>
> ```
> for (i = 0, j = 999; i < 10; i++, j--) {
>  printf("%d, %d\n", i, j);
> }
> ```

An empty `for` will run forever:

```
for(;;) {  // "forever"
    printf("I will print this again and again and again\n" );
    printf("for all eternity until the heat-death of the universe.\n");

    printf("Or until you hit CTRL-C.\n");
}
```

3.3.5 The `switch` Statement

Depending on what languages you're coming from, you might or might not be familiar with `switch`, or C's version might even be more restrictive than you're used to. This is a statement that allows you to take a variety of actions depending on the value of an integer expression.

Basically, it evaluates an expression to an integer value, jumps to the `case` that corresponds to that value. Execution resumes from that point. If a `break` statement is encountered, then execution jumps out of the `switch`.

Let's do an example where the user enters a number of goats and we print out a gut-feel of how many goats that is.

```c
#include <stdio.h>

int main(void)
{
    int goat_count;

    printf("Enter a goat count: ");
    scanf("%d", &goat_count);       // Read an integer from the keyboard

    switch (goat_count) {
        case 0:
            printf("You have no goats.\n");
            break;

        case 1:
            printf("You have a singular goat.\n");
            break;

        case 2:
            printf("You have a brace of goats.\n");
            break;

        default:
            printf("You have a bona fide plethora of goats!\n");
            break;
    }
}
```

In that example, if the user enters, say, `2`, the `switch` will jump to the `case 2` and execute from there. When (if) it hits a `break`, it jumps out of the `switch`.

Also, you might see that `default` label there at the bottom. This is what happens when no cases match.

Every `case`, including `default`, is optional. And they can occur in any order, but it's really typical for `default`, if any, to be listed last.

So the whole thing acts like an `if-else` cascade:

```c
if (goat_count == 0)
    printf("You have no goats.\n");
else if (goat_count == 1)
    printf("You have a singular goat.\n");
else if (goat_count == 2)
    printf("You have a brace of goats.\n");
else
    printf("You have a bona fide plethora of goats!\n");
```

With some key differences:

- `switch` is often faster to jump to the correct code (though the spec makes no such guarantee).
- `if-else` can do things like relational conditionals like `<` and `>=` and floating point and other types, while `switch` cannot.

There's one more neat thing about switch that you sometimes see that is quite interesting: *fall through*.

Remember how `break` causes us to jump out of the switch?

Well, what happens if we *don't* `break`?

Turns out we just keep on going into the next `case`! Demo!

```
switch (x) {
    case 1:
        printf("1\n");
        // Fall through!
    case 2:
        printf("2\n");
        break;
    case 3:
        printf("3\n");
        break;
}
```

If `x == 1`, this `switch` will first hit `case 1`, it'll print the `1`, but then it just continues on to the next line of code… which prints `2`!

And then, at last, we hit a `break` so we jump out of the `switch`.

if `x == 2`, then we just hit the `case 2`, print `2`, and `break` as normal.

Not having a `break` is called *fall through*.

ProTip: *ALWAYS* put a comment in the code where you intend to fall through, like I did above. It will save other programmers from wondering if you meant to do that.

In fact, this is one of the common places to introduce bugs in C programs: forgetting to put a `break` in your `case`. You gotta do it if you don't want to just roll into the next case[8].

Earlier I said that `switch` works with integer types—keep it that way. Don't use floating point or string types in there. One loophole-ish thing here is that you can use character types because those are secretly integers themselves. So this is perfectly acceptable:

```
char c = 'b';

switch (c) {
    case 'a':
        printf("It's 'a'!\n");
        break;

    case 'b':
        printf("It's 'b'!\n");
        break;

    case 'c':
        printf("It's 'c'!\n");
        break;
}
```

Finally, you can use `enum`s in `switch` since they are also integer types. But more on that in the `enum` chapter.

[8]This was considered such a hazard that the designers of the Go Programming Language made `break` the default; you have to explicitly use Go's `fallthrough` statement if you want to fall into the next case.

Chapter 4

Functions

> *"Sir, not in an environment such as this. That's why I've also been programmed for over thirty secondary functions that—"*
>
> —C3PO, before being rudely interrupted, reporting a now-unimpressive number of additional functions, *Star Wars* script

Very much like other languages you're used to, C has the concept of *functions*.

Functions can accept a variety of *arguments* and return a value. One important thing, though: the arguments and return value types are predeclared—because that's how C likes it!

Let's take a look at a function. This is a function that takes an `int` as an argument, and returns an `int`.

```
#include <stdio.h>

int plus_one(int n)  // The "definition"
{
    return n + 1;
}

```

The `int` before the `plus_one` indicates the return type.

The `int n` indicates that this function takes one `int` argument, stored in *parameter* `n`. A parameter is a special type of local variable into which the arguments are copied.

I'm going to drive home the point that the arguments are copied into the parameters, here. Lots of things in C are easier to understand if you know that the parameter is a *copy* of the argument, not the argument itself. More on that in a minute.

Continuing the program down into `main()`, we can see the call to the function, where we assign the return value into local variable `j`:

```
int main(void)
{
    int i = 10, j;

    j = plus_one(i);  // The "call"

    printf("i + 1 is %d\n", j);
}
```


www.dbooks.org

> Before I forget, notice that I defined the function before I used it. If I hadn't done that, the compiler wouldn't know about it yet when it compiles `main()` and it would have given an unknown function call error. There is a more proper way to do the above code with *function prototypes*, but we'll talk about that later.

Also notice that `main()` is a function!

It returns an `int`.

But what's this `void` thing? This is a keyword that's used to indicate that the function accepts no arguments.

You can also return `void` to indicate that you don't return a value:

```
#include <stdio.h>

// This function takes no arguments and returns no value:

void hello(void)
{
    printf("Hello, world!\n");
}

int main(void)
{
    hello();  // Prints "Hello, world!"
}
```

4.1 Passing by Value

I'd mentioned earlier that when you pass an argument to a function, a copy of that argument gets made and stored in the corresponding parameter.

If the argument is a variable, a copy of the value of that variable gets made and stored in the parameter.

More generally, the entire argument expression is evaluated and its value determined. That value is copied to the parameter.

In any case, the value in the parameter is its own thing. It is independent of whatever values or variables you used as arguments when you made the function call.

So let's look at an example here. Study it and see if you can determine the output before running it:

```
#include <stdio.h>

void increment(int a)
{
    a++;
}

int main(void)
{
    int i = 10;

    increment(i);

    printf("i == %d\n", i);  // What does this print?
}
```

At first glance, it looks like `i` is `10`, and we pass it to the function `increment()`. There the value gets incremented, so when we print it, it must be `11`, right?

> *"Get used to disappointment."*
>
> —Dread Pirate Roberts, *The Princess Bride*

But it's not `11`—it prints `10`! How?

It's all about the fact that the expressions you pass to functions get *copied* onto their corresponding parameters. The parameter is a copy, not the original.

So `i` is `10` out in `main()`. And we pass it to `increment()`. The corresponding parameter is called `a` in that function.

And the copy happens, as if by assignment. Loosely, `a = i`. So at that point, `a` is `10`. And out in `main()`, `i` is also `10`.

Then we increment `a` to `11`. But we're not touching `i` at all! It remains `10`.

Finally, the function is complete. All its local variables are discarded (bye, `a`!) and we return to `main()`, where `i` is still `10`.

And we print it, getting `10`, and we're done.

This is why in the previous example with the `plus_one()` function, we `return`ed the locally modified value so that we could see it again in `main()`.

Seems a little bit restrictive, huh? Like you can only get one piece of data back from a function, is what you're thinking. There is, however, another way to get data back; C folks call it *passing by reference* and that's a story we'll tell another time.

But no fancy-schmancy name will distract you from the fact that *EVERYTHING* you pass to a function *WITHOUT EXCEPTION* is copied into its corresponding parameter, and the function operates on that local copy, *NO MATTER WHAT*. Remember that, even when we're talking about this so-called passing by reference.

4.2 Function Prototypes

So if you recall back in the ice age a few sections ago, I mentioned that you had to define the function before you used it, otherwise the compiler wouldn't know about it ahead of time, and would bomb out with an error.

This isn't quite strictly true. You can notify the compiler in advance that you'll be using a function of a certain type that has a certain parameter list. That way the function can be defined anywhere (even in a different file), as long as the *function prototype* has been declared before you call that function.

Fortunately, the function prototype is really quite easy. It's merely a copy of the first line of the function definition with a semicolon tacked on the end for good measure. For example, this code calls a function that is defined later, because a prototype has been declared first:

```
#include <stdio.h>

int foo(void);  // This is the prototype!

int main(void)
{
    int i;

    // We can call foo() here before it's definition because the
    // prototype has already been declared, above!

    i = foo();
```

```

    printf("%d\n", i);  // 3490
}

int foo(void)  // This is the definition, just like the prototype!
{
    return 3490;
}
```

If you don't declare your function before you use it (either with a prototype or its definition), you're performing something called an *implicit declaration*. This was allowed in the first C standard (C89), and that standard has rules about it, but is no longer allowed today. And there is no legitimate reason to rely on it in new code.

You might notice something about the sample code we've been using… That is, we've been using the good old `printf()` function without defining it or declaring a prototype! How do we get away with this lawlessness? We don't, actually. There is a prototype; it's in that header file `stdio.h` that we included with `#include`, remember? So we're still legit, officer!

4.3 Empty Parameter Lists

You might see these from time to time in older code, but you shouldn't ever code one up in new code. Always use `void` to indicate that a function takes no parameters. There's never[1] a reason to do this in modern code.

If you're good just remembering to put `void` in for empty parameter lists in functions and prototypes, you can skip the rest of this section.

There are two contexts for this:

- Omitting all parameters where the function is defined
- Omitting all parameters in a prototype

Let's look at a potential function definition first:

```
void foo()  // Should really have a `void` in there
{
    printf("Hello, world!\n");
}
```

While the spec spells out that the behavior in this instance is *as-if* you'd indicated `void` (C11 §6.7.6.3¶14), the `void` type is there for a reason. Use it.

But in the case of a function prototype, there is a *significant* difference between using `void` and not:

```
void foo();
void foo(void);  // Not the same!
```

Leaving `void` out of the prototype indicates to the compiler that there is no additional information about the parameters to the function. It effectively turns off all that type checking.

With a prototype **definitely** use `void` when you have an empty parameter list.

[1] Never say "never".

Chapter 5

Pointers—Cower In Fear!

> *"How do you get to Carnegie Hall?"*
> *"Practice!"*
>
> —20th-century joke of unknown origin

Pointers are one of the most feared things in the C language. In fact, they are the one thing that makes this language challenging at all. But why?

Because they, quite honestly, can cause electric shocks to come up through the keyboard and physically *weld* your arms permanently in place, cursing you to a life at the keyboard in this language from the 70s!

Really? Well, not really. I'm just trying to set you up for success.

Depending on what language you came from, you might already understand the concept of *references*, where a variable refers to an object of some type.

This is very much the same, except we have to be more explicit with C about when we're talking about the reference or the thing it refers to.

5.1 Memory and Variables

Computer memory holds data of all kinds, right? It'll hold `floats`, `ints`, or whatever you have. To make memory easy to cope with, each byte of memory is identified by an integer. These integers increase sequentially as you move up through memory[1]. You can think of it as a bunch of numbered boxes, where each box holds a byte[2] of data. Or like a big array where each element holds a byte, if you come from a language with arrays. The number that represents each box is called its *address*.

Now, not all data types use just a byte. For instance, an `int` is often four bytes, as is a `float`, but it really depends on the system. You can use the `sizeof` operator to determine how many bytes of memory a certain type uses.

```
// %zu is the format specifier for type size_t

printf("an int uses %zu bytes of memory\n", sizeof(int));

// That prints "4" for me, but can vary by system.
```

[1] Typically. I'm sure there are exceptions out there in the dark corridors of computing history.

[2] A byte is a number made up of no more than 8 binary digits, or *bits* for short. This means in decimal digits just like grandma used to use, it can hold an unsigned number between 0 and 255, inclusive.

www.dbooks.org

> **Memory Fun Facts**: When you have a data type that uses more than a byte of memory, the bytes that make up the data are always adjacent to one another in memory. Sometimes they're in order, and sometimes they're not[3], but that's platform-dependent, and often taken care of for you without you needing to worry about pesky byte orderings.

So *anyway*, if we can get on with it and get a drum roll and some foreboding music playing for the definition of a pointer, *a pointer is a variable that holds an address*. Imagine the classical score from 2001: A Space Odyssey at this point. Ba bum ba bum ba bum BAAAAH!

Ok, so maybe a bit overwrought here, yes? There's not a lot of mystery about pointers. They are the address of data. Just like an `int` variable can hold the value `12`, a pointer variable can hold the address of data.

This means that all these things mean the same thing, i.e. a number that represents a point in memory:

- Index into memory (if you're thinking of memory like a big array)
- Address
- Location

I'm going to use these interchangeably. And yes, I just threw *location* in there because you can never have enough words that mean the same thing.

And a pointer variable holds that address number. Just like a `float` variable might hold `3.14159`.

Imagine you have a bunch of Post-it® notes all numbered in sequence with their address. (The first one is at index numbered `0`, the next at index `1`, and so on.)

In addition to the number representing their positions, you can also write another number of your choice on each. It could be the number of dogs you have. Or the number of moons around Mars…

…Or, *it could be the index of another Post-it note!*

If you have written the number of dogs you have, that's just a regular variable. But if you wrote the index of another Post-it in there, *that's a pointer*. It points to the other note!

Another analogy might be with house addresses. You can have a house with certain qualities, yard, metal roof, solar, etc. Or you could have the address of that house. The address isn't the same as the house itself. One's a full-blown house, and the other is just a few lines of text. But the address of the house is a *pointer* to that house. It's not the house itself, but it tells you where to find it.

And we can do the same thing in the computer with data. You can have a data variable that's holding some value. And that value is in memory at some address. And you could have a different *pointer variable* hold the address of that data variable.

It's not the data variable itself, but, like with a house address, it tells us where to find it.

When we have that, we say we have a "pointer to" that data. And we can follow the pointer to access the data itself.

(Though it doesn't seem particularly useful yet, this all becomes indispensable when used with function calls. Bear with me until we get there.)

So if we have an `int`, say, and we want a pointer to it, what we want is some way to get the address of that `int`, right? After all, the pointer just holds the *address of* the data. What operator do you suppose we'd use to find the *address of* the `int`?

Well, by a shocking surprise that must come as something of a shock to you, gentle reader, we use the `address-of` operator (which happens to be an ampersand: "`&`")to find the address of the data. Ampersand.

So for a quick example, we'll introduce a new *format specifier* for `printf()` so you can print a pointer. You know already how `%d` prints a decimal integer, yes? Well, `%p` prints a pointer. Now, this pointer is going to

[3]The order that bytes come in is referred to as the *endianness* of the number. Common ones are *big endian* and *little endian*. This usually isn't something you need to worry about.

look like a garbage number (and it might be printed in hexadecimal[4] instead of decimal), but it is merely the index into memory the data is stored in. (Or the index into memory that the first byte of data is stored in, if the data is multi-byte.) In virtually all circumstances, including this one, the actual value of the number printed is unimportant to you, and I show it here only for demonstration of the `address-of` operator.

```
#include <stdio.h>

int main(void)
{
    int i = 10;

    printf("The value of i is %d\n", i);
    printf("And its address is %p\n", (void *)&i);

    // %p expects the argument to be a pointer to void
    // so we cast it to make the compiler happy.
}
```

On my computer, this prints:

```
The value of i is 10
And its address is 0x7ffddf7072a4
```

If you're curious, that hexadecimal number is 140,727,326,896,068 in decimal (base 10 just like Grandma used to use). That's the index into memory where the variable `i`'s data is stored. It's the address of `i`. It's the location of `i`. It's a pointer to `i`.

It's a pointer because it lets you know where `i` is in memory. Like a home address written on a scrap of paper tells you where you can find a particular house, this number indicates to us where in memory we can find the value of `i`. It points to `i`.

Again, we don't really care what the address's exact number is, generally. We just care that it's a pointer to `i`.

5.2 Pointer Types

So… this is all well and good. You can now successfully take the address of a variable and print it on the screen. There's a little something for the ol' resume, right? Here's where you grab me by the scruff of the neck and ask politely what the frick pointers are good for.

Excellent question, and we'll get to that right after these messages from our sponsor.

> ACME ROBOTIC HOUSING UNIT CLEANING SERVICES. YOUR HOMESTEAD WILL BE DRAMATICALLY IMPROVED OR YOU WILL BE TERMINATED. MESSAGE ENDS.

Welcome back to another installment of Beej's Guide. When we met last we were talking about how to make use of pointers. Well, what we're going to do is store a pointer off in a variable so that we can use it later. You can identify the *pointer type* because there's an asterisk (`*`) before the variable name and after its type:

```
int main(void)
{
    int i;  // i's type is "int"
    int *p; // p's type is "pointer to an int", or "int-pointer"
}
```

Hey, so we have here a variable that is a pointer type, and it can point to other `int`s. That is, it can hold the address of other `int`s. We know it points to `int`s, since it's of type `int*` (read "int-pointer").

[4] That is, base 16 with digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

www.dbooks.org

When you do an assignment into a pointer variable, the type of the right hand side of the assignment has to be the same type as the pointer variable. Fortunately for us, when you take the `address-of` a variable, the resultant type is a pointer to that variable type, so assignments like the following are perfect:

```
int i;
int *p;  // p is a pointer, but is uninitialized and points to garbage

p = &i;  // p is assigned the address of i--p now "points to" i
```

On the left of the assignment, we have a variable of type pointer-to-`int` (`int*`), and on the right side, we have expression of type pointer-to-`int` since `i` is an `int` (because address-of `int` gives you a pointer to `int`). The address of a thing can be stored in a pointer to that thing.

Get it? I know it still doesn't quite make much sense since you haven't seen an actual use for the pointer variable, but we're taking small steps here so that no one gets lost. So now, let's introduce you to the anti-address-of operator. It's kind of like what `address-of` would be like in Bizarro World.

5.3 Dereferencing

A pointer variable can be thought of as *referring* to another variable by pointing to it. It's rare you'll hear anyone in C land talking about "referring" or "references", but I bring it up just so that the name of this operator will make a little more sense.

When you have a pointer to a variable (roughly "a reference to a variable"), you can use the original variable through the pointer by *dereferencing* the pointer. (You can think of this as "de-pointering" the pointer, but no one ever says "de-pointering".)

Back to our analogy, this is vaguely like looking at a home address and then going to that house.

Now, what do I mean by "get access to the original variable"? Well, if you have a variable called `i`, and you have a pointer to `i` called `p`, you can use the dereferenced pointer `p` *exactly as if it were the original variable `i`*!

You almost have enough knowledge to handle an example. The last tidbit you need to know is actually this: what is the dereference operator? It's actually called the *indirection operator*, because you're accessing values indirectly via the pointer. And it is the asterisk, again: `*`. Now, don't get this confused with the asterisk you used in the pointer declaration, earlier. They are the same character, but they have different meanings in different contexts[5].

Here's a full-blown example:

```
#include <stdio.h>

int main(void)
{
    int i;
    int *p;  // this is NOT a dereference--this is a type "int*"

    p = &i;  // p now points to i, p holds address of i

    i = 10;  // i is now 10
    *p = 20; // the thing p points to (namely i!) is now 20!!

    printf("i is %d\n", i);   // prints "20"
```

[5]That's not all! It's used in `/*comments*/` and multiplication and in function prototypes with variable length arrays! It's all the same `*`, but the context gives it different meaning.

```
14     printf("i is %d\n", *p);  // "20"! dereference-p is the same as i!
15 }
```

Remember that `p` holds the address of `i`, as you can see where we did the assignment to `p` on line 8. What the indirection operator does is tells the computer to *use the object the pointer points to* instead of using the pointer itself. In this way, we have turned `*p` into an alias of sorts for `i`.

Great, but *why*? Why do any of this?

5.4 Passing Pointers as Arguments

Right about now, you're thinking that you have an awful lot of knowledge about pointers, but absolutely zero application, right? I mean, what use is `*p` if you could just simply say `i` instead?

Well, my friend, the real power of pointers comes into play when you start passing them to functions. Why is this a big deal? You might recall from before that you could pass all kinds of arguments to functions and they'd be dutifully copied into parameters, and then you could manipulate local copies of those variables from within the function, and then you could return a single value.

What if you wanted to bring back more than one single piece of data from the function? I mean, you can only return one thing, right? What if I answered that question with another question? …Er, two questions?

What happens when you pass a pointer as an argument to a function? Does a copy of the pointer get put into its corresponding parameter? *You bet your sweet peas it does.* Remember how earlier I rambled on and on about how *EVERY SINGLE ARGUMENT* gets copied into parameters and the function uses a copy of the argument? Well, the same is true here. The function will get a copy of the pointer.

But, and this is the clever part: we will have set up the pointer in advance to point at a variable… and then the function can dereference its copy of the pointer to get back to the original variable! The function can't see the variable itself, but it can certainly dereference a pointer to that variable!

This is analogous to writing a home address on a piece of paper, and then copying that onto another piece of paper. You now have *two* pointers to that house, and both are equally good at getting you to the house itself.

In the case of a function call. one of the copies is stored in a pointer variable out in the calling scope, and the other is stored in a pointer variable that is the parameter of the function.

Example! Let's revisit our old `increment()` function, but this time let's make it so that it actually increments the value out in the caller.

```
1  #include <stdio.h>
2
3  void increment(int *p)  // note that it accepts a pointer to an int
4  {
5      *p = *p + 1;        // add one to the thing p points to
6  }
7
8  int main(void)
9  {
10     int i = 10;
11     int *j = &i;  // note the address-of; turns it into a pointer to i
12
13     printf("i is %d\n", i);        // prints "10"
14     printf("i is also %d\n", *j);  // prints "10"
15
16     increment(j);                  // j is an int*--to i
17
```

```
    printf("i is %d\n", i);        // prints "11"!
}
```

Ok! There are a couple things to see here… not the least of which is that the `increment()` function takes an `int*` as an argument. We pass it an `int*` in the call by changing the `int` variable `i` to an `int*` using the `address-of` operator. (Remember, a pointer holds an address, so we make pointers to variables by running them through the `address-of` operator.)

The `increment()` function gets a copy of the pointer. Both the original pointer `j` (in `main()`) and the copy of that pointer `p` (the parameter in `increment()`) point to the same address, namely the one holding the value `i`. (Again, by analogy, like two pieces of paper with the same home address written on them.) Dereferencing either will allow you to modify the original variable `i`! The function can modify a variable in another scope! Rock on!

The above example is often more concisely written in the call just by using address-of right in the argument list:

```
printf("i is %d\n", i);  // prints "10"
increment(&i);
printf("i is %d\n", i);  // prints "11"!
```

Pointer enthusiasts will recall from early on in the guide, we used a function to read from the keyboard, `scanf()`… and, although you might not have recognized it at the time, we used the `address-of` to pass a pointer to a value to `scanf()`. We had to pass a pointer, see, because `scanf()` reads from the keyboard (typically) and stores the result in a variable. The only way it can see that variable out in the calling function's scope is if we pass a pointer to that variable:

```
int i = 0;

scanf("%d", &i);          // pretend you typed "12"
printf("i is %d\n", i);  // prints "i is 12"
```

See, `scanf()` dereferences the pointer we pass it in order to modify the variable it points to. And now you know why you have to put that pesky ampersand in there!

5.5 The `NULL` Pointer

Any pointer variable of any pointer type can be set to a special value called `NULL`. This indicates that this pointer doesn't point to anything.

```
int *p;

p = NULL;
```

Since it doesn't point to a value, dereferencing it is undefined behavior, and probably will result in a crash:

```
int *p = NULL;

*p = 12;  // CRASH or SOMETHING PROBABLY BAD. BEST AVOIDED.
```

Despite being called the billion dollar mistake by its creator[6], the `NULL` pointer is a good sentinel value[7] and general indicator that a pointer hasn't yet been initialized.

(Of course, like other variables, the pointer points to garbage unless you explicitly assign it to point to an address or `NULL`.)

[6] https://en.wikipedia.org/wiki/Null_pointer#History
[7] https://en.wikipedia.org/wiki/Sentinel_value

5.6 A Note on Declaring Pointers

The syntax for declaring a pointer can get a little weird. Let's look at this example:

```
int a;
int b;
```

We can condense that into a single line, right?

```
int a, b;  // Same thing
```

So `a` and `b` are both `int`s. No problem.

But what about this?

```
int a;
int *p;
```

Can we make that into one line? We can. But where does the `*` go?

The rule is that the `*` goes in front of any variable that is a pointer type. That is. the `*` is *not* part of the `int` in this example. it's a part of variable `p`.

With that in mind, we can write this:

```
int a, *p;  // Same thing
```

It's important to note that the following line does *not* declare two pointers:

```
int *p, q;  // p is a pointer to an int; q is just an int.
```

This can be particularly insidious-looking if the programmer writes this following (valid) line of code which is functionally identical to the one above.

```
int* p, q;  // p is a pointer to an int; q is just an int.
```

So take a look at this and determine which variables are pointers and which are not:

```
int *a, b, c, *d, e, *f, g, h, *i;
```

I'll drop the answer in a footnote[8].

5.7 `sizeof` and Pointers

Just a little bit of syntax here that might be confusing and you might see from time to time.

Recall that `sizeof` operates on the *type* of the expression.

```
int *p;

sizeof(int); // Returns size of an `int`
sizeof p     // p is type int*, so returns size of `int*`
sizeof *p    // *p is type int, so returns size of `int`
```

You might see code with that last `sizeof` in there. Just remember that `sizeof` is all about the type of the expression, not the variables in the expression themselves.

[8] The pointer type variables are `a`, `d`, `f`, and `i`, because those are the ones with `*` in front of them.

Chapter 6

Arrays

> *"Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration."*
>
> —Stan Kelly-Bootle, computer scientist

Luckily, C has arrays. I mean, I know it's considered a low-level language[1] but it does at least have the concept of arrays built-in. And since a great many languages drew inspiration from C's syntax, you're probably already familiar with using `[` and `]` for declaring and using arrays.

But C only *barely* has arrays! As we'll find out later, arrays are just syntactic sugar in C—they're actually all pointers and stuff deep down. *Freak out!* But for now, let's just use them as arrays. *Phew*.

6.1 Easy Example

Let's just crank out an example:

```c
#include <stdio.h>

int main(void)
{
    int i;
    float f[4];  // Declare an array of 4 floats

    f[0] = 3.14159;  // Indexing starts at 0, of course.
    f[1] = 1.41421;
    f[2] = 1.61803;
    f[3] = 2.71828;

    // Print them all out:

    for (i = 0; i < 4; i++) {
        printf("%f\n", f[i]);
    }
}
```

When you declare an array, you have to give it a size. And the size has to be fixed[2].

[1] These days, anyway.

[2] Again, not really, but variable-length arrays—of which I'm not really a fan—are a story for another time.

In the above example, we made an array of 4 `floats`. The value in the square brackets in the declaration lets us know that.

Later on in subsequent lines, we access the values in the array, setting them or getting them, again with square brackets.

Hopefully this looks familiar from languages you already know!

6.2 Getting the Length of an Array

You can't…ish. C doesn't record this information[3]. You have to manage it separately in another variable.

When I say "can't", I actually mean there are some circumstances when you *can*. There is a trick to get the number of elements in an array in the scope in which an array is declared. But, generally speaking, this won't work the way you want if you pass the array to a function[4].

Let's take a look at this trick. The basic idea is that you take the `sizeof` the array, and then divide that by the size of each element to get the length. For example, if an `int` is 4 bytes, and the array is 32 bytes long, there must be room for $\frac{32}{4}$ or 8 `ints` in there.

```
int x[12];  // 12 ints

printf("%zu\n", sizeof x);     // 48 total bytes
printf("%zu\n", sizeof(int));  // 4 bytes per int

printf("%zu\n", sizeof x / sizeof(int));  // 48/4 = 12 ints!
```

If it's an array of `chars`, then `sizeof` the array *is* the number of elements, since `sizeof(char)` is defined to be 1. For anything else, you have to divide by the size of each element.

But this trick only works in the scope in which the array was defined. If you pass the array to a function, it doesn't work. Even if you make it "big" in the function signature:

```
void foo(int x[12])
{
    printf("%zu\n", sizeof x);     // 8?! What happened to 48?
    printf("%zu\n", sizeof(int));  // 4 bytes per int

    printf("%zu\n", sizeof x / sizeof(int));  // 8/4 = 2 ints?? WRONG.
}
```

This is because when you "pass" arrays to functions, you're only passing a pointer to the first element, and that's what `sizeof` measures. More on this in the Passing Single Dimensional Arrays to Functions section, below.

One more thing you can do with `sizeof` and arrays is get the size of an array of a fixed number of elements without declaring the array. This is like how you can get the size of an `int` with `sizeof(int)`.

For example, to see how many bytes would be needed for an array of 48 `doubles`, you can do this:

```
sizeof(double [48]);
```

6.3 Array Initializers

You can initialize an array with constants ahead of time:

[3] Since arrays are just pointers to the first element of the array under the hood, there's no additional information recording the length.

[4] Because when you pass an array to a function, you're actually just passing a pointer to the first element of that array, not the "entire" array.

www.dbooks.org

```c
#include <stdio.h>

int main(void)
{
    int i;
    int a[5] = {22, 37, 3490, 18, 95};  // Initialize with these values

    for (i = 0; i < 5; i++) {
        printf("%d\n", a[i]);
    }
}
```

Catch: initializer values must be constant terms. Can't throw variables in there. Sorry, Illinois!

You should never have more items in your initializer than there is room for in the array, or the compiler will get cranky:

```
foo.c: In function 'main':
foo.c:6:39: warning: excess elements in array initializer
    6 |     int a[5] = {22, 37, 3490, 18, 95, 999};
      |                                       ^~~
foo.c:6:39: note: (near initialization for 'a')
```

But (fun fact!) you can have *fewer* items in your initializer than there is room for in the array. The remaining elements in the array will be automatically initialized with zero. This is true in general for all types of array initializers: if you have an initializer, anything not explicitly set to a value will be set to zero.

```c
int a[5] = {22, 37, 3490};

// is the same as:

int a[5] = {22, 37, 3490, 0, 0};
```

It's a common shortcut to see this in an initializer when you want to set an entire array to zero:

```c
int a[100] = {0};
```

Which means, "Make the first element zero, and then automatically make the rest zero, as well."

You can set specific array elements in the initializer, as well, by specifying an index for the value! When you do this, C will happily keep initializing subsequent values for you until the initializer runs out, filling everything else with `0`.

To do this, put the index in square brackets with an `=` after, and then set the value.

Here's an example where we build an array:

```c
int a[10] = {0, 11, 22, [5]=55, 66, 77};
```

Because we listed index 5 as the start for `55`, the resulting data in the array is:

```
0 11 22 0 0 55 66 77 0 0
```

You can put simple constant expressions in there, as well.

```c
#define COUNT 5

int a[COUNT] = {[COUNT-3]=3, 2, 1};
```

which gives us:

```
0 0 3 2 1
```

Lastly, you can also have C compute the size of the array from the initializer, just by leaving the size off:

```
int a[3] = {22, 37, 3490};

// is the same as:

int a[] = {22, 37, 3490};  // Left the size off!
```

6.4 Out of Bounds!

C doesn't stop you from accessing arrays out of bounds. It might not even warn you.

Let's steal the example from above and keep printing off the end of the array. It only has 5 elements, but let's try to print 10 and see what happens:

```
#include <stdio.h>

int main(void)
{
    int i;
    int a[5] = {22, 37, 3490, 18, 95};

    for (i = 0; i < 10; i++) {  // BAD NEWS: printing too many elements!
        printf("%d\n", a[i]);
    }
}
```

Running it on my computer prints:

```
22
37
3490
18
95
32765
1847052032
1780534144
-56487472
21890
```

Yikes! What's that? Well, turns out printing off the end of an array results in what C developers call *undefined behavior*. We'll talk more about this beast later, but for now it means, "You've done something bad, and anything could happen during your program run."

And by anything, I mean typically things like finding zeroes, finding garbage numbers, or crashing. But really the C spec says in this circumstance the compiler is allowed to emit code that does *anything*[5].

Short version: don't do anything that causes undefined behavior. Ever[6].

[5] In the good old MS-DOS days before memory protection was a thing, I was writing some particularly abusive C code that deliberately engaged in all kinds of undefined behavior. But I knew what I was doing, and things were working pretty well. Until I made a misstep that caused a lockup and, as I found upon reboot, nuked all my BIOS settings. That was fun. (Shout-out to @man for those fun times.)

[6] There are a lot of things that cause undefined behavior, not just out-of-bounds array accesses. This is what makes the C language so *exciting*.

www.dbooks.org

6.5 Multidimensional Arrays

You can add as many dimensions as you want to your arrays.

```
int a[10];
int b[2][7];
int c[4][5][6];
```

These are stored in memory in row-major order[7]. This means with a 2D array, the first index listed indicates the row, and the second the column.

You an also use initializers on multidimensional arrays by nesting them:

```
#include <stdio.h>

int main(void)
{
    int row, col;

    int a[2][5] = {      // Initialize a 2D array
        {0, 1, 2, 3, 4},
        {5, 6, 7, 8, 9}
    };

    for (row = 0; row < 2; row++) {
        for (col = 0; col < 5; col++) {
            printf("(%d,%d) = %d\n", row, col, a[row][col]);
        }
    }
}
```

For output of:

```
(0,0) = 0
(0,1) = 1
(0,2) = 2
(0,3) = 3
(0,4) = 4
(1,0) = 5
(1,1) = 6
(1,2) = 7
(1,3) = 8
(1,4) = 9
```

And you can initialize with explicit indexes:

```
// Make a 3x3 identity matrix

int a[3][3] = {[0][0]=1, [1][1]=1, [2][2]=1};
```

which builds a 2D array like this:

```
1 0 0
0 1 0
0 0 1
```

[7] https://en.wikipedia.org/wiki/Row-_and_column-major_order

6.6 Arrays and Pointers

[*Casually*] So… I kinda might have mentioned up there that arrays were pointers, deep down? We should take a shallow dive into that now so that things aren't completely confusing. Later on, we'll look at what the real relationship between arrays and pointers is, but for now I just want to look at passing arrays to functions.

6.6.1 Getting a Pointer to an Array

I want to tell you a secret. Generally speaking, when a C programmer talks about a pointer to an array, they're talking about a pointer *to the first element* of the array[8].

So let's get a pointer to the first element of an array.

```c
#include <stdio.h>

int main(void)
{
    int a[5] = {11, 22, 33, 44, 55};
    int *p;

    p = &a[0];  // p points to the array
                // Well, to the first element, actually

    printf("%d\n", *p);  // Prints "11"
}
```

This is so common to do in C that the language allows us a shorthand:

```c
p = &a[0];  // p points to the array

// is the same as:

p = a;      // p points to the array, but much nicer-looking!
```

Just referring to the array name in isolation is the same as getting a pointer to the first element of the array! We're going to use this extensively in the upcoming examples.

But hold on a second—isn't `p` an `int*`? And `*p` gives us `11`, same as `a[0]`? Yessss. You're starting to get a glimpse of how arrays and pointers are related in C.

6.6.2 Passing Single Dimensional Arrays to Functions

Let's do an example with a single dimensional array. I'm going to write a couple functions that we can pass the array to that do different things.

Prepare for some mind-blowing function signatures!

```c
#include <stdio.h>

// Passing as a pointer to the first element
void times2(int *a, int len)
{
    for (int i = 0; i < len; i++)
        printf("%d\n", a[i] * 2);
}
```

[8] This is technically incorrect, as a pointer to an array and a pointer to the first element of an array have different types. But we can burn that bridge when we get to it.

www.dbooks.org

```

// Same thing, but using array notation
void times3(int a[], int len)
{
    for (int i = 0; i < len; i++)
        printf("%d\n", a[i] * 3);
}

// Same thing, but using array notation with size
void times4(int a[5], int len)
{
    for (int i = 0; i < len; i++)
        printf("%d\n", a[i] * 4);
}

int main(void)
{
    int x[5] = {11, 22, 33, 44, 55};

    times2(x, 5);
    times3(x, 5);
    times4(x, 5);
}
```

All those methods of listing the array as a parameter in the function are identical.

```
void times2(int *a, int len)
void times3(int a[], int len)
void times4(int a[5], int len)
```

In usage by C regulars, the first is the most common, by far.

And, in fact, in the latter situation, the compiler doesn't even care what number you pass in (other than it has to be greater than zero[9]). It doesn't enforce anything at all.

Now that I've said that, the size of the array in the function declaration actually *does* matter when you're passing multidimensional arrays into functions, but let's come back to that.

6.6.3 Changing Arrays in Functions

We've said that arrays are just pointers in disguise. This means that if you pass an array to a function, you're likely passing a pointer to the first element in the array.

But if the function has a pointer to the data, it is able to manipulate that data! So changes that a function makes to an array will be visible back out in the caller.

Here's an example where we pass a pointer to an array to a function, the function manipulates the values in that array, and those changes are visible out in the caller.

```
#include <stdio.h>

void double_array(int *a, int len)
{
```

[9] C11 §6.7.6.2¶1 requires it be greater than zero. But you might see code out there with arrays declared of zero length at the end of `structs` and GCC is particularly lenient about it unless you compile with `-pedantic`. This zero-length array was a hackish mechanism for making variable-length structures. Unfortunately, it's technically undefined behavior to access such an array even though it basically worked everywhere. C99 codified a well-defined replacement for it called *flexible array members*, which we'll chat about later.

```
    // Multiple each element by 2
    //
    // This doubles the values in x in main() since x and a both point
    // to the same array in memory!

    for (int i = 0; i < len; i++)
        a[i] *= 2;
}

int main(void)
{
    int x[5] = {1, 2, 3, 4, 5};

    double_array(x, 5);

    for (int i = 0; i < 5; i++)
        printf("%d\n", x[i]);  // 2, 4, 6, 8, 10!
}
```

Even though we passed the array in as parameter `a` which is type `int*`, look at how we access it using array notation with `a[i]`! Whaaaat. This is totally allowed.

Later when we talk about the equivalence between arrays and pointers, we'll see how this makes a lot more sense. For now, it's enough to know that functions can make changes to arrays that are visible out in the caller.

6.6.4 Passing Multidimensional Arrays to Functions

The story changes a little when we're talking about multidimensional arrays. C needs to know all the dimensions (except the first one) so it has enough information to know where in memory to look to find a value.

Here's an example where we're explicit with all the dimensions:

```
#include <stdio.h>

void print_2D_array(int a[2][3])
{
    for (int row = 0; row < 2; row++) {
        for (int col = 0; col < 3; col++)
            printf("%d ", a[row][col]);
        printf("\n");
    }
}

int main(void)
{
    int x[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };

    print_2D_array(x);
}
```

But in this case, these two[10] are equivalent:

```
void print_2D_array(int a[2][3])
void print_2D_array(int a[][3])
```

The compiler really only needs the second dimension so it can figure out how far in memory to skip for each increment of the first dimension. In general, it needs to know all the dimensions except the first one.

Also, remember that the compiler does minimal compile-time bounds checking (if you're lucky), and C does zero runtime checking of bounds. No seat belts! Don't crash by accessing array elements out of bounds!

[10]This is also equivalent: `void print_2D_array(int (*a)[3])`, but that's more than I want to get into right now.

Chapter 7

Strings

Finally! Strings! What could be simpler?

Well, turns out strings aren't actually strings in C. That's right! They're pointers! Of course they are!

Much like arrays, strings in C *barely exist*.

But let's check it out—it's not really such a big deal.

7.1 String Literals

Before we start, let's talk about string literals in C. These are sequences of characters in *double* quotes (`"`). (Single quotes enclose characters, and are a different animal entirely.)

Examples:

```
"Hello, world!\n"
"This is a test."
"When asked if this string had quotes in it, she replied, \"It does.\""
```

The first one has a newline at the end—quite a common thing to see.

The last one has quotes embedded within it, but you see each is preceded by (we say "escaped by") a backslash (`\`) indicating that a literal quote belongs in the string at this point. This is how the C compiler can tell the difference between printing a double quote and the double quote at the end of the string.

7.2 String Variables

Now that we know how to make a string literal, let's assign it to a variable so we can do something with it.

```
char *s = "Hello, world!";
```

Check out that type: pointer to a `char`. The string variable `s` is actually a pointer to the first character in that string, namely the `H`.

And we can print it with the `%s` (for "string") format specifier:

```
char *s = "Hello, world!";

printf("%s\n", s);  // "Hello, world!"
```


www.dbooks.org

7.3 String Variables as Arrays

Another option is this, nearly equivalent to the above `char*` usage:

```
char s[14] = "Hello, world!";

// or, if we were properly lazy and have the compiler
// figure the length for us:

char s[] = "Hello, world!";
```

This means you can use array notation to access characters in a string. Let's do exactly that to print all the characters in a string on the same line:

```
#include <stdio.h>

int main(void)
{
    char s[] = "Hello, world!";

    for (int i = 0; i < 13; i++)
        printf("%c\n", s[i]);
}
```

Note that we're using the format specifier `%c` to print a single character.

Also, check this out. The program will still work fine if we change the definition of `s` to be a `char*` type:

```
#include <stdio.h>

int main(void)
{
    char *s = "Hello, world!";   // char* here

    for (int i = 0; i < 13; i++)
        printf("%c\n", s[i]);    // But still use arrays here...?
}
```

And we still can use array notation to get the job done when printing it out! This is surprising, but is still only because we haven't talked about array/pointer equivalence yet. But this is yet another hint that arrays and pointers are the same thing, deep down.

7.4 String Initializers

We've already seen some examples with initializing string variables with string literals:

```
char *s = "Hello, world!";
char t[] = "Hello, again!";
```

But these two are subtly different.

This one is a pointer to a string literal (i.e. a pointer to the first character in a string):

```
char *s = "Hello, world!";
```

If you try to mutate that string with this:

```
char *s = "Hello, world!";

s[0] = 'z';  // BAD NEWS: tried to mutate a string literal!
```

The behavior is undefined. Probably, depending on your system, a crash will result.

But declaring it as an array is different. This one is a mutable *copy* of the string that we can change at will:

```
char t[] = "Hello, again!";  // t is an array copy of the string
t[0] = 'z'; //  No problem

printf("%s\n", t);  // "zello, again!"
```

So remember: if you have a pointer to a string literal, don't try to change it! And if you use a string in double quotes to initialize an array, that's not actually a string literal.

7.5 Getting String Length

You can't, since C doesn't track it for you. And when I say "can't", I actually mean "can"[1]. There's a function in `<string.h>` called `strlen()` that can be used to compute the length of any string in bytes[2].

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *s = "Hello, world!";

    printf("The string is %zu bytes long.\n", strlen(s));
}
```

The `strlen()` function returns type `size_t`, which is an integer type so you can use it for integer math. We print `size_t` with `%zu`.

The above program prints:

```
The string is 13 bytes long.
```

Great! So it *is* possible to get the string length!

But… if C doesn't track the length of the string anywhere, how does it know how long the string is?

7.6 String Termination

C does strings a little differently than many programming languages, and in fact differently than almost every modern programming language.

When you're making a new language, you have basically two options for storing a string in memory:

1. Store the bytes of the string along with a number indicating the length of the string.
2. Store the bytes of the string, and mark the end of the string with a special byte called the *terminator*.

If you want strings longer than 255 characters, option 1 requires at least two bytes to store the length. Whereas option 2 only requires one byte to terminate the string. So a bit of savings there.

[1] Though it is true that C doesn't track the length of strings.

[2] If you're using the basic character set or an 8-bit character set, you're used to one character being one byte. This isn't true in all character encodings, though.

www.dbooks.org

Of course, these days it seems ridiculous to worry about saving a byte (or 3—lots of languages will happily let you have strings that are 4 gigabytes in length). But back in the day, it was a bigger deal.

So C took approach #2. In C, a "string" is defined by two basic characteristics:

- A pointer to the first character in the string.
- A zero-valued byte (or `NUL` character[3]) somewhere in memory after the pointer that indicates the end of the string.

A `NUL` character can be written in C code as `\0`, though you don't often have to do this.

When you include a string in double quotes in your code, the `NUL` character is automatically, implicitly included.

```
char *s = "Hello!";  // Actually "Hello!\0" behind the scenes
```

So with this in mind, let's write our own `strlen()` function that counts `char`s in a string until it finds a `NUL`.

The procedure is to look down the string for a single `NUL` character, counting as we go[4]:

```
int my_strlen(char *s)
{
    int count = 0;

    while (s[count] != '\0')  // Single quotes for single char
        count++;

    return count;
}
```

And that's basically how the built-in `strlen()` gets the job done.

7.7 Copying a String

You can't copy a string through the assignment operator (`=`). All that does is make a copy of the pointer to the first character… so you end up with two pointers to the same string:

```
#include <stdio.h>

int main(void)
{
    char s[] = "Hello, world!";
    char *t;

    // This makes a copy of the pointer, not a copy of the string!
    t = s;

    // We modify t
    t[0] = 'z';

    // But printing s shows the modification!
    // Because t and s point to the same string!

    printf("%s\n", s);  // "zello, world!"
}
```

[3] This is different than the `NULL` pointer, and I'll abbreviate it `NUL` when talking about the character versus `NULL` for the pointer.
[4] Later we'll learn a neater way to do it with pointer arithmetic.

If you want to make a copy of a string, you have to copy it a byte at a time—but this is made easier with the `strcpy()` function[5].

Before you copy the string, make sure you have room to copy it into, i.e. the destination array that's going to hold the characters needs to be at least as long as the string you're copying.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char s[] = "Hello, world!";
    char t[100];  // Each char is one byte, so plenty of room

    // This makes a copy of the string!
    strcpy(t, s);

    // We modify t
    t[0] = 'z';

    // And s remains unaffected because it's a different string
    printf("%s\n", s);  // "Hello, world!"

    // But t has been changed
    printf("%s\n", t);  // "zello, world!"
}
```

Notice with `strcpy()`, the destination pointer is the first argument, and the source pointer is the second. A mnemonic I use to remember this is that it's the order you would have put `t` and `s` if an assignment = worked for strings, with the source on the right and the destination on the left.

[5]There's a safer function called `strncpy()` that you should probably use instead, but we'll get to that later.

www.dbooks.org

Chapter 8

Structs

In C, we have something called a `struct`, which is a user-definable type that holds multiple pieces of data, potentially of different types.

It's a convenient way to bundle multiple variables into a single one. This can be beneficial for passing variables to functions (so you just have to pass one instead of many), and useful for organizing data and making code more readable.

If you've come from another language, you might be familiar with the idea of *classes* and *objects*. These don't exist in C, natively[1]. You can think of a `struct` as a class with only data members, and no methods.

8.1 Declaring a Struct

You can declare a `struct` in your code like so:

```
struct car {
    char *name;
    float price;
    int speed;
};
```

This is often done at the global scope outside any functions so that the `struct` is globally available.

When you do this, you're making a new *type*. The full type name is `struct car`. (Not just `car`—that won't work.)

There aren't any variables of that type yet, but we can declare some:

```
struct car saturn;  // Variable "saturn" of type "struct car"
```

And now we have an uninitialized variable `saturn`[2] of type `struct car`.

We should initialize it! But how do we set the values of those individual fields?

Like in many other languages that stole it from C, we're going to use the dot operator (`.`) to access the individual fields.

```
saturn.name = "Saturn SL/2";
saturn.price = 15999.99;
```

[1] Although in C individual items in memory like `int`s are referred to as "objects", they're not objects in an object-oriented programming sense.

[2] The Saturn was a popular brand of economy car in the United States until it was put out of business by the 2008 crash, sadly so to us fans.

```
saturn.speed = 175;

printf("Name:           %s\n", saturn.name);
printf("Price (USD):    %f\n", saturn.price);
printf("Top Speed (km): %d\n", saturn.speed);
```

There on the first lines, we set the values in the `struct car`, and then in the next bit, we print those values out.

8.2 Struct Initializers

That example in the previous section was a little unwieldy. There must be a better way to initialize that `struct` variable!

You can do it with an initializer by putting values in for the fields *in the order they appear in the `struct`* when you define the variable. (This won't work after the variable has been defined—it has to happen in the definition).

```
struct car {
    char *name;
    float price;
    int speed;
};

// Now with an initializer! Same field order as in the struct declaration:
struct car saturn = {"Saturn SL/2", 16000.99, 175};

printf("Name:      %s\n", saturn.name);
printf("Price:     %f\n", saturn.price);
printf("Top Speed: %d km\n", saturn.speed);
```

The fact that the fields in the initializer need to be in the same order is a little freaky. If someone changes the order in `struct car`, it could break all the other code!

We can be more specific with our initializers:

```
struct car saturn = {.speed=175, .name="Saturn SL/2"};
```

Now it's independent of the order in the `struct` declaration. Which is safer code, for sure.

Similar to array initializers, any missing field designators are initialized to zero (in this case, that would be `.price`, which I've omitted).

8.3 Passing Structs to Functions

You can do a couple things to pass a `struct` to a function.

1. Pass the `struct`.
2. Pass a pointer to the `struct`.

Recall that when you pass something to a function, a *copy* of that thing gets made for the function to operate on, whether it's a copy of a pointer, an `int`, a `struct`, or anything.

There are basically two cases when you'd want to pass a pointer to the `struct`:

1. You need the function to be able to make changes to the `struct` that was passed in, and have those changes show in the caller.

2. The `struct` is somewhat large and it's more expensive to copy that onto the stack than it is to just copy a pointer[3].

For those two reasons, it's far more common to pass a pointer to a `struct` to a function, though its by no means illegal to pass the `struct` itself.

Let's try passing in a pointer, making a function that will allow you to set the `.price` field of the `struct car`:

```
#include <stdio.h>

struct car {
    char *name;
    float price;
    int speed;
};

int main(void)
{
    struct car saturn = {.speed=175, .name="Saturn SL/2"};

    // Pass a pointer to this struct car, along with a new,
    // more realistic, price:
    set_price(&saturn, 799.99);

    printf("Price: %f\n", saturn.price);
}
```

You should be able to come up with the function signature for `set_price()` just by looking at the types of the arguments we have there.

`saturn` is a `struct car`, so `&saturn` must be the address of the `struct car`, AKA a pointer to a `struct car`, namely a `struct car*`.

And `799.99` is a `float`.

So the function declaration must look like this:

```
void set_price(struct car *c, float new_price)
```

We just need to write the body. One attempt might be:

```
void set_price(struct car *c, float new_price) {
    c.price = new_price;  // ERROR!!
}
```

That won't work because the dot operator only works on `structs`… it doesn't work on *pointers* to `structs`.

Ok, so we can dereference the `struct` to de-pointer it to get to the `struct` itself. Dereferencing a `struct car*` results in the `struct car` that the pointer points to, which we should be able to use the dot operator on:

```
void set_price(struct car *c, float new_price) {
    (*c).price = new_price;  // Works, but is ugly and non-idiomatic :(
}
```

And that works! But it's a little clunky to type all those parens and the asterisk. C has some syntactic sugar called the *arrow operator* that helps with that.

[3] A pointer is likely 8 bytes on a 64-bit system.

8.4 The Arrow Operator

The arrow operator helps refer to fields in pointers to `structs`.

```
void set_price(struct car *c, float new_price) {
    // (*c).price = new_price;  // Works, but non-idiomatic :(
    //
    // The line above is 100% equivalent to the one below:

    c->price = new_price;  // That's the one!
}
```

So when accessing fields, when do we use dot and when do we use arrow?

- If you have a `struct`, use dot (`.`).
- If you have a pointer to a `struct`, use arrow (`->`).

8.5 Copying and Returning `structs`

Here's an easy one for you!

Just assign from one to the other!

```
struct car a, b;

b = a;  // Copy the struct
```

And returning a `struct` (as opposed to a pointer to one) from a function also makes a similar copy to the receiving variable.

This is not a "deep copy"[4]. All fields are copied as-is, including pointers to things.

8.6 Comparing `structs`

There's only one safe way to do it: compare each field one at a time.

You might think you could use `memcmp()`, but that doesn't handle the case of the possible padding bytes that might be in there.

If you clear the `struct` to zero first with `memset()`, then it *might* work, though there could be weird elements that might not compare as you expect[5].

[4] A *deep copy* follows pointer in the `struct` and copies the data they point to, as well. A *shallow copy* just copies the pointers, but not the things they point to. C doesn't come with any built-in deep copy functionality.

[5] https://stackoverflow.com/questions/141720/how-do-you-compare-structs-for-equality-in-c

www.dbooks.org

Chapter 9

File Input/Output

We've already seen a couple examples of I/O with `scanf()` and `printf()` for doing I/O at the console (screen/keyboard).

But we'll push those concepts a little farther this chapter.

9.1 The `FILE*` Data Type

When we do any kind of I/O in C, we do so through a piece of data that you get in the form of a `FILE*` type. This `FILE*` holds all the information needed to communicate with the I/O subsystem about which file you have open, where you are in the file, and so on.

The spec refers to these as *streams*, i.e. a stream of data from a file or from any source. I'm going to use "files" and "streams" interchangeably, but really you should think of a "file" as a special case of a "stream". There are other ways to stream data into a program than just reading from a file.

We'll see in a moment how to go from having a filename to getting an open `FILE*` for it, but first I want to mention three streams that are already open for you and ready for use.

|`FILE*` name|Description| |-|-| |`stdin`|Standard Input, generally the keyboard by default| |`stdout`|Standard Output, generally the screen by default| |`stderr`|Standard Error, generally the screen by default, as well|

We've actually been using these implicitly already, it turns out. For example, these two calls are the same:

```
printf("Hello, world!\n");
fprintf(stdout, "Hello, world!\n");  // printf to a file
```

But more on that later.

Also you'll notice that both `stdout` and `stderr` go to the screen. While this seems at first either like an oversight or redundancy, it actually isn't. Typical operating systems allow you to *redirect* the output of either of those into different files, and it can be convenient to be able to separate error messages from regular non-error output.

For example, in a POSIX shell (like sh, ksh, bash, zsh, etc.) on a Unix-like system, we could run a program and send just the non-error (`stdout`) output to one file, and all the error (`stderr`) output to another file.

```
./foo > output.txt 2> errors.txt   # This command is Unix-specific
```

For this reason, you should send serious error messages to `stderr` instead of `stdout`.

More on how to do that later.

9.2 Reading Text Files

Streams are largely categorized two different ways: *text* and *binary*.

Text streams are allowed to do significant translation of the data, most notably translations of newlines to their different representations[1]. Text files are logically a sequence of *lines* separated by newlines. To be portable, your input data should always end with a newline.

But the general rule is that if you're able to edit the file in a regular text editor, it's a text file. Otherwise, it's binary. More on binary later.

So let's get to work—how do we open a file for reading, and pull data out of it?

Let's create a file called `hello.txt` that has just this in it:

```
Hello, world!
```

And let's write a program to open the file, read a character out of it, and then close the file when we're done. That's the game plan!

```c
#include <stdio.h>

int main(void)
{
    FILE *fp;                       // Variable to represent open file

    fp = fopen("hello.txt", "r");   // Open file for reading

    int c = fgetc(fp);              // Read a single character
    printf("%c\n", c);              // Print char to stdout

    fclose(fp);                     // Close the file when done
}
```

See how when we opened the file with `fopen()`, it returned the `FILE*` to us so we could use it later.

(I'm leaving it out for brevity, but `fopen()` will return `NULL` if something goes wrong, like file-not-found, so you should really error check it!)

Also notice the `"r"` that we passed in—this means "open a text stream for reading". (There are various strings we can pass to `fopen()` with additional meaning, like writing, or appending, and so on.)

After that, we used the `fgetc()` function to get a character from the stream. You might be wondering why I've made `c` an `int` instead of a `char`—hold that thought!

Finally, we close the stream when we're done with it. All streams are automatically closed when the program exits, but it's good form and good housekeeping to explicitly close any files yourself when done with them.

The `FILE*` keeps track of our position in the file. So subsequent calls to `fgetc()` would get the next character in the file, and then the next, until the end.

But that sounds like a pain. Let's see if we can make it easier.

9.3 End of File: `EOF`

There is a special character defined as a macro: `EOF`. This is what `fgetc()` will return when the end of the file has been reached and you've attempted to read another character.

[1] We used to have three different newlines in broad effect: Carriage Return (CR, used on old Macs), Linefeed (LF, used on Unix systems), and Carriage Return/Linefeed (CRLF, used on Windows systems). Thankfully the introduction of OS X, being Unix-based, reduced this number to two.

www.dbooks.org

How about I share that Fun Fact™, now. Turns out `EOF` is the reason why `fgetc()` and functions like it return an `int` instead of a `char`. `EOF` isn't a character proper, and its value likely falls outside the range of `char`. Since `fgetc()` needs to be able to return any byte **and** `EOF`, it needs to be a wider type that can hold more values. so `int` it is. But unless you're comparing the returned value against `EOF`, you can know, deep down, it's a `char`.

All right! Back to reality! We can use this to read the whole file in a loop.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      FILE *fp;
6      int c;
7
8      fp = fopen("hello.txt", "r");
9
10     while ((c = fgetc(fp)) != EOF)
11         printf("%c", c);
12
13     fclose(fp);
14 }
```

(If line 10 is too weird, just break it down starting with the innermost-nested parens. The first thing we do is assign the result of `fgetc()` into `c`, and *then* we compare *that* against `EOF`. We've just crammed it into a single line. This might look hard to read, but study it—it's idiomatic C.)

And running this, we see:

```
Hello, world!
```

But still, we're operating a character at a time, and lots of text files make more sense at the line level. Let's switch to that.

9.3.1 Reading a Line at a Time

So how can we get an entire line at once? `fgets()` to the rescue! For arguments, it takes a pointer to a `char` buffer to hold bytes, a maximum number of bytes to read, and a `FILE*` to read from. It returns `NULL` on end-of-file or error. `fgets()` is even nice enough to NUL-terminate the string when its done[2].

Let's do a similar loop as before, except let's have a multiline file and read it in a line at a time.

Here's a file `quote.txt`:

```
A wise man can learn more from
a foolish question than a fool
can learn from a wise answer.
                  --Bruce Lee
```

And here's some code that reads that file a line at a time and prints out a line number before each one:

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    char s[1024];  // Big enough for any line this program will encounter
```

[2] If the buffer's not big enough to read in an entire line, it'll just stop reading mid-line, and the next call to `fgets()` will continue reading the rest of the line.

```
    int linecount = 0;

    fp = fopen("quote.txt", "r");

    while (fgets(s, sizeof s, fp) != NULL)
        printf("%d: %s", ++linecount, s);

    fclose(fp);
}
```

Which gives the output:

```
1: A wise man can learn more from
2: a foolish question than a fool
3: can learn from a wise answer.
4:                   --Bruce Lee
```

9.4 Formatted Input

You know how you can get formatted output with `printf()` (and, thus, `fprintf()` like we'll see, below)?

You can do the same thing with `fscanf()`.

Let's have a file with a series of data records in it. In this case, whales, with name, length in meters, and weight in tonnes. `whales.txt`:

```
blue 29.9 173
right 20.7 135
gray 14.9 41
humpback 16.0 30
```

Yes, we could read these with `fgets()` and then parse the string with `sscanf()` (and in some ways that's more resilient against corrupted files), but in this case, let's just use `fscanf()` and pull it in directly.

The `fscanf()` function skips leading whitespace when reading, and returns `EOF` on end-of-file or error.

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    char name[1024];  // Big enough for any line this program will encounter
    float length;
    int mass;

    fp = fopen("whales.txt", "r");

    while (fscanf(fp, "%s %f %d", name, &length, &mass) != EOF)
        printf("%s whale, %d tonnes, %.1f meters\n", name, mass, length);

    fclose(fp);
}
```

Which gives the result:

```
blue whale, 173 tonnes, 29.9 meters
right whale, 135 tonnes, 20.7 meters
```

```
gray whale, 41 tonnes, 14.9 meters
humpback whale, 30 tonnes, 16.0 meters
```

9.5 Writing Text Files

In much the same way we can use `fgetc()`, `fgets()`, and `fscanf()` to read text streams, we can use `fputc()`, `fputs()`, and `fprintf()` to write text streams.

To do so, we have to `fopen()` the file in write mode by passing `"w"` as the second argument. Opening an existing file in `"w"` mode will instantly truncate that file to 0 bytes for a full overwrite.

We'll put together a simple program that outputs a file `output.txt` using a variety of output functions.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      FILE *fp;
6      int x = 32;
7
8      fp = fopen("output.txt", "w");
9
10     fputc('B', fp);
11     fputc('\n', fp);   // newline
12     fprintf(fp, "x = %d\n", x);
13     fputs("Hello, world!\n", fp);
14
15     fclose(fp);
16 }
```

And this produces a file, `output.txt`, with these contents:

```
B
x = 32
Hello, world!
```

Fun fact: since `stdout` is a file, you could replace line 8 with:

```
fp = stdout;
```

and the program would have outputted to the console instead of to a file. Try it!

9.6 Binary File I/O

So far we've just been talking text files. But there's that other beast we mentioned early on called *binary* files, or binary streams.

These work very similarly to text files, except the I/O subsystem doesn't perform any translations on the data like it might with a text file. With binary files, you get a raw stream of bytes, and that's all.

The big difference in opening the file is that you have to add a `"b"` to the mode. That is, to read a binary file, open it in `"rb"` mode. To write a file, open it in `"wb"` mode.

Because it's streams of bytes, and streams of bytes can contain NUL characters, and the NUL character is the end-of-string marker in C, it's rare that people use the `fprintf()`-and-friends functions to operate on binary files.

Instead the most common functions are `fread()` and `fwrite()`. The functions read and write a specified number of bytes to the stream.

To demo, we'll write a couple programs. One will write a sequence of byte values to disk all at once. And the second program will read a byte at a time and print them out[3].

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    unsigned char bytes[6] = {5, 37, 0, 88, 255, 12};

    fp = fopen("output.bin", "wb");  // wb mode for "write binary"!

    // In the call to fwrite, the arguments are:
    //
    // * Pointer to data to write
    // * Size of each "piece" of data
    // * Count of each "piece" of data
    // * FILE*

    fwrite(bytes, sizeof(char), 6, fp);

    fclose(fp);
}
```

Those two middle arguments to `fwrite()` are pretty odd. But basically what we want to tell the function is, "We have items that are *this* big, and we want to write *that* many of them." This makes it convenient if you have a record of a fixed length, and you have a bunch of them in an array. You can just tell it the size of one record and how many to write.

In the example above, we tell it each record is the size of a `char`, and we have 6 of them.

Running the program gives us a file `output.bin`, but opening it in a text editor doesn't show anything friendly! It's binary data—not text. And random binary data I just made up, at that!

If I run it through a hex dump[4] program, we can see the output as bytes:

```
05 25 00 58 ff 0c
```

And those values in hex do match up to the values (in decimal) that we wrote out.

But now let's try to read them back in with a different program. This one will open the file for binary reading (`"rb"` mode) and will read the bytes one at a time in a loop.

`fread()` has the neat feature where it returns the number of bytes read, or `0` on EOF. So we can loop until we see that, printing numbers as we go.

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    unsigned char c;

```

[3] Normally the second program would read all the bytes at once, and *then* print them out in a loop. That would be more efficient. But we're going for demo value, here.

[4] https://en.wikipedia.org/wiki/Hex_dump

www.dbooks.org

```
    fp = fopen("output.bin", "rb"); // rb for "read binary"!

    while (fread(&c, sizeof(char), 1, fp) > 0)
        printf("%d\n", c);
}
```

And, running it, we see our original numbers!

```
5
37
0
88
255
12
```

Woo hoo!

9.6.1 `struct` and Number Caveats

As we saw in the `structs` section, the compiler is free to add padding to a `struct` as it sees fit. And different compilers might do this differently. And the same compiler on different architectures could do it differently. And the same compiler on the same architectures could do it differently.

What I'm getting at is this: it's not portable to just `fwrite()` an entire `struct` out to a file when you don't know where the padding will end up.

How do we fix this? Hold that thought—we'll look at some ways to do this after looking at another related problem.

Numbers!

Turns out all architectures don't represent numbers in memory the same way.

Let's look at a simple `fwrite()` of a 2-byte number. We'll write it in hex so each byte is clear. The most significant byte will have the value `0x12` and the least significant will have the value `0x34`.

```
unsigned short v = 0x1234;  // Two bytes, 0x12 and 0x34

fwrite(&v, sizeof v, 1, fp);
```

What ends up in the stream?

Well, it seems like it should be `0x12` followed by `0x34`, right?

But if I run this on my machine and hex dump the result, I get:

```
34 12
```

They're reversed! What gives?

This has something to do with what's called the *endianess*[5] of the architecture. Some write the most significant bytes first, and some the least significant bytes first.

This means that if you write a multibyte number out straight from memory, you can't do it in a portable way[6].

A similar problem exists with floating point. Most systems use the same format for their floating point numbers, but some do not. No guarantees!

So… how can we fix all these problems with numbers and `structs` to get our data written in a portable way?

[5] https://en.wikipedia.org/wiki/Endianess

[6] And this is why I used individual bytes in my `fwrite()` and `fread()` examples, above, shrewdly.

The summary is to *serialize* the data, which is a general term that means to take all the data and write it out in a format that you control, that is well-known, and programmable to work the same way on all platforms.

As you might imagine, this is a solved problem. There are a bunch of serialization libraries you can take advantage of, such as Google's *protocol buffers*[7], out there and ready to use. They will take care of all the gritty details for you, and even will allow data from your C programs to interoperate with other languages that support the same serialization methods.

Do yourself and everyone a favor! Serialize your binary data when you write it to a stream! This will keep things nice and portable, even if you transfer data files from one architecture to another.

[7] https://en.wikipedia.org/wiki/Protocol_buffers

www.dbooks.org

Chapter 10

`typedef`: Making New Types

Well, not so much making *new* types as getting new names for existing types. Sounds kinda pointless on the surface, but we can really use this to make our code cleaner.

10.1 `typedef` in Theory

Basically, you take an existing type and you make an alias for it with `typedef`.

Like this:

```
typedef int antelope;  // Make "antelope" an alias for "int"

antelope x = 10;       // Type "antelope" is the same as type "int"
```

You can take any existing type and do it. You can even make a number of types with a comma list:

```
typedef int antelope, bagel, mushroom;  // These are all "int"
```

That's really useful, right? That you can type `mushroom` instead of `int`? You must be *super excited* about this feature!

OK, Professor Sarcasm—we'll get to some more common applications of this in a moment.

10.1.1 Scoping

`typedef` follows regular scoping rules.

For this reason, it's quite common to find `typedef` at file scope ("global") so that all functions can use the new types at will.

10.2 `typedef` in Practice

So renaming `int` to something else isn't that exciting. Let's see where `typedef` commonly makes an appearance.

10.2.1 `typedef` and `struct`s

Sometimes a `struct` will be `typedef`'d to a new name so you don't have to type the word `struct` over and over.

```
struct animal {
    char *name;
    int leg_count, speed;
};

//  original name      new name
//            |         |
//            v         v
//      |-----------| |----|
typedef struct animal animal;

struct animal y;  // This works
animal z;         // This also works because "animal" is an alias
```

Personally, I don't care for this practice. I like the clarity the code has when you add the word `struct` to the type; programmers know what they're getting. But it's really common so I'm including it here.

Now I want to run the exact same example in a way that you might commonly see. We're going to put the `struct animal` *in* the `typedef`. You can mash it all together like this:

```
//  original name
//            |
//            v
//      |-----------|
typedef struct animal {
    char *name;
    int leg_count, speed;
} animal;                          // <-- new name

struct animal y;  // This works
animal z;         // This also works because "animal" is an alias
```

That's exactly the same as the previous example, just more concise.

But that's not all! There's another common shortcut that you might see in code using what are called *anonymous structures*[1]. It turns out you don't actually need to name the structure in a variety of places, and with `typedef` is one of them.

Let's do the same example with an anonymous structure:

```
//  Anonymous struct! It has no name!
//         |
//         v
//      |----|
typedef struct {
    char *name;
    int leg_count, speed;
} animal;                          // <-- new name

//struct animal y;  // ERROR: this no longer works--no such struct!
animal z;           // This works because "animal" is an alias
```

As another example, we might find something like this:

```
typedef struct {
    int x, y;
```

[1] We'll talk more about these later.

```
} point;

point p = {.x=20, .y=40};

printf("%d, %d\n", p.x, p.y);  // 20, 40
```

10.2.2 typedef and Other Types

It's not that using `typedef` with a simple type like `int` is completely useless… it helps you abstract the types to make it easier to change them later.

For example, if you have `float` all over your code in 100 zillion places, it's going to be painful to change them all to `double` if you find you have to do that later for some reason.

But if you prepared a little with:

```
typedef float app_float;

// and

app_float f1, f2, f3;
```

Then if later you want to change to another type, like `long double`, you just need to change the `typedef`:

```
//        voila!
//      |---------|
typedef long double app_float;

// and no need to change this line:

app_float f1, f2, f3;  // Now these are all long doubles
```

10.2.3 typedef and Pointers

You can make a type that is a pointer.

```
typedef int *intptr;

int a = 10;
intptr x = &a;  // "intptr" is type "int*"
```

I really don't like this practice. It hides the fact that `x` is a pointer type because you don't see a `*` in the declaration.

IMHO, it's better to explicitly show that you're declaring a pointer type so that other devs can clearly see it and don't mistake `x` for having a non-pointer type.

But at last count, say, 832,007 people had a different opinion.

10.2.4 typedef and Capitalization

I've seen all kinds of capitalization on `typedef`.

```
typedef struct {
    int x, y;
} my_point;          // lower snake case

typedef struct {
```

```
    int x, y;
} MyPoint;          // CamelCase

typedef struct {
    int x, y;
} Mypoint;          // Leading uppercase

typedef struct {
    int x, y;
} MY_POINT;          // UPPER SNAKE CASE
```

The C11 specification doesn't dictate one way or another, and shows examples in all uppercase and all lowercase.

K&R2 uses leading uppercase predominantly, but show some examples in uppercase and snake case (with `_t`).

If you have a style guide in use, stick with it. If you don't, grab one and stick with it.

10.3 Arrays and `typedef`

The syntax is a little weird, and this is rarely seen in my experience, but you can `typedef` an array of some number of items.

```
// Make type five_ints an array of 5 ints
typedef int five_ints[5];

five_ints x = {11, 22, 33, 44, 55};
```

I don't like it because it hides the array nature of the variable, but it's possible to do.

Chapter 11

Pointers II: Arithmetic

Time to get more into it with a number of new pointer topics! If you're not up to speed with pointers, check out the first section in the guide on the matter.

11.1 Pointer Arithmetic

Turns out you can do math on pointers, notably addition and subtraction.

But what does it mean when you do that?

In short, if you have a pointer to a type, adding one to the pointer moves to the next item of that type directly after it in memory.

It's **important** to remember that as we move pointers around and look at different places in memory, we need to make sure that we're always pointing to a valid place in memory before we dereference. If we're off in the weeds and we try to see what's there, the behavior is undefined and a crash is a common result.

This is a little chicken-and-eggy with Array/Pointer Equivalence, below, but we're going to give it a shot, anyway.

11.1.1 Adding to Pointers

First, let's take an array of numbers.

```
int a[5] = {11, 22, 33, 44, 55};
```

Then let's get a pointer to the first element in that array:

```
int a[5] = {11, 22, 33, 44, 55};

int *p = &a[0];  // Or "int *p = a;" works just as well
```

Then let's print the value there by dereferencing the pointer:

```
printf("%d\n", *p);  // Prints 11
```

Now let's use pointer arithmetic to print the next element in the array, the one at index 1:

```
printf("%d\n", *(p + 1));  // Prints 22!!
```

What happened there? C knows that `p` is a pointer to an `int`. So it knows the `sizeof` an `int`[1] and it knows to skip that many bytes to get to the next `int` after the first one!

[1] Recall that the `sizeof` operator tells you the size in bytes of an object in memory.

In fact, the prior example could be written these two equivalent ways:

```
printf("%d\n", *p);         // Prints 11
printf("%d\n", *(p + 0));  // Prints 11
```

because adding `0` to a pointer results in the same pointer.

Let's think of the upshot here. We can iterate over elements of an array this way instead of using an array:

```
int a[5] = {11, 22, 33, 44, 55};

int *p = &a[0];  // Or "int *p = a;" works just as well

for (int i = 0; i < 5; i++) {
    printf("%d\n", *(p + i));  // Same as p[i]!
}
```

And that works the same as if we used array notation! Oooo! Getting closer to that array/pointer equivalence thing! More on this later in this chapter.

But what's actually happening, here? How does it work?

Remember from early on that memory is like a big array, where a byte is stored at each array index?

And the array index into memory has a few names:

- Index into memory
- Location
- Address
- *Pointer!*

So a point is an index into memory, somewhere.

For a random example, say that a number 3490 was stored at address ("index") 23,237,489,202. If we have an `int` pointer to that 3490, that value of that pointer is 23,237,489,202… because the pointer is the memory address. Different words for the same thing.

And now let's say we have another number, 4096, stored right after the 3490 at address 23,237,489,210 (8 higher than the 3490 because each `int` in this example is 8 bytes long).

If we add `1` to that pointer, it actually jumps ahead `sizeof(int)` bytes to the next `int`. It knows to jump that far ahead because it's an `int` pointer. If it were a `float` pointer, it'd jump `sizeof(float)` bytes ahead to get to the next float!

So you can look at the next `int`, by adding `1` to the pointer, the one after that by adding `2` to the pointer, and so on.

11.1.2 Changing Pointers

We saw how we could add an integer to a pointer in the previous section. This time, let's *modify the pointer, itself*.

You can just add (or subtract) integer values directly to (or from) any pointer!

Let's do that example again, except with a couple changes. First, I'm going to add a `999` to the end of our numbers to act as a sentinel value. This will let us know where the end of the data is.

```
int a[] = {11, 22, 33, 44, 55, 999};  // Add 999 here as a sentinel

int *p = &a[0];  // p points to the 11
```

And we also have `p` pointing to the element at index `0` of `a`, namely `11`, just like before.

Now—let's starting *incrementing* `p` so that it points at subsequent elements of the array. We'll do this until `p` points to the `999`; that is, we'll do it until `*p == 999`:

```
while (*p != 999) {       // While the thing p points to isn't 999
    printf("%d\n", *p);   // Print it
    p++;                  // Move p to point to the next int!
}
```

Pretty crazy, right?

When we give it a run, first `p` points to `11`. Then we increment `p`, and it points to `22`, and then again, it points to `33`. And so on, until it points to `999` and we quit.

11.1.3 Subtracting Pointers

You can subtract a value from a pointer to get to earlier address, as well, just like we were adding to them before.

But we can also subtract two pointers to find the difference between them, e.g. we can calculate how many `int`s there are between two `int*`s. The catch is that this only works within a single array[2]—if the pointers point to anything else, you get undefined behavior.

Remember how strings are `char*`s in C? Let's see if we can use this to write another variant of `strlen()` to compute the length of a string that utilizes pointer subtraction.

The idea is that if we have a pointer to the beginning of the string, we can find a pointer to the end of the string by scanning ahead for the `NUL` character.

And if we have a pointer to the beginning of the string, and we computed the pointer to the end of the string, we can just subtract the two pointers to come up with the length!

```
#include <stdio.h>

int my_strlen(char *s)
{
    // Start scanning from the beginning of the string
    char *p = s;

    // Scan until we find the NUL character
    while (*p != '\0')
        p++;

    // Return the difference in pointers
    return p - s;
}

int main(void)
{
    printf("%d\n", my_strlen("Hello, world!"));  // Prints "13"
}
```

Remember that you can only use pointer subtraction between two pointers that point to the same array!

[2] Or string, which is really an array of `char`s. Somewhat peculiarly, you can also have a pointer that references *one past* the end of the array without a problem and still do math on it. You just can't dereference it when it's out there.

11.2 Array/Pointer Equivalence

We're finally ready to talk about this! We've seen plenty of examples of places where we've intermixed array notation, but let's give out the *fundamental formula of array/pointer equivalence*:

```
a[b] == *(a + b)
```

Study that! Those are equivalent and can be used interchangeably!

I've oversimplified a bit, because in my above example `a` and `b` can both be expressions, and we might want a few more parentheses to force order of operations in case the expressions are complex.

The spec is specific, as always, declaring (in C11 §6.5.2.1¶2):

> `E1[E2]` is identical to `(*((E1)+(E2)))`

but that's a little harder to grok. Just make sure you include parentheses if the expressions are complicated so all your math happens in the right order.

This means we can *decide* if we're going to use array or pointer notation for any array or pointer (assuming it points to an element of an array).

Let's use an array and pointer with both array and pointer notation:

```c
#include <stdio.h>

int main(void)
{
    int a[] = {11, 22, 33, 44, 55};

    int *p = a;  // p points to the first element of a, 11

    // Print all elements of the array a variety of ways:

    for (int i = 0; i < 5; i++)
        printf("%d\n", a[i]);      // Array notation with a

    for (int i = 0; i < 5; i++)
        printf("%d\n", p[i]);      // Array notation with p

    for (int i = 0; i < 5; i++)
        printf("%d\n", *(a + i));  // Pointer notation with a

    for (int i = 0; i < 5; i++)
        printf("%d\n", *(p + i));  // Pointer notation with p

    for (int i = 0; i < 5; i++)
        printf("%d\n", *(p++));    // Moving pointer p
        //printf("%d\n", *(a++));    // Moving array variable a--ERROR!
}
```

So you can see that in general, if you have an array variable, you can use pointer or array notion to access elements. Same with a pointer variable.

The one big difference is that you can *modify* a pointer to point to a different address, but you can't do that with an array variable.

11.2.1 Array/Pointer Equivalence in Function Calls

This is where you'll encounter this concept the most, for sure.

If you have a function that takes a pointer argument, e.g.:

```
int my_strlen(char *s)
```

this means you can pass either an array or a pointer to this function and have it work!

```
char s[] = "Antelopes";
char *t = "Wombats";

printf("%d\n", my_strlen(s));  // Works!
printf("%d\n", my_strlen(t));  // Works, too!
```

And it's also why these two function signatures are equivalent:

```
int my_strlen(char *s)    // Works!
int my_strlen(char s[])   // Works, too!
```

11.3 `void` Pointers

You've already seen the `void` keyword used with functions, but this is an entirely separate, unrelated animal.

Sometimes it's useful to have a pointer to a thing *that you don't know the type of*.

I know. Bear with me just a second.

There are basically two use cases for this.

1. A function is going to operate on something byte-by-byte. For example, `memcpy()` copies bytes of memory from one pointer to another, but those pointers can point to any type. `memcpy()` takes advantage of the fact that if you iterate through `char*`s, you're iterating through the bytes of an object no matter what type the object is. More on this in the Multibyte Values subsection.

2. Another function is calling a function you passed to it (a callback), and it's passing you data. You know the type of the data, but the function calling you doesn't. So it passes you `void*`s—'cause it doesn't know the type—and you convert those to the type you need. The built-in `qsort()` and `bsearch()` use this technique.

Let's look at an example, the built-in `memcpy()` function:

```
void *memcpy(void *s1, void *s2, size_t n);
```

This function copies `n` bytes of memory starting from address `s1` into the memory starting at address `s2`.

But look! `s1` and `s2` are `void*`s! Why? What does it mean? Let's run more examples to see.

For instance, we could copy a string with `memcpy()` (though `strcpy()` is more appropriate for strings):

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char s[] = "Goats!";
    char t[100];

    memcpy(t, s, 7);  // Copy 7 bytes--including the NUL terminator!

```

```
    printf("%s\n", t);  // "Goats!"
}
```

Or we can copy some `int`s:

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    int a[] = {11, 22, 33};
    int b[3];

    memcpy(b, a, 3 * sizeof(int));  // Copy 3 ints of data

    printf("%d\n", b[1]);  // 22
}
```

That one's a little wild—you see what we did there with `memcpy()`? We copied the data from `a` to `b`, but we had to specify how many *bytes* to copy, and an `int` is more than one byte.

OK, then—how many bytes does an `int` take? Answer: depends on the system. But we can tell how many bytes any type takes with the `sizeof` operator.

So there's the answer: an `int` takes `sizeof(int)` bytes of memory to store.

And if we have 3 of them in our array, like we did in that example, the entire space used for the 3 `int`s must be `3 * sizeof(int)`.

(In the string example, earlier, it would have been more technically accurate to copy `7 * sizeof(char)` bytes. But `char`s are always one byte large, by definition, so that just devolves into `7 * 1`.)

We could even copy a `float` or a `struct` with `memcpy()`! (Though this is abusive—we should just use `=` for that):

```
struct antelope my_antelope;
struct antelopy my_clone_antelope;

// ...

memcpy(&my_clone_antelope, &my_antelope, sizeof my_antelope);
```

Look at how versatile `memcpy()` is! If you have a pointer to a source and a pointer to a destination, and you have the number of bytes you want to copy, you can copy *any type of data*.

Imagine if we didn't have `void*`. We'd have to write specialized `memcpy()` functions for each type:

```
memcpy_int(int *a, int *b, int count);
memcpy_float(float *a, float *b, int count);
memcpy_double(double *a, double *b, int count);
memcpy_char(char *a, char *b, int count);
memcpy_unsigned_char(unsigned char *a, unsigned char *b, int count);

// etc... blech!
```

Much better to just use `void*` and have one function that can do it all.

That's the power of `void*`. You can write functions that don't care about the type and is still able to do things with it.

But with great power comes great responsibility. Maybe not *that* great in this case, but there are some limits.

1. You cannot do pointer arithmetic on a `void*`. 2. You cannot dereference a `void*`. 3. You cannot use the arrow operator on a `void*`, since it's also a dereference. 4. You cannot use array notation on a `void*`, since it's also a dereference, as well[3].

And if you think about it, these rules make sense. All those operations rely on knowing the `sizeof` the type of data pointed to, and with `void*`, we don't know the size of the data being pointed to—it could be anything!

But wait—if you can't dereference a `void*` what good can it ever do you?

Like with `memcpy()`, it helps you write generic functions that can handle multiple types of data. But the secret is that, deep down, *you convert the `void*` to another type before you use it*!

And conversion is easy: you can just assign into a variable of the desired type[4].

```
char a = 'X';  // A single char

void *p = &a;  // p points to the 'X'
char *q = p;   // q also points to the 'X'

printf("%c\n", *p);  // ERROR--cannot dereference void*!
printf("%c\n", *q);  // Prints "X"
```

Let's write our own `memcpy()` to try this out. We can copy bytes (`char`s), and we know the number of bytes because it's passed in.

```
void *my_memcpy(void *dest, void *src, int byte_count)
{
    // Convert void*s to char*s
    char *s = src, *d = dest;

    // Now that we have char*s, we can dereference and copy them
    while (byte_count--) {
        *d++ = *s++;
    }

    // Most of these functions return the destination, just in case
    // that's useful to the caller.
    return dest;
}
```

Right there at the beginning, we copy the `void*`s into `char*`s so that we can use them as `char*`s. It's as easy as that.

Then some fun in a while loop, where we decrement `byte_count` until it becomes false (`0`). Remember that with post-decrement, the value of the expression is computed (for `while` to use) and *then* the variable is decremented.

And some fun in the copy, where we assign `*d = *s` to copy the byte, but we do it with post-increment so that both `d` and `s` move to the next byte after the assignment is made.

Lastly, most memory and string functions return a copy of a pointer to the destination string just in case the caller wants to use it.

[3] Because remember that array notation is just a dereference and some pointer math, and you can't dereference a `void*`!

[4] You can also *cast* the `void*` to another type, but we haven't gotten to casts yet.

Now that we've done that, I just want to quickly point out that we can use this technique to iterate over the bytes of *any* object in C, `floats`, `structs`, or anything!

Let's run one more real-world example with the built-in `qsort()` routine that can sort *anything* thanks to the magic of `void*`s.

(In the following example, you can ignore the word `const`, which we haven't covered yet.)

```
#include <stdio.h>
#include <stdlib.h>

// The type of structure we're going to sort
struct animal {
    char *name;
    int leg_count;
};

// This is a comparison function called by qsort() to help it determine
// what exactly to sort by. We'll use it to sort an array of struct
// animals by leg_count.
int compar(const void *elem1, const void *elem2)
{
    // We know we're sorting struct animals, so let's make both
    // arguments pointers to struct animals
    const struct animal *animal1 = elem1;
    const struct animal *animal2 = elem2;

    // Return <0 =0 or >0 depending on whatever we want to sort by.

    // Let's sort ascending by leg_count, so we'll return the difference
    // in the leg_counts
    if (animal1->leg_count > animal2->leg_count)
        return 1;

    if (animal1->leg_count < animal2->leg_count)
        return -1;

    return 0;
}

int main(void)
{
    // Let's build an array of 4 struct animals with different
    // characteristics. This array is out of order by leg_count, but
    // we'll sort it in a second.
    struct animal a[4] = {
        {.name="Dog", .leg_count=4},
        {.name="Monkey", .leg_count=2},
        {.name="Antelope", .leg_count=4},
        {.name="Snake", .leg_count=0}
    };

    // Call qsort() to sort the array. qsort() needs to be told exactly
    // what to sort this data by, and we'll do that inside the compar()
    // function.
```

```
    //
    // This call is saying: qsort array a, which has 4 elements, and
    // each element is sizeof(struct animal) bytes big, and this is the
    // function that will compare any two elements.
    qsort(a, 4, sizeof(struct animal), compar);

    // Print them all out
    for (int i = 0; i < 4; i++) {
        printf("%d: %s\n", a[i].leg_count, a[i].name);
    }
}
```

As long as you give `qsort()` a function that can compare two items that you have in your array to be sorted, it can sort anything. And it does this without needing to have the types of the items hardcoded in there anywhere. `qsort()` just rearranges blocks of bytes based on the results of the `compar()` function you passed in.

Chapter 12

Manual Memory Allocation

This is one of the big areas where C likely diverges from languages you already know: *manual memory management*.

Other languages uses reference counting, garbage collection, or other means to determine when to allocate new memory for some data—and when to deallocate it when no variables refer to it.

And that's nice. It's nice to be able to not worry about it, to just drop all the references to an item and trust that at some point the memory associated with it will be freed.

But C's not like that, entirely.

Of course, in C, some variables are automatically allocated and deallocated when they come into scope and leave scope. We call these automatic variables. They're your average run-of-the-mill block scope "local" variables. No problem.

But what if you want something to persist longer than a particular block? This is where manual memory management comes into play.

You can tell C explicitly to allocate for you a certain number of bytes that you can use as you please. And these bytes will remain allocated until you explicitly free that memory[1].

It's important to free the memory you're done with! If you don't, we call that a *memory leak* and your process will continue to reserve that memory until it exits.

If you manually allocated it, you have to manually free it when you're done with it.

So how do we do this? We're going to learn a couple new functions, and make use of the `sizeof` operator to help us learn how many bytes to allocate.

In common C parlance, devs say that automatic local variables are allocated "on the stack", and manually-allocated memory is "on the heap". The spec doesn't talk about either of those things, but all C devs will know what you're talking about if you bring them up.

All functions we're going to learn in this chapter can be found in `<stdlib.h>`.

12.1 Allocating and Deallocating, `malloc()` and `free()`

The `malloc()` function accepts a number of bytes to allocate, and returns a void pointer to that block of newly-allocated memory.

[1] Or until the program exits, in which case all the memory allocated by it is freed. Asterisk: some systems allow you to allocate memory that persists after a program exits, but it's system dependent, out of scope for this guide, and you'll certainly never do it on accident.

www.dbooks.org

Since it's a `void*`, you can assign it into whatever pointer type you want… normally this will correspond in some way to the number of bytes you're allocating.

So… how many bytes should I allocate? We can use `sizeof` to help with that. If we want to allocate enough room for a single `int`, we can use `sizeof(int)` and pass that to `malloc()`.

After we're done with some allocated memory, we can call `free()` to indicate we're done with that memory and it can be used for something else. As an argument, you pass the same pointer you got from `malloc()` (or a copy of it). It's undefined behavior to use a memory region after you `free()` it.

Let's try. We'll allocate enough memory for an `int`, and then store something there, and the print it.

```
// Allocate space for a single int (sizeof(int) bytes-worth):

int *p = malloc(sizeof(int));

*p = 12;  // Store something there

printf("%d\n", *p);  // Print it: 12

free(p);  // All done with that memory

//*p = 3490;  // ERROR: undefined behavior! Use after free()!
```

Now, in that contrived example, there's really no benefit to it. We could have just used an automatic `int` and it would have worked. But we'll see how the ability to allocate memory this way has its advantages, especially with more complex data structures.

One more thing you'll commonly see takes advantage of the fact that `sizeof` can give you the size of the result type of any constant expression. So you could put a variable name in there, too, and use that. Here's an example of that, just like the previous one:

```
int *p = malloc(sizeof *p);  // *p is an int, so same as sizeof(int)
```

12.2 Error Checking

All the allocation functions return a pointer to the newly-allocated stretch of memory, or `NULL` if the memory cannot be allocated for some reason.

Some OSes like Linux can be configured in such a way that `malloc()` never returns `NULL`, even if you're out of memory. But despite this, you should always code it up with protections in mind.

```
int *x;

x = malloc(sizeof(int) * 10);

if (x == NULL) {
    printf("Error allocating 10 ints\n");
    // do something here to handle it
}
```

Here's a common pattern that you'll see, where we do the assignment and the condition on the same line:

```
int *x;

if ((x = malloc(sizeof(int) * 10)) == NULL)
    printf("Error allocating 10 ints\n");
```

```
    // do something here to handle it
}
```

12.3 Allocating Space for an Array

We've seen how to allocate space for a single thing; now what about for a bunch of them in an array?

In C, an array is a bunch of the same thing back-to-back in a contiguous stretch of memory.

We can allocate a contiguous stretch of memory—we've seen how to do that. If we wanted 3490 bytes of memory, we could just ask for it:

```
char *p = malloc(3490);  // Voila
```

And—indeed!—that's an array of 3490 `char`s (AKA a string!) since each `char` is 1 byte. In other words, `sizeof(char)` is `1`.

Note: there's no initialization done on the newly-allocated memory—it's full of garbage. Clear it with `memset()` if you want to, or see `calloc()`, below.

But we can just multiply the size of the thing we want by the number of elements we want, and then access them using either pointer or array notation. Example!

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    // Allocate space for 10 ints
    int *p = malloc(sizeof(int) * 10);

    // Assign them values 0-45:
    for (int i = 0; i < 10; i++)
        p[i] = i * 5;

    // Print all values 0, 5, 10, 15, ..., 40, 45
    for (int i = 0; i < 10; i++)
        printf("%d\n", p[i]);

    // Free the space
    free(p);
}
```

The key's in that `malloc()` line. If we know each `int` takes `sizeof(int)` bytes to hold it, and we know we want 10 of them, we can just allocate exactly that many bytes with:

```
sizeof(int) * 10
```

And this trick works for every type. Just pass it to `sizeof` and multiply by the size of the array.

12.4 An Alternative: `calloc()`

This is another allocation function that works similarly to `malloc()`, with two key differences:

- Instead of a single argument, you pass the size of one element, and the number of elements you wish to allocate. It's like it's made for allocating arrays.
- It clears the memory to zero.

You still use `free()` to deallocate memory obtained through `calloc()`.

Here's a comparison of `calloc()` and `malloc()`.

```
// Allocate space for 10 ints with calloc(), initialized to 0:
int *p = calloc(sizeof(int), 10);

// Allocate space for 10 ints with malloc(), initialized to 0:
int *q = malloc(sizeof(int) * 10);
memset(q, 0, sizeof(int) * 10);   // set to 0
```

Again, the result is the same for both except `malloc()` doesn't zero the memory by default.

12.5 Changing Allocated Size with `realloc()`

If you've already allocated 10 `ints`, but later you decide you need 20, what can you do?

One option is to allocate some new space, and then `memcpy()` the memory over… but it turns out that sometimes you don't need to move anything. And there's one function that's just smart enough to do the right thing in all the right circumstances: `realloc()`.

It takes a pointer to some previously-allocted memory (by `malloc()` or `calloc()`) and a new size for the memory region to be.

It then grows or shrinks that memory, and returns a pointer to it. Sometimes it might return the same pointer (if the data didn't have to be copied elsewhere), or it might return a different one (if the data did have to be copied).

Be sure when you call `realloc()`, you specify the number of *bytes* to allocate, and not just the number of array elements! That is:

```
num_floats *= 2;

np = realloc(p, num_floats);  // WRONG: need bytes, not number of elements!

np = realloc(p, num_floats * sizeof(float));  // Better!
```

Let's allocate an array of 20 `floats`, and then change our mind and make it an array of 40.

We're going to assign the return value of `realloc()` into another pointer just to make sure it's not `NULL`. If it's not, then we can reassign it into our original pointer. (If we just assigned the return value directly into the original pointer, we'd lose that pointer if the function returned `NULL` and we'd have no way to get it back.)

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    // Allocate space for 20 floats
    float *p = malloc(sizeof *p * 20);  // sizeof *p same as sizeof(float)

    // Assign them fractional values 0.0-1.0:
    for (int i = 0; i < 20; i++)
        p[i] = i / 20.0;

    // But wait! Let's actually make this an array of 40 elements
    float *new_p = realloc(p, sizeof *p * 40);

```

```
16      // Check to see if we successfully reallocated
17      if (new_p == NULL) {
18          printf("Error reallocing\n");
19          return 1;
20      }
21
22      // If we did, we can just reassign p
23      p = new_p;
24
25      // And assign the new elements values in the range 1.0-2.0
26      for (int i = 20; i < 40; i++)
27          p[i] = 1.0 + (i - 20) / 20.0;
28
29      // Print all values 0.0-2.0 in the 40 elements:
30      for (int i = 0; i < 40; i++)
31          printf("%f\n", p[i]);
32
33      // Free the space
34      free(p);
35  }
```

Notice in there how we took the return value from `realloc()` and reassigned it into the same pointer variable `p` that we passed in. That's pretty common to do.

Also if line 7 is looking weird, with that `sizeof *p` in there, remember that `sizeof` works on the size of the type of the expression. And the type of `*p` is `float`, so that line is equivalent to `sizeof(float)`.

12.5.1 Reading in Lines of Arbitrary Length

I want to demonstrate two things with this full-blown example.

1. Use of `realloc()` to grow a buffer as we read in more data.
2. Use of `realloc()` to shrink the buffer down to the perfect size after we've completed the read.

What we see here is a loop that calls `fgetc()` over and over to append to a buffer until we see that the last character is a newline.

Once it finds the newline, it shrinks the buffer to just the right size and returns it.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   // Read a line of arbitrary size from a file
5   //
6   // Returns a pointer to the line.
7   // Returns NULL on EOF or error.
8   //
9   // It's up to the caller to free() this pointer when done with it.
10  //
11  // Note that this strips the newline from the result. If you need
12  // it in there, probably best to switch this to a do-while.
13
14  char *readline(FILE *fp)
15  {
16      int offset = 0;   // Index next char goes in the buffer
17      int bufsize = 4;  // Preferably power of 2 initial size
```

```c
    char *buf;        // The buffer
    int c;            // The character we've read in

    buf = malloc(bufsize);  // Allocate initial buffer

    if (buf == NULL)   // Error check
        return NULL;

    // Main loop--read until newline or EOF
    while (c = fgetc(fp), c != '\n' && c != EOF) {

        // Check if we're out of room in the buffer accounting
        // for the extra byte for the NUL terminator
        if (offset == bufsize - 1) {  // -1 for the NUL terminator
            bufsize *= 2;  // 2x the space

            char *new_buf = realloc(buf, bufsize);

            if (new_buf == NULL) {
                free(buf);   // On error, free and bail
                return NULL;
            }

            buf = new_buf;  // Successful realloc
        }

        buf[offset++] = c;  // Add the byte onto the buffer
    }

    // We hit newline or EOF...

    // If at EOF and we read no bytes, free the buffer and
    // return NULL to indicate we're at EOF:
    if (c == EOF && offset == 0) {
        free(buf);
        return NULL;
    }

    // Shrink to fit
    if (offset < bufsize - 1) {  // If we're short of the end
        char *new_buf = realloc(buf, offset + 1); // +1 for NUL terminator

        // If successful, point buf to new_buf;
        // otherwise we'll just leave buf where it is
        if (new_buf != NULL)
            buf = new_buf;
    }

    // Add the NUL terminator
    buf[offset] = '\0';

    return buf;
}
```

```

int main(void)
{
    FILE *fp = fopen("foo.txt", "r");

    char *line;

    while ((line = readline(fp)) != NULL) {
        printf("%s\n", line);
        free(line);
    }

    fclose(fp);
}
```

When growing memory like this, it's common (though hardly a law) to double the space needed each step just to minimize the number of `realloc()`s that occur.

Finally you might note that `readline()` returns a pointer to a `malloc()`d buffer. As such, it's up to the caller to explicitly `free()` that memory when it's done with it.

12.5.2 `realloc()` with `NULL`

Trivia time! These two lines are equivalent:

```
char *p = malloc(3490);
char *p = realloc(NULL, 3490);
```

That could be convenient if you have some kind of allocation loop and you don't want to special-case the first `malloc()`.

```
int *p = NULL;
int length = 0;

while (!done) {
    // Allocate 10 more ints:
    length += 10;
    p = realloc(p, sizeof *p * length);

    // Do amazing things
    // ...
}
```

In that example, we didn't need an initial `malloc()` since `p` was `NULL` to start.

12.6 Aligned Allocations

You probably aren't going to need to use this.

And I don't want to get too far off in the weeds talking about it right now, but there's this thing called *memory alignment*, which has to do with the memory address (pointer value) being a multiple of a certain number.

For example, a system might require that 16-bit values begin on memory addresses that are multiples of 2. Or that 64-bit values begin on memory addresses that are multiples of 2, 4, or 8, for example. It depends on the CPU.

Some systems require this kind of alignment for fast memory access, or some even for memory access at all.

www.dbooks.org

Now, if you use `malloc()`, `calloc()`, or `realloc()`, C will give you a chunk of memory that's well-aligned for any value at all, even `structs`. Works in all cases.

But there might be times that you know that some data can be aligned at a smaller boundary, or must be aligned at a larger one for some reason. I imagine this is more common with embedded systems programming.

In those cases, you can specify an alignment with `aligned_alloc()`.

The alignment is an integer power of two greater than zero, so `2`, `4`, `8`, `16`, etc. and you give that to `aligned_alloc()` before the number of bytes you're interested in.

The other restriction is that the number of bytes you allocate needs to be a multiple of the alignment. But this might be changing. See C Defect Report 460[2]

Let's do an example, allocating on a 64-byte boundary:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    // Allocate 256 bytes aligned on a 64-byte boundary
    char *p = aligned_alloc(64, 256);  // 256 == 64 * 4

    // Copy a string in there and print it
    strcpy(p, "Hello, world!");
    printf("%s\n", p);

    // Free the space
    free(p);
}
```

I want to throw a note here about `realloc()` and `aligned_alloc()`. `realloc()` doesn't have any alignment guarantees, so if you need to get some aligned reallocated space, you'll have to do it the hard way with `memcpy()`.

Here's a non-standard `aligned_realloc()` function, if you need it:

```
void *aligned_realloc(void *ptr, size_t old_size, size_t alignment, size_t size)
{
    char *new_ptr = aligned_alloc(alignment, size);

    if (new_ptr == NULL)
        return NULL;

    size_t copy_size = old_size < size? old_size: size;  // get min

    if (ptr != NULL)
        memcpy(new_ptr, ptr, copy_size);

    free(ptr);

    return new_ptr;
}
```

[2] http://www.open-std.org/jtc1/sc22/wg14/www/docs/summary.htm#dr_460

Note that it *always* copies data, taking time, while real `realloc()` will avoid that if it can. So this is hardly efficient. Avoid needing to reallocate custom-aligned data.

www.dbooks.org

Chapter 13

Scope

Scope is all about what variables are visible in what contexts.

13.1 Block Scope

This is the scope of almost all the variables devs define. It includes what other languages might call "function scope", i.e. variables that are declared inside functions.

The basic rule is that if you've declared a variable in a block delimited by squirrelly braces, the scope of that variable is that block.

If there's a block inside a block, then variables declared in the *inner* block are local to that block, and cannot be seen in the outer scope.

Once a variable's scope ends, that variable can no longer be referenced, and you can consider its value to be gone into the great bit bucket[1] in the sky.

An example with nested scope:

```c
#include <stdio.h>

int main(void)
{
    int a = 12;         // Local to outer block, but visible in inner block

    if  (a == 12) {
        int b = 99;     // Local to inner block, not visible in outer block

        printf("%d %d\n", a, b);  // OK: "12 99"
    }

    printf("%d\n", a);  // OK, we're still in a's scope

    printf("%d\n", b);  // ILLEGAL, out of b's scope
}
```

[1] https://en.wikipedia.org/wiki/Bit_bucket

13.1.1 Where To Define Variables

Another fun fact is that you can define variables anywhere in the block, within reason—they have the scope of that block, but cannot be used before they are defined.

```
#include <stdio.h>

int main(void)
{
    int i = 0;

    printf("%d\n", i);     // OK: "0"

    //printf("%d\n", j);   // ILLEGAL--can't use j before it's defined

    int j = 5;

    printf("%d %d\n", i, j);   // OK: "0 5"
}
```

Historically, C required all the variables be defined before any code in the block, but this is no longer the case in the C99 standard.

13.1.2 Variable Hiding

If you have a variable named the same thing at an inner scope as one at an outer scope, the one at the inner scope takes precedence at long as you're running in the inner scope. That is, it *hides* the one at outer scope for the duration of its lifetime.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i = 10;
6
7      {
8          int i = 20;
9
10         printf("%d\n", i);  // Inner scope i, 20 (outer i is hidden)
11     }
12
13     printf("%d\n", i);  // Outer scope i, 10
14 }
```

You might have noticed in that example that I just threw a block in there at line 7, not so much as a `for` or `if` statement to kick it off! This is perfectly legal. Sometimes a dev will want to group a bunch of local variables together for a quick computation and will do this, but it's rare to see.

13.2 File Scope

If you define a variable outside of a block, that variable has *file scope*. It's visible in all functions in the file that come after it, and shared between them. (An exception is if a block defines a variable of the same name, it would hide the one at file scope.)

This is closest to what you would consider to be "global" scope in another language.

For example:

```
#include <stdio.h>

int shared = 10;    // File scope! Visible to the whole file after this!

void func1(void)
{
    shared += 100;  // Now shared holds 110
}

void func2(void)
{
    printf("%d\n", shared);  // Prints "110"
}

int main(void)
{
    func1();
    func2();
}
```

Note that if `shared` were declared at the bottom of the file, it wouldn't compile. It has to be declared *before* any functions use it.

There are ways to further modify items at file scope, namely with static and extern, but we'll talk more about those later.

13.3 `for`-loop Scope

I really don't know what to call this, as C11 §6.8.5.3¶1 doesn't give it a proper name. We've done it already a few times in this guide, as well. It's when you declare a variable inside the first clause of a `for`-loop:

```
for (int i = 0; i < 10; i++)
    printf("%d\n", i);

printf("%d\n", i);  // ILLEGAL--i is only in scope for the for-loop
```

In that example, `i`'s lifetime begins the moment it is defined, and continues for the duration of the loop.

If the loop body is enclosed in a block, the variables defined in the `for`-loop are visible from that inner scope.

Unless, of course, that inner scope hides them. This crazy example prints `999` five times:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 5; i++) {
        int i = 999;  // Hides the i in the for-loop scope
        printf("%d\n", i);
    }
}
```

13.4 A Note on Function Scope

The C spec does refer to *function scope*, but it's used exclusively with *labels*, something we haven't discussed yet. More on that another day.

www.dbooks.org

Chapter 14

Types II: Way More Types!

We're used to `char`, `int`, and `float` types, but it's now time to take that stuff to the next level and see what else we have out there in the types department!

14.1 Signed and Unsigned Integers

So far we've used `int` as a *signed* type, that is, a value that can be either negative or positive. But C also has specific *unsigned* integer types that can only hold positive numbers.

These types are prefaced by the keyword `unsigned`.

```
int a;           // signed
signed int a;    // signed
signed a;        // signed, "shorthand" for "int" or "signed int", rare
unsigned int b;  // unsigned
unsigned c;      // unsigned, shorthand for "unsigned int"
```

Why? Why would you decide you only wanted to hold positive numbers?

Answer: you can get larger numbers in an unsigned variable than you can in a signed ones.

But why is that?

You can think of integers being represented by a certain number of *bits*[1]. On my computer, an `int` is represented by 64 bits.

And each permutation of bits that are either `1` or `0` represents a number. We can decide how to divvy up these numbers.

With signed numbers, we use (roughly) half the permutations to represent negative numbers, and the other half to represent positive numbers.

With unsigned, we use *all* the permutations to represent positive numbers.

On my computer with 64-bit `int`s using two's complement[2] to represent unsigned numbers, I have the following limits on integer range:

[1] "Bit" is short for *binary digit*. Binary is just another way of representing numbers. Instead of digits 0-9 like we're used to, it's digits 0-1.

[2] https://en.wikipedia.org/wiki/Two%27s_complement

Type	Minimum	Maximum
`int`	`-9,223,372,036,854,775,808`	`9,223,372,036,854,775,807`
`unsigned int`	`0`	`18,446,744,073,709,551,615`

Notice that the largest positive `unsigned int` is approximately twice as large as the largest positive `int`. So you can get some flexibility there.

14.2 Character Types

Remember `char`? The type we can use to hold a single character?

```
char c = 'B';

printf("%c\n", c);  // "B"
```

I have a shocker for you: it's actually an integer.

```
char c = 'B';

// Change this from %c to %d:
printf("%d\n", c);  // 66 (!!)
```

Deep down, `char` is just a small `int`, namely an integer that uses just a single byte of space, limiting its range to…

Here the C spec gets just a little funky. It assures us that a `char` is a single byte, i.e. `sizeof(char) == 1`. But then in C11 §3.6¶3 it goes out of its way to say:

> A byte is composed of a contiguous sequence of bits, *the number of which is implementation-defined.*

Wait—what? Some of you might be used to the notion that a byte is 8 bits, right? I mean, that's what it is, right? And the answer is, "Almost certainly."[3] But C is an old language, and machines back in the day had, shall we say, a more *relaxed* opinion over how many bits were in a byte. And through the years, C has retained this flexibility.

But assuming your bytes in C are 8 bits, like they are for virtually all machines in the world that you'll ever see, the range of a `char` is…

—So before I can tell you, it turns out that `char`s might be signed or unsigned depending on your compiler. Unless you explicitly specify.

In many cases, just having `char` is fine because you don't care about the sign of the data. But if you need signed or unsigned `char`s, you *must* be specific:

```
char a;           // Could be signed or unsigned
signed char b;    // Definitely signed
unsigned char c;  // Definitely unsigned
```

OK, now, finally, we can figure out the range of numbers if we assume that a `char` is 8 bits and your system uses the virtually universal two's complement representation for signed and unsigned[4].

So, assuming those constraints, we can finally figure our ranges:

[3] The industry term for a sequence of exactly, indisputably 8 bits is an *octet*.

[4] In general, f you have an n bit two's complement number, the signed range is -2^{n-1} to $2^{n-1}-1$. And the unsigned range is 0 to 2^n-1.

www.dbooks.org

`char` type	Minimum	Maximum
`signed char`	-128	127
`unsigned char`	0	255

And the ranges for `char` are implementation-defined.

Let me get this straight. `char` is actually a number, so can we do math on it?

Yup! Just remember to keep things in the range of a `char`!

```
#include <stdio.h>

int main(void)
{
    char a = 10, b = 20;

    printf("%d\n", a + b);  // 30!
}
```

What about those constant characters in single quotes, like `'B'`? How does that have a numeric value?

The spec is also hand-wavey here, since C isn't designed to run on a single type of underlying system.

But let's just assume for the moment that your character set is based on ASCII[5] for at least the first 128 characters. In that case, the character constant will be converted to a `char` whose value is the same as the ASCII value of the character.

That was a mouthful. Let's just have an example:

```
#include <stdio.h>

int main(void)
{
    char a = 10;
    char b = 'B';  // ASCII value 66

    printf("%d\n", a + b);  // 76!
}
```

This depends on your execution environment and the character set used[6]. One of the most popular character sets today is Unicode[7] (which is a superset of ASCII), so for your basic 0-9, A-Z, a-z and punctuation, you'll almost certainly get the ASCII values out of them.

14.3 More Integer Types: `short`, `long`, `long long`

So far we've just generally been using two integer types:

- `char`
- `int`

and we recently learned about the unsigned variants of the integer types. And we learned that `char` was secretly a small `int` in disguise. So we know the `int`s can come in multiple bit sizes.

[5] https://en.wikipedia.org/wiki/ASCII
[6] https://en.wikipedia.org/wiki/List_of_information_system_character_sets
[7] https://en.wikipedia.org/wiki/Unicode

But there are a couple more integer types we should look at, and the *minimum* minimum and maximum values they can hold.

Yes, I said "minimum" twice. The spec says that these types will hold numbers of *at least* these sizes, so your implementation might be different. The header file `<limits.h>` defines macros that hold the minimum and maximum integer values; rely on that to be sure, and *never hardcode or assume these values*.

These additional types are `short int`, `long int`, and `long long int`. Commonly, when using these types, C developers leave the `int` part off (e.g. `long long`), and the compiler is perfectly happy.

```
// These two lines are equivalent:
long long int x;
long long x;

// And so are these:
short int x;
short x;
```

Let's take a look at the integer data types and sizes in ascending order, grouped by signedness.

Type	Minimum Bytes	Minimum Value	Maximum Value
`char`	1	-127 or 0	127 or 255[8]
`signed char`	1	-127	127
`short`	2	-32767	32767
`int`	2	-32767	32767
`long`	4	-2147483647	2147483647
`long long`	8	-9223372036854775807	9223372036854775807
`unsigned char`	1	0	255
`unsigned short`	2	0	65535
`unsigned int`	2	0	65535
`unsigned long`	4	0	44294967295
`unsigned long long`	8	0	9223372036854775807

There is no `long long long` type. You can't just keep adding `long`s like that. Don't be silly.

> Two's complement fans might have noticed something funny about those numbers. Why does, for example, the `signed char` stop at -127 instead of -128? Remember: these are only the minimums required by the spec. Some number representations (like sign and magnitude[9]) top off at ±127.

Let's run the same table on my 64-bit, two's complement system and see what comes out:

Type	My Bytes	Minimum Value	Maximum Value
`char`	1	-128	127[10]
`signed char`	1	-128	127
`short`	2	-32768	32767
`int`	4	-2147483648	2147483647
`long`	8	-9223372036854775808	9223372036854775807
`long long`	8	-9223372036854775808	9223372036854775807
`unsigned char`	1	0	255
`unsigned short`	2	0	65535
`unsigned int`	4	0	4294967295

[8] Depends on if a `char` defaults to `signed char` or `unsigned char`

[9] https://en.wikipedia.org/wiki/Signed_number_representations#Signed_magnitude_representation

www.dbooks.org

Type	My Bytes	Minimum Value	Maximum Value
`unsigned long`	8	0	18446744073709551615
`unsigned long long`	8	0	18446744073709551615

That's a little more sensible, but we can see how my system has larger limits than the minimums in the specification.

So what are the macros in `<limits.h>`?

Type	Min Macro	Max Macro
`char`	`CHAR_MIN`	`CHAR_MAX`
`signed char`	`SCHAR_MIN`	`SCHAR_MAX`
`short`	`SHRT_MIN`	`SHRT_MAX`
`int`	`INT_MIN`	`INT_MAX`
`long`	`LONG_MIN`	`LONG_MAX`
`long long`	`LLONG_MIN`	`LLONG_MAX`
`unsigned char`	`0`	`UCHAR_MAX`
`unsigned short`	`0`	`USHRT_MAX`
`unsigned int`	`0`	`UINT_MAX`
`unsigned long`	`0`	`ULONG_MAX`
`unsigned long long`	`0`	`ULLONG_MAX`

Notice there's a way hidden in there to determine if a system uses signed or unsigned `char`s. If `CHAR_MAX == UCHAR_MAX`, it must be unsigned.

Also notice there's no minimum macro for the `unsigned` variants—they're just `0`.

14.4 More Float: `double` and `long double`

Let's see what the spec has to say about floating point numbers in §5.2.4.2.2¶1-2:

> The following parameters are used to define the model for each floating-point type:

Parameter	Definition
s	sign (± 1)
b	base or radix of exponent representation (an integer > 1)
e	exponent (an integer between a minimum e_{min} and a maximum e_{max})
p	precision (the number of base-b digits in the significand)
f_k	nonnegative integers less than b (the significand digits)

> A *floating-point number* (x) is defined by the following model:

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}, \quad e_{min} \le e \le e_{max}$$

[10] My `char` is signed.

I hope that cleared it right up for you.

Okay, fine. Let's step back a bit and see what's practical.

Note: we refer to a bunch of macros in this section. They can be found in the header `<float.h>`.

Floating point number are encoded in a specific sequence of bits (IEEE-754 format[11] is tremendously popular) in bytes.

Diving in a bit more, the number is basically represented as the *significand* (which is the number part—the significant digits themselves, also sometimes referred to as the *mantissa*) and the *exponent*, which is what power to raise the digits to. Recall that a negative exponent can make a number smaller.

Imagine we're using 10 as a number to raise by an exponent. We could represent the following numbers by using a significand of 12345, and exponents of -3, 4, and 0 to encode the following floating point values:

$12345 \times 10^{-3} = 12.345$

$12345 \times 10^{4} = 123450000$

$12345 \times 10^{0} = 12345$

For all those numbers, the significand stays the same. The only difference is the exponent.

On your machine, the base for the exponent is probably 2, not 10, since computers like binary. You can check it by printing the `FLT_RADIX` macro.

So we have a number that's represented by a number of bytes, encoded in some way. Because there are a limited number of bit patterns, a limited number of floating point numbers can be represented.

But more particularly, only a certain number of significant decimal digits can be represented accurately.

How can you get more? You can use larger data types!

And we have a couple of them. We know about `float` already, but for more precision we have `double`. And for even more precision, we have `long double` (unrelated to `long int` except by name).

The spec doesn't go into how many bytes of storage each type should take, but on my system, we can see the relative size increases:

|Type|`sizeof`| |:-|-:| |`float`|4| |`double`|8| |`long double`|16|

So each of the types (on my system) uses those additional bits for more precision.

But *how much* precision are we talking, here? How many decimal numbers can be represented by these values?

Well, C provides us with a bunch of macros in `<float.h>` to help us figure that out.

It gets a little wonky if you are using a base-2 (binary) system for storing the numbers (which is virtually everyone on the planet, probably including you), but bear with me while we figure it out.

14.4.1 How Many Decimal Digits?

The million dollar question is, "How many significant decimal digits can I store in a given floating point type so that I get out the same decimal number when I print it?"

The number of decimal digits you can store in a floating point type and surely get the same number back out when you print it is given by these macros:

Type	Decimal Digits You Can Store	Minimum
`float`	`FLT_DIG`	6

[11] https://en.wikipedia.org/wiki/IEEE_754

www.dbooks.org

Type	Decimal Digits You Can Store	Minimum
`double`	`DBL_DIG`	10
`long double`	`LDBL_DIG`	10

On my system, `FLT_DIG` is 6, so I can be sure that if I print out a 6 digit `float`, I'll get the same thing back. (It could be more digits—some numbers will come back correctly with more digits. But 6 is definitely coming back.)

For example, printing out `floats` following this pattern of increasing digits, we apparently make it to 8 digits before something goes wrong, but after that we're back to 7 correct digits.

```
0.12345
0.123456
0.1234567
0.12345678
0.123456791  <-- Things start going wrong
0.1234567910
```

Let's do another demo. In this code we'll have two `floats` that both hold numbers that have `FLT_DIG` significant decimal digits[12]. Then we add those together, for what should be 12 significant decimal digits. But that's more than we can store in a `float` and correctly recover as a string—so we see when we print it out, things start going wrong after the 7th significant digit.

```c
#include <stdio.h>
#include <float.h>

int main(void)
{
    // Both these numbers have 6 significant digits, so they can be
    // stored accurately in a float:

    float f = 3.14159f;
    float g = 0.00000265358f;

    printf("%.5f\n", f);   // 3.14159       -- correct!
    printf("%.11f\n", g);  // 0.00000265358 -- correct!

    // Now add them up
    f += g;                // 3.14159265358 is what f _should_ be

    printf("%.11f\n", f);  // 3.14159274101 -- wrong!
}
```

(The above code has an `f` after the numeric constants—this indicates that the constant is type `float`, as opposed to the default of `double`. More on this later.)

Remember that `FLT_DIG` is the safe number of digits you can store in a `float` and retrieve correctly.

Sometimes you might get one or two more out of it. But sometimes you'll only get `FLT_DIG` digits back. The sure thing: if you store any number of digits up to and including `FLT_DIG` in a `float`, you're sure to get them back correctly.

So that's the story. `FLT_DIG`. The End.

…Or is it?

[12]This program runs as its comments indicate on a system with `FLT_DIG` of `6` that uses IEEE-754 base-2 floating point numbers. Otherwise, you might get different output.

14.4.2 Converting to Decimal and Back

But storing a base 10 number in a floating point number and getting it back out is only half the story.

Turns out floating point numbers can encode numbers that require more decimal places to print out completely. It's just that your big decimal number might not map to one of those numbers.

That is, when you look at floating point numbers from one to the next, there's a gap. If you try to encode a decimal number in that gap, it'll use the closest floating point number. That's why you can only encode `FLT_DIG` for a `float`.

But what about those floating point numbers that *aren't* in the gap? How many places do you need to print those out accurately?

Another way to phrase this question is for any given floating point number, how many decimal digits do I have to preserve if I want to convert the decimal number back into an identical floating point number? That is, how many digits do I have to print in base 10 to recover **all** the digits in base 2 in the original number?

Sometimes it might only be a few. But to be sure, you'll want to convert to decimal with a certain safe number of decimal places. That number is encoded in the following macros:

Macro	Description
`FLT_DECIMAL_DIG`	Number of decimal digits encoded in a `float`.
`DBL_DECIMAL_DIG`	Number of decimal digits encoded in a `double`.
`LDBL_DECIMAL_DIG`	Number of decimal digits encoded in a `long double`.
`DECIMAL_DIG`	Same as the widest encoding, `LDBL_DECIMAL_DIG`.

Let's see an example where `DBL_DIG` is 15 (so that's all we can have in a constant), but `DBL_DECIMAL_DIG` is 17 (so we have to convert to 17 decimal numbers to preserve all the bits of the original `double`).

Let's assign the 15 significant digit number `0.123456789012345` to `x`, and let's assign the 1 significant digit number `0.0000000000000006` to `y`.

```
x is exact: 0.12345678901234500    Printed to 17 decimal places
y is exact: 0.00000000000000060
```

But let's add them together. This should give `0.1234567890123456`, but that's more than `DBL_DIG`, so strange things might happen… let's look:

```
x + y not quite right: 0.12345678901234559    Should end in 4560!
```

That's what we get for printing more than `DBL_DIG`, right? But check this out… that number, above, is exactly representable as it is!

If we assign `0.12345678901234559` (17 digits) to `z` and print it, we get:

```
z is exact: 0.12345678901234559   17 digits correct! More than DBL_DIG!
```

If we'd truncated `z` down to 15 digits, it wouldn't have been the same number. That's why to preserve all the bits of a `double`, we need `DBL_DECIMAL_DIG` and not just the lesser `DBL_DIG`.

All that being said, it's clear that when we're messing with decimal numbers in general, it's not safe to print more than `FLT_DIG`, `DBL_DIG`, or `LDBL_DIG` digits to be sensible in relation to the original base 10 numbers and any subsequent math.

But when converting from `float` to a decimal representation and *back* to `float`, definitely use `FLT_DECIMAL_DIG` to do that so that all the bits are preserved exactly.

www.dbooks.org

14.5 Constant Numeric Types

When you write down a constant number, like `1234`, it has a type. But what type is it? Let's look at the how C decides what type the constant is, and how to force it to choose a specific type.

14.5.1 Hexadecimal and Octal

In addition to good ol' decimal like Grandma used to bake, C also supports constants of different bases.

If you lead a number with `0x`, it is read as a hex number:

```
int a = 0x1A2B;   // Hexadecimal
int b = 0x1a2b;   // Case doesn't matter for hex digits

printf("%x", a);  // Print a hex number, "1a2b"
```

If you lead a number with a `0`, it is read as an octal number:

```
int a = 012;

printf("%o\n", a);  // Print an octal number, "12"
```

This is particularly problematic for beginner programmers who try to pad decimal numbers on the left with `0` to line things up nice and pretty, inadvertently changing the base of the number:

```
int x = 11111;  // Decimal 11111
int y = 00111;  // Decimal 73 (Octal 111)
int z = 01111;  // Decimal 585 (Octal 1111)
```

14.5.1.1 A Note on Binary

An unofficial extension[13] in many C compilers allows you to represent a binary number with a `0b` prefix:

```
int x = 0b101010;    // Binary 101010

printf("%d\n", x);   // Prints 42 decimal
```

There's no `printf()` format specifier for printing a binary number. You have to do it a character at a time with bitwise operators.

14.5.2 Integer Constants

You can force a constant integer to be a certain type by appending a suffix to it that indicates the type.

We'll do some assignments to demo, but most often devs leave off the suffixes unless needed to be precise. The compiler is pretty good at making sure the types are compatible.

```
int           x = 1234;
long int      x = 1234L;
long long int x = 1234LL

unsigned int           x = 1234U;
unsigned long int      x = 1234UL;
unsigned long long int x = 1234ULL;
```

[13] It's really surprising to me that C doesn't have this in the spec yet. In the C99 Rationale document, they write, "A proposal to add binary constants was rejected due to lack of precedent and insufficient utility." Which seems kind of silly in light of some of the other features they kitchen-sinked in there! I'll bet one of the next releases has it.

The suffix can be uppercase or lowercase. And the `U` and `L` or `LL` can appear either one first.

Type	Suffix
`int`	None
`long int`	`L`
`long long int`	`LL`
`unsigned int`	`U`
`unsigned long int`	`UL`
`unsigned long long int`	`ULL`

I mentioned in the table that "no suffix" means `int`… but it's actually more complex than that.

So what happens when you have an unsuffixed number like:

```
int x = 1234;
```

What type is it?

What C will generally do is choose the smallest type from `int` up that can hold the value.

But specifically, that depends on the number's base (decimal, hex, or octal), as well.

The spec has a great table indicating which type gets used for what unsuffixed value. In fact, I'm just going to copy it wholesale right here.

C11 §6.4.4.1¶5 reads, "The type of an integer constant is the first of the first of the corresponding list in which its value can be represented."

And then goes on to show this table:

Suffix	Decimal Constant	Octal or Hexadecimal Constant

none	`int`
	`long int`
		`int`
		`unsigned int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`u` or `U`	`unsigned int`
	`unsigned long int`
	`unsigned long long int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long int`

`l` or `L`	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

Both `u` or `U` and `l` or `L`	`unsigned long int` `unsigned long long int`	`unsigned long int` `unsigned long long int`
`ll` or `LL`	`long long int`	`long long int` `unsigned long long int`

Suffix	Decimal Constant	Octal or Hexadecimal Constant
Both `u` or `U` and `ll` or `LL`	`unsigned long long int`	`unsigned long long int`

What that's saying is that, for example, if you specify a number like `123456789U`, first C will see if it can be `unsigned int`. If it doesn't fit there, it'll try `unsigned long int`. And then `unsigned long long int`. It'll use the smallest type that can hold the number.

14.5.3 Floating Point Constants

You'd think that a floating point constant like `1.23` would have a default type of `float`, right?

Surprise! Turns out unsuffiexed floating point numbers are type `double`! Happy belated birthday!

You can force it to be of type `float` by appending an `f` (or `F`—it's case-insensitive). You can force it to be of type `long double` by appending `l` (or `L`).

Type	Suffix
`float`	`F`
`double`	None
`long double`	`L`

For example:

```
float x       = 3.14f;
double x      = 3.14;
long double x = 3.14L;
```

This whole time, though, we've just been doing this, right?

```
float x = 3.14;
```

Isn't the left a `float` and the right a `double`? Yes! But C's pretty good with automatic numeric conversions, so it's more common to have an unsuffixed floating point constant than not. More on that later.

14.5.3.1 Scientific Notation

Remember earlier when we talked about how a floating point number can be represented by a significand, base, and exponent?

Well, there's a common way of writing such a number, shown here followed by it's more recognizable equivalent which is what you get when you actually run the math:

$1.2345 \times 10^3 = 1234.5$

Writing numbers in the form $s \times b^e$ is called *scientific notation*[14]. In C, these are written using "E notation", so these are equivalent:

Scientific Notation	E notation
$1.2345 \times 10^{-3} = 0.0012345$	`1.2345e-3`
$1.2345 \times 10^{8} = 123450000$	`1.2345e+8`

[14] https://en.wikipedia.org/wiki/Scientific_notation

You can print a number in this notation with `%e`:

```
printf("%e\n", 123456.0);  // Prints 1.234560e+05
```

A couple little fun facts about scientific notation:

- You don't have to write them with a single leading digit before the decimal point. Any number of numbers can go in front.

  ```
  double x = 123.456e+3;  // 123456
  ```

  However, when you print it, it will change the exponent so there is only one digit in front of the decimal point.

- The plus can be left off the exponent, as it's default, but this is uncommon in practice from what I've seen.

  ```
  1.2345e10 == 1.2345e+10
  ```

- You can apply the `F` or `L` suffixes to E-notation constants:

  ```
  1.2345e10F
  1.2345e10L
  ```

14.5.3.2 Hexadecimal Floating Point Constants

But wait, there's more floating to be done!

Turns out there are hexadecimal floating point constants, as well!

These work similar to decimal floating point numbers, but they begin with a `0x` just like integer numbers.

The catch is that you *must* specify an exponent, and this exponent produces a power of 2. That is: 2^x.

And then you use a `p` instead of an `e` when writing the number:

So `0xa.1p3` is $10.0625 \times 2^3 == 80.5$.

When using floating point hex constants, We can print hex scientific notation with `%a`:

```
double x = 0xa.1p3;

printf("%a\n", x);  // 0x1.42p+6
printf("%f\n", x);  // 80.500000
```

Chapter 15

Types III: Conversions

In this chapter, we want to talk all about converting from one type to another. C has a variety of ways of doing this, and some might be a little different that you're used to in other languages.

Before we talk about how to make conversions happen, let's talk about how they work when they *do* happen.

15.1 String Conversions

Unlike many languages, C doesn't do string-to-number (and vice-versa) conversions in quite as streamlined a manner as it does numeric conversions.

For these, we'll have to call functions to do the dirty work.

15.1.1 Numeric Value to String

When we want to convert a number to a string, we can use either `sprintf()` (pronounced *SPRINT-f*) or `snprintf()` (*s-n-print-f*)[1]

These basically work like `printf()`, except they output to a string instead, and you can print that string later, or whatever.

For example, turning part of the value π into a string:

```
#include <stdio.h>

int main(void)
{
    char s[10];
    float f = 3.14159;

    // Convert "f" to string, storing in "s", writing at most 10 characters
    // including the NUL terminator

    snprintf(s, 10, "%f", f);

    printf("String value: %s\n", s);  // String value: 3.141590
}
```

If we wanted to convert a `double`, we'd use `%lf`. Or a `long double`, `%Lf`.

[1] They're the same except `snprintf()` allows you to specify a maximum number of bytes to output, preventing the overrunning of the end of your string. So it's safer.

15.1.2 String to Numeric Value

There are a couple families of functions to do this in C. We'll call these the `atoi` (pronounced *a-to-i*) family and the `strtol` (*stir-to-long*) family.

For basic conversion from a string to a number, try the `atoi` functions from `<stdlib.h>`. These have bad error-handling characteristics (including undefined behavior if you pass in a bad string), so use them carefully.

Function	Description
`atoi`	String to `int`
`atof`	String to `float`
`atol`	String to `long int`
`atoll`	String to `long long int`

Though the spec doesn't cop to it, the `a` at the beginning of the function stands for ASCII[2], so really `atoi()` is "ASCII-to-integer", but saying so today is a bit ASCII-centric.

Here's an example converting a string to a `float`:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *pi = "3.14159";
    float f;

    f = atof(pi);

    printf("%f\n", f);
}
```

But, like I said, we get undefined behavior from weird things like this:

```
int x = atoi("what");  // "What" ain't no number I ever heard of
```

(When I run that, I get `0` back, but you really shouldn't count on that in any way. You could get something completely different.)

For better error handling characteristics, let's check out all those `strtol` functions, also in `<stdlib.h>`. Not only that, but they convert to more types and more bases, too!

Function	Description
`strtol`	String to `long int`
`strtoll`	String to `long long int`
`strtoul`	String to `unsigned long int`
`strtoull`	String to `unsigned long long int`
`strtof`	String to `float`
`strtod`	String to `double`
`strtold`	String to `long double`

These functions all follow a similar pattern of use, and are a lot of people's first experience with pointers to pointers! But never fret—it's easier than it looks.

[2] https://en.wikipedia.org/wiki/ASCII

www.dbooks.org

Let's do an example where we convert a string to an `unsigned long`, discarding error information (i.e. information about bad characters in the input string):

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *s = "3490";

    // Convert string s, a number in base 10, to an unsigned long int.
    // NULL means we don't care to learn about any error information.

    unsigned long int x = strtoul(s, NULL, 10);

    printf("%lu\n", x);  // 3490
}
```

Notice a couple things there. Even though we didn't deign to capture any information about error characters in the string, `strtoul()` won't give us undefined behavior; it will just return `0`.

Also, we specified that this was a decimal (base 10) number.

Does this mean we can convert numbers of different bases? Sure! Let's do binary!

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *s = "101010";  // What's the meaning of this number?

    // Convert string s, a number in base 2, to an unsigned long int.

    unsigned long int x = strtoul(s, NULL, 2);

    printf("%lu\n", x);  // 42
}
```

OK, that's all fun and games, but what's with that `NULL` in there? What's that for?

That helps us figure out if an error occurred in the processing of the string. It's a pointer to a pointer to a `char`, which sounds scary, but isn't once you wrap your head around it.

Let's do an example where we feed in a deliberately bad number, and we'll see how `strtol()` lets us know where the first invalid digit is.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *s = "34x90";  // "x" is not a valid digit in base 10!
    char *badchar;

    // Convert string s, a number in base 10, to an unsigned long int.

    unsigned long int x = strtoul(s, &badchar, 10);
```

```

    // It tries to convert as much as possible, so gets this far:

    printf("%lu\n", x);  // 34

    // But we can see the offending bad character because badchar
    // points to it!

    printf("Invalid character: %c\n", *badchar);  // "x"
}
```

So there we have `strtoul()` modifying what `badchar` points to in order to show us where things went wrong[3].

But what if nothing goes wrong? In that case, `badchar` will point to the `NUL` terminator at the end of the string. So we can test for it:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *s = "3490";  // "x" is not a valid digit in base 10!
    char *badchar;

    // Convert string s, a number in base 10, to an unsigned long int.

    unsigned long int x = strtoul(s, &badchar, 10);

    // Check if things went well

    if (*badchar == '\0') {
        printf("Success! %lu\n", x);
    } else  {
        printf("Partial conversion: %lu\n", x);
        printf("Invalid character: %c\n", *badchar);
    }
}
```

So there you have it. The `atoi()`-style functions are good in a controlled pinch, but the `strtol()`-style functions give you far more control over error handling and the base of the input.

15.2 char Conversions

What if you have a single character with a digit in it, like `'5'`… Is that the same as the value `5`?

Let's try it and see.

```
printf("%d %d\n", 5, '5');
```

On my UTF-8 system, this prints:

```
5 53
```

[3]We have to pass a pointer to `badchar` to `strtoul()` or it won't be able to modify it in any way we can see, analogous to why you have to pass a pointer to an `int` to a function if you want that function to be able to change that value of that `int`.

www.dbooks.org

So… no. And 53? What is that? That's the UTF-8 (and ASCII) code point for the character symbol `'5'`[4]

So how do we convert the character `'5'` (which apparently has value 53) into the value `5`?

With one clever trick, that's how!

The C Standard guarantees that these character will have code points that are in sequence and in this order:

```
0  1  2  3  4  5  6  7  8  9
```

Ponder for a second–how can we use that? Spoilers ahead…

Let's take a look at the characters and their code points in UTF-8:

```
0  1  2  3  4  5  6  7  8  9
48 49 50 51 52 53 54 55 56 57
```

You see there that `'5'` is `53`, just like we were getting. And `'0'` is `48`.

So we can subtract `'0'` from any digit character to get its numeric value:

```
char c = '6';

int x = c;  // x has value 54, the code point for '6'

int y = c - '0'; // y has value 6, just like we want
```

And we can convert the other way, too, just by adding the value on.

```
int x = 6;

char c = x + '0';  // c has value 54

printf("%d\n", c);  // prints 54
printf("%c\n", c);  // prints 6 with %c
```

You might think this is a weird way to do this conversion, and by today's standards, it certainly is. But back in the olden days when computers were made literally out of wood, this was the method for doing this conversion. And it wasn't broke, so C never fixed it.

15.3 Numeric Conversions

15.3.1 Boolean

If you convert a zero to `bool`, the result is `0`. Otherwise it's `1`.

15.3.2 Integer to Integer Conversions

If an integer type is converted to unsigned and doesn't fit in it, the unsigned result wraps around odometer-style until it fits in the unsigned[5].

If an integer type is converted to a signed number and doesn't fit, the result is implementation-defined! Something documented will happen, but you'll have to look it up[6]

[4] Each character has a value associated with it for any given character encoding scheme.

[5] In practice, what's probably happening on your implementation is that the high-order bits are just being dropped from the result, so a 16-bit number `0x1234` being converted to an 8-bit number ends up as `0x0034`, or just `0x34`.

[6] Again, in practice, what will likely happen on your system is that the bit pattern for the original will be truncated and then just used to represent the signed number, two's complement. For example, my system takes an `unsigned char` of `192` and converts it to `signed char -64`. In two's complement, the bit pattern for both these numbers is binary `11000000`.

15.3.3 Integer and Floating Point Conversions

If a floating point type is converted to an integer type, the fractional part is discarded with prejudice[7].

But—and here's the catch—if the number is too large to fit in the integer, you get undefined behavior. So don't do that.

Going From integer or floating point to floating point, C makes a best effort to find the closest floating point number to the integer that it can.

Again, though, if the original value can't be represented, it's undefined behavior.

15.4 Implicit Conversions

These are conversions the compiler does automatically for you when you mix and match types.

15.4.1 The Integer Promotions

In a number of places, if an `int` can be used to represent a value from `char` or `short` (signed or unsigned), that value is *promoted* up to `int`. If it doesn't fit in an `int`, it's promoted to `unsigned int`.

This is how we can do something like this:

```
char x = 10, y = 20;
int i = x + y;
```

In that case, `x` and `y` get promoted to `int` by C before the math takes place.

The integer promotions take place during The Usual Arithmetic Conversions, with variadic functions[8], unary `+` and `-` operators, or when passing values to functions without prototypes[9].

15.4.2 The Usual Arithmetic Conversions

These are automatic conversions that C does around numeric operations that you ask for. (That's actually what they're called, by the way, by C11 §6.3.1.8.) Note that for this section, we're just talking about numeric types—strings will come later.

These conversions answer questions about what happens when you mix types, like this:

```
int x = 3 + 1.2;   // Mixing int and double
                   // 4.2 is converted to int
                   // 4 is stored in x

float y = 12 * 2;  // Mixing float and int
                   // 24 is converted to float
                   // 24.0 is stored in y
```

Do they become `int`s? Do they become `float`s? How does it work?

Here are the steps, paraphrased for easy consumption.

1. If one thing in the expression is a floating type, convert the other things to that floating type.
2. Otherwise, if both types are integer types, perform the integer promotions on each, then make the operand types as big as they need to be hold the common largest value. Sometimes this involves changing signed to unsigned.

[7] Not really—it's just discarded regularly.

[8] Functions with a variable number of arguments.

[9] This is rarely done because the compiler will complain and having a prototype is the *Right Thing* to do. I think this still works for historic reasons, before prototypes were a thing.

www.dbooks.org

If you want to know the gritty details, check out C11 §6.3.1.8. But you probably don't.

Just generally remember that int types become float types if there's a floating point type anywhere in there, and the compiler makes an effort to make sure mixed integer types don't overflow.

Finally, if you convert from one floating point type to another, the compiler will try to make an exact conversion. If it can't, it'll do the best approximation it can. If the number is too large to fit in the type you're converting into, *boom*: undefined behavior!

15.4.3 `void*`

The `void*` type is interesting because it can be converted from or to any pointer type.

```
int x = 10;

void *p = &x;  // &x is type int*, but we store it in a void*

int *q = p;    // p is void*, but we store it in an int*
```

15.5 Explicit Conversions

These are conversions from type to type that you have to ask for; the compiler won't do it for you.

You can convert from one type to another by assigning one type to another with an `=`.

You can also convert explicitly with a *cast*.

15.5.1 Casting

You can explicitly change the type of an expression by putting a new type in parentheses in front of it. Some C devs frown on the practice unless absolutely necessary, but it's likely you'll come across some C code with these in it.

Let's do an example where we want to convert an `int` into a `long` so that we can store it in a `long`.

Note: this example is contrived and the cast in this case is completely unnecessary because the `x + 12` expression would automatically be changed to `long int` to match the wider type of `y`.

```
int x = 10;
long int y = (long int)x + 12;
```

In that example, even those `x` was type `int` before, the expression `(long int)x` has type `long int`. We say, "We cast `x` to `long int`."

More commonly, you might see a cast being used to convert a `void*` into a specific pointer type so it can be dereferenced.

A callback from the built-in `qsort()` function might display this behavior since it has `void*`s passed into it:

```
int compar(const void *elem1, const void *elem2)
{
    if (*((const int*)elem2) > *((const int*)elem1)) return 1;
    if (*((const int*)elem2) < *((const int*)elem1)) return -1;
    return 0;
}
```

But you could also clearly write it with an assignment:

```
int compar(const void *elem1, const void *elem2)
{
    const int *e1 = elem1;
    const int *e2 = elem2;

    return *e2 - *e1;
}
```

One place you'll see casts more commonly is to avoid a warning when printing pointer values with the rarely-used `%p` which gets picky with anything other than a `void*`:

```
int x = 3490;
int *p = &x;

printf("%p\n", p);
```

generates this warning:

```
warning: format '%p' expects argument of type 'void *', but argument
         2 has type 'int *'
```

You can fix it with a cast:

```
printf("%p\n", (void *)p);
```

Another place is with explicit pointer changes, if you don't want to use an intervening `void*`, but these are also pretty uncommon:

```
long x = 3490;
long *p = &x;
unsigned char *c = (unsigned char *)p;
```

A third place it's often required is with the character conversion functions in `<ctype.h>` where you should cast questionably-signed values to `unsigned char` to avoid undefined behavior.

Again, casting is rarely *needed* in practice. If you find yourself casting, there might be another way to do the same thing, or maybe you're casting unnecessarily.

Or maybe it is necessary. Personally, I try to avoid it, but am not afraid to use it if I have to.

www.dbooks.org

Chapter 16

Types IV: Qualifiers and Specifiers

Now that we have some more types under our belts, turns out we can give these types some additional attributes that control their behavior. These are the *type qualifiers* and *storage-class specifiers*.

16.1 Type Qualifiers

These are going to allow you to declare constant values, and also to give the compiler optimization hints that it can use.

16.1.1 `const`

This is the most common type qualifier you'll see. It means the variable is constant, and any attempt to modify it will result in a very angry compiler.

```
const int x = 2;

x = 4;  // COMPILER PUKING SOUNDS, can't assign to a constant
```

You can't change a `const` value.

Often you see `const` in parameter lists for functions:

```
void foo(const int x)
{
    printf("%d\n", x + 30);  // OK, doesn't modify "x"
}
```

16.1.1.1 `const` and Pointers

This one gets a little funky, because there are two usages that have two meanings when it comes to pointers.

For one thing, we can make it so you can't change the thing the pointer points to. You do this by putting the `const` up front with the type name (before the asterisk) in the type declaration.

```
int x[] = {10, 20};
const int *p = x;

p++;  // We can modify p, no problem

*p = 30; // Compiler error! Can't change what it points to
```

Somewhat confusingly, these two things are equivalent:

```
const int *p;  // Can't modify what p points to
int const *p;  // Can't modify what p points to, just like the previous line ```

Great, so we can't change the thing the pointer points to, but we can
change the pointer itself. What if we want the other way around? We want
to be able to change what the pointer points to, but _not_ the pointer
itself?

Just move the `const` after the asterisk in the declaration:

```{.c}
int *const p;   // We can't modify "p" with pointer arithmetic

p++;  // Compiler error!
```

But we can modify what they point to:

```
int x = 10;
int *const p = &x;

*p = 20;   // Set "x" to 20, no problem
```

You can also do make both things `const`:

```
const int *const p;  // Can't modify p or *p!
```

Finally, if you have multiple levels of indirection, you should `const` the appropriate levels. Just because a pointer is `const`, doesn't mean the pointer it points to must also be. You can explicitly set them like in the following examples:

```
char **p;
p++;     // OK!
(*p)++;  // OK!

char **const p;
p++;     // Error!
(*p)++;  // OK!

char *const *p;
p++;     // OK!
(*p)++;  // Error!

char *const *const p;
p++;     // Error!
(*p)++;  // Error!
```

16.1.1.2 `const` Correctness

One more thing I have to mention is that the compiler will warn on something like this:

```
const int x = 20;
int *p = &x;
```

saying something to the effect of:
```

```
initialization discards 'const' qualifier from pointer type target
```

What's happening there?

Well, we need to look at the types on either side of the assignment:

```
    const int x = 20;
    int *p = &x;
//    ^       ^
//    |       |
//  int*    const int*
```

The compiler is warning us that the value on the right side of the assignment is `const`, but the one of the left is not. And the compiler is letting us know that it is discarding the "const-ness" of the expression on the right.

That is, we *can* still try to do the following, but it's just wrong. The compiler will warn, and it's undefined behavior:

```
const int x = 20;
int *p = &x;

*p = 40;  // Undefined behavior--maybe it modifies "x", maybe not!

printf("%d\n", x);  // 40, if you're lucky
```

16.1.2 `restrict`

TLDR: you never have to use this and you can ignore it every time you see it. If you use it correctly, you will likely realize some performance gain. If you use it incorrectly, you will realize undefined behavior.

`restrict` is a hint to the compiler that a particular piece of memory will only be accessed by one pointer and never another. (That is, there will be no aliasing of the particular object the `restrict` pointer points to.) If a developer declares a pointer to be `restrict` and then accesses the object it points to in another way (e.g. via another pointer), the behavior is undefined.

Basically you're telling C, "Hey—I guarantee that this one single pointer is the only way I access this memory, and if I'm lying, you can pull undefined behavior on me."

And C uses that information to perform certain optimizations. For instance, if you're dereferencing the `restrict` pointer repeatedly in a loop, C might decide to cache the result in a register and only store the final result once the loop completes. If any other pointer referred to that same memory and accessed it in the loop, the results would not be accurate.

(Note that `restrict` has no effect if the object pointed to is never written to. It's all about optimizations surrounding writes to memory.)

Let's write a function to swap two variables, and we'll use the `restrict` keyword to assure C that we'll never pass in pointers to the same thing. And then let's blow it an try passing in pointers to the same thing.

```
void swap(int *restrict a, int *restrict b)
{
    int t;

    t = *a;
    *a = *b;
    *b = t;
}

```

```
int main(void)
{
    int x = 10, y = 20;

    swap(&x, &y);  // OK! "a" and "b", above, point to different things

    swap(&x, &x);  // Undefined behavior! "a" and "b" point to the same thing
}
```

If we were to take out the `restrict` keywords, above, that would allow both calls to work safely. But then the compiler might not be able to optimize.

`restrict` has block scope, that is, the restriction only lasts for the scope it's used. If it's in a parameter list for a function, it's in the block scope of that function.

If the restricted pointer points to an array, it only applies to the individual objects in the array. Other pointers could read and write from the array as long as they didn't read or write any of the same elements as the restricted one.

If it's outside any function in file scope, the restriction covers the entire program.

You're likely to see this in library functions like `printf()`:

```
int printf(const char * restrict format, ...);
```

Again, that's just telling the compiler that inside the `printf()` function, there will be only one pointer that refers to any part of that `format` string.

One last note: if you're using array notation in your function parameter for some reason instead of pointer notation, you can use `restrict` like so:

```
void foo(int p[restrict])     // With no size

void foo(int p[restrict 10])  // Or with a size
```

But pointer notation would be more common.

16.1.3 `volatile`

You're unlikely to see or need this unless you're dealing with hardware directly.

`volatile` tells the compiler that a value might change behind its back and should be looked up every time.

An example might be where the compiler is looking in memory at an address that continuously updates behind the scenes, e.g. some kind of hardware timer.

If the compiler decides to optimize that and store the value in a register for a protracted time, the value in memory will update and won't be reflected in the register.

By declaring something `volatile`, you're telling the compiler, "Hey, the thing this points at might change at any time for reasons outside this program code."

```
volatile int *p;
```

16.1.4 `_Atomic`

This is an optional C feature that we'll talk about in the Atomics chapter.

16.2 Storage-Class Specifiers

Storage-class specifiers are similar to type quantifiers. They give the compiler more information about the type of a variable.

16.2.1 `auto`

You barely ever see this keyword, since `auto` is the default for block scope variables. It's implied.

These are the same:

```
{
    int a;          // auto is the default...
    auto int a;     // So this is redundant
}
```

The `auto` keyword indicates that this object has *automatic storage duration*. That is, it exists in the scope in which it is defined, and is automatically deallocated when the scope is exited.

One gotcha about automatic variables is that their value is indeterminate until you explicitly initialize them. We say they're full of "random" or "garbage" data, though neither of those really makes me happy. In any case, you won't know what's in it unless you initialize it.

Always initialize all automatic variables before use!

16.2.2 `static`

This keyword has two meanings, depending on if the variable is file scope or block scope.

Let's start with block scope.

16.2.2.1 `static` in Block Scope

In this case, we're basically saying, "I just want a single instance of this variable to exist, shared between calls."

That is, its value will persist between calls.

`static` in block scope with an initializer will only be initialized one time on program startup, not each time the function is called.

Let's do an example:

```
#include <stdio.h>

void counter(void)
{
    static int count = 1;  // This is initialized one time

    printf("This has been called %d time(s)\n", count);

    count++;
}

int main(void)
{
    counter();  // "This has been called 1 time(s)"
    counter();  // "This has been called 2 time(s)"
    counter();  // "This has been called 3 time(s)"
```

```
    counter();  // "This has been called 4 time(s)"
}
```

See how the value of `count` persists between calls?

One thing of note is that `static` block scope variables are initialized to `0` by default.

```
static int foo;      // Default starting value is `0`...
static int foo = 0;  // So the `0` assignment is redundant
```

Finally, be advised that if you're writing multithreaded programs, you have to be sure you don't let multiple threads trample the same variable.

16.2.2.2 `static` in File Scope

When you get out to file scope, outside any blocks, the meaning rather changes.

Variables at file scope already persist between function calls, so that behavior is already there.

Instead what `static` means in this context is that this variable isn't visible outside of this particular source file. Kinda like "global", but only in this file.

More on that in the section about building with multiple source files.

16.2.3 `extern`

The `extern` storage-class specifier gives us a way to refer to objects in other source files.

Let's say, for example, the file `bar.c` had the following as its entirety:

```
// bar.c

int a = 37;
```

Just that. Declaring a new `int a` in file scope.

But what if we had another source file, `foo.c`, and we wanted to refer to the `a` that's in `bar.c`?

It's easy with the `extern` keyword:

```
// foo.c

extern int a;

int main(void)
{
    printf("%d\n", a);  // 37, from bar.c!

    a = 99;

    printf("%d\n", a);  // Same "a" from bar.c, but it's now 99
}
```

We could have also made the `extern int a` in block scope, and it still would have referred to the `a` in `bar.c`:

```
// foo.c

int main(void)
{
```

```
    extern int a;

    printf("%d\n", a);  // 37, from bar.c!

    a = 99;

    printf("%d\n", a);  // Same "a" from bar.c, but it's now 99
}
```

Now, if a in `bar.c` had been marked `static`. this wouldn't have worked. `static` variables at file scope are not visible outside that file.

A final note about `extern` on functions. For functions, `extern` is the default, so it's redundant. You can declare a function `static` if you only want it visible in a single source file.

16.2.4 `register`

This is a keyword to hint to the compiler that this variable is frequently-used, and should be made as fast as possible to access. The compiler is under no obligation to agree to it.

Now, modern C compiler optimizers are pretty effective at figuring this out themselves, so it's rare to see these days.

But if you must:

```
#include <stdio.h>

int main(void)
{
    register int a;   // Make "a" as fast to use as possible.

    for (a = 0; a < 10; a++)
        printf("%d\n", a);
}
```

It does come at a price, however. You can't take the address of a register:

```
register int a;
int *p = &a;    // COMPILER ERROR! Can't take address of a register
```

The same applies to any part of an array:

```
register int a[] = {11, 22, 33, 44, 55};
int *p = a;  // COMPILER ERROR! Can't take address of a[0]
```

Or dereferencing part of an array:

```
register int a[] = {11, 22, 33, 44, 55};

int a = *(a + 2);  // COMPILER ERROR! Address of a[0] taken
```

Interestingly, for the equivalent with array notation, gcc only warns:

```
register int a[] = {11, 22, 33, 44, 55};

int a = a[2];  // COMPILER WARNING!
```

with:

```
warning: ISO C forbids subscripting ‘register’ array
```

The fact that you can't take the address of a register variable frees the compiler up to make optimizations around that assumption if it hasn't figured them out already. Also adding `register` to a `const` variable prevents one from accidentally passing its pointer to another function that willfully ignore its constness[1].

A bit of historic backstory, here: deep inside the CPU are little dedicated "variables" called *registers*[2]. They are super fast to access compared to RAM, so using them gets you a speed boost. But they're not in RAM, so they don't have an associated memory address (which is why you can't take the address-of or get a pointer to them).

But, like I said, modern compilers are really good at producing optimal code, using registers whenever possible regardless of whether or not you specified the `register` keyword. Not only that, but the spec allows them to just treat it as if you'd typed `auto`, if they want. So no guarantees.

16.2.5 `_Thread_local`

When you're using multiple threads and you have some variables in either global or `static` block scope, this is a way to make sure that each thread gets its own copy of the variable. This'll help you avoid race conditions and threads stepping on each other's toes.

If you're in block scope, you have to use this along with either `extern` or `static`.

Also, if you include `<threads.h>`, you can use the rather more palatable `thread_local` as an alias for the uglier `_Thread_local`.

More information can be found in the Threads section.

[1] https://gustedt.wordpress.com/2010/08/17/a-common-misconsception-the-register-keyword/
[2] https://en.wikipedia.org/wiki/Processor_register

www.dbooks.org

Chapter 17

Multifile Projects

So far we've been looking at toy programs that for the most part fit in a single file. But complex C programs are made up of many files that are all compiled and linked together into a single executable.

In this chapter we'll check out some of the common patterns and practices for putting together larger projects.

17.1 Includes and Function Prototypes

A really common situation is that some of your functions are defined in one file, and you want to call them from another.

This actually works out of the box with a warning… let's first try it and then look at the right way to fix the warning.

For these examples, we'll put the filename as the first comment in the source.

To compile them, you'll need to specify all the sources on the command line:

```
# output file   source files
#     v            v
#   |----| |---------|
gcc -o foo foo.c bar.c
```

In that examples, `foo.c` and `bar.c` get built into the executable named `foo`.

So let's take a look at the source file `bar.c`:

```
// File bar.c

int add(int x, int y)
{
    return x + y;
}
```

And the file `foo.c` with main in it:

```
// File foo.c

#include <stdio.h>

int main(void)
{
```

```
    printf("%d\n", add(2, 3));  // 5!
}
```

See how from `main()` we call `add()`—but `add()` is in a completely different source file! It's in `bar.c`, while the call to it is in `foo.c`!

If we build this with:

```
gcc -o foo foo.c bar.c
```

we get this error:

```
error: implicit declaration of function 'add' is invalid in C99
```

(Or you might get a warning. Which you should not ignore. Never ignore warnings in C; address them all.)

If you recall from the section on prototypes, implicit declarations are banned in modern C and there's no legitimate reason to introduce them into new code. We should fix it.

What `implicit declaration` means is that we're using a function, namely `add()` in this case, without letting C know anything about it ahead of time. C wants to know what it returns, what types it takes as arguments, and things such as that.

We saw how to fix that earlier with a *function prototype*. Indeed, if we add one of those to `foo.c` before we make the call, everything works well:

```
// File foo.c

#include <stdio.h>

int add(int, int);  // Add the prototype

int main(void)
{
    printf("%d\n", add(2, 3));  // 5!
}
```

No more error!

But that's a pain—needing to type in the prototype every time you want to use a function. I mean, we used `printf()` right there and didn't need to type in a prototype; what gives?

If you remember from what back with `hello.c` at the beginning of the book, *we actually did include the prototype for `printf()`*! It's in the file `stdio.h`! And we included that with `#include`!

Can we do the same with our `add()` function? Make a prototype for it and put it in a header file?

Sure!

Header files in C have a `.h` extension by default. And they often, but not always, have the same name as their corresponding `.c` file. So let's make a `bar.h` file for our `bar.c` file, and we'll stick the prototype in it:

```
// File bar.h

int add(int, int);
```

And now let's modify `foo.c` to include that file. Assuming it's in the same directory, we include it inside double quotes (as opposed to angle brackets):

```
// File foo.c

#include <stdio.h>
```

```

#include "bar.h"  // Include from current directory

int main(void)
{
    printf("%d\n", add(2, 3));  // 5!
}
```

Notice how we don't have the prototype in `foo.c` anymore—we included it from `bar.h`. Now *any* file that wants that `add()` functionality can just `#include "bar.h"` to get it, and you don't need to worry about typing in the function prototype.

As you might have guessed, `#include` literally includes the named file *right there* in your source code, just as if you'd typed it in.

And building and running:

```
./foo
5
```

Indeed, we get the result of $2 + 3$! Yay!

But don't crack open your drink of choice quite yet. We're almost there! There's just one more piece of boilerplate we have to add.

17.2 Dealing with Repeated Includes

It's not uncommon that a header file will itself `#include` other headers needed for the functionality of its corresponding C files. I mean, why not?

And it could be that you have a header `#included` multiple times from different places. Maybe that's no problem, but maybe it would cause compiler errors. And we can't control how many places `#include` it!

Even, worse we might get into a crazy situation where header `a.h` includes header `b.h`, and `b.h` includes `a.h`! It's an `#include` infinite cycle!

Trying to build such a thing gives an error:

```
error: #include nested depth 200 exceeds maximum of 200
```

What we need to do is make it so that if a file gets included once, subsequent `#includes` for that file are ignored.

The stuff that we're about to do is so common that you should just automatically do it every time you make a header file!

And the common way to do this is with a preprocessor variable that we set the first time we `#include` the file. And then for subsequent `#includes`, we first check to make sure that the variable isn't defined.

For that variable name, it's super common to take the name of the header file, like `bar.h`, make it uppercase, and replace the period with an underscore: `BAR_H`.

So put a check at the very, very top of the file where you see if it's already been included, and effectively comment the whole thing out if it has.

(Don't put a leading underscore (because a leading underscore followed by a capital letter is reserved) or a double leading underscore (because that's also reserved.))

```
#ifndef BAR_H   // If BAR_H isn't defined...
#define BAR_H   // Define it (with no particular value)

```

```
// File bar.h

int add(int, int);

#endif          // End of the #ifndef BAR_H
```

This will effectively cause the header file to be included only a single time, no matter how many places try to `#include` it.

17.3 `static` and `extern`

When it comes to multifile projects, you can make sure file-scope variables and functions are *not* visible from other source files with the `static` keyword.

And you can refer to objects in other files with `extern`.

For more info, check out the sections in the book on the `static` and `extern` storage-class specifiers.

17.4 Compiling with Object Files

This isn't part of the spec, but it's 99.999% common in the C world.

You can compile C files into an intermediate representation called *object files*. These are compiled machine code that hasn't been put into an executable yet.

Object files in Windows have a `.OBJ` extension; in Unix-likes, they're `.o`.

In gcc, we can build some like this, with the `-c` (compile only!) flag:

```
gcc -c foo.c     # produces foo.o
gcc -c bar.c     # produces bar.o
```

And then we can *link* those together into a single executable:

```
gcc -o foo foo.o bar.o
```

Voila, we've produced an executable `foo` from the two object files.

But you're thinking, why bother? Can't we just:

```
gcc -o foo foo.c bar.c
```

and kill two boids[1] with one stone?

For little programs, that's fine. I do it all the time.

But for larger programs, we can take advantage of the fact that compiling from source to object files is relatively slow, and linking together a bunch of object files is relatively fast.

This really shows with the `make` utility that only rebuilds sources that are newer than their outputs.

Let's say you had a thousand C files. You could compile them all to object files to start (slowly) and then combine all those object files into an executable (fast).

Now say you modified just one of those C source files—here's the magic: *you only have to rebuild that one object file for that source file*! And then you rebuild the executable (fast). All the other C files don't have to be touched.

In other words, by only rebuilding the object files we need to, we cut down on compilation times radically. (Unless of course you're doing a "clean" build, in which case all the object files have to be created.)

[1] https://en.wikipedia.org/wiki/Boids

www.dbooks.org

Chapter 18

The Outside Environment

When you run a program, it's actually you talking to the shell, saying, "Hey, please run this thing." And the shell says, "Sure," and then tells the operating system, "Hey, could you please make a new process and run this thing?" And if all goes well, the OS complies and your program runs.

But there's a whole world outside your program in the shell that can be interacted with from within C. We'll look at a few of those in this chapter.

18.1 Command Line Arguments

Many command line utilities accept *command line arguments*. For example, if we want to see all files that end in `.txt`, we can type something like this on a Unix-like system:

```
ls *.txt
```

(or `dir` instead of `ls` on a Windows system).

In this case, the command is `ls`, but it arguments are all all files that end with `.txt`[1].

So how can we see what is passed into program from the command line?

Say we have a program called `add` that adds all numbers passed on the command line and prints the result:

```
./add 10 30 5
45
```

That's gonna pay the bills for sure!

But seriously, this is a great tool for seeing how to get those arguments from the command line and break them down.

First, let's see how to get them at all. For this, we're going to need a new `main()`!

Here's a program that prints out all the command line arguments. For example, if we name the executable `foo`, we can run it like this:

```
./foo i like turtles
```

and we'll see this output:

[1] Historially, MS-DOS and Windows programs would do this differently than Unix. In Unix, the shell would *expand* the wildcard into all matching files before your program saw it, whereas the Microsoft variants would pass the wildcard expression into the program to deal with. In any case, there are arguments that get passed into the program.

```
arg 0: ./foo
arg 1: i
arg 2: like
arg 3: turtles
```

It's a little weird, because the zeroth argument is the name of the executable, itself. But that's just something to get used to. The arguments themselves follow directly.

Source:

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    for (int i = 0; i < argc; i++) {
        printf("arg %d: %s\n", i, argv[i]);
    }
}
```

Whoa! What's going on with the `main()` function signature? What's `argc` and `argv`[2] (pronounced *arg-cee* and *arg-vee*)?

Let's start with the easy one first: `argc`. This is the *argument count*, including the program name, itself. If you think of all the arguments as an array of strings, which is exactly what they are, then you can think of `argc` as the length of that array, which is exactly what it is.

And so what we're doing in that loop is going through all the `argv`s and printing them out one at a time, so for a given input:

```
./foo i like turtles
```

we get a corresponding output:

```
arg 0: ./foo
arg 1: i
arg 2: like
arg 3: turtles
```

With that in mind, we should be good to go with our adder program.

Our plan:

- Look at all the command line arguments (past `argv[0]`, the program name)
- Convert them to integers
- Add them to a running total
- Print the result

Let's get to it!

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    int total = 0;

    for (int i = 1; i < argc; i++) {  // Start at 1, the first argument
```

[2] Since they're just regular parameter names, you don't actually have to call them `argc` and `argv`. But it's so very idiomatic to use those names, if you get creative, other C programmers will look at you with a suspicious eye, indeed!

```
        int value = atoi(argv[i]);    // Use strtol() for better error handling

        total += value;
    }

    printf("%d\n", total);
}
```

Sample runs:

```
$ ./add
0
$ ./add 1
1
$ ./add 1 2
3
$ ./add 1 2 3
6
$ ./add 1 2 3 4
10
```

Of course, it might puke if you pass in a non-integer, but hardening against that is left as an exercise to the reader.

18.1.1 The Last `argv` is `NULL`

One bit of fun trivia about `argv` is that after the last string is a pointer to `NULL`.

That is:

```
argv[argc] == NULL
```

is always true!

This might seem pointless, but it turns out to be useful in a couple places; we'll take a look at one of those right now.

18.1.2 The Alternate: `char **argv`

Remember that when you call a function, C doesn't differentiate between array notation and pointer notation in the function signature.

That is, these are the same:

```
void foo(char a[])
void foo(char *a)
```

Now, it's been convenient to think of `argv` as an array of strings, i.e. an array of `char*`s, so this made sense:

```
int main(int argc, char *argv[])
```

but because of the equivalence, you could also write:

```
int main(int argc, char **argv)
```

Yeah, that's a pointer to a pointer, all right! If it makes it easier, think of it as a pointer to a string. But really, it's a pointer to a value that points to a `char`.

Also recall that these are equivalent:

```
argv[i]
*(argv + i)
```

which means you can do pointer arithmetic on `argv`.

So an alternate way to consume the command line arguments might be to just walk along the `argv` array by bumping up a pointer until we hit that `NULL` at the end.

Let's modify our adder to do that:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    int total = 0;

    // Cute trick to get the compiler to stop warning about the
    // unused variable argc:
    (void)argc;

    for (char **p = argv + 1; *p != NULL; p++) {
        int value = atoi(*p);  // Use strtol() for better error handling

        total += value;
    }

    printf("%d\n", total);
}
```

Personally, I use array notation to access `argv`, but have seen this style floating around, as well.

18.1.3 Fun Facts

Just a few more things about `argc` and `argv`.

- Some environments might not set `argv[0]` to the program name. If it's not available, `argv[0]` will be an empty string. I've never seen this happen.
- The spec is actually pretty liberal with what an implementation can do with `argv` and where those values come from. But every system I've been on works the same way, as we've discussed in this section.
- You can modify `argc`, `argv`, or any of the strings that `argv` points to. (Just don't make those strings longer than they already are!)
- On some Unix-like systems, modifying the string `argv[0]` results in the output of `ps` changing[3].

  Normally, if you have a program called `foo` that you've run with `./foo`, you might see this in the output of `ps`:

  ```
   4078 tty1     S      0:00 ./foo
  ```

  But if you modify `argv[0]` like so, being careful that the new string `"Hi!  "` is the same length as the old one `"./foo"`:

  ```
  strcpy(argv[0], "Hi!  ");
  ```

[3] `ps`, Process Status, is a Unix command to see what processes are running at the moment.

and then run `ps` while the program `./foo` is still executing, we'll see this instead:

```
4079 tty1     S      0:00 Hi!
```

This behavior is not in the spec and is highly system-dependent.

18.2 Exit Status

Did you notice that the function signatures for `main()` have it returning type `int`? What's that all about? It has to do with a thing called the *exit status*, which is an integer that can be returned to the program that launched yours to let it know how things went.

Now, there are a number of ways a program can exit in C, including `return`ing from `main()`, or calling one of the `exit()` variants.

All of these methods accept an `int` as an argument.

Side note: did you see that in basically all my examples, even though `main()` is supposed to return an `int`, I don't actually `return` anything? In any other function, this would be illegal, but there's a special case in C: if execution reaches the end of `main()` without finding a `return`, it automatically does a `return 0`.

But what does the `0` mean? What other numbers can we put there? And how are they used?

The spec is both clear and vague on the matter, as is common. Clear because it spells out what you can do, but vague in that it doesn't particularly limit it, either.

Nothing for it but to *forge ahead* and figure it out!

Let's get Inception[4] for a second: turns out that when you run your program, *you're running it from another program*.

Usually this other program is some kind of shell[5] that doesn't do much on its own except launch other programs.

But this is a multi-phase process, especially visible in command-line shells:

1. The shell launches your program
2. The shell typically goes to sleep (for command-line shells)
3. Your program runs
4. Your program terminates
5. The shell wakes up and waits for another command

Now, there's a little piece of communication that takes place between steps 4 and 5: the program can return a *status value* that the shell can interrogate. Typically, this value is used to indicate the success or failure of your program, and, if a failure, what type of failure.

This value is what we've been `return`ing from `main()`. That's the status.

Now, the C spec allows for two different status values, which have macro names defined in `<stdlib.h>`:

Status	Description
`EXIT_SUCCESS` or `0`	Program terminated successfully.
`EXIT_FAILURE`	Program terminated with an error.

Let's write a short program that multiplies two numbers from the command line. We'll require that you specify exactly two values. If you don't, we'll print an error message, and exit with an error status.

[4] https://en.wikipedia.org/wiki/Inception
[5] https://en.wikipedia.org/wiki/Shell_(computing)

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    if (argc != 3) {
        printf("usage: mult x y\n");
        return EXIT_FAILURE;   // Indicate to shell that it didn't work
    }

    printf("%d\n", atoi(argv[1]) * atoi(argv[2]));

    return 0;  // same as EXIT_SUCCESS, everything was good.
}
```

Now if we try to run this, we get the expected effect until we specify exactly the right number of command-line arguments:

```
$ ./mult
usage: mult x y

$ ./mult 3 4 5
usage: mult x y

$ ./mult 3 4
12
```

But that doesn't really show the exit status that we returned, does it? We can get the shell to print it out, though. Assuming you're running Bash or another POSIX shell, you can use `echo $?` to see it[6].

Let's try:

```
$ ./mult
usage: mult x y
$ echo $?
1

$ ./mult 3 4 5
usage: mult x y
$ echo $?
1

$ ./mult 3 4
12
$ echo $?
0
```

Interesting! We see that on my system, `EXIT_FAILURE` is `1`. The spec doesn't spell this out, so it could be any number. But try it; it's probably `1` on your system, too.

18.2.1 Other Exit Status Values

The status `0` most definitely means success, but what about all the other integers, even negative ones?

Here we're going off the C spec and into Unix land. In general, while `0` means success, a positive non-zero

[6] In Windows `cmd.exe`, type `echo %errorlevel%`. In PowerShell, type `$LastExitCode`.

number means failure. So you can only have one type of success, and multiple types of failure. Bash says the exit code should be between 0 and 255, though a number of codes are reserved.

In short, if you want to indicate different error exit statuses in a Unix environment, you can start with `1` and work your way up.

On Linux, if you try any code outside the range 0-255, it will bitwise AND the code with `0xff`, effectively clamping it to that range.

You can script the shell to later use these status codes to make decisions about what to do next.

18.3 Environment Variables

Before I get into this, I need to warn you that C doesn't specify what an environment variable is. So I'm going to describe the environment variable system that works on every major platform I'm aware of.

Basically, the environment is the program that's going to run your program, e.g. the bash shell. And it might have some bash variables defined. In case you didn't know, the shell can make its own variables. Each shell is different, but in bash you can just type `set` and it'll show you all of them.

Here's an excerpt from the 61 variables that are defined in my bash shell:

```
HISTFILE=/home/beej/.bash_history
HISTFILESIZE=500
HISTSIZE=500
HOME=/home/beej
HOSTNAME=FBILAPTOP
HOSTTYPE=x86_64
IFS=$' \t\n'
```

Notice they are in the form of key/value pairs. For example, one key is `HOSTTYPE` and its value is `x86_64`. From a C perspective, all values are strings, even if they're numbers[7].

So, *anyway*! Long story short, it's possible to get these values from inside your C program.

Let's write a program that uses the standard `getenv()` function to look up a value that you set in the shell.

`getenv()` will return a pointer to the value string, or else `NULL` if the environment variable doesn't exist.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *val = getenv("FROTZ");  // Try to get the value

    // Check to make sure it exists
    if (val == NULL) {
        printf("Cannot find the FROTZ environment variable\n");
        return EXIT_FAILURE;
    }

    printf("Value: %s\n", val);
}
```

If I run this directly, I get this:

[7] If you need a numeric value, convert the string with something like `atoi()` or `strtol()`.

```
$ ./foo
Cannot find the FROTZ environment variable
```

which makes sense, since I haven't set it yet.

In bash, I can set it to something with[8]:

```
$ export FROTZ="C is awesome!"
```

Then if I run it, I get:

```
$ ./foo
Value: C is awesome!
```

In this way, you can set up data in environment variables, and you can get it in your C code and modify your behavior accordingly.

18.3.1 Setting Environment Variables

This isn't standard, but a lot of systems provide ways to set environment variables.

If on a Unix-like, look up the documentation for `putenv()`, `setenv()`, and `unsetenv()`. On Windows, see `_putenv()`.

18.3.2 Unix-like Alternative Environment Variables

If you're on a Unix-like system, odds are you have another couple ways of getting access to environment variables. Note that although the spec points this out as a common extension, it's not truly part of the C standard. It is, however, part of the POSIX standard.

One of these is a variable called `environ` that must be declared like so:

```
extern char **environ;
```

It's an array of strings terminated with a `NULL` pointer.

You should declare it yourself before you use it, or you might find it in the non-standard `<unistd.h>` header file.

Each string is in the form `"key=value"` so you'll have to split it and parse it yourself if you want to get the keys and values out.

Here's an example of looping through and printing out the environment variables a couple different ways:

```
#include <stdio.h>

extern char **environ;  // MUST be extern AND named "environ"

int main(void)
{
    for (char **p = environ; *p != NULL; p++) {
        printf("%s\n", *p);
    }

    // Or you could do this:
    for (int i = 0; environ[i] != NULL; i++) {
        printf("%s\n", environ[i]);
    }
}
```

[8] In Windows CMD.EXE, use `set FROTZ=value`. In PowerShell, use `$Env:FROTZ=value`.

www.dbooks.org

For a bunch of output that looks like this:

```
SHELL=/bin/bash
COLORTERM=truecolor
TERM_PROGRAM_VERSION=1.53.2
LOGNAME=beej
HOME=/home/beej
... etc ...
```

Use `getenv()` if at all possible because it's more portable. But if you have to iterate over environment variables, using `environ` might be the way to go.

Another non-standard way to get the environment variables is as a parameter to `main()`. It works much the same way, but you avoid needing to add your `extern environ` variable. Not even the POSIX spec supports this[9] as far as I can tell, but it's common in Unix land.

```c
#include <stdio.h>

int main(int argc, char **argv, char **env)  // <-- env!
{
    (void)argc; (void)argv;  // Suppress unused warnings

    for (char **p = env; *p != NULL; p++) {
        printf("%s\n", *p);
    }

    // Or you could do this:
    for (int i = 0; env[i] != NULL; i++) {
        printf("%s\n", env[i]);
    }
}
```

Just like using `environ` but *even less portable*. It's good to have goals.

[9] https://pubs.opengroup.org/onlinepubs/9699919799/functions/exec.html

Chapter 19

The C Preprocessor

Before your program gets compiled, it actually runs through a phase called *preprocessing*. It's almost like there's a language *on top* of the C language that runs first. And it outputs the C code, which then gets compiled.

We've already seen this to an extent with `#include`! That's the C Preprocessor! Where it sees that directive, it includes the named file right there, just as if you'd typed it in there. And *then* the compiler builds the whole thing.

But it turns out it's a lot more powerful than just being able to include things. You can define *macros* that are substituted… and even macros that take arguments!

19.1 `#include`

Let's start with the one we've already seen a bunch. This is, of course, a way to include other sources in your source. Very commonly used with header files.

While the spec allows for all kinds of behavior with `#include`, we're going to take a more pragmatic approach and talk about the way it works on every system I've ever seen.

We can split header files into two categories: system and local. Things that are built-in, like `stdio.h`, `stdlib.h`, `math.h`, and so on, you can include with angle brackets:

```
#include <stdio.h>
#include <stdlib.h>
```

The angle brackets tell C, "Hey, don't look in the current directory for this header file—look in the system-wide include directory instead."

Which, of course, implies that there must be a way to include local files from the current directory. And there is: with double quotes:

```
#include "myheader.h"
```

Or you can very probably look in relative directories using forward slashes and dots, like this:

```
#include "mydir/myheader.h"
#include "../someheader.py"
```

Don't use a backslash (`\`) for your path separators in your `#include`! It's undefined behavior! Use forward slash (`/`) only, even on Windows.

www.dbooks.org

In summary, used angle brackets (< and >) for the system includes, and use double quotes (") for your personal includes.

19.2 Simple Macros

A *macro* is an identifier that gets *expanded* to another piece of code before the compiler even sees it. Think of it like a placeholder—when the preprocessor sees one of those identifiers, it replaces it with another value that you've defined.

We do this with `#define` (often read "pound define"). Here's an example:

```
1 #include <stdio.h>
2
3 #define HELLO "Hello, world"
4 #define PI 3.14159
5
6 int main(void)
7 {
8     printf("%s, %f\n", HELLO, PI);
9 }
```

On lines 3 and 4 we defined a couple macros. Wherever these appear elsewhere in the code (line 8), they'll be substituted with the defined values.

From the C compiler's perspective, it's exactly as if we'd written this, instead:

```
#include <stdio.h>

int main(void)
{
    printf("%s, %f\n", "Hello, world", 3.14159);
}
```

See how `HELLO` was replaced with `"Hello, world"` and `PI` was replaced with `3.14159`? From the compiler's perspective, it's just like those values had appeared right there in the code.

Note that the macros don't have a specific type, *per se*. Really all that happens is they get replaced wholesale with whatever they're `#defined` as. If the resulting C code is invalid, the compiler will puke.

You can also define a macro with no value:

```
#define EXTRA_HAPPY
```

in that case, the macro exists and is defined, but is defined to be nothing. So anyplace it occurs in the text will just be replaced with nothing. We'll see a use for this later.

It's conventional to write macro names in `ALL_CAPS` even though that's not technically required.

Overall, this gives you a way to define constant values that are effectively global and can be used *any* place. Even in those places where a `const` variable won't work, e.g. in `switch` `case`s and fixed array lengths.

That said, the debate rages online whether a typed `const` variable is better than `#define` macro in the general case.

It can also be used to replace or modify keywords, a concept completely foreign to `const`, though this practice should be used sparingly.

19.3 Conditional Compilation

It's possible to get the preprocessor to decide whether or not to present certain blocks of code to the compiler, or just remove them entirely before compilation.

We do that by basically wrapping up the code in conditional blocks, similar to `if-else` statements.

19.3.1 If Defined, `#ifdef` and `#endif`

First of all, let's try to compile specific code depending on whether or not a macro is even defined.

```
1  #include <stdio.h>
2
3  #define EXTRA_HAPPY
4
5  int main(void)
6  {
7
8  #ifdef EXTRA_HAPPY
9      printf("I'm extra happy!\n");
10 #endif
11
12     printf("OK!\n");
13 }
```

In that example, we define `EXTRA_HAPPY` (to be nothing, but it *is* defined), then on line 8 we check to see if it is defined with an `#ifdef` directive. If it is defined, the subsequent code will be included up until the `#endif`.

So because it is defined, the code will be included for compilation and the output will be:

```
I'm extra happy!
OK!
```

If we were to comment out the `#define`, like so:

```
//#define EXTRA_HAPPY
```

then it wouldn't be defined, and the code wouldn't be included in compilation. And the output would just be:

```
OK!
```

It's important to remember that these decisions happen at compile time! The code actually get compiled or removed depending on the condition. This is in contrast to a standard `if` statement that gets evaluated while the program is running.

19.3.2 If Not Defined, `#ifndef`

There's also the negative sense of "if defined": "if not defined", or `#ifndef`. We could change the previous example to read to output different things based on whether or not something was defined:

```
#ifdef EXTRA_HAPPY
    printf("I'm extra happy!\n");
#endif

#ifndef EXTRA_HAPPY
    printf("I'm just regular\n");
#endif
```

We'll see a cleaner way to do that in the next section.

Tying it all back in to header files, we've seen how we can cause header files to only be included one time by wrapping them in preprocessor directives like this:

```
#ifndef MYHEADER_H  // First line of myheader.h
#define MYHEADER_H

int x = 12;

#endif  // Last line of myheader.h
```

This demonstrates how a macro persists across files and multiple `#includes`. If it's not yet defined, let's define it and compile the whole header file.

But the next time it's included, we see that `MYHEADER_H` *is* defined, so we don't send the header file to the compiler—it gets effectively removed.

19.3.3 `#else`

But that's not all we can do! There's also an `#else` that we can throw in the mix.

Let's mod the previous example:

```
#ifdef EXTRA_HAPPY
    printf("I'm extra happy!\n");
#else
    printf("I'm just regular\n");
#endif
```

Now if `EXTRA_HAPPY` is not defined, it'll hit the `#else` clause and print:

```
I'm just regular
```

19.3.4 General Conditional: `#if`, `#elif`

This works very much like the `#ifdef` and `#ifndef` directives in that you can also have an `#else` and the whole thing wraps up with `#endif`.

The only difference is that the constant expression after the `#if` must evaluate to true (non-zero) for the code in the `#if` to be compiled. So instead of whether or not something is defined, we want an expression that evaluates to true.

```
#include <stdio.h>

#define HAPPY_FACTOR 1

int main(void)
{

#if HAPPY_FACTOR == 0
    printf("I'm not happy!\n");
#elif HAPPY_FACTOR == 1
    printf("I'm just regular\n");
#else
    printf("I'm extra happy!\n");
#endif

```

```
    printf("OK!\n");
}
```

Again, for the unmatched `#if` clauses, the compiler won't even see those lines. For the above code, after the preprocessor gets finished with it, all the compiler sees is:

```
#include <stdio.h>

int main(void)
{

    printf("I'm just regular\n");

    printf("OK!\n");
}
```

One hackish thing this is used for is to comment out large numbers of lines quickly[1].

If you put an `#if 0` ("if false") at the front of the block to be commented out and an `#endif` at the end, you can get this effect:

```
#if 0
    printf("All this code"); /* is effectively */
    printf("commented out"); // by the #if 0
#endif
```

You might have noticed that there's no `#elifdef` or `#elifndef` directives. How can we get the same effect with `#if`? That is, what if I wanted this:

```
#ifdef FOO
    x = 2;
#elifdef BAR  // ERROR: Not supported by standard C
    x = 3;
#endif
```

How could I do it?

Turns out there's a preprocessor operator called `defined` that we can use with an `#if` statement.

These are equivalent:

```
#ifdef FOO
#if defined FOO
#if defined(FOO)   // Parentheses optional
```

As are these:

```
#ifndef FOO
#if !defined FOO
#if !defined(FOO)   // Parentheses optional
```

Notice how we can use the standard logical NOT operator (`!`) for "not defined".

So now we're back in `#if` land and we can use `#elif` with impunity!

This broken code:

```
#ifdef FOO
    x = 2;
#elifdef BAR  // ERROR: Not supported by standard C
```

[1] You can't always just wrap the code in `/* */` comments because those won't nest.

www.dbooks.org

```
    x = 3;
#endif
```

can be replaced with:

```
#if defined FOO
    x = 2;
#elif defined BAR
    x = 3;
#endif
```

19.3.5 Losing a Macro: `#undef`

If you've defined something but you don't need it any longer, you can undefine it with `#undef`.

```
#include <stdio.h>

int main(void)
{
#define GOATS

#ifdef GOATS
    printf("Goats detected!\n");  // prints
#endif

#undef GOATS  // Make GOATS no longer defined

#ifdef GOATS
    printf("Goats detected, again!\n"); // doesn't print
#endif
}
```

19.4 Built-in Macros

The standard defines a lot of built-in macros that you can test and use for conditional compilation. Let's look at those here.

19.4.1 Mandatory Macros

These are all defined:

Macro	Description
`__DATE__`	The date of compilation—like when you're compiling this file—in `Mmm dd yyyy` format
`__TIME__`	The time of compilation in `hh:mm:ss` format
`__FILE__`	A string containing this file's name
`__LINE__`	The line number of the file this macro appears on
`__func__`	The name of the function this appears in, as a string[2]
`__STDC__`	Defined with `1` if this is a standard C compiler
`__STDC_HOSTED__`	This will be `1` if the compiler is a *hosted implementation*[3], otherwise `0`
`__STDC_VERSION__`	This version of C, a constant `long int` in the form `yyyymmL`, e.g. `201710L`

Let's put these together.

```c
#include <stdio.h>

int main(void)
{
    printf("This function: %s\n", __func__);
    printf("This file: %s\n", __FILE__);
    printf("This line: %d\n", __LINE__);
    printf("Compiled on: %s %s\n", __DATE__, __TIME__);
    printf("C Version: %ld\n", __STDC_VERSION__);
}
```

The output on my system is:

```
This function: main
This file: foo.c
This line: 7
Compiled on: Nov 23 2020 17:16:27
C Version: 201710
```

`__FILE__`, `__func__` and `__LINE__` are particularly useful to report error conditions in messages to developers. The `assert()` macro in `<assert.h>` uses these to call out where in the code the assertion failed.

19.4.1.1 `__STDC_VERSION__`s

In case you're wondering, here are the version numbers for different major releases of the C Language Spec:

Release	ISO/IEC version	`__STDC_VERSION__`
C89	ISO/IEC 9899:1990	undefined
C89	ISO/IEC 9899:1990/Amd.1:1995	`199409L`
C99	ISO/IEC 9899:1999	`199901L`
C11	ISO/IEC 9899:2011/Amd.1:2012	`201112L`

Note the macro did not exist originally in C89.

Also note that the plan is that the version numbers will strictly increase, so you could always check for, say, "at least C99" with:

```c
#if __STDC_VERSION__ >= 1999901L
```

19.4.2 Optional Macros

Your implementation might define these, as well. Or it might not.

Macro	Description
`__STDC_ISO_10646__`	If defined, `wchar_t` holds Unicode values, otherwise something else
`__STDC_MB_MIGHT_NEQ_WC__`	A `1` indicates that the values in multibyte characters might not map equally to values in wide characters

[2] This isn't really a macro—it's technically an identifier. But it's the only predefined identifier and it feels very macro-like, so I'm including it here. Like a rebel.

[3] A hosted implementation basically means you're running the full C standard, probably on an operating system of some kind. Which you probably are. If you're running on bare metal in some kind of embedded system, you're probably on a *standalone implementation*.

www.dbooks.org

Macro	Description
`__STDC_UTF_16__`	A `1` indicates that the system uses UTF-16 encoding in type `char16_t`
`__STDC_UTF_32__`	A `1` indicates that the system uses UTF-32 encoding in type `char32_t`
`__STDC_ANALYZABLE__`	A `1` indicates the code is analyzable[4]
`__STDC_IEC_559__`	`1` if IEEE-754 (aka IEC 60559) floating point is supported
`__STDC_IEC_559_COMPLEX__`	`1` if IEC 60559 complex floating point is supported
`__STDC_LIB_EXT1__`	`1` if this implementation supports a variety of "safe" alternate standard library functions (they have `_s` suffixes on the name)
`__STDC_NO_ATOMICS__`	`1` if this implementation does **not** support `_Atomic` or `<stdatomic.h>`
`__STDC_NO_COMPLEX__`	`1` if this implementation does **not** support complex types or `<complex.h>`
`__STDC_NO_THREADS__`	`1` if this implementation does **not** support `<threads.h>`
`__STDC_NO_VLA__`	`1` if this implementation does **not** support variable-length arrays

19.5 Macros with Arguments

Macros are more powerful than simple substitution, though. You can set them up to take arguments that are substituted in, as well.

A question often arises for when to use parameterized macros versus functions. Short answer: use functions. But you'll see lots of macros in the wild and in the standard library. People tend to use them for short, mathy things, and also for features that might change from platform to platform. You can define different keywords for one platform or another.

19.5.1 Macros with One Argument

Let's start with a simple one that squares a number:

```
1 #include <stdio.h>
2
3 #define SQR(x) x * x  // Not quite right, but bear with me
4
5 int main(void)
6 {
7     printf("%d\n", SQR(12));  // 144
8 }
```

What that's saying is "everywhere you see `SQR` with some value, replace it with that value times itself".

So line 7 will be changed to:

```
7     printf("%d\n", 12 * 12);  // 144
```

which C comfortably converts to 144.

[4] OK, I know that was a cop-out answer. Basically there's an optional extension compilers can implement wherein they agree to limit certain types of undefined behavior so that the C code is more amenable to static code analysis. It is unlikely you'll need to use this.

But we've made an elementary error in that macro, one that we need to avoid.

Let's check it out. What if we wanted to compute `SQR(3 + 4)`? Well, $3 + 4 = 7$, so we must want to compute $7^2 = 49$. That's it; `49`—final answer.

Let's drop it in our code and see that we get… 19?

```
    printf("%d\n", SQR(3 + 4));  // 19!!??
```

What happened?

If we follow the macro expansion, we get

```
    printf("%d\n", 3 + 4 * 3 + 4);  // 19!
```

Oops! Since multiplication takes precedence, we do the $4 \times 3 = 12$ first, and get $3 + 12 + 4 = 19$. Not what we were after.

So we have to fix this to make it right.

This is so common that you should automatically do it every time you make a parameterized math macro!

The fix is easy: just add some parentheses!

```
#define SQR(x) (x) * (x)   // Better... but still not quite good enough!
```

And now our macro expands to:

```
    printf("%d\n", (3 + 4) * (3 + 4));  // 49! Woo hoo!
```

But we actually still have the same problem which might manifest if we have a higher-precedence operator than multiply (`*`) nearby.

So the safe, proper way to put the macro together is to wrap the whole thing in additional parentheses, like so:

```
#define SQR(x) ((x) * (x))   // Good!
```

Just make it a habit to do that when you make a math macro and you can't go wrong.

19.5.2 Macros with More than One Argument

You can stack these things up as much as you want:

```
#define TRIANGLE_AREA(w, h) (0.5 * (w) * (h))
```

Let's do some macros that solve for x using the quadratic formula. Just in case you don't have it on the top of your head, it says for equations of the form:

$$ax^2 + bx + c = 0$$

you can solve for x with the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Which is crazy. Also notice the plus-or-minus ($\pm$) in there, indicating that there are actually two solutions.

So let's make macros for both:

```
#define QUADP(a, b, c) ((-(b) + sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
#define QUADM(a, b, c) ((-(b) - sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
```

So that gets us some math. But let's define one more that we can use as arguments to `printf()` to print both answers.

```
//          macro              replacement
//      |-----------| |----------------------------|
#define QUAD(a, b, c) QUADP(a, b, c), QUADM(a, b, c)
```

That's just a couple values separated by a comma—and we can use that as a "combined" argument of sorts to `printf()` like this:

```
printf("x = %f or x = %f\n", QUAD(2, 10, 5));
```

Let's put it together into some code:

```
1  #include <stdio.h>
2  #include <math.h>  // For sqrt()
3
4  #define QUADP(a, b, c) ((-(b) + sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
5  #define QUADM(a, b, c) ((-(b) - sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
6  #define QUAD(a, b, c) QUADP(a, b, c), QUADM(a, b, c)
7
8  int main(void)
9  {
10     printf("2*x^2 + 10*x + 5 = 0\n");
11     printf("x = %f or x = %f\n", QUAD(2, 10, 5));
12 }
```

And this gives us the output:

```
2*x^2 + 10*x + 5 = 0
x = -0.563508 or x = -4.436492
```

Plugging in either of those values gives us roughly zero (a bit off because the numbers aren't exact):

$$2 \times -0.563508^2 + 10 \times -0.563508 + 5 \approx 0.000003$$

19.5.3 Macros with Variable Arguments

There's also a way to have a variable number of arguments passed to a macro, using ellipses (`...`) after the known, named arguments. When the macro is expanded, all of the extra arguments will be in a comma-separated list in the `__VA_ARGS__` macro, and can be replaced from there:

```
1  #include <stdio.h>
2
3  // Combine the first two arguments to a single number,
4  // then have a commalist of the rest of them:
5
6  #define X(a, b, ...) (10*(a) + 20*(b)), __VA_ARGS__
7
8  int main(void)
9  {
10     printf("%d %f %s %d\n", X(5, 4, 3.14, "Hi!", 12));
11 }
```

The substitution that takes place on line 10 would be:

```
10     printf("%d %f %s %d\n", (10*(5) + 20*(4)), 3.14, "Hi!", 12);
```

for output:

```
130 3.140000 Hi! 12
```

You can also "stringify" `__VA_ARGS__` by putting a `#` in front of it:

```
#define X(...) #__VA_ARGS__

printf("%s\n", X(1,2,3));  // Prints "1, 2, 3"
```

19.5.4 Stringification

Already mentioned, just above, you can turn any argument into a string by preceding it with a `#` in the replacement text.

For example, we could print anything as a string with this macro and `printf()`:

```
#define STR(x) #x

printf("%s\n", STR(3.14159));
```

In that case, the substitution leads to:

```
printf("%s\n", "3.14159");
```

Let's see if we can use this to greater effect so that we can pass any `int` variable name into a macro, and have it print out it's name and value.

```
1  #include <stdio.h>
2
3  #define PRINT_INT_VAL(x) printf("%s = %d\n", #x, x)
4
5  int main(void)
6  {
7      int a = 5;
8
9      PRINT_INT_VAL(a);  // prints "a = 5"
10 }
```

On line 9, we get the following macro replacement:

```
9      printf("%s = %d\n", "a", 5);
```

19.5.5 Concatenation

We can concatenate two arguments together with `##`, as well. Fun times!

```
#define CAT(a, b) a ## b

printf("%f\n", CAT(3.14, 1592));   // 3.141592
```

19.6 Multiline Macros

It's possible to continue a macro to multiple lines if you escape the newline with a backslash (\).

Let's write a multiline macro that prints numbers from `0` to the product of the two arguments passed in.

```
1  #include <stdio.h>
2
3  #define PRINT_NUMS_TO_PRODUCT(a, b) do { \
4      int product = (a) * (b); \
5      for (int i = 0; i < product; i++) { \
6          printf("%d\n", i); \
```

```
    } \
} while(0)

int main(void)
{
    PRINT_NUMS_TO_PRODUCT(2, 4);  // Outputs numbers from 0 to 7
}
```

A couple things to note there:

- Escapes at the end of every line except the last one to indicate that the macro continues.
- The whole thing is wrapped in a `do-while(0)` loop with squirrley braces.

The latter point might be a little weird, but it's all about absorbing the trailing `;` the coder drops after the macro.

At first I thought that just using squirrely braces would be enough, but there's a case where it fails if the coder puts a semicolon after the macro. Here's that case:

```
#include <stdio.h>

#define FOO(x) { (x)++; }

int main(void)
{
    int i = 0;

    if (i == 0)
        FOO(i);
    else
        printf(":-(\n");

    printf("%d\n", i);
}
```

Looks simple enough, but it won't build without a syntax error:

```
foo.c:11:5: error: 'else' without a previous 'if'
```

Do you see it?

Let's look at the expansion:

```
    if (i == 0) {
        (i)++;
    };              // <-- Trouble with a capital-T!

    else
        printf(":-(\n");
```

The `;` puts an end to the `if` statement, so the `else` is just floating out there illegally[5].

So wrap that multiline macro with a `do-while(0)`.

[5] *Breakin' the law… breakin' the law…*

19.7 Example: An Assert Macro

Adding asserts to your code is a good way to catch conditions that you think shouldn't happen. C provides `assert()` functionality. It checks a condition, and if it's false, the program bombs out telling you the file and line number on which the assertion failed.

But this is wanting.

1. First of all, you can't specify an additional message with the assert.
2. Secondly, there's no easy on-off switch for all the asserts.

We can address the first with macros.

Basically, when I have this code:

```
ASSERT(x < 20, "x must be under 20");
```

I want something like this to happen (assuming the `ASSERT()` is on line 220 of `foo.c`):

```
if (!(x < 20)) {
    fprintf(stderr, "foo.c:220: assertion x < 20 failed: ");
    fprintf(stderr, "x must be under 20\n");
    exit(1);
}
```

We can get the filename out of the `__FILE__` macro, and the line number from `__LINE__`. The message is already a string, but `x < 20` is not, so we'll have to stringify it with `#`. We can make a multiline macro by using backslash escapes at the end of the line.

```
#define ASSERT(c, m) \
do { \
    if (!(c)) { \
        fprintf(stderr, __FILE__ ":%d: assertion %s failed: %s\n", \
                        __LINE__, #c, m); \
        exit(1); \
    } \
} while(0)
```

(It looks a little weird with `__FILE__` out front like that, but remember it is a string literal, and string literals next to each other are automagically concatenated. `__LINE__` on the other hand, it's just an `int`.)

And that works! If I run this:

```
int x = 30;

ASSERT(x < 20, "x must be under 20");
```

I get this output:

```
foo.c:23: assertion x < 20 failed: x must be under 20
```

Very nice!

The only thing left is a way to turn it on and off, and we could do that with conditional compilation.

Here's the complete example:

```
#include <stdio.h>
#include <stdlib.h>

#define ASSERT_ENABLED 1

```

```
#if ASSERT_ENABLED
#define ASSERT(c, m) \
do { \
    if (!(c)) { \
        fprintf(stderr, __FILE__ ":%d: assertion %s failed: %s\n", \
                        __LINE__, #c, m); \
        exit(1); \
    } \
} while(0)
#else
#define ASSERT(c, m)  // Empty macro if not enabled
#endif

int main(void)
{
    int x = 30;

    ASSERT(x < 20, "x must be under 20");
}
```

This has the output:

```
foo.c:23: assertion x < 20 failed: x must be under 20
```

19.8 The `#error` Directive

This directive causes the compiler to error out as soon as it sees it.

Commonly, this is used inside a conditional to prevent compilation unless some prerequisites are met:

```
#ifndef __STDC_IEC_559__
    #error I really need IEEE-754 floating point to compile. Sorry!
#endif
```

Some compilers have a non-standard complementary `#warning` directive that will output a warning but not stop compilation, but this is not in the C11 spec.

19.9 The `#pragma` Directive

This is one funky directive, short for "pragmatic". You can use it to do… well, anything your compiler supports you doing with it.

Basically the only time you're going to add this to your code is if some documentation tells you to do so.

19.9.1 Non-Standard Pragmas

Here's one non-standard example of using `#pragma` to cause the compiler to execute a `for` loop in parallel with multiple threads (if the compiler supports the OpenMP[6] extension):

```
#pragma omp parallel for
for (int i = 0; i < 10; i++) { ... }
```

There are all kinds of `#pragma` directives documented across all four corners of the globe.

All unrecognized `#pragma`s are ignored by the compiler.

[6] https://www.openmp.org/

19.9.2 Standard Pragmas

There are also a few standard ones, and these start with `STDC`, and follow the same form:

```
#pragma STDC pragma_name on-off
```

The `on-off` portion can be either `ON`, `OFF`, or `DEFAULT`.

And the `pragma_name` can be one of these:

Pragma Name	Description
`FP_CONTRACT`	Allow floating point expressions to be contracted into a single operation to avoid rounding errors that might occur from multiple operations.
`FENV_ACCESS`	Set to `ON` if you plan to access the floating point status flags. If `OFF`, the compiler might perform optimizations that cause the values in the flags to be inconsistent or invalid.
`CX_LIMITED_RANGE`	Set to `ON` to allow the compiler to skip overflow checks when performing complex arithmetic. Defaults to `OFF`.

For example:

```
#pragma STDC FP_CONTRACT OFF
#pragma STDC CX_LIMITED_RANGE ON
```

As for `CX_LIMITED_RANGE`, the spec points out:

> The purpose of the pragma is to allow the implementation to use the formulas:
>
> $(x + iy) \times (u + iv) = (xu - yv) + i(yu + xv)$
>
> $(x + iy)/(u + iv) = [(xu + yv) + i(yu - xv)]/(u^2 + v^2)$
>
> $|x + iy| = \sqrt{x^2 + y^2}$
>
> where the programmer can determine they are safe.

19.9.3 `_Pragma` Operator

This is another way to declare a pragma that you could use in a macro.

These are equivalent:

```
#pragma "Unnecessary" quotes
_Pragma("\"Unnecessary\" quotes")
```

This can be used in a macro, if need be:

```
#define PRAGMA(x) _Pragma(#x)
```

19.10 The `#line` Directive

This allows you to override the values for `__LINE__` and `__FILE__`. If you want.

I've never wanted to do this, but in K&R2, they write:

> For the benefit of other preprocessors that generate C programs […]

So maybe there's that.

To override the line number to, say 300:

```
#line 300
```

and `__LINE__` will keep counting up from there.

To override the line number and the filename:

```
#line 300 "newfilename"
```

19.11 The Null Directive

A # on a line by itself is ignored by the preprocessor. Now, to be entirely honest, I don't know what the use case is for this.

I've seen examples like this:

```
#ifdef FOO
    #
#else
    printf("Something");
#endif
```

which is just cosmetic; the line with the solitary # can be deleted with no ill effect.

Or maybe for cosmetic consistency, like this:

```
#
#ifdef FOO
    x = 2;
#endif
#
#if BAR == 17
    x = 12;
#endif
#
```

But, with respect to cosmetics, that's just ugly.

Another post mentions elimination of comments—that in GCC, a comment after a # will not be seen by the compiler. Which I don't doubt, but the specification doesn't seem to say this is standard behavior.

My searches for rationale aren't bearing much fruit. So I'm going to just say this is some good ol' fashioned C esoterica.

Chapter 20

structs II: More Fun with structs

Turns out there's a lot more you can do with `structs` than we've talked about, but it's just a big pile of miscellaneous things. So we'll throw them in this chapter.

If you're good with `struct` basics, you can round out your knowledge here.

20.1 Initializers of Nested structs and Arrays

Remember how you could initialize structure members along these lines?

```
struct foo x = {.a=12, .b=3.14};
```

Turns out we have more power in these initializers than we'd originally shared. Exciting!

For one thing, if you have a nested substructure like the following, you can initialize members of that substructure by following the variable names down the line:

```
struct foo x = {.a.b.c=12};
```

Let's look at an example:

```
#include <stdio.h>

struct cabin_information {
    int window_count;
    int o2level;
};

struct spaceship {
    char *manufacturer;
    struct cabin_information ci;
};

int main(void)
{
    struct spaceship s = {
        .manufacturer="General Products",
        .ci.window_count = 8,   // <-- NESTED INITIALIZER!
        .ci.o2level = 21
    };

```


www.dbooks.org

```
21     printf("%s: %d seats, %d%% oxygen\n",
22         s.manufacturer, s.ci.window_count, s.ci.o2level);
23 }
```

Check out lines 16-17! That's where we're initializing members of the `struct cabin_information` in the definition of `s`, our `struct spaceship`.

And here is another option for that same initializer—this time we'll do something more standard-looking, but either approach works:

```
15     struct spaceship s = {
16         .manufacturer="General Products",
17         .ci={
18             .window_count = 8,
19             .o2level = 21
20         }
21     };
```

Now, as if the above information isn't spectacular enough, we can also mix in array initializers in there, too.

Let's change this up to get an array of passenger information in there, and we can check out how the initializers work in there, too.

```
1  #include <stdio.h>
2
3  struct passenger {
4      char *name;
5      int covid_vaccinated; // Boolean
6  };
7
8  #define MAX_PASSENGERS 8
9
10 struct spaceship {
11     char *manufacturer;
12     struct passenger passenger[MAX_PASSENGERS];
13 };
14
15 int main(void)
16 {
17     struct spaceship s = {
18         .manufacturer="General Products",
19         .passenger = {
20             // Initialize a field at a time
21             [0].name = "Gridley, Lewis",
22             [0].covid_vaccinated = 0,
23
24             // Or all at once
25             [7] = {.name="Brown, Teela", .covid_vaccinated=1},
26         }
27     };
28
29     printf("Passengers for %s ship:\n", s.manufacturer);
30
31     for (int i = 0; i < MAX_PASSENGERS; i++)
32         if (s.passenger[i].name != NULL)
33             printf("    %s (%svaccinated)\n",
```

```
                s.passenger[i].name,
                s.passenger[i].covid_vaccinated? "": "not ");
}
```

20.2 Anonymous structs

These are "the struct with no name". We also mention these in the typedef section, but we'll refresh here.

Here's a regular struct:

```
struct animal {
    char *name;
    int leg_count, speed;
};
```

And here's the anonymous equivalent:

```
struct {              // <-- No name!
    char *name;
    int leg_count, speed;
};
```

Okaaaaay. So we have a struct, but it has no name, so we have no way of using it later? Seems pretty pointless.

Admittedly, in that example, it is. But we can still make use of it a couple ways.

One is rare, but since the anonymous struct represents a type, we can just put some variable names after it and use them.

```
struct {                    // <-- No name!
    char *name;
    int leg_count, speed;
} a, b, c;                  // 3 variables of this struct type

a.name = "antelope";
c.leg_count = 4;            // for example
```

But that's still not that useful.

Far more common is use of anonymous structs with a typedef so that we can use it later (e.g. to pass variables to functions).

```
typedef struct {                    // <-- No name!
    char *name;
    int leg_count, speed;
} animal;                           // New type: animal

animal a, b, c;

a.name = "antelope";
c.leg_count = 4;            // for example
```

Personally, I don't use many anonymous structs. I think it's more pleasant to see the entire struct animal before the variable name in a declaration.

But that's just, like, my opinion, man.

20.3 Self-Referential `structs`

For any graph-like data structure, it's useful to be able to have pointers to the connected nodes/vertices. But this means that in the definition of a node, you need to have a pointer to a node. It's chicken and eggy!

But it turns out you can do this in C with no problem whatsoever.

For example, here's a linked list node:

```
struct node {
    int data;
    struct node *next;
};
```

It's important to note that `next` is a pointer. This is what allows the whole thing to even build. Even though the compiler doesn't know what the entire `struct node` looks like yet, all pointers are the same size.

Here's a cheesy linked list program to test it out:

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *next;
};

int main(void)
{
    struct node *head;

    // Hackishly set up a linked list (11)->(22)->(33)
    head = malloc(sizeof(struct node));
    head->data = 11;
    head->next = malloc(sizeof(struct node));
    head->next->data = 22;
    head->next->next = malloc(sizeof(struct node));
    head->next->next->data = 33;
    head->next->next->next = NULL;

    // Traverse it
    for (struct node *cur = head; cur != NULL; cur = cur->next) {
        printf("%d\n", cur->data);
    }
}
```

Running that prints:

```
11
22
33
```

20.4 Flexible Array Members

Back in the good old days, when people carved C code out of wood, some folks thought would be neat if they could allocate `structs` that had variable length arrays at the end of them.

I want to be clear that the first part of the section is the old way of doing things, and we're going to do things the new way after that.

For example, maybe you could define a `struct` for holding strings and the length of that string. It would have a length and an array to hold the data. Maybe something like this:

```
struct len_string {
    int length;
    char data[8];
};
```

But that has `8` hardcoded as the maximum length of a string, and that's not much. What if we did something *clever* and just `malloc()`d some extra space at the end after the struct, and then let the data overflow into that space?

Let's do that, and then allocate another 40 bytes on top of it:

```
struct len_string *s = malloc(sizeof *s + 40);
```

Because `data` is the last field of the `struct`, if we overflow that field, it runs out into space that we already allocated! For this reason, this trick only works if the short array is the *last* field in the `struct`.

```
// Copy more than 8 bytes!

strcpy(s->data, "Hello, world!");  // Won't crash. Probably.
```

In fact, there was a common compiler workaround for doing this, where you'd allocate a zero length array at the end:

```
struct len_string {
    int length;
    char data[0];
};
```

And then every extra byte you allocated was ready for use in that string.

Because `data` is the last field of the `struct`, if we overflow that field, it runs out into space that we already allocated!

```
// Copy more than 8 bytes!

strcpy(s->data, "Hello, world!");  // Won't crash. Probably.
```

But, of course, actually accessing the data beyond the end of that array is undefined behavior! In these modern times, we no longer deign to resort to such savagery.

Luckily for us, we can still get the same effect with C99 and later, but now it's legal.

Let's just change our above definition to have no size for the array[1]:

```
struct len_string {
    int length;
    char data[];
};
```

Again, this only works if the flexible array member is the *last* field in the `struct`.

And then we can allocate all the space we want for those strings by `malloc()`ing larger than the `struct len_string`, as we do in this example that makes a new `struct len_string` from a C string:

[1]Technically we say that it has an *incomplete type*.

www.dbooks.org

```
struct len_string *len_string_from_c_string(char *s)
{
    int len = strlen(s);

    // Allocate "len" more bytes than we'd normally need
    struct len_string *ls = malloc(sizeof *ls + len);

    ls->length = len;

    // Copy the string into those extra bytes
    memcpy(ls->data, s, len);

    return ls;
}
```

20.5 Padding Bytes

Beware that C is allowed to add padding bytes within or after a `struct` as it sees fit. You can't trust that they will be directly adjacent in memory[2].

Let's take a look at this program. We output two numbers. One is the sum of the `sizeof`s the individual field types. The other is the `sizeof` the entire `struct`.

One would expect them to be the same. The size of the total is the size of the sum of its parts, right?

```
#include <stdio.h>

struct foo {
    int a;
    char b;
    int c;
    char d;
};

int main(void)
{
    printf("%zu\n", sizeof(int) + sizeof(char) + sizeof(int) + sizeof(char));
    printf("%zu\n", sizeof(struct foo));
}
```

But on my system, this outputs:

```
10
16
```

They're not the same! The compiler has added 6 bytes of padding to help it be more performant. Maybe you got different output with your compiler, but unless you're forcing it, you can't be sure there's no padding.

20.6 `offsetof`

In the previous section, we saw that the compiler could inject padding bytes at will inside a structure.

What if we needed to know where those were? We can measure it with `offsetof`, defined in `<stddef.h>`.

[2]Though some compilers have options to force this to occur—search for `__attribute__((packed))` to see how to do this with GCC.

Let's modify the code from above to print the offsets of the individual fields in the `struct`:

```
#include <stdio.h>
#include <stddef.h>

struct foo {
    int a;
    char b;
    int c;
    char d;
};

int main(void)
{
    printf("%zu\n", offsetof(struct foo, a));
    printf("%zu\n", offsetof(struct foo, b));
    printf("%zu\n", offsetof(struct foo, c));
    printf("%zu\n", offsetof(struct foo, d));
}
```

For me, this outputs:

```
0
4
8
12
```

indicating that we're using 4 bytes for each of the fields. It's a little weird, because `char` is only 1 byte, right? The compiler is putting 3 padding bytes after each `char` so that all the fields are 4 bytes long. Presumably this will run faster on my CPU.

20.7 Bit-Fields

In my experience, these are rarely used, but you might see them out there from time to time, especially in lower-level applications that pack bits together into larger spaces.

Let's take a look at some code to demonstrate a use case:

```
#include <stdio.h>

struct foo {
    unsigned int a;
    unsigned int b;
    unsigned int c;
    unsigned int d;
};

int main(void)
{
    printf("%zu\n", sizeof(struct foo));
}
```

For me, this prints `16`. Which makes sense, since `unsigned`s are 4 bytes on my system.

But what if we knew that all the values that were going to be stored in `a` and `b` could be stored in 5 bits, and the values in `c`, and `d` could be stored in 3 bits? That's only a total 16 bits. Why have 128 bits reserved for them if we're only going to use 16?

Well, we can tell C to pretty-please try to pack these values in. We can specify the maximum number of bits that values can take (from 1 up the size of the containing type).

We do this by putting a colon after the field name, followed by the field width in bits.

```
struct foo {
    unsigned int a:5;
    unsigned int b:5;
    unsigned int c:3;
    unsigned int d:3;
};
```

Now when I ask C how big my `struct foo` is, it tells me 4! It was 16 bytes, but now it's only 4. It has "packed" those 4 values down into 4 bytes, which is a four-fold memory savings.

The tradeoff is, of course, that the 5-bit fields can only hold values from 0-31 and the 3-bit fields can only hold values from 0-7. But life's all about compromise, after all.

20.7.1 Non-Adjacent Bit-Fields

A gotcha: C will only combine **adjacent** bit-fields. If they're interrupted by non-bit-fields, you get no savings:

```
struct foo {            // sizeof(struct foo) == 16 (for me)
    unsigned int a:1;   // since a is not adjacent to c.
    unsigned int b;
    unsigned int c:1;
    unsigned int d;
};
```

In that example, since `a` is not adjacent to `c`, they are both "packed" in their own `int`s.

So we have one `int` each for `a`, `b`, `c`, and `d`. Since my `int`s are 4 bytes, that's a grand total of 16 bytes.

A quick rearrangement yields some space savings from 16 bytes down to 12 bytes (on my system):

```
struct foo {            // sizeof(struct foo) == 12 (for me)
    unsigned int a:1;
    unsigned int c:1;
    unsigned int b;
    unsigned int d;
};
```

And now, since `a` is next to `c`, the compiler puts them together into a single `int`.

So we have one `int` for a combined `a` and `c`, and one `int` each for `b` and `d`. For a grand total of 3 `int`s, or 12 bytes.

Put all your bitfields together to get the compiler to combine them.

20.7.2 Signed or Unsigned `int`s

If you just declare a bit-field to be `int`, the different compilers will treat it as `signed` or `unsigned`. Just like the situation with `char`.

Be specific about the signedness when using bit-fields.

20.7.3 Unnamed Bit-Fields

In some specific circumstances, you might need to reserve some bits for hardware reasons, but not need to use them in code.

For example, let's say you have a byte where the top 2 bits have a meaning, the bottom 1 bit has a meaning, but the middle 5 bits do not get used by you[3].

We *could* do something like this:

```
struct foo {
    unsigned char a:2;
    unsigned char dummy:5;
    unsigned char b:1;
};
```

And that works—in our code we use `a` and `b`, but never `dummy`. It's just there to eat up 5 bits to make sure `a` and `b` are in the "required" (by this contrived example) positions within the byte.

C allows us a way to clean this up: *unnamed bit-fields*. You can just leave the name (`dummy`) out in this case, and C is perfectly happy for the same effect:

```
struct foo {
    unsigned char a:2;
    unsigned char :5;   // <-- unnamed bit-field!
    unsigned char b:1;
};
```

20.7.4 Zero-Width Unnamed Bit-Fields

Some more esoterica out here… Let's say you were packing bits into an `unsigned int`, and you needed some adjacent bit-fields to pack into the *next* `unsigned int`.

That is, if you do this:

```
struct foo {
    unsigned int a:1;
    unsigned int b:2;
    unsigned int c:3;
    unsigned int d:4;
};
```

the compiler packs all those into a single `unsigned int`. But what if you needed `a` and `b` in one `int`, and `c` and `d` in a different one?

There's a solution for that: put an unnamed bit-field of width `0` where you want the compiler to start anew with packing bits in a different `int`:

```
struct foo {
    unsigned int a:1;
    unsigned int b:2;
    unsigned int :0;   // <--Zero-width unnamed bit-field
    unsigned int c:3;
    unsigned int d:4;
};
```

It's analogous to an explicit page break in a word processor. You're telling the compiler, "Stop packing bits in this `unsigned`, and start packing them in the next one."

[3] Assuming 8-bit `chars`, i.e. `CHAR_BIT == 8`.

www.dbooks.org

By adding the zero-width unnamed bit field in that spot, the compiler puts `a` and `b` in one `unsigned int`, and `c` and `d` in another `unsigned int`. Two total, for a size of 8 bytes on my system (`unsigned ints` are 4 bytes each).

20.8 Unions

These are basically just like `structs`, except the fields overlap in memory. The `union` will be only large enough for the largest field, and you can only use one field at a time.

It's a way to reuse the same memory space for different types of data.

You declare them just like `structs`, except it's `union`. Take a look at this:

```
union foo {
    int a, b, c, d, e, f;
    float g, h;
    char i, j, k, l;
};
```

Now, that's a lot of fields. If this were a `struct`, my system would tell me it took 36 bytes to hold it all.

But it's a `union`, so all those fields overlap in the same stretch of memory. The biggest one is `int` (or `float`), taking up 4 bytes on my system. And, indeed, if I ask for the `sizeof` the `union foo`, it tells me 4!

The tradeoff is that you can only portably use one of those fields at a time. However…

20.8.1 Unions and Type Punning

You can non-portably write to one `union` field and read from another!

Doing so is called type punning[4], and you'd use it if you really knew what you were doing, typically with some kind of low-level programming.

Since the members of a union share the same memory, writing to one member necessarily affects the others. And if you read from one what was written to another, you get some weird effects.

```
#include <stdio.h>

union foo {
    float b;
    short a;
};

int main(void)
{
    union foo x;

    x.b = 3.14159;

    printf("%f\n", x.b);  // 3.14159, fair enough

    printf("%d\n", x.a);  // But what about this?
}
```

On my system, this prints out:

[4] https://en.wikipedia.org/wiki/Type_punning

```
3.141590
4048
```

because under the hood, the object representation for the float `3.14159` was the same as the object representation for the short `4048`. On my system. Your results may vary.

20.8.2 Pointers to unions

If you have a pointer to a `union`, you can cast that pointer to any of the types of the fields in that `union` and get the values out that way.

In this example, we see that the `union` has `ints` and `floats` in it. And we get pointers to the `union`, but we cast them to `int*` and `float*` types (the cast silences compiler warnings). And then if we dereference those, we see that they have the values we stored directly in the `union`.

```c
#include <stdio.h>

union foo {
    int a, b, c, d, e, f;
    float g, h;
    char i, j, k, l;
};

int main(void)
{
    union foo x;

    int *foo_int_p = (int *)&x;
    float *foo_float_p = (float *)&x;

    x.a = 12;
    printf("%d\n", x.a);           // 12
    printf("%d\n", *foo_int_p);    // 12, again

    x.g = 3.141592;
    printf("%f\n", x.g);           // 3.141592
    printf("%f\n", *foo_float_p);  // 3.141592, again
}
```

The reverse is also true. If we have a pointer to a type inside the `union`, we can cast that to a pointer to the `union` and access its members.

```c
union foo x;
int *foo_int_p = (int *)&x;             // Pointer to int field
union foo *p = (union foo *)foo_int_p;  // Back to pointer to union

p->a = 12;  // This line the same as...
x.a = 12;   // this one.
```

All this just lets you know that, under the hood, all these values in a `union` start at the same place in memory, and that's the same as where the entire `union` is.

20.8.3 Common Initial Sequences in Unions

If you have a `union` of `structs`, and all those `structs` begin with a *common initial sequence*, it's valid to access members of that sequence from any of the `union` members.

What?

Here are two `structs` with a common initial sequence:

```
struct a {
    int x;      //
    float y;    // Common initial sequence

    char *p;
};

struct b {
    int x;      //
    float y;    // Common initial sequence

    double *p;
    short z;
};
```

Do you see it? It's that they start with `int` followed by `float`—that's the common initial sequence. The members in the sequence of the `structs` have to be compatible types. And we see that with `x` and `y`, which are `int` and `float` respectively.

Now let's build a union of these:

```
union foo {
    struct a sa;
    struct b sb;
};
```

What this rule tells us is that we're guaranteed that the members of the common initial sequences are interchangeable in code. That is:

- `f.sa.x` is the same as `f.sb.x`.

and

- `f.sa.y` is the same as `f.sb.y`.

Because fields `x` and `y` are both in the common initial sequence.

Also, the names of the members in the common initial sequence don't matter—all that matters is that the types are the same.

All together, this allows us a way to safely add some shared information between `structs` in the `union`. The best example of this is probably using a field to determine the type of `struct` out of all the `structs` in the `union` that is currently "in use".

That is, if we weren't allowed this and we passed the `union` to some function, how would that function know which member of the `union` was the one it should look at?

Take a look at these `structs`. Note the common initial sequence:

```
#include <stdio.h>

struct common {
    int type;   // common initial sequence
};

struct antelope {
    int type;   // common initial sequence
```

```

    int loudness;
};

struct octopus {
    int type;   // common initial sequence

    int sea_creature;
    float intelligence;
};
```

Now let's throw them into a union:

```
union animal {
    struct common common;
    struct antelope antelope;
    struct octopus octopus;
};
```

Also, please indulge me these two #defines for the demo:

```
#define ANTELOPE 1
#define OCTOPUS  2
```

So far, nothing special has happened here. It seems like the type field is completely useless.

But now let's make a generic function that prints a union animal. It has to somehow be able to tell if it's looking at a struct antelope or a struct octopus.

Because of the magic of common initial sequences, it can look up the animal type in any of these places for a particular union animal x:

```
int type = x.common.type;    \\ or...
int type = x.antelope.type;  \\ or...
int type = x.octopus.type;
```

All those refer to the same value in memory.

And, as you might have guessed, the struct common is there so code can agnostically look at the type without mentioning a particular animal.

Let's look at the code to print a union animal:

```
void print_animal(union animal *x)
{
    switch (x->common.type) {
        case ANTELOPE:
            printf("Antelope: loudness=%d\n", x->antelope.loudness);
            break;

        case OCTOPUS:
            printf("Octopus : sea_creature=%d\n", x->octopus.sea_creature);
            printf("          intelligence=%f\n", x->octopus.intelligence);
            break;

        default:
            printf("Unknown animal type\n");
    }

```

```
45 }
46
47 int main(void)
48 {
49     union animal a = {.antelope.type=ANTELOPE, .antelope.loudness=12};
50     union animal b = {.octopus.type=OCTOPUS, .octopus.sea_creature=1,
51                                         .octopus.intelligence=12.8};
52
53     print_animal(&a);
54     print_animal(&b);
55 }
```

See how on line 29 we're just passing in the `union`—we have no idea what type of animal `struct` is in use within it.

But that's OK! Because on line 31 we check the type to see if it's an antelope or an octopus. And then we can look at the proper `struct` to get the members.

It's definitely possible to get this same effect using just `structs`, but you can do it this way if you want the memory-saving effects of a `union`.

20.9 Unions and Unnamed Structs

You know how you can have an unnamed `struct`, like this:

```
struct {
    int x, y;
} s;
```

That defines a variable `s` that is of anonymous `struct` type (because the `struct` has no name tag), with members `x` and `y`.

So things like this are valid:

```
s.x = 34;
s.y = 90;

printf("%d %d\n", s.x, s.y);
```

Turns out you can drop those unnamed `structs` in `unions` just like you might expect:

```
union foo {
    struct {        // unnamed!
        int x, y;
    } a;

    struct {        // unnamed!
        int z, w;
    } b;
};
```

And then access them as per normal:

```
union foo f;

f.a.x = 1;
f.a.y = 2;
```

```
f.b.z = 3;
f.b.w = 4;
```

No problem!

20.10 Passing and Returning structs and unions

You can pass a struct or union to a function by value (as opposed to a pointer to it)—a copy of that object to the parameter will be made as if by assignment as per usual.

You can also return a struct or union from a function and it is also passed by value back.

```
#include <stdio.h>

struct foo {
    int x, y;
};

struct foo f(void)
{
    return (struct foo){.x=34, .y=90};
}

int main(void)
{
    struct foo a = f();  // Copy is made

    printf("%d %d\n", a.x, a.y);
}
```

Fun fact: if you do this, you can use the . operator right off the function call:

```
    printf("%d %d\n", f().x, f().y);
```

(Of course that example calls the function twice, inefficiently.)

And the same holds true for returning pointers to structs and unions—just be sure to use the -> arrow operator in that case.

Chapter 21

Characters and Strings II

We've talked about how `char` types are actually just small integer types… but it's the same for a character in single quotes.

But a string in double quotes is type `const char *`.

Turns out there are few more types of strings and characters, and it leads down one of the most infamous rabbit holes in the language: the whole multibyte/wide/Unicode/localization thingy.

We're going to peer into that rabbit hole, but not go in. …Yet!

21.1 Escape Sequences

We're used to strings and characters with regular letters, punctuation, and numbers:

```
char *s = "Hello!";
char t = 'c';
```

But what if we want some special characters in there that we can't type on the keyboard because they don't exist (e.g. "€"), or even if we want a character that's a single quote? We clearly can't do this:

```
char t = '''';
```

To do these things, we use something called *escape sequences*. These are the backslash character (\) followed by another character. The two (or more) characters together have special meaning.

For our single quote character example, we can put an escape (that is, \) in front of the central single quote to solve it:

```
char t = '\'';
```

Now C knows that `\'` means just a regular quote we want to print, not the end of the character sequence.

You can say either "backslash" or "escape" in this context ("escape that quote") and C devs will know what you're talking about. Also, "escape" in this context is different than your `Esc` key or the ASCII `ESC` code.

21.1.1 Frequently-used Escapes

In my humble opinion, these escape characters make up 99.2%[1] of all escapes.

[1] I just made up that number, but it's probably not far off

Code	Description
`\n`	Newline character—when printing, continue subsequent output on the next line
`\'`	Single quote—used for a single quote character constant
`\"`	Double quote—used for a double quote in a string literal
`\\`	Backslash—used for a literal `\` in a string or character

Here are some examples of the escapes and what they output when printed.

```
printf("Use \\n for newline\n");  // Use \n for newline
printf("Say \"hello\"!\n");       // Say "hello"!
printf("%c\n", '\'');             // '
```

21.1.2 Rarely-used Escapes

But there are more escapes! You just don't see these as often.

Code	Description
`\a`	Alert. This makes the terminal make a sound or flash, or both!
`\b`	Backspace. Moves the cursor back a character. Doesn't delete the character.
`\f`	Formfeed. This moves to the next "page", but that doesn't have much modern meaning. On my system, this behaves like `\v`.
`\r`	Return. Move to the beginning of the same line.
`\t`	Horizontal tab. Moves to the next horizontal tab stop. On my machine, this lines up on columns that are multiples of 8, but YMMV.
`\v`	Vertical tab. Moves to the next vertical tab stop. On my machine, this moves to the same column on the next line.
`\?`	Literal question mark. Sometimes you need this to avoid trigraphs, as shown below.

21.1.2.1 Single Line Status Updates

A use case for `\b` or `\r` is to show status updates that appear on the same line on the screen and don't cause the display to scroll. Here's an example that does a countdown from 10. (Note this makes use of the non-standard POSIX function `sleep()` from `<unistd.h>`—if you're not on a Unix-like, search for your platform and `sleep` for the equivalent.)

```
#include <stdio.h>
#include <threads.h>

int main(void)
{
    for (int i = 10; i >= 0; i--) {
        printf("\rT minus %d second%s... \b", i, i != 1? "s": "");

        fflush(stdout);  // Force output to update

        // Sleep for 1 second
        thrd_sleep(&(struct timespec){.tv_sec=1}, NULL);
    }

    printf("\rLiftoff!              \n");
}
```

Quite a few things are happening on line 7. First of all, we lead with a `\r` to get us to the beginning of the current line, then we overwrite whatever's there with the current countdown. (There's ternary operator out there to make sure we print `1 second` instead of `1 seconds`.)

Also, there's a space after the `...` That's so that we properly overwrite the last `.` when `i` drops from `10` to `9` and we get a column narrower. Try it without the space to see what I mean.

And we wrap it up with a `\b` to back up over that space so the cursor sits at the exact end of the line in an aesthetically-pleasing way.

Note that line 14 also has a lot of spaces at the end to overwrite the characters that were already there from the countdown.

Finally, we have a weird `fflush(stdout)` in there, whatever that means. Short answer is that most terminals are *line buffered* by default, meaning they don't actually display anything until a newline character is encountered. Since we don't have a newline (we just have `\r`), without this line, the program would just sit there until `Liftoff!` and then print everything all in one instant. `fflush()` overrides this behavior and forces output to happen *right now*.

21.1.2.2 The Question Mark Escape

Why bother with this? After all, this works just fine:

```
printf("Doesn't it?\n");
```

And it works fine with the escape, too:

```
printf("Doesn't it\?\n");   // Note \?
```

So what's the point??!

Let's get more emphatic with another question mark and an exclamation point:

```
printf("Doesn't it??!\n");
```

When I compile this, I get this warning:

```
foo.c: In function ‘main’:
foo.c:5:23: warning: trigraph ??! converted to | [-Wtrigraphs]
    5 |     printf("Doesn't it??!\n");
      |
```

And running it gives this unlikely result:

```
Doesn't it|
```

So *trigraphs*? What the heck is this??!

I'm sure we'll revisit this dusty corner of the language later, but the short of it is the compiler looks for certain triplets of characters starting with `??` and it substitutes other characters in their place. So if you're on some ancient terminal without a pipe symbol (`|`) on the keyboard, you can type `??!` instead.

You can fix this by escaping the second question mark, like so:

```
printf("Doesn't it?\?!\n");
```

And then it compiles and works as-expected.

These days, of course, no one ever uses trigraphs. But that whole `??!` does sometimes appear if you decide to use it in a string for emphasis.

21.1.3 Numeric Escapes

In addition, there are ways to specify numeric constants or other character values inside strings or character constants.

If you know an octal or hexadecimal representation of a byte, you can include that in a string or character constant.

The following table has example numbers, but any hex or octal numbers may be used. Pad with leading zeros if necessary to read the proper digit count.

Code	Description
`\123`	Embed the byte with octal value `123`, 3 digits exactly.
`\x4D`	Embed the byte with hex value `4D`, 2 digits.
`\u2620`	Embed the Unicode character at code point with hex value `2620`, 4 digits.
`\U0001243F`	Embed the Unicode character at code point with hex value `1243F`, 8 digits.

Here's an example of the less-commonly used octal notation to represent the letter `B` in between `A` and `C`. Normally this would be used for some kind of special unprintable character, but we have other ways to do that, below, and this is just an octal demo:

```
printf("A\102C\n");  // 102 is `B` in ASCII/UTF-8
```

Note there's no leading zero on the octal number when you include it this way. But it does need to be three characters, so pad with leading zeros if you need to.

But far more common is to use hex constants these days. Here's a demo that you shouldn't use, but it demos embedding the UTF-8 bytes 0xE2, 0x80, and 0xA2 in a string, which corresponds to the Unicode "bullet" character (•).

```
printf("\xE2\x80\xA2 Bullet 1\n");
printf("\xE2\x80\xA2 Bullet 2\n");
printf("\xE2\x80\xA2 Bullet 3\n");
```

Produces the following output if you're on a UTF-8 console (or probably garbage if you're not):

```
• Bullet 1
• Bullet 2
• Bullet 3
```

But that's a crummy way to do Unicode. You can use the escapes `\u` (16-bit) or `\U` (32-bit) to just refer to Unicode by code point number. The bullet is `2022` (hex) in Unicode, so you can do this and get more portable results:

```
printf("\u2022 Bullet 1\n");
printf("\u2022 Bullet 2\n");
printf("\u2022 Bullet 3\n");
```

Be sure to pad `\u` with enough leading zeros to get to four characters, and `\U` with enough to get to eight.

For example, that bullet could be done with `\U` and four leading zeros:

```
printf("\U00002022 Bullet 1\n");
```

But who has time to be that verbose?

www.dbooks.org

Chapter 22

Enumerated Types: `enum`

C offers us another way to have constant integer values by name: `enum`.

For example:

```
enum {
  ONE=1,
  TWO=2
};

printf("%d %d", ONE, TWO);  // 1 2
```

In some ways, it can be better—or different—than using a `#define`. Key differences:

- `enum`s can only be integer types.
- `#define` can define anything at all.
- `enum`s are often shown by their symbolic identifier name in a debugger.
- `#define`d numbers just show as raw numbers which are harder to know the meaning of while debugging.

Since they're integer types, they can be used any place integers can be used, including in array dimensions and `case` statements.

Let's tear into this more.

22.1 Behavior of `enum`

22.1.1 Numbering

`enum`s are automatically numbered unless you override them.

They start at `0`, and autoincrement up from there, by default:

```
enum {
    SHEEP,  // Value is 0
    WHEAT,  // Value is 1
    WOOD,   // Value is 2
    BRICK,  // Value is 3
    ORE     // Value is 4
};

printf("%d %d\n", SHEEP, BRICK);  // 0 3
```

You can force particular integer values, as we saw earlier:

```
enum {
  X=2,
  Y=18,
  Z=-2
};
```

Duplicates are not a problem:

```
enum {
  X=2,
  Y=2,
  Z=2
};
```

if values are omitted, numbering continues counting in the positive direction from whichever value was last specified. For example:

```
enum {
  A,    // 0, default starting value
  B,    // 1
  C=4,  // 4, manually set
  D,    // 5
  E,    // 6
  F=3   // 3, manually set
  G,    // 4
  H     // 5
}
```

22.1.2 Trailing Commas

This is perfectly fine, if that's your style:

```
enum {
  X=2,
  Y=18,
  Z=-2,   // <-- Trailing comma
};
```

It's gotten more popular in languages of the recent decades so you might be pleased to see it.

22.1.3 Scope

`enum`s scope as you'd expect. If at file scope, the whole file can see it. If in a block, it's local to that block.

It's really common for `enum`s to be defined in header files so they can be `#include`d at file scope.

22.1.4 Style

As you've noticed, it's common to declare the `enum` symbols in uppercase (with underscores).

This isn't a requirement, but is a very, very common idiom.

22.2 Your `enum` is a Type

This is an important thing to know about `enum`: they're a type, analogous to how a `struct` is a type.

You can give them a tag name so you can refer to the type later and declare variables of that type.

Now, since `enum`s are integer types, why not just use `int`?

In C, the best reason for this is code clarity–it's a nice, typed way to describe your thinking in code. C (unlike C++) doesn't actually enforce any values being in range for a particular `enum`.

Let's do an example where we declare a variable `r` of type `enum resource` that can hold those values:

```
// Named enum, type is "enum resource"

enum resource {
    SHEEP,
    WHEAT,
    WOOD,
    BRICK,
    ORE
};

// Declare a variable "r" of type "enum resource"

enum resource r = BRICK;

if (r == BRICK) {
    printf("I'll trade you a brick for two sheep.\n");
}
```

You can also `typedef` these, of course, though I personally don't like to.

```
typedef enum {
    SHEEP,
    WHEAT,
    WOOD,
    BRICK,
    ORE
} RESOURCE;

RESOURCE r = BRICK;
```

Another shortcut that's legal but rare is to declare variables when you declare the `enum`:

```
// Declare an enum and some initialized variables of that type:

enum {
    SHEEP,
    WHEAT,
    WOOD,
    BRICK,
    ORE
} r = BRICK, s = WOOD;
```

You can also give the `enum` a name so you can use it later, which is probably what you want to do in most cases:

```
// Declare an enum and some initialized variables of that type:

enum resource {   // <-- type is now "enum resource"
    SHEEP,
```

```
    WHEAT,
    WOOD,
    BRICK,
    ORE
} r = BRICK, s = WOOD;
```

In short, enums are a great way to write nice, scoped, typed, clean code.

www.dbooks.org

Chapter 23

Pointers III: Pointers to Pointers and More

Here's where we cover some intermediate and advanced pointer usage. If you don't have pointers down well, review the previous chapters on pointers and pointer arithmetic before starting on this stuff.

23.1 Pointers to Pointers

If you can have a pointer to a variable, and a variable can be a pointer, can you have a pointer to a variable that it itself a pointer?

Yes! This is a pointer to a pointer, and it's held in variable of type pointer-pointer.

Before we tear into that, I want to try for a *gut feel* for how pointers to pointers work.

Remember that a pointer is just a number. It's a number that represents an index in computer memory, typically one that holds a value we're interested in for some reason.

That pointer, which is a number, has to be stored somewhere. And that place is memory, just like everything else[1].

But because it's stored in memory, it must have an index it's stored at, right? The pointer must have an index in memory where it is stored. And that index is a number. It's the address of the pointer. It's a pointer to the pointer.

Let's start with a regular pointer to an `int`, back from the earlier chapters:

```c
#include <stdio.h>

int main(void)
{
    int x = 3490;  // Type: int
    int *p = &x;   // Type: pointer to an int

    printf("%d\n", *p);  // 3490
}
```

[1]There's some devil in the details with values that are stored in registers only, but we can safely ignore that for our purposes here. Also the C spec makes no stance on these "register" things beyond the `register` keyword, the description for which doesn't mention registers.

Straightforward enough, right? We have two types represented: `int` and `int*`, and we set up `p` to point to `x`. Then we can dereference `p` on line 8 and print out the value `3490`.

But, like we said, we can have a pointer to any variable… so does that mean we can have a pointer to `p`?

In other words, what type is this expression?

```
int x = 3490;  // Type: int
int *p = &x;   // Type: pointer to an int

&p  // <-- What type is the address of p? AKA a pointer to p?
```

If `x` is an `int`, then `&x` is a pointer to an `int` that we've stored in `p` which is type `int*`. Follow? (Repeat this paragraph until you do!)

And therefore `&p` is a pointer to an `int*`, AKA a "pointer to a pointer to an `int`". AKA "`int`-pointer-pointer".

Got it? (Repeat the previous paragraph until you do!)

We write this type with two asterisks: `int **`. Let's see it in action.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int x = 3490;  // Type: int
6      int *p = &x;   // Type: pointer to an int
7      int **q = &p;  // Type: pointer to pointer to int
8
9      printf("%d %d\n", *p, **q);  // 3490 3490
10 }
```

Let's make up some pretend addresses for the above values as examples and see what these three variables might look like in memory. The address values, below are just made up by me for example purposes:

Variable	Stored at Address	Value Stored There
`x`	`28350`	`3490`—the value from the code
`p`	`29122`	`28350`—the address of `x`!
`q`	`30840`	`29122`—the address of `p`!

Indeed, let's try it for real on my computer[2] and print out the pointer values with `%p` and I'll do the same table again with actual references (printed in hex).

Variable	Stored at Address	Value Stored There
`x`	`0x7ffd96a07b94`	`3490`—the value from the code
`p`	`0x7ffd96a07b98`	`0x7ffd96a07b94`—the address of `x`!
`q`	`0x7ffd96a07ba0`	`0x7ffd96a07b98`—the address of `p`!

You can see those addresses are the same except the last byte, so just focus on those.

On my system, `int`s are 4 bytes, which is why we're seeing the address go up by 4 from `x` to `p`[3] and then goes up by 8 from `p` to `q`. On my system, all pointers are 8 bytes.

[2] You're very likely to get different numbers on yours.

[3] There is absolutely nothing in the spec that says this will always work this way, but it happens to work this way on my system.

Does it matter if it's an `int*` or an `int**`? Is one more bytes than the other? Nope! Remember that all pointers are addresses, that is indexes into memory. And on my machine you can represent an index with 8 bytes… doesn't matter what's stored at that index.

Now check out what we did there on line 9 of the previous example: we *double dereferenced* `q` to get back to our `3490`.

This is the important bit about pointers and pointers to pointers:

- You can get a pointer to anything with `&` (including to a pointer!)
- You can get the thing a pointer points to with `*` (including a pointer!)

So you can think of `&` as being used to make pointers, and `*` being the inverse—it goes the opposite direction of `&`—to get to the thing pointed to.

In terms of type, each time you `&`, that adds another pointer level to the type.

If you have	Then you run	The result type is
`int x`	`&x`	`int *`
`int *x`	`&x`	`int **`
`int **x`	`&x`	`int ***`
`int ***x`	`&x`	`int ****`

And each time you use dereference (`*`), it does the opposite:

If you have	Then you run	The result type is
`int ****x`	`*x`	`int ***`
`int ***x`	`*x`	`int **`
`int **x`	`*x`	`int *`
`int *x`	`*x`	`int`

Note that you can use multiple `*`s in a row to quickly dereference, just like we saw in the example code with `**q`, above. Each one strips away one level of indirection.

If you have	Then you run	The result type is
`int ****x`	`***x`	`int *`
`int ***x`	`**x`	`int *`
`int **x`	`**x`	`int`

In general, `&*E == E`[4]. The dereference "undoes" the address-of.

But `&` doesn't work the same way—you can only do those one at a time, and have to store the result in an intermediate variable:

```
int x = 3490;      // Type: int
int *p = &x;       // Type: int *, pointer to an int
int **q = &p;      // Type: int **, pointer to pointer to int
int ***r = &q;     // Type: int ***, pointer to pointer to pointer to int
int ****s = &r;    // Type: int ****, you get the idea
int *****t = &s;   // Type: int *****
```

[4]Even if `E` is `NULL`, it turns out, weirdly.

23.1.1 Pointer Pointers and `const`

If you recall, declaring a pointer like this:

```
int *const p;
```

means that you can't modify `p`. Trying to `p++` would give you a compile-time error.

But how does that work with `int **` or `int ***`? Where does the `const` go, and what does it mean?

Let's start with the simple bit. The `const` right next to the variable name refers to that variable. So if you want an `int***` that you can't change, you can do this:

```
int ***const p;

p++;  // Not allowed
```

But here's where things get a little weird.

What if we had this situation:

```
int main(void)
{
    int x = 3490;
    int *const p = &x;
    int **q = &p;
}
```

When I build that, I get a warning:

```
warning: initialization discards 'const' qualifier from pointer target type
    7 |     int **q = &p;
      |               ^
```

What's going on? The

That is, we're saying that `q` is type `int **`, and if you dereference that, the rightmost `*` in the type goes away. So after the dereference, we have type `int *`.

And we're assigning `&p` into it which is *a pointer to* an `int *const`, or, in other words, `int *const *`.

But `q` is `int **`! A type with different `const`ness on the first `*`! So we get a warning that the `const` in `p`'s `int *const *` is being ignored and thrown away.

We can fix that by making sure `q`'s type is at least as `const` as `p`.

```
int x = 3490;
int *const p = &x;
int *const *q = &p;
```

And now we're happy.

We could make `q` even more `const`. As it is, above, we're saying, "`q` isn't itself `const`, but the thing it points to is `const`." But we could make them both `const`:

```
int x = 3490;
int *const p = &x;
int *const *const q = &p;  // More const!
```

And that works, too. Now we can't modify `q`, or the pointer `q` points to.

23.2 Multibyte Values

We kinda hinted at this in a variety of places earlier, but clearly not every value can be stored in a single byte of memory. Things take up multiple bytes of memory (assuming they're not `char`s). You can tell how many bytes by using `sizeof`. And you can tell which address in memory is the *first* byte of the object by using the standard `&` operator, which always returns the address of the first byte.

And here's another fun fact! If you iterate over the bytes of any object, you get its *object representation*. Two things with the same object representation in memory are equal.

If you want to iterate over the object representation, you should do it with pointers to `unsigned char`.

Let's make our own version of `memcpy()` that does exactly this:

```
void *my_memcpy(void *dest, const void *src, size_t n)
{
    // Make local variables for src and dest, but of type unsigned char

    const unsigned char *s = src;
    unsigned char *d = dest;

    while (n-- > 0)   // For the given number of bytes
        *d++ = *s++;  // Copy source byte to dest byte

    // Most copy functions return a pointer to the dest as a convenience
    // to the caller

    return dest;
}
```

(There are some good examples of post-increment and post-decrement in there for you to study, as well.)

It's important to note that the version, above, is probably less efficient than the one that comes with your system.

But you can pass pointers to anything into it, and it'll copy those objects. Could be `int*`, `struct animal*`, or anything.

Let's do another example that prints out the object representation bytes of a `struct` so we can see if there's any padding in there and what values it has[5].

```
#include <stdio.h>

struct foo {
    char a;
    int b;
};

int main(void)
{
    struct foo x = {0x12, 0x12345678};
    unsigned char *p = (unsigned char *)&x;

    for (size_t i = 0; i < sizeof x; i++) {
        printf("%02X\n", p[i]);
    }
}
```

[5]Your C compiler is not required to add padding bytes, and the values of any padding bytes that are added are indeterminate.

What we have there is a `struct foo` that's built in such a way that should encourage a compiler to inject padding bytes (though it doesn't have to). And then we get an `unsigned char *` to the first byte of the `struct foo` variable `x`.

From there, all we need to know is the `sizeof x` and we can loop through that many bytes, printing out the values (in hex for ease).

Running this gives a bunch of numbers as output. I've annotated it below to identify where the values were stored:

```
12  | x.a == 0x12

AB  |
BF  | padding bytes with "random" value
26  |

78  |
56  | x.b == 0x12345678
34  |
12  |
```

On all systems, `sizeof(char)` is 1, and we see that first byte at the top of the output holding the value `0x12` that we stored there.

Then we have some padding bytes—for me, these varied from run to run.

Finally, on my system, `sizeof(int)` is 4, and we can see those 4 bytes at the end. Notice how they're the same bytes as are in the hex value `0x12345678`, but strangely in reverse order[6].

So that's a little peek under the hood at the bytes of a more complex entity in memory.

23.3 The `NULL` Pointer and Zero

These things can be used interchangeably:

- `NULL`
- `0`
- `'\0'`
- `(void *)0`

Personally, I always use `NULL` when I mean `NULL`, but you might see some other variants from time to time. Though `'\0'` (a byte with all bits set to zero) will also compare equal, it's *weird* to compare it to a pointer; you should compare `NULL` against the pointer. (Of course, lots of times in string processing, you're comparing *the thing the pointer points to* to `'\0'`, and that's right.)

`0` is called the *null pointer constant*, and, when compared to or assigned into another pointer, it is converted to a null pointer of the same type.

23.4 Pointers as Integers

You can cast pointers to integers and vice-versa (since a pointer is just an index into memory), but you probably only ever need to do this if you're doing some low-level hardware stuff. The results of such machinations are implementation-defined, so they aren't portable. And *weird things* could happen.

C does make one guarantee, though: you can convert a pointer to a `uintptr_t` type and you'll be able to convert it back to a pointer without losing any data.

[6] This will vary depending on the architecture, but my system is *little endian*, which means the least-significant byte of the number is stored first. *Big endian* systems will have the `12` first and the `78` last. But the spec doesn't dictate anything about this representation.

www.dbooks.org

`uintptr_t` is defined in `<stdint.h>`[7].

Additionally, if you feel like being signed, you can use `intptr_t` to the same effect.

23.5 Casting Pointers to other Pointers

There's only one safe pointer conversion:

1. Converting to `intptr_t` or `uintptr_t`.
2. Converting to and from `void*`.

TWO! Two safe pointer conversions.

3. Converting to and from `char*`.

THREE! Three safe conversions!

If you cast to a pointer of another type and then access the object it points to, the behavior is undefined due to something called *strict aliasing*.

Plain old *aliasing* refers to the ability to have more than one way to access the same object. The access points are aliases for each other.

Strict aliasing says you are only allowed to access an object via pointers to *compatible types*.

For example, this is definitely allowed:

```
int a = 1;
int *p = &a;
```

`p` is a pointer to an `int`, and it points to a compatible type—namely `int`—so we're golden.

But the following isn't good because `int` and `float` are not compatible types:

```
int a = 1;
float *p = (float *)&a;
```

Here's a demo program that does some aliasing. It takes a variable `v` of type `int32_t` and aliases it to a pointer to a `struct words`. That `struct` has two `int16_t`s in it. These types are incompatible, so we're in violation of strict aliasing rules.

Let's see if we can break something.

```
#include <stdio.h>
#include <stdint.h>

struct words {
    int16_t v[2];
};

void fun(int32_t *pv, struct words *pw)
{
    for (int i = 0; i < 5; i++) {
        (*pv)++;

        // Print the 32-bit value and the 16-bit values:

        printf("%x, %x-%x\n", *pv, pw->v[1], pw->v[0]);
    }
```

[7] It's an optional feature, so it might not be there—but it probably is.

```
}

int main(void)
{
    int32_t v = 0x12345678;

    struct words *pw = (struct words *)&v;  // Violates strict aliasing

    fun(&v, pw);
}
```

See how I pass in the two incompatible pointers to `fun()`? One of the types is `int32_t*` and the other is `struct words*`.

But they both point to the same object: the 32-bit value initialized to `0x12345678`.

So if we look at the fields in the `struct words`, we should see the two 16-bit halves of that number. Right?

And in the `fun()` loop, we increment the pointer to the `int32_t`. That's it. But since the `struct` points to that same memory, it, too, should be updated to the same value.

So let's run it and get this, with the 32-bit value on the left and the two 16-bit portions on the right. It should match[8]:

```
12345679, 1234-5679
1234567a, 1234-567a
1234567b, 1234-567b
1234567c, 1234-567c
1234567d, 1234-567d
```

and it does… *UNTIL TOMORROW!*

Let's try it compiling GCC with `-O3` and `-fstrict-aliasing`:

```
12345679, 1234-5678
1234567a, 1234-5679
1234567b, 1234-567a
1234567c, 1234-567b
1234567d, 1234-567c
```

They're off by one! But they point to the same memory! How could this be? Answer: it's undefined behavior to alias memory like that. *Anything is possible*, except not in a good way.

If your code violates strict aliasing rules, whether it works or not depends on how someone decides to compile it. And that's a bummer since that's beyond your control. Unless you're some kind of omnipotent deity.

Unlikely, sorry.

Speaking of no fun, these rules also break an age-old tradition in C of "subclassing" `structs` that have a common initial sequence. Folks used to do this with abandon:

```
struct base_class {
    int a, b;    // Common elements
};

struct subclass_a {
    struct base_class super;  // inheritance
```

[8] I'm printing out the 16-bit values reversed since I'm on a little-endian machine at it makes it easier to read here.

www.dbooks.org

```
    int x, y, z; // specialization
};

struct subclass_b {
    struct base_class super;  // inheritance

    float f, g, h; // specialization
};
```

And if you had some of these subclasses like this:

```
struct subclass_a sa = {.super.a=1, .super.b=2};
struct subclass_b sb = {.super.a=3, .super.b=4};
```

And a function like this, taking a `void*` argument:

```
void print_info(void *vp)
{
    struct base_class *p = vp;

    printf("%d %d\n", p->a, p->b);
}
```

We know the `structs` would have the same initial memory layout as the base class `structs`, owing to that mix-in being the first member declared.

People then felt empowered to call `print_info()` with pointers to either subclass `struct` type.

```
print_info(&sa);
print_info(&sb);
```

But this breaks the strict aliasing rules since `p` in `print_info()` is a pointer to a different type than the actual objects.

So don't do this unless you're OK being non-portable and are sure your compiler isn't relying on the strict aliasing rules.

GCC can be forced to not use the strict aliasing rules with `-fno-strict-aliasing`. Compiling the demo program, above, with `-O3` and this flag causes the output to be as expected.

Lastly, *type punning* is using pointers of different types to look at the same data. Before strict aliasing, this kind of things was fairly common:

```
int a = 0x12345678;
short b = *((short *)&a);   // Violates strict aliasing
```

If you want to do type punning (relatively) safely, see the section on Unions and Type Punning.

23.6 Pointer Differences

As you know from the section on pointer arithmetic, you can subtract one pointer from another[9] to get the difference between them in count of array elements.

Now the *type of that difference* is something that's up to the implementation, so it could vary from system to system.

To be more portable, you can store the result in a variable of type `ptrdiff_t` defined in `<stddef.h>`.

[9] Assuming they point to the same array object.

```
int cats[100];

int *f = cats + 20;
int *g = cats + 60;

ptrdiff_t d = g - f;  // difference is 40
```

And you can print it by prefixing the integer format specifier with `t`:

```
printf("%td\n", d);  // Print decimal: 40
printf("%tX\n", d);  // Print hex:     28
```

23.7 Pointers to Functions

Functions are just collections of machine instructions in memory, so there's no reason we can't get a pointer to the first instruction of the function.

And then call it.

This can be useful for passing a pointer to a function into another function as an argument. Then the second one could call whatever was passed in.

The tricky part with these, though, is that C needs to know the type of the variable that is the pointer to the function.

And it would really like to know all the details.

Like "this is a pointer to a function that takes two `int` arguments and returns `void`".

How do you write all that down so you can declare a variable?

Well, it turns out it looks very much like a function prototype, except with some extra parentheses:

```
// Declare p to be a pointer to a function.
// This function returns a float, and takes two ints as arguments.

float (*p)(int, int);
```

Also notice that you don't have to give the parameters names. But you can if you want; they're just ignored.

```
// Declare p to be a pointer to a function.
// This function returns a float, and takes two ints as arguments.

float (*p)(int a, int b);
```

So now that we know how to declare a variable, how do we know what to assign into it? How do we get the address of a function?

Turns out there's a shortcut just like with getting a pointer to an array: you can just refer to the bare function name without parens. (You can put an `&` in front of this if you like, but it's unnecessary and not idiomatic.)

Once you have a pointer to a function, you can call it just by adding parens and an argument list.

Let's do a simple example where I effectively make an alias for a function by setting a pointer to it. Then we'll call it.

This code prints out `3490`:

```
#include <stdio.h>

void print_int(int n)
```

```
{
    printf("%d\n", n);
}

int main(void)
{
    // Assign p to point to print_int:

    void (*p)(int) = print_int;

    p(3490);          // Call print_int via the pointer
}
```

Notice how the type of `p` represents the return value and parameter types of `print_int`. It has to, or else C will complain about incompatible pointer types.

One more example here shows how we might pass a pointer to a function as an argument to another function.

We'll write a function that takes a couple integer arguments, plus a pointer to a function that operates on those two arguments. Then it prints the result.

```
1  #include <stdio.h>
2
3  int add(int a, int b)
4  {
5      return a + b;
6  }
7
8  int mult(int a, int b)
9  {
10     return a * b;
11 }
12
13 void print_math(int (*op)(int, int), int x, int y)
14 {
15     int result = op(x, y);
16
17     printf("%d\n", result);
18 }
19
20 int main(void)
21 {
22     print_math(add, 5, 7);   // 12
23     print_math(mult, 5, 7);  // 35
24 }
```

Take a moment to digest that. The idea here is that we're going to pass a pointer to a function to `print_math()`, and it's going to call that function to do some math.

This way we can change the behavior of `print_math()` by passing another function into it. You can see we do that on lines 22-23 when we pass in pointers to functions `add` and `mult`, respectively.

Now, on line 13, I think we can all agree the function signature of `print_math()` is a sight to behold. And, if you can believe it, this one is actually pretty straight-forward compared to some things you can construct[10].

But let's digest it. Turns out there are only three parameters, but they're a little hard to see:

[10] The Go Programming Language drew its type declaration syntax inspiration from the opposite of what C does.

```
//                        op              x       y
//               |-----------------|  |---|   |---|
void print_math(int (*op)(int, int), int x, int y)
```

The first, `op`, is a pointer to a function that takes two `int`s as arguments and returns an `int`. This matches the signatures for both `add()` and `mult()`.

The second and third, `x` and `y`, are just standard `int` parameters.

Slowly and deliberately let your eyes play over the signature while you identify the working parts. One thing that always stands out for me is the sequence `(*op)(`, the parens and the asterisk. That's the giveaway it's a pointer to a function.

Finally, jump back to the *Pointers II* chapter for a pointer-to-function example using the built-in `qsort()`.

Chapter 24

Bitwise Operations

These numeric operations effectively allow you to manipulate individual bits in variables, fitting since C is such a low-level langauge[1].

If you're not familiar with bitwise operations, Wikipedia has a good bitwise article[2].

24.1 Bitwise AND, OR, XOR, and NOT

For each of these, the usual arithmetic conversions take place on the operands (which in this case must be an integer type), and then the appropriate bitwise operation is performed.

Operation	Operator	Example
AND	`&`	`a = b & c`
OR	`|`	`a = b | c`
XOR	`^`	`a = b ^ c`
NOT	`~`	`a = ~c`

Note how they're similar to the Boolean operators `&&` and `||`.

These have assignment shorthand variants similar to `+=` and `-=`:

Operator	Example	Longhand equivalent
`&=`	`a &= c`	`a = a & c`
`|=`	`a |= c`	`a = a | c`
`^=`	`a ^= c`	`a = a ^ c`

24.2 Bitwise Shift

For these, the integer promotions are performed on each operand (which must be an integer type) and then a bitwise shift is executed. The type of the result is the type of the promoted left operand.

New bits are filled with zeros, with a possible exception noted in the implementation-defined behavior, below.

[1] Not that other languages don't do this—they do. It is interesting how many modern languages use the same operators for bitwise that C does.

[2] https://en.wikipedia.org/wiki/Bitwise_operation

Operation	Operator	Example
Shift left	`<<`	`a = b << c`
Shift right	`>>`	`a = b >> c`

There's also the same similar shorthand for shifting:

Operator	Example	Longhand equivalent
`>>=`	`a >>= c`	`a = a >> c`
`<<=`	`a <<= c`	`a = a << c`

Watch for undefined behavior: no negative shifts, and no shifts that are larger than the size of the promoted left operand.

Also watch for implementation-defined behavior: if you right-shift a negative number, the results are implementation-defined. (It's perfectly fine to right-shift a signed `int`, just make sure it's positive.)

www.dbooks.org

Chapter 25

Variadic Functions

Variadic is a fancy word for functions that take arbitrary numbers of arguments.

A regular function takes a specific number of arguments, for example:

```
int add(int x, int y)
{
    return x + y;
}
```

You can only call that with exactly two arguments which correspond to parameters `x` and `y`.

```
add(2, 3);
add(5, 12);
```

But if you try it with more, the compiler won't let you:

```
add(2, 3, 4);  // ERROR
add(5);        // ERROR
```

Variadic functions get around this limitation to a certain extent.

We've already seen a famous example in `printf()`! You can pass all kinds of things to it.

```
printf("Hello, world!\n");
printf("The number is %d\n", 2);
printf("The number is %d and pi is %f\n", 2, 3.14159);
```

It seems to not care how many arguments you give it!

Well, that's not entirely true. Zero arguments will give you an error:

```
printf();  // ERROR
```

This leads us to one of the limitations of variadic functions in C: they must have at least one argument.

But aside from that, they're pretty flexible, even allows arguments to have different types just like `printf()` does.

Let's see how they work!

25.1 Ellipses in Function Signatures

So how does it work, syntactically?

What you do is put all the arguments that *must* be passed first (and remember there has to be at least one) and after that, you put `...`. Just like this:

```
void func(int a, ...)   // Literally 3 dots here
```

Here's some code to demo that:

```
#include <stdio.h>

void func(int a, ...)
{
    printf("a is %d\n", a);  // Prints "a is 2"
}

int main(void)
{
    func(2, 3, 4, 5, 6);
}
```

So, great, we can get that first argument that's in variable `a`, but what about the rest of the arguments? How do you get to them?

Here's where the fun begins!

25.2 Getting the Extra Arguments

You're going to want to include `<stdarg.h>` to make any of this work.

First things first, we're going to use a special variable of type `va_list` (variable argument list) to keep track of which variable we're accessing at a time.

The idea is that we first start processing arguments with a call to `va_start()`, process each argument in turn with `va_arg()`, and then, when done, wrap it up with `va_end()`.

When you call `va_start()`, you need to pass in the *last named parameter* (the one just before the `...`) so it knows where to start looking for the additional arguments.

And when you call `va_arg()` to get the next argument, you have to tell it the type of argument to get next.

Here's a demo that adds together an arbitrary number of integers. The first argument is the number of integers to add together. We'll make use of that to figure out how many times we have to call `va_arg()`.

```
#include <stdio.h>
#include <stdarg.h>

int add(int count, ...)
{
    int total = 0;
    va_list va;

    va_start(va, count);   // Start with arguments after "count"

    for (int i = 0; i < count; i++) {
        int n = va_arg(va, int);   // Get the next int

        total += n;
    }

```

```
    va_end(va);  // All done

    return total;
}

int main(void)
{
    printf("%d\n", add(4, 6, 2, -4, 17));  // 6 + 2 - 4 + 17 = 21
    printf("%d\n", add(2, 22, 44));        // 22 + 44 = 66
}
```

When `printf()` is called, it uses the number of `%d`s (or whatever) in the format string to know how many more arguments there are!

If the syntax of `va_arg()` is looking strange to you (because of that loose type name floating around in there), you're not alone. These are implemented with preprocessor macros in order to get all the proper magic in there.

25.3 `va_list` Functionality

What is that `va_list` variable we're using up there? It's an opaque variable[1] that holds information about which argument we're going to get next with `va_arg()`. You see how we just call `va_arg()` over and over? The `va_list` variable is a placeholder that's keeping track of progress so far.

But we have to initialize that variable to some sensible value. That's where `va_start()` comes into play.

When we called `va_start(va, count)`, above, we were saying, "Initialize the `va` variable to point to the variable argument *immediately after* `count`."

And that's *why* we need to have at least one named variable in our argument list[2].

Once you have that pointer to the initial parameter, you can easily get subsequent argument values by calling `va_arg()` repeatedly. When you do, you have to pass in your `va_list` variable (so it can keep on keeping track of where you are), as well as the type of argument you're about to copy off.

It's up to you as a programmer to figure out which type you're going to pass to `va_arg()`. In the above example, we just did `int`s. But in the case of `printf()`, it uses the format specifier to determine which type to pull off next.

And when you're done, call `va_end()` to wrap it up. You **must** (the spec says) call this on a particular `va_list` variable before you decide to call either `va_start()` or `va_copy()` on it again. I know we haven't talked about `va_copy()` yet.

So the standard progression is:

- `va_start()` to initialize your `va_list` variable
- Repeatedly `va_arg()` to get the values
- `va_end()` to deinitialize your `va_list` variable

I also mentioned `va_copy()` up there; it makes a copy of your `va_list` variable in the exact same state. That is, if you haven't started with `va_arg()` with the source variable, the new one won't be started, either. If you've consumed 5 variables with `va_arg()` so far, the copy will also reflect that.

[1] That is, us lowly developers aren't supposed to know what's in there or what it means. The spec doesn't dictate what it is in detail.

[2] Honestly, it would be possible to remove that limitation from the language, but the idea is that the macros `va_start()`, `va_arg()`, and `va_end()` should be able to be written in C. And to make that happen, we need some way to initialize a pointer to the location of the first parameter. And to do that, we need the *name* of the first parameter. It would require a language extension to make this possible, and so far the committee hasn't found a rationale for doing so.

va_copy() can be useful if you need to scan ahead through the arguments but need to also remember your current place.

25.4 Library Functions That Use va_lists

One of the other uses for these is pretty cool: writing your own custom printf() variant. It would be a pain to have to handle all those format specifiers right? All zillion of them?

Luckily, there are printf() variants that accept a working va_list as an argument. You can use these to wrap up and make your own custom printf()s!

These functions start with the letter v, such as vprintf(), vfprintf(), vsprintf(), and vsnprintf(). Basically all your printf() golden oldies except with a v in front.

Let's make a function my_printf() that works just like printf() except it takes an extra argument up front.

```
1  #include <stdio.h>
2  #include <stdarg.h>
3
4  int my_printf(int serial, const char *format, ...)
5  {
6      va_list va;
7
8      // Do my custom work
9      printf("The serial number is: %d\n", serial);
10
11     // Then pass the rest off to vprintf()
12     va_start(va, format);
13     int rv = vprintf(format, va);
14     va_end(va);
15
16     return rv;
17 }
18
19 int main(void)
20 {
21     int x = 10;
22     float y = 3.2;
23
24     my_printf(3490, "x is %d, y is %f\n", x, y);
25 }
```

See what we did there? On lines 12-14 we started a new va_list variable, and then just passed it right into vprintf(). And it knows just want to do with it, because it has all the printf() smarts built-in.

We still have to call va_end() when we're done, though, so don't forget that!

Chapter 26

Locale and Internationalization

Localization is the process of making your app ready to work well in different locales (or countries).

As you might know, not everyone uses the same character for decimal points or for thousands separators… or for currency.

These locales have names, and you can select one to use. For example, a US locale might write a number like:

100,000.00

Whereas in Brazil, the same might be written with the commas and decimal points swapped:

100.000,00

Makes it easier to write your code so it ports to other nationalities with ease!

Well, sort of. Turns out C only has one built-in locale, and it's limited. The spec really leaves a lot of ambiguity here; it's hard to be completely portable.

But we'll do our best!

26.1 Setting the Localization, Quick and Dirty

For these calls, include `<locale.h>`.

There is basically one thing you can portably do here in terms of declaring a specific locale. This is likely what you want to do if you're going to do locale anything:

```
setlocale(LC_ALL, "");  // Use this environment's locale for everything
```

You'll want to call that so that the program gets initialized with your current locale.

Getting into more details, there is one more thing you can do and stay portable:

```
setlocale(LC_ALL, "C");  // Use the default C locale
```

but that's called by default every time your program starts, so there's not much need to do it yourself.

In that second string, you can specify any locale supported by your system. This is completely system-dependent, so it will vary. On my system, I can specify this:

```
setlocale(LC_ALL, "en_US.UTF-8");  // Non-portable!
```

And that'll work. But it's only portable to systems which have that exact same name for that exact same locale, and you can't guarantee it.

By passing in an empty string ("") for the second argument, you're telling C, "Hey, figure out what the current locale on this system is so I don't have to tell you."

26.2 Getting the Monetary Locale Settings

Because moving green pieces of paper around promises to be the key to happiness[1], let's talk about monetary locale. When you're writing portable code, you have to know what to type for cash, right? Whether that's "$", "€", "¥", or "£".

How can you write that code without going insane? Luckily, once you call `setlocale(LC_ALL, "")`, you can just look these up with a call to `localeconv()`:

```
struct lconv *x = localeconv();
```

This function returns a pointer to a statically-allocated `struct lconv` that has all that juicy information you're looking for.

Here are the fields of `struct lconv` and their meanings.

First, some conventions. An _p_ means "positive", and _n_ means "negative", and `int_` means "international". Though a lot of these are type `char` or `char*`, most (or the strings they point to) are actually treated as integers[2].

Before we go further, know that `CHAR_MAX` (from `<limits.h>`) is the maximum value that can be held in a `char`. And that many of the following `char` values use that to indicate the value isn't available in the given locale.

Field	Description
`char *mon_decimal_point`	Decimal pointer character for money, e.g. `"."`.
`char *mon_thousands_sep`	Thousands separator character for money, e.g. `","`.
`char *mon_grouping`	Grouping description for money (see below).
`char *positive_sign`	Positive sign for money, e.g. `"+"` or `""`.
`char *negative_sign`	Negative sign for money, e.g. `"-"`.
`char *currency_symbol`	Currency symbol, e.g. `"$"`.
`char frac_digits`	When printing monetary amounts, how many digits to print past the decimal point, e.g. `2`.
`char p_cs_precedes`	`1` if the `currency_symbol` comes before the value for a non-negative monetary amount, `0` if after.
`char n_cs_precedes`	`1` if the `currency_symbol` comes before the value for a negative monetary amount, `0` if after.
`char p_sep_by_space`	Determines the separation of the `currency symbol` from the value for non-negative amounts (see below).
`char n_sep_by_space`	Determines the separation of the `currency symbol` from the value for negative amounts (see below).
`char p_sign_posn`	Determines the `positive_sign` position for non-negative values.
`char p_sign_posn`	Determines the `positive_sign` position for negative values.
`char *int_curr_symbol`	International currency symbol, e.g. `"USD "`.
`char int_frac_digits`	International value for `frac_digits`.
`char int_p_cs_precedes`	International value for `p_cs_precedes`.
`char int_n_cs_precedes`	International value for `n_cs_precedes`.
`char int_p_sep_by_space`	International value for `p_sep_by_space`.

[1] "This planet has—or rather had—a problem, which was this: most of the people living on it were unhappy for pretty much of the time. Many solutions were suggested for this problem, but most of these were largely concerned with the movement of small green pieces of paper, which was odd because on the whole it wasn't the small green pieces of paper that were unhappy." —The Hitchhiker's Guide to the Galaxy, Douglas Adams

[2] Remember that `char` is just a byte-sized integer.

www.dbooks.org

Field	Description
`char int_n_sep_by_space`	International value for `n_sep_by_space`.
`char int_p_sign_posn`	International value for `p_sign_posn`.
`char int_n_sign_posn`	International value for `n_sign_posn`.

26.2.1 Monetary Digit Grouping

OK, this is a trippy one. `mon_grouping` is a `char*`, so you might be thinking it's a string. But in this case, no, it's really an array of `chars`. It should always end either with a `0` or `CHAR_MAX`.

These values describe how to group sets of numbers in currency to the *left* of the decimal (the whole number part).

For example, we might have:

```
  2   1   0
 --- --- ---
$100,000,000.00
```

These are groups of three. Group 0 (just left of the decimal) has 3 digits. Group 1 (next group to the left) has 3 digits, and the last one also has 3.

So we could describe these groups, from the right (the decimal) to the left with a bunch of integer values representing the group sizes:

```
3 3 3
```

And that would work for values up to $100,000,000.

But what if we had more? We could keep adding `3`s…

```
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

but that's crazy. Luckily, we can specify `0` to indicate that the previous group size repeats:

```
3 0
```

Which means to repeat every 3. That would handle $100, $1,000, $10,000, $10,000,000, $100,000,000,000, and so on.

You can go legitimately crazy with these to indicate some weird groupings.

For example:

```
4 3 2 1 0
```

would indicate:

```
$1,0,0,0,0,00,000,0000.00
```

One more value that can occur is `CHAR_MAX`. This indicates that no more grouping should occur, and can appear anywhere in the array, including the first value.

```
3 2 CHAR_MAX
```

would indicate:

```
100000000,00,000.00
```

for example.

And simply having `CHAR_MAX` in the first array position would tell you there was to be no grouping at all.

26.2.2 Separators and Sign Position

All the `sep_by_space` variants deal with spacing around the currency sign. Valid values are:

Value	Description
0	No space between currency symbol and value.
1	Separate the currency symbol (and sign, if any) from the value with a space.
2	Separate the sign symbol from the currency symbol (if adjacent) with a space, otherwise separate the sign symbol from the value with a space.

The `sign_posn` variants are determined by the following values:

Value	Description
0	Put parens around the value and the currency symbol.
1	Put the sign string in front of the currency symbol and value.
2	Put the sign string after the currency symbol and value.
3	Put the sign string directly in front of the currency symbol.
4	Put the sign string directly behind the currency symbol.

26.2.3 Example Values

When I get the values on my system, this is what I see (grouping string displayed as individual byte values):

```
mon_decimal_point  = "."
mon_thousands_sep  = ","
mon_grouping       = 3 3 0
positive_sign      = ""
negative_sign      = "-"
currency_symbol    = "$"
frac_digits        = 2
p_cs_precedes      = 1
n_cs_precedes      = 1
p_sep_by_space     = 0
n_sep_by_space     = 0
p_sign_posn        = 1
n_sign_posn        = 1
int_curr_symbol    = "USD "
int_frac_digits    = 2
int_p_cs_precedes  = 1
int_n_cs_precedes  = 1
int_p_sep_by_space = 1
int_n_sep_by_space = 1
int_p_sign_posn    = 1
int_n_sign_posn    = 1
```

26.3 Localization Specifics

Notice how we passed the macro `LC_ALL` to `setlocale()` earlier… this hints that there might be some variant that allows you to be more precise about which *parts* of the locale you're setting.

Let's take a look at the values you can see for these:

Macro	Description
`LC_ALL`	Set all of the following to the given locale.
`LC_COLLATE`	Controls the behavior of the `strcoll()` and `strxfrm()` functions.
`LC_CTYPE`	Controls the behavior of the character-handling functions[3].
`LC_MONETARY`	Controls the values returned by `localeconv()`.
`LC_NUMERIC`	Controls the decimal point for the `printf()` family of functions.
`LC_TIME`	Controls time formatting of the `strftime()` and `wcsftime()` time and date printing functions.

It's pretty common to see `LC_ALL` being set, but, hey, at least you have options.

Also I should point out that `LC_CTYPE` is one of the biggies because it ties into wide characters, a significant can of worms that we'll talk about later.

[3]Except for `isdigit()` and `isxdigit()`.

Chapter 27

Unicode, Wide Characters, and All That

Before we begin, note that this is an active area of language development in C as it works to get past some, erm, *growing pains*. When C2x comes out, updates here are probable.

Most people are basically interested in the deceptively simple question, "How do I use such-and-such character set in C?" We'll get to that. But as we'll see, it might already work on your system. Or you might have to punt to a third-party library.

We're going to talk about a lot of things this chapter—some are platform agnostic, and some are C-specific.

Let's get an outline first of what we're going to look at:

- Unicode background
- Character encoding background
- Source and Execution character Sets
- Using Unicode and UTF-8
- Using other character types like `wchar_t`, `char16_t`, and `char32_t`

Let's dive in!

27.1 What is Unicode?

Back in the day, it was popular in the US and much of the world to use a 7-bit or 8-bit encoding for characters in memory. This meant we could have 128 or 256 characters (including non-printable characters) total. That was fine for a US-centric world, but it turns out there are actually other alphabets out there—who knew? Chinese has over 50,000 characters, and that's not fitting in a byte.

So people came up with all kinds of alternate ways to represent their own custom character sets. And that was fine, but turned into a compatibility nightmare.

To escape it, Unicode was invented. One character set to rule them all. It extends off into infinity (effectively) so we'll never run out of space for new characters. It has Chinese, Latin, Greek, cuneiform, chess symbols, emojis… just about everything, really! And more is being added all the time!

27.2 Code Points

I want to talk about two concepts here. It's confusing because they're both numbers… different numbers for the same thing. But bear with me.

Let's loosely define *code point* to mean a numeric value representing a character. (Code points can also represent unprintable control characters, but just assume I mean something like the letter "B" or the character

www.dbooks.org

"π".)

Each code point represents a unique character. And each character has a unique numeric code point associated with it.

For example, in Unicode, the numeric value 66 represents "B", and 960 represents "π". Other character mappings that aren't Unicode use different values, potentially, but let's forget them and concentrate on Unicode, the future!

So that's one thing: there's a number that represents each character. In Unicode, these numbers run from 0 to over 1 million.

Got it?

Because we're about to flip the table a little.

27.3 Encoding

If you recall, an 8-bit byte can hold values from 0-255, inclusive. That's great for "B" which is 66—that fits in a byte. But "π" is 960, and that doesn't fit in a byte! We need another byte. How do we store all that in memory? Or what about bigger numbers, like 195,024? That's going to need a number of bytes to hold.

The Big Question: how are these numbers represented in memory? This is what we call the *encoding* of the characters.

So we have two things: one is the code point which tells us effectively the serial number of a particular character. And we have the encoding which tells us how we're going to represent that number in memory.

There are plenty of encodings. You can make up your own right now, if you want[1]. But we're going to look at some really common encodings that are in use with Unicode.

Encoding	Description
UTF-8	A byte-oriented encoding that uses a variable number of bytes per character. This is the one to use.
UTF-16	A 16-bit per character[2] encoding.
UTF-32	A 32-bit per character encoding.

With UTF-16 and UTF-32, the byte order matters, so you might see UTF-16BE for big-endian and UTF-16LE for little-endian. Same for UTF-32. Technically, if unspecified, you should assume big-endian. But since Windows uses UTF-16 extensively and is little-endian, sometimes that is assumed[3].

Let's look at some examples. I'm going to write the values in hex because that's exactly two digits per 8-bit byte, and it makes it easier to see how things are arranged in memory.

Character	Code Point	UTF-16BE	UTF-32BE	UTF-16LE	UTF-32LE	UTF-8
A	41	0041	00000041	4100	41000000	41
B	42	0042	00000042	4200	42000000	42
~	7E	007E	0000007E	7E00	7E000000	7E
π	3C0	03C0	000003C0	C003	C0030000	CF80

[1] For example, we could store the code point in a big-endian 32-bit integer. Straightforward! We just invented an encoding! Actually not; that's what UTF-32BE encoding is. Oh well—back to the grind!

[2] Ish. Technically, it's variable width—there's a way to represent code points higher than 2^{16} by putting two UTF-16 characters together.

[3] There's a special character called the *Byte Order Mark* (BOM), code point 0xFEFF, that can optionally precede the data stream and indicate the endianess. It is not required, however.

Character	Code Point	UTF-16BE	UTF-32BE	UTF-16LE	UTF-32LE	UTF-8
€	20AC	20AC	000020AC	AC20	AC200000	E282AC

Look in there for the patterns. Note that UTF-16BE and UTF-32BE are simply the code point represented directly as 16- and 32-bit values[4].

Little-endian is the same, except the bytes are in little-endian order.

Then we have UTF-8 at the end. First you might notice that the single-byte code points are represented as a single byte. That's nice. You might also notice that different code points take different number of bytes. This is a variable-width encoding.

So as soon as we get above a certain value, UTF-8 starts using additional bytes to store the values. And they don't appear to correlate with the code point value, either.

The details of UTF-8 encoding[5] are beyond the scope of this guide, but it's enough to know that it has a variable number of bytes per code point, and those byte values don't match up with the code point *except for the first 128 code points*. If you really want to learn more, Computerphile has a great UTF-8 video with Tom Scott[6].

That last bit is a neat thing about Unicode and UTF-8 from a North American perspective: it's backward compatible with 7-bit ASCII encoding! So if you're used to ASCII, UTF-8 is the same! Every ASCII-encoded document is also UTF-8 encoded! (But not the other way around, obviously.)

It's probably that last point more than any other that is driving UTF-8 to take over the world.

27.4 Source and Execution Character Sets

When programming in C, there are (at least) three character sets that are in play:

- The one that your code exists on disk as.
- The one the compiler translates that into just as compilation begins (the *source character set*). This might be the same as the one on disk, or it might not.
- The one the compiler translates the source character set into for execution (the *execution character set*). This might be the same as the source character set, or it might not.

Your compiler probably has options to select these character sets at build-time.

The basic character set for both source and execution will contain the following characters:

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
! " # % & ' ( ) * + , - . / :
; < = > ? [ \ ] ^ _ { | } ~
space tab vertical-tab
form-feed end-of-line
```

Those are the characters you can use in your source and remain 100% portable.

The execution character set will additionally have characters for alert (bell/flash), backspace, carriage return, and newline.

[4] Again, this is only true in UTF-16 for characters that fit in two bytes.
[5] https://en.wikipedia.org/wiki/UTF-8
[6] https://www.youtube.com/watch?v=MijmeoH9LT4

www.dbooks.org

But most people don't go to that extreme and freely use their extended character sets in source and executable, especially now that Unicode and UTF-8 are getting more common. I mean, the basic character set doesn't even allow for `@`, `$`, or `` `! ``

Notably, it's a pain (though possible with escape sequences) to enter Unicode characters using only the basic character set.

27.5 Unicode in C

Before I get into encoding in C, let's talk about Unicode from a code point standpoint. There is a way in C to specify Unicode characters and these will get translated by the compiler into the execution character set[7].

So how do we do it?

How about the euro symbol, code point 0x20AC. (I've written it in hex because both ways of representing it in C require hex.) How can we put that in our C code?

Use the `\u` escape to put it in a string, e.g. `"\u20AC"` (case for the hex doesn't matter). You must put **exactly four** hex digits after the `\u`, padding with leading zeros if necessary.

Here's an example:

```
char *s = "\u20AC1.23";

printf("%s\n", s);  // €1.23
```

So `\u` works for 16-bit Unicode code points, but what about ones bigger than 16 bits? For that, we need capitals: `\U`.

For example:

```
char *s = "\U0001D4D1";

printf("%s\n", s);  // Prints a mathematical letter "B"
```

It's the same as `\u`, just with 32 bits instead of 16. These are equivalent:

```
\u03C0
\U000003C0
```

Again, these are translated into the execution character set during compilation. They represent Unicode code points, not any specific encoding. Furthermore, if a Unicode code point is not representable in the execution character set, the compiler can do whatever it wants with it.

Now, you might wonder why you can't just do this:

```
char *s = "€1.23";

printf("%s\n", s);  // €1.23
```

And you probably can, given a modern compiler. The source character set will be translated for you into the execution character set by the compiler. But compilers are free to puke out if they find any characters that aren't included in their extended character set, and the € symbol certainly isn't in the basic character set.

Caveat from the spec: you can't use `\u` or `\U` to encode any code points below 0xA0 except for 0x24 (`$`), 0x40 (`@`), and 0x60 (`` ``)—yes, those are precisely the trio of common punctuation marks missing from the basic character set. Apparently this restriction is relaxed in the upcoming version of the spec.

[7] Presumably the compiler makes the best effort to translate the code point to whatever the output encoding is, but I can't find any guarantees in the spec.

Finally, you can also use these in identifiers in your code, with some restrictions. But I don't want to get into that here. We're all about string handling in this chapter.

And that's about it for Unicode in C (except encoding).

27.6 A Quick Note on UTF-8 Before We Swerve into the Weeds

It could be that your source file on disk, the extended source characters, and the extended execution characters are all in UTF-8 format. And the libraries you use expect UTF-8. This is the glorious future of UTF-8 everywhere.

If that's the case, and you don't mind being non-portable to systems that aren't like that, then just run with it. Stick Unicode characters in your source and data at will. Use regular C strings and be happy.

A lot of things will just work (albeit non-portably) because UTF-8 strings can safely be NUL-terminated just like any other C string. But maybe losing portability in exchange for easier character handling is a tradeoff that's worth it to you.

There are some caveats, however:

- Things like `strlen()` report the number of bytes in a string, not the number of characters, necessarily. (The `mbstowcs()` returns the number of characters in a string when you convert it to wide characters. POSIX extends this so you can pass `NULL` for the first argument if you just want the character count.)
- The following won't work properly with characters of more than one byte: `strtok()`, `strchr()` (use `strstr()` instead), `strspn()`-type functions, `toupper()`, `tolower()`, `isalpha()`-type functions, and probably more. Beware anything that operates on bytes.
- `printf()` variants allow for a way to only print so many bytes of a string[8]. You want to make certain you print the correct number of bytes to end on a character boundary.
- If you want to `malloc()` space for a string, or declare an array of `chars` for one, be aware that the maximum size could be more than you were expecting. Each character could take up to `MB_LEN_MAX` bytes (from `<limits.h>`)—except characters in the basic character set which are guaranteed to be one byte.

And probably others I haven't discovered. Let me know what pitfalls there are out there…

27.7 Different Character Types

I want to introduce more character types. We're used to `char`, right?

But that's too easy. Let's make things a lot more difficult! Yay!

27.7.1 Multibyte Characters

First of all, I want to potentially change your thinking about what a string (array of `chars`) is. These are *multibyte strings* made up of *multibyte characters*.

That's right—your run-of-the-mill string of characters is multibyte. When someone says "C string", they mean "C multibyte string".

Even if a particular character in the string is only a single byte, or if a string is made up of only single characters, it's known as a multibyte string.

For example:

[8] With a format specifier like `"%s.12"`, for example.

www.dbooks.org

```
char c[128] = "Hello, world!";  // Multibyte string
```

What we're saying here is that a particular character that's not in the basic character set could be composed of multiple bytes. Up to `MB_LEN_MAX` of them (from `<limits.h>`). Sure, it only looks like one character on the screen, but it could be multiple bytes.

You can throw Unicode values in there, as well, as we saw earlier:

```
char *s = "\u20AC1.23";

printf("%s\n", s);  // €1.23
```

But here we're getting into some weirdness, because check this out:

```
char *s = "\u20AC1.23";  // €1.23

printf("%zu\n", strlen(s));  // 7!
```

The string length of `"€1.23"` is `7`?! Yes! Well, on my system, yes! Remember that `strlen()` returns the number of bytes in the string, not the number of characters. (When we get to "wide characters", coming up, we'll see a way to get the number of characters in the string.)

Note that while C allows individual multibyte `char` constants (as opposed to `char*`), the behavior of these varies by implementation and your compiler might warn on it.

GCC, for example, warns of multi-character character constants for the following two lines (and, on my system, prints out the UTF-8 encoding):

```
printf("%x\n", '€');
printf("%x\n", '\u20ac');
```

27.7.2 Wide Characters

If you're not a multibyte character, then you're a *wide character*.

A wide character is a single value that can uniquely represent any character in the current locale. It's analogous to Unicode code points. But it might not be. Or it might be.

Basically, where multibyte character strings are arrays of bytes, wide character strings are arrays of *characters*. So you can start thinking on a character-by-character basis rather than a byte-by-byte basis (the latter of which gets all messy when characters start taking up variable numbers of bytes).

Wide characters can be represented by a number of types, but the big standout one is `wchar_t`. It's the main one. It's like `char`, except wide.

You might be wondering if you can't tell if it's Unicode or not, how does that allow you much flexibility in terms of writing code? `wchar_t` opens some of those doors, as there are a rich set of functions you can use to deal with `wchar_t` strings (like getting the length, etc.) without caring about the encoding.

27.8 Using Wide Characters and `wchar_t`

Time for a new type: `wchar_t`. This is the main wide character type. Remember how a `char` is only one byte? And a byte's not enough to represent all characters, potentially? Well, this one is enough.

To use `wchar_t`, include `<wchar.h>`.

How many bytes big is it? Well, it's not totally clear. Could be 16 bits. Could be 32 bits.

But wait, you're saying—if it's only 16 bits, it's not big enough to hold all the Unicode code points, is it? You're right—it's not. The spec doesn't require it to be. It just has to be able to represent all the characters in the current locale.

This can cause grief with Unicode on platforms with 16-bit `wchar_ts` (ahem—Windows). But that's out of scope for this guide.

You can declare a string or character of this type with the `L` prefix, and you can print them with the `%ls` ("ell ess") format specifier. Or print an individual `wchar_t` with `%lc`.

```
wchar_t *s = L"Hello, world!";
wchar_t c = L'B';

printf("%ls %lc\n", s, c);
```

Now—are those characters stored as Unicode code points, or not? Depends on the implementation. But you can test if they are with the macro `__STDC_ISO_10646__`. If this is defined, the answer is, "It's Unicode!"

More detailedly, the value in that macro is an integer in the form `yyyymm` that lets you know what Unicode standard you can rely on—whatever was in effect on that date.

But how do you use them?

27.8.1 Multibyte to `wchar_t` Conversions

So how do we get from the byte-oriented standard strings to the character-oriented wide strings and back?

We can use a couple string conversion functions to make this happen.

First, some naming conventions you'll see in these functions:

- `mb`: multibyte
- `wc`: wide character
- `mbs`: multibyte string
- `wcs`: wide character string

So if we want to convert a multibyte string to a wide character string, we can call the `mbstowcs()`. And the other way around: `wcstombs()`.

Conversion Function	Description
`mbtowc()`	Convert a multibyte character to a wide character.
`wctomb()`	Convert a wide character to a multibyte character.
`mbstowcs()`	Convert a multibyte string to a wide string.
`wcstombs()`	Convert a wide string to a multibyte string.

Let's do a quick demo where we convert a multibyte string to a wide character string, and compare the string lengths of the two using their respective functions.

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <string.h>
#include <locale.h>

int main(void)
{
    // Get out of the C locale to one that likely has the euro symbol
    setlocale(LC_ALL, "");
```

```

    // Original multibyte string with a euro symbol (Unicode point 20ac)
    char *mb_string = "The cost is \u20ac1.23";  // €1.23
    size_t mb_len = strlen(mb_string);

    // Wide character array that will hold the converted string
    wchar_t wc_string[128];  // Holds up to 128 wide characters

    // Convert the MB string to WC; this returns the number of wide chars
    size_t wc_len = mbstowcs(wc_string, mb_string, 128);

    // Print result--note the %ls for wide char strings
    printf("multibyte: \"%s\" (%zu bytes)\n", mb_string, mb_len);
    printf("wide char: \"%ls\" (%zu characters)\n", wc_string, wc_len);
}
```

On my system, this outputs:

```
multibyte: "The cost is €1.23" (19 bytes)
wide char: "The cost is €1.23" (17 characters)
```

(Your system might vary on the number of bytes depending on your locale.)

One interesting thing to note is that `mbstowcs()`, in addition to converting the multibyte string to wide, returns the length (in characters) of the wide character string. On POSIX-compliant systems, you can take advantage of a special mode where it *only* returns the length-in-characters of a given multibyte string: you just pass `NULL` to the destination, and `0` to the maximum number of characters to convert (this value is ignored).

(In the code below, I'm using my extended source character set—you might have to replace those with `\u` escapes.)

```
setlocale(LC_ALL, "");

// The following string has 7 characters
size_t len_in_chars = mbstowcs(NULL, "§¶°±π€•", 0);

printf("%zu", len_in_chars);  // 7
```

Again, that's a non-portable POSIX extension.

And, of course, if you want to convert the other way, it's `wcstombs()`.

27.9 Wide Character Functionality

Once we're in wide character land, we have all kinds of functionality at our disposal. I'm just going to summarize a bunch of the functions here, but basically what we have here are the wide character versions of the multibyte string functions that we're use to. (For example, we know `strlen()` for multibyte strings; there's a `wcslen()` for wide character strings.)

27.9.1 `wint_t`

A lot of these functions use a `wint_t` to hold single characters, whether they are passed in or returned.

It is related to `wchar_t` in nature. A `wint_t` is an integer that can represent all values in the extended character set, and also a special end-of-file character, `WEOF`.

This is used by a number of single-character-oriented wide character functions.

27.9.2 I/O Stream Orientation

The tl;dr here is to not mix and match byte-oriented functions (like `fprintf()`) with wide-oriented functions (like `fwprintf()`). Decide if a stream will be byte-oriented or wide-oriented and stick with those types of I/O functions.

In more detail: streams can be either byte-oriented or wide-oriented. When a stream is first created, it has no orientation, but the first read or write will set the orientation.

If you first use a wide operation (like `fwprintf()`) it will orient the stream wide.

If you first use a byte operation (like `fprintf()`) it will orient the stream by bytes.

You can manually set an unoriented stream one way or the other with a call to `fwide()`. You can use that same function to get the orientation of a stream.

If you need to change the orientation mid-flight, you can do it with `freopen()`.

27.9.3 I/O Functions

Typically include `<stdio.h>` and `<wchar.h>` for these.

I/O Function	Description
`wprintf()`	Formatted console output.
`wscanf()`	Formatted console input.
`getwchar()`	Character-based console input.
`putwchar()`	Character-based console output.
`fwprintf()`	Formatted file output.
`fwscanf()`	Formatted file input.
`fgetwc()`	Character-based file input.
`fputwc()`	Character-based file output.
`fgetws()`	String-based file input.
`fputws()`	String-based file output.
`swprintf()`	Formatted string output.
`swscanf()`	Formatted string input.
`vfwprintf()`	Variadic formatted file output.
`vfwscanf()`	Variadic formatted file input.
`vswprintf()`	Variadic formatted string output.
`vswscanf()`	Variadic formatted string input.
`vwprintf()`	Variadic formatted console output.
`vwscanf()`	Variadic formatted console input.
`ungetwc()`	Push a wide character back on an output stream.
`fwide()`	Get or set stream multibyte/wide orientation.

27.9.4 Type Conversion Functions

Typically include `<wchar.h>` for these.

Conversion Function	Description
`wcstod()`	Convert string to `double`.
`wcstof()`	Convert string to `float`.
`wcstold()`	Convert string to `long double`.
`wcstol()`	Convert string to `long`.
`wcstoll()`	Convert string to `long long`.
`wcstoul()`	Convert string to `unsigned long`.

www.dbooks.org

Conversion Function	Description
`wcstoull()`	Convert string to `unsigned long long`.

27.9.5 String and Memory Copying Functions

Typically include `<wchar.h>` for these.

Copying Function	Description
`wcscpy()`	Copy string.
`wcsncpy()`	Copy string, length-limited.
`wmemcpy()`	Copy memory.
`wmemmove()`	Copy potentially-overlapping memory.
`wcscat()`	Concatenate strings.
`wcsncat()`	Concatenate strings, length-limited.

27.9.6 String and Memory Comparing Functions

Typically include `<wchar.h>` for these.

Comparing Function	Description
`wcscmp()`	Compare strings lexicographically.
`wcsncmp()`	Compare strings lexicographically, length-limited.
`wcscoll()`	Compare strings in dictionary order by locale.
`wmemcmp()`	Compare memory lexicographically.
`wcsxfrm()`	Transform strings into versions such that `wcscmp()` behaves like `wcscoll()`[9].

27.9.7 String Searching Functions

Typically include `<wchar.h>` for these.

Searching Function	Description
`wcschr()`	Find a character in a string.
`wcsrchr()`	Find a character in a string from the back.
`wmemchr()`	Find a character in memory.
`wcsstr()`	Find a substring in a string.
`wcspbrk()`	Find any of a set of characters in a string.
`wcsspn()`	Find length of substring including any of a set of characters.
`wcscspn()`	Find length of substring before any of a set of characters.
`wcstok()`	Find tokens in a string.

[9] `wcscoll()` is the same as `wcsxfrm()` followed by `wcscmp()`.

27.9.8 Length/Miscellaneous Functions

Typically include `<wchar.h>` for these.

Length/Misc Function	Description
`wcslen()`	Return the length of the string.
`wmemset()`	Set characters in memory.
`wcsftime()`	Formatted date and time output.

27.9.9 Character Classification Functions

Include `<wctype.h>` for these.

Length/Misc Function	Description
`iswalnum()`	True if the character is alphanumeric.
`iswalpha()`	True if the character is alphabetic.
`iswblank()`	True if the character is blank (space-ish, but not a newline).
`iswcntrl()`	True if the character is a control character.
`iswdigit()`	True if the character is a digit.
`iswgraph()`	True if the character is printable (except space).
`iswlower()`	True if the character is lowercase.
`iswprint()`	True if the character is printable (including space).
`iswpunct()`	True if the character is punctuation.
`iswspace()`	True if the character is whitespace.
`iswupper()`	True if the character is uppercase.
`iswxdigit()`	True if the character is a hex digit.
`towlower()`	Convert character to lowercase.
`towupper()`	Convert character to uppercase.

27.10 Parse State, Restartable Functions

We're going to get a little bit into the guts of multibyte conversion, but this is a good thing to understand, conceptually.

Imagine how your program takes a sequence of multibyte characters and turns them into wide characters, or vice-versa. It might, at some point, be partway through parsing a character, or it might have to wait for more bytes before it makes the determination of the final value.

This parse state is stored in an opaque variable of type `mbstate_t` and is used every time conversion is performed. That's how the conversion functions keep track of where they are mid-work.

And if you change to a different character sequence mid-stream, or try to seek to a different place in your input sequence, it could get confused over that.

Now you might want to call me on this one: we just did some conversions, above, and I never mentioned any `mbstate_t` anywhere.

That's because the conversion functions like `mbstowcs()`, `wctomb()`, etc. each have their own `mbstate_t` variable that they use. There's only one per function, though, so if you're writing multithreaded code, they're not safe to use.

Fortunately, C defines *restartable* versions of these functions where you can pass in your own `mbstate_t` on per-thread basis if you need to. If you're doing multithreaded stuff, use these!

www.dbooks.org

Quick note on initializing an `mbstate_t` variable: just `memset()` it to zero. There is no built-in function to force it to be initialized.

```
mbstate_t mbs;

// Set the state to the initial state
memset(&mbs, 0, sizeof mbs);
```

Here is a list of the restartable conversion functions—note the naming convension of putting an "`r`" after the "from" type:

- `mbrtowc()`—multibyte to wide character
- `wcrtomb()`—wide character to multibyte
- `mbsrtowcs()`—multibyte string to wide character string
- `wcsrtombs()`—wide character string to multibyte string

These are really similar to their non-restartable counterparts, except they require you pass in a pointer to your own `mbstate_t` variable. And also they modify the source string pointer (to help you out if invalid bytes are found), so it might be useful to save a copy of the original.

Here's the example from earlier in the chapter reworked to pass in our own `mbstate_t`.

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <wchar.h>
#include <string.h>
#include <locale.h>

int main(void)
{
    // Get out of the C locale to one that likely has the euro symbol
    setlocale(LC_ALL, "");

    // Original multibyte string with a euro symbol (Unicode point 20ac)
    char *mb_string = "The cost is \u20ac1.23";  // €1.23
    size_t mb_len = strlen(mb_string);

    // Wide character array that will hold the converted string
    wchar_t wc_string[128];  // Holds up to 128 wide characters

    // Set up the conversion state
    mbstate_t mbs;
    memset(&mbs, 0, sizeof mbs);  // Initial state

    // mbsrtowcs() modifies the input pointer to point at the first
    // invalid character, or NULL if successful. Let's make a copy of
    // the pointer for mbsrtowcs() to mess with so our original is
    // unchanged.
    //
    // This example will probably be successful, but we check farther
    // down to see.
    const char *invalid = mb_string;

    // Convert the MB string to WC; this returns the number of wide chars
    size_t wc_len = mbsrtowcs(wc_string, &invalid, 128, &mbs);

```

```
    if (invalid == NULL) {
        printf("No invalid characters found\n");

        // Print result--note the %ls for wide char strings
        printf("multibyte: \"%s\" (%zu bytes)\n", mb_string, mb_len);
        printf("wide char: \"%ls\" (%zu characters)\n", wc_string, wc_len);
    } else {
        ptrdiff_t offset = invalid - mb_string;
        printf("Invalid character at offset %td\n", offset);
    }
}
```

For the conversion functions that manage their own state, you can reset their internal state to the initial one by passing in `NULL` for their `char*` arguments, for example:

```
mbstowcs(NULL, NULL, 0);   // Reset the parse state for mbstowcs()
mbstowcs(dest, src, 100);  // Parse some stuff
```

For I/O, each wide stream manages its own `mbstate_t` and uses that for input and output conversions as it goes.

And some of the byte-oriented I/O functions like `printf()` and `scanf()` keep their own internal state while doing their work.

Finally, these restartable conversion functions do actually have their own internal state if you pass in `NULL` for the `mbstate_t` parameter. This makes them behave more like their non-restartable counterparts.

27.11 Unicode Encodings and C

In this section, we'll see what C can (and can't) do when it comes to three specific Unicode encodings: UTF-8, UTF-16, and UTF-32.

27.11.1 UTF-8

To refresh before this section, read the UTF-8 quick note, above.

Aside from that, what are C's UTF-8 capabilities?

Well, not much, unfortunately.

You can tell C that you specifically want a string literal to be UTF-8 encoded, and it'll do it for you. You can prefix a string with `u8`:

```
char *s = u8"Hello, world!";

printf("%s\n", s);   // Hello, world!--if you can output UTF-8
```

Now, can you put Unicode characters in there?

```
char *s = u8"€123";
```

Sure! If the extended source character set supports it. (gcc does.)

What if it doesn't? You can specify a Unicode code point with your friendly neighborhood `\u` and `\U`, as noted above.

But that's about it. There's no portable way in the standard library to take arbirary input and turn it into UTF-8 unless your locale is UTF-8. Or to parse UTF-8 unless your locale is UTF-8.

So if you want to do it, either be in a UTF-8 locale and:

```
setlocale(LC_ALL, "");
```

or figure out a UTF-8 locale name on your local machine and set it explicitly like so:

```
setlocale(LC_ALL, "en_US.UTF-8");  // Non-portable name
```

Or use a third-party library.

27.11.2 UTF-16, UTF-32, `char16_t`, and `char32_t`

`char16_t` and `char32_t` are a couple other potentially wide character types with sizes of 16 bits and 32 bits, respectively. Not necessarily wide, because if they can't represent every character in the current locale, they lose their wide character nature. But the spec refers them as "wide character" types all over the place, so there we are.

These are here to make things a little more Unicode-friendly, potentially.

To use, include `<uchar.h>`. (That's "u", not "w".)

This header file doesn't exist on OS X—bummer. If you just want the types, you can:

```
#include <stdint.h>

typedef int_least16_t char16_t;
typedef int_least32_t char32_t;
```

But if you also want the functions, that's all on you.

Assuming you're still good to go, you can declare a string or character of these types with the `u` and `U` prefixes:

```
char16_t *s = u"Hello, world!";
char16_t c = u'B';

char32_t *t = U"Hello, world!";
char32_t d = U'B';
```

Now—are values in these stored in UTF-16 or UTF-32? Depends on the implementation.

But you can test to see if they are. If the macros `__STDC_UTF_16__` or `__STDC_UTF_32__` are defined (to `1`) it means the types hold UTF-16 or UTF-32, respectively.

If you're curious, and I know you are, the values, if UTF-16 or UTF-32, are stored in the native endianess. That is, you should be able to compare them straight up to Unicode code point values:

```
char16_t pi = u"\u03C0";  // pi symbol

#if __STDC_UTF_16__
pi == 0x3C0;  // Always true
#else
pi == 0x3C0;  // Probably not true
#endif
```

27.11.3 Multibyte Conversions

You can convert from your multibyte encoding to `char16_t` or `char32_t` with a number of helper functions.

(Like I said, though, the result might not be UTF-16 or UTF-32 unless the corresponding macro is set to `1`.)

All of these functions are restartable (i.e. you pass in your own `mbstate_t`), and all of them operate character by character[10].

Conversion Function	Description
`mbrtoc16()`	Convert a multibyte character to a `char16_t` character.
`mbrtoc32()`	Convert a multibyte character to a `char32_t` character.
`c16rtomb()`	Convert a `char16_t` character to a multibyte character.
`c32rtomb()`	Convert a `char32_t` character to a multibyte character.

27.11.4 Third-Party Libraries

For heavy-duty conversion between different specific encodings, there are a couple mature libraries worth checking out. Note that I haven't used either of these.

- iconv[11]—Internationalization Conversion, a common POSIX-standard API available on the major platforms.
- ICU[12]—International Components for Unicode. At least one blogger found this easy to use.

If you have more noteworthy libraries, let me know.

[10] Ish—things get funky with multi-`char16_t` UTF-16 encodings.
[11] https://en.wikipedia.org/wiki/Iconv
[12] http://site.icu-project.org/

www.dbooks.org

Chapter 28

Exiting a Program

Turns out there are a lot of ways to do this, and even ways to set up "hooks" so that a function runs when a program exits.

In this chapter we'll dive in and check them out.

We already covered the meaning of the exit status code in the Exit Status section, so jump back there and review if you have to.

All the functions in this section are in `<stdlib.h>`.

28.1 Normal Exits

We'll start with the regular ways to exit a program, and then jump to some of the rarer, more esoteric ones.

When you exit a program normally, all open I/O streams are flushed and temporary files removed. Basically it's a nice exit where everything gets cleaned up and handled. It's what you want to do almost all the time unless you have reasons to do otherwise.

28.1.1 Returning From `main()`

If you've noticed, `main()` has a return type of `int`… and yet I've rarely, if ever, been `return`ing anything from `main()` at all.

This is because for `main()` only (and I can't stress enough this special case *only* applies to `main()` and no other functions anywhere) has an *implicit* `return 0` if you fall off the end.

You can explicitly `return` from `main()` any time you want, and some programmers feel it's more *Right* to always have a `return` at the end of `main()`. But if you leave it off, C will put one there for you.

So… here are the `return` rules for `main()`:

- You can return an exit status from `main()` with a `return` statement. `main()` is the only function with this special behavior. Using `return` in any other function just returns from that function to the caller.
- If you don't explicitly `return` and just fall off the end of `main()`, it's just as if you'd returned `0` or `EXIT_SUCCESS`.

28.1.2 `exit()`

This one has also made an appearance a few times. If you call `exit()` from anywhere in your program, it will exit at that point.

The argument you pass to `exit()` is the exit status.

28.1.3 Setting Up Exit Handlers with `atexit()`

You can register functions to be called when a program exits whether by returning from `main()` or calling the `exit()` function.

A call to `atexit()` with the handler function name will get it done. You can register multiple exit handlers, and they'll be called in the reverse order of registration.

Here's an example:

```c
#include <stdio.h>
#include <stdlib.h>

void on_exit_1(void)
{
    printf("Exit handler 1 called!\n");
}

void on_exit_2(void)
{
    printf("Exit handler 2 called!\n");
}

int main(void)
{
    atexit(on_exit_1);
    atexit(on_exit_2);

    printf("About to exit...\n");
}
```

And the output is:

```
About to exit...
Exit handler 2 called!
Exit handler 1 called!
```

28.2 Quicker Exits with `quick_exit()`

This is similar to a normal exit, except:

- Open files might not be flushed.
- Temporary files might not be removed.
- `atexit()` handlers won't be called.

But there is a way to register exit handlers: call `at_quick_exit()` analogously to how you'd call `atexit()`.

```c
#include <stdio.h>
#include <stdlib.h>

void on_quick_exit_1(void)
{
    printf("Quick exit handler 1 called!\n");
}

void on_quick_exit_2(void)
{
```

```
    printf("Quick exit handler 2 called!\n");
}

void on_exit(void)
{
    printf("Normal exit--I won't be called!\n");
}

int main(void)
{
    at_quick_exit(on_quick_exit_1);
    at_quick_exit(on_quick_exit_2);

    atexit(on_exit);  // This won't be called

    printf("About to quick exit...\n");

    quick_exit(0);
}
```

Which gives this output:

```
About to quick exit...
Quick exit handler 2 called!
Quick exit handler 1 called!
```

It works just like `exit()`/`atexit()`, except for the fact that file flushing and cleanup might not be done.

28.3 Nuke it from Orbit: `_Exit()`

Calling `_Exit()` exits immediately, period. No on-exit callback functions are executed. Files won't be flushed. Temp files won't be removed.

Use this if you have to exit *right fargin' now*.

28.4 Exiting Sometimes: `assert()`

The `assert()` statement is used to insist that something be true, or else the program will exit.

Devs often use an assert to catch Should-Never-Happen type errors.

```
#define PI 3.14159

assert(PI > 3);   // Sure enough, it is, so carry on
```

versus:

```
goats -= 100;

assert(goats >= 0);  // Can't have negative goats
```

In that case, if I try to run it and `goats` falls under `0`, this happens:

```
goat_counter: goat_counter.c:8: main: Assertion `goats >= 0' failed.
Aborted
```

and I'm dropped back to the command line.

This isn't very user-friendly, so it's only used for things the user will never see. And often people write their own assert macros that can more easily be turned off.

28.5 Abnormal Exit: `abort()`

You can use this if something has gone horribly wrong and you want to indicate as much to the outside environment. This also won't necessarily clean up any open files, etc.

I've rarely seen this used.

Some foreshadowing about *signals*: this actually works by raising a `SIGABRT` which will end the process.

What happens after that is up to the system, but on Unix-likes, it was common to dump core[1] as the program terminated.

[1] https://en.wikipedia.org/wiki/Core_dump

www.dbooks.org

Chapter 29

Signal Handling

Before we start, I'm just going to advise you to generally ignore this entire chapter and use your OS's (very likely) superior signal handling functions. Unix-likes have the `sigaction()` family of functions, and Windows has… whatever it does[1].

With that out of the way, what are signals?

29.1 What Are Signals?

A *signal* is *raised* on a variety of external events. Your program can be configured to be interrupted to *handle* the signal, and, optionally, continue where it left off once the signal has been handled.

Think of it like a function that's automatically called when one of these external events occurs.

What are these events? On your system, there are probably a lot of them, but in the C spec there are just a few:

Signal	Description
`SIGABRT`	Abnormal termination—what happens when `abort()` is called.
`SIGFPE`	Floating point exception.
`SIGILL`	Illegal instruction.
`SIGINT`	Interrupt—usually the result of `CTRL-C` being hit.
`SIGSEGV`	"Segmentation Violation": invalid memory access.
`SIGTERM`	Termination requested.

You can set up your program to ignore, handle, or allow the default action for each of these by using the `signal()` function.

29.2 Handling Signals with `signal()`

The `signal()` call takes two parameters: the signal in question, and an action to take when that signal is raised.

The action can be one of three things:

- A pointer to a handler function.

[1] Apparently it doesn't do Unix-style signals at all deep down, and they're simulated for console apps.

- `SIG_IGN` to ignore the signal.
- `SIG_DFL` to restore the default handler for the signal.

Let's write a program that you can't `CTRL-C` out of. (Don't fret—in the following program, you can also hit `RETURN` and it'll exit.)

```
1  #include <stdio.h>
2  #include <signal.h>
3
4  int main(void)
5  {
6      char s[1024];
7
8      signal(SIGINT, SIG_IGN);    // Ignore SIGINT, caused by ^C
9
10     printf("Try hitting ^C... (hit RETURN to exit)\n");
11
12     // Wait for a line of input so the program doesn't just exit
13     fgets(s, sizeof s, stdin);
14 }
```

Check out line 8—we tell the program to ignore `SIGINT`, the interrupt signal that's raised when `CTRL-C` is hit. No matter how much you hit it, the signal remains ignored. If you comment out line 8, you'll see you can `CTRL-C` with impunity and quit the program on the spot.

29.3 Writing Signal Handlers

I mentioned you could also write a handler function that gets called with the signal is raised.

These are pretty straightforward, are also very capability-limited when it comes to the spec.

Before we start, let's look at the function prototype for the `signal()` call:

```
void (*signal(int sig, void (*func)(int)))(int);
```

Pretty easy to read, right?

WRONG! :)

Let's take a moment to take it apart for practice.

`signal()` takes two arguments: an integer `sig` representing the signal, and a pointer `func` to the handler (the handler returns `void` and takes an `int` as an argument), highlighted below:

```
                sig          func
              |-----|  |---------------|
void (*signal(int sig, void (*func)(int)))(int);
```

Basically, we're going to pass in the signal number we're interesting in catching, and we're going to pass a pointer to a function of the form:

```
void f(int x);
```

that will do the actual catching.

Now—what about the rest of that prototype? It's basically all the return type. See, `signal()` will return whatever you passed as `func` on success… so that means it's returning a pointer to a function that returns `void` and takes an `int` as an argument.

```
returned
function    indicates we're              and
returns     returning a                  that function
void        pointer to function          takes an int
|--|        |                                   |---|
void       (*signal(int sig, void (*func)(int)))(int);
```

Also, it can return `SIG_ERR` in case of an error.

Let's do an example where we make it so you have to hit `CTRL-C` twice to exit.

I want to be clear that this program engages in undefined behavior in a couple ways. But it'll probably work for you, and it's hard to come up with portable non-trivial demos.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4
5  int count = 0;
6
7  void sigint_handler(int signum)
8  {
9      // The compiler is allowed to run:
10     //
11     //   signal(signum, SIG_DFL)
12     //
13     // when the handler is called. So we reset the handler here:
14     signal(SIGINT, sigint_handler);
15
16     (void)signum;   // Get rid of unused variable warning
17
18     count++;                        // Undefined behavior
19     printf("Count: %d\n", count);  // Undefined behavior
20
21     if (count == 2) {
22         printf("Exiting!\n");      // Undefined behavior
23         exit(0);
24     }
25 }
26
27 int main(void)
28 {
29     signal(SIGINT, sigint_handler);
30
31     printf("Try hitting ^C...\n");
32
33     for(;;);  // Wait here forever
34 }
```

One of the things you'll notice is that on line 14 we reset the signal handler. This is because C has the option of resetting the signal handler to its `SIG_DFL` behavior before running your custom handler. In other words, it could be a one-off. So we reset it first thing so that we handle it again for the next one.

We're ignoring the return value from `signal()` in this case. If we'd set it to a different handler earlier, it would return a pointer to that handler, which we could get like this:

```
// old_handler is type "pointer to function that takes a single
// int parameter and returns void":

void (*old_handler)(int);

old_handler = signal(SIGINT, sigint_handler);
```

That said, I'm not sure of a common use case for this. But if you need the old handler for some reason, you can get it that way.

Quick note on line 16—that's just to tell the compiler to not warn that we're not using this variable. It's like saying, "I know I'm not using it; you don't have to warn me."

And lastly you'll see that I've marked undefined behavior in a couple places. More on that in the next section.

29.4 What Can We Actually Do?

Turns out we're pretty limited in what we can and can't do in our signal handlers. This is one of the reasons why I say you shouldn't even bother with this and instead use your OS's signal handling instead (e.g. `sigaction()` for Unix-like systems).

Wikipedia goes so far as to say the only really portable thing you can do is call `signal()` with `SIG_IGN` or `SIG_DFL` and that's it.

Here's what we **can't** portably do:

- Call any standard library function.
  - Like `printf()`, for example.
  - I think it's probably safe to call restartable/reentrant functions, but the spec doesn't allow that liberty.
- Get or set values from a local `static`, file scope, or thread-local variable.
  - Unless it's a lock-free atomic object or…
  - You're assigning into a variable of type `volatile sig_atomic_t`.

That last bit–`sig_atomic_t`–is your ticket to getting data out of a signal handler. (Unless you want to use lock-free atomic objects, which is outside the scope of this section[2].) It's an integer type that might or might not be signed. And it's bounded by what you can put in there.

You can look at the minimum and maximum allowable values in the macros `SIG_ATOMIC_MIN` and `SIG_ATOMIC_MAX`[3].

Confusingly, the spec also says you can't refer "to any object with static or thread storage duration that is not a lock-free atomic object other than by assigning a value to an object declared as `volatile sig_atomic_t` […]"

My read on this is that you can't read or write anything that's not a lock-free atomic object. Also you can assign to an object that's `volatile sig_atomic_t`.

But can you read from it? I honestly don't see why not, except that the spec is very pointed about mentioning assigning into. But if you have to read it and make any kind of decision based on it, you might be opening up room for some kind of race conditions.

With that in mind, we can rewrite our "hit `CTRL-C` twice to exit" code to be a little more portable, albeit less verbose on the output.

Let's change our `SIGINT` handler to do nothing except increment a value that's of type `volatile sig_atomic_t`. So it'll count the number of `CTRL-C`s that have been hit.

[2] Confusingly, `sig_atomic_t` predates the lock-free atomics and is not the same thing.
[3] If `sig_action_t` is signed, the range will be at least `-127` to `127`. If unsigned, at least `0` to `255`.

www.dbooks.org

Then in our main loop, we'll check to see if that counter is over `2`, then bail out if it is.

```
#include <stdio.h>
#include <signal.h>

volatile sig_atomic_t count = 0;

void sigint_handler(int signum)
{
    (void)signum;                    // Unused variable warning

    signal(SIGINT, sigint_handler);  // Reset signal handler

    count++;                         // Undefined behavior
}

int main(void)
{
    signal(SIGINT, sigint_handler);

    printf("Hit ^C twice to exit.\n");

    while(count < 2);
}
```

Undefined behavior again? It's my read that this is, because we have to read the value in order to increment and store it.

If we only want to postpone the exit by one hitting of `CTRL-C`, we can do that without too much trouble. But any more postponement would require some ridiculous function chaining.

What we'll do is handle it once, and the handler will reset the signal to its default behavior (that is, to exit):

```
#include <stdio.h>
#include <signal.h>

void sigint_handler(int signum)
{
    (void)signum;                      // Unused variable warning
    signal(SIGINT, SIG_DFL);           // Reset signal handler
}

int main(void)
{
    signal(SIGINT, sigint_handler);

    printf("Hit ^C twice to exit.\n");

    while(1);
}
```

Later when we look at lock-free atomic variables, we'll see a way to fix the `count` version (assuming lock-free atomic variables are available on your particular system).

This is why at the beginning, I was suggesting checking out your OS's built-in signal system as a probably-superior alternative.

29.5 Friends Don't Let Friends `signal()`

Again, use your OS's built-in signal handling or the equivalent. It's not in the spec, not as portable, but probably is far more capable. Plus your OS probably has a number of signals defined that aren't in the C spec. And it's difficult to write portable code using `signal()` anyway.

www.dbooks.org

Chapter 30

Variable-Length Arrays (VLAs)

C provides a way for you to declare an array whose size is determined at runtime. This gives you the benefits of dynamic runtime sizing like you get with `malloc()`, but without needing to worry about `free()`ing the memory after.

Now, a lot of people don't like VLAs. They've been banned from the Linux kernel, for example. We'll dig into more of that rationale later.

This is an optional feature of the language. The macro `__STDC_NO_VLA__` is set to `1` if VLAs are *not* present. (They were mandatory in C99, and then became optional in C11.)

```
#if __STDC_NO_VLA__ == 1
   #error Sorry, need VLAs for this program!
#endif
```

But since neither GCC nor Clang bother to define this macro, you may get limited mileage from this.

Let's dive in first with an example, and then we'll look for the devil in the details.

30.1 The Basics

A normal array is declared with a constant size, like this:

```
int v[10];
```

But with VLAs, we can use a size determined at runtime to set the array, like this:

```
int n = 10;
int v[n];
```

Now, that looks like the same thing, and in many ways is, but this gives you the flexibility to compute the size you need, and then get an array of exactly that size.

Let's ask the user to input the size of the array, and then store the index-times-10 in each of those array elements:

```
#include <stdio.h>

int main(void)
{
    int n;

    printf("Enter a number: "); fflush(stdout);
```

```
8       scanf(" %d", &n);
9
10      int v[n];
11
12      for (int i = 0; i < n; i++)
13          v[i] = i * 10;
14
15      for (int i = 0; i < n; i++)
16          printf("v[%d] = %d\n", i, v[i]);
17  }
```

(On line 7, I have an `fflush()` that should force the line to output even though I don't have a newline at the end.)

Line 10 is where we declare the VLA—once execution gets past that line, the size of the array is set to whatever `n` was at that moment. The array length can't be changed later.

You can put an expression in the brackets, as well:

```
int v[x * 100];
```

Some restrictions:

- You can't declare a VLA at file scope, and you can't make a `static` one in block scope[1].
- You can't use an initializer list to initialize the array.

Also, entering a negative value for the size of the array invokes undefined behavior—in this universe, anyway.

30.2 `sizeof` and VLAs

We're used to `sizeof` giving us the size in bytes of any particular object, including arrays. And VLAs are no exception.

The main difference is that `sizeof` on a VLA is executed at *runtime*, whereas on a non-variably-sized variable it is computed at *compile time*.

But the usage is the same.

You can even compute the number of elements in a VLA with the usual array trick:

```
size_t num_elems = sizeof v / sizeof v[0];
```

There's a subtle and correct implication from the above line: pointer arithmetic works just like you'd expect for a regular array. So go ahead and use it to your heart's content:

```
#include <stdio.h>

int main(void)
{
    int n = 5;
    int v[n];

    int *p = v;

    *(p+2) = 12;
    printf("%d\n", v[2]);  // 12
```

[1]This is due to how VLAs are typically allocated on the stack, whereas `static` variables are on the heap. And the whole idea with VLAs is they'll be automatically dellocated when the stack frame is popped at the end of the function.

www.dbooks.org

```

    p[3] = 34;
    printf("%d\n", v[3]);  // 34
}
```

Like with regular arrays, you can use parentheses with `sizeof()` to get the size of a would-be VLA without actually declaring one:

```
int x = 12;

printf("%zu\n", sizeof(int [x]));  // Prints 48 on my system
```

30.3 Multidimensional VLAs

You can go ahead and make all kinds of VLAs with one or more dimensions set to a variable

```
int w = 10;
int h = 20;

int x[h][w];
int y[5][w];
int z[10][w][20];
```

Again, you can navigate these just like you would a regular array.

30.4 Passing One-Dimensional VLAs to Functions

Passing single-dimensional VLAs into a function can be no different than passing a regular array in. You just go for it.

```
#include <stdio.h>

int sum(int count, int *v)
{
    int total = 0;

    for (int i = 0; i < count; i++)
        total += v[i];

    return total;
}

int main(void)
{
    int x[5];   // Standard array

    int a = 5;
    int y[a];   // VLA

    for (int i = 0; i < a; i++)
        x[i] = y[i] = i + 1;

    printf("%d\n", sum(5, x));
    printf("%d\n", sum(a, y));
```

```
}
```

But there's a bit more to it than that. You can also let C know that the array is a specific VLA size by passing that in first and then giving that dimension in the parameter list:

```
int sum(int count, int v[count])
{
    // ...
}
```

Incidentally, there are a couple ways of listing a prototype for the above function; one of them involves an `*` if you don't want to specifically name the value in the VLA. It just indicates that the type is a VLA as opposed to a regular pointer.

VLA prototypes:

```
void do_something(int count, int v[count]);  // With names
void do_something(int, int v[*]);            // Without names
```

Again, that `*` thing only works with the prototype—in the function itself, you'll have to put the explicit size.

Now—*let's get multidimensional*! This is where the fun begins.

30.5 Passing Multi-Dimensional VLAs to Functions

Same thing as we did with the second form of one-dimensional VLAs, above, but this time we're passing in two dimensions and using those.

In the following example, we build a multiplication table matrix of a variable width and height, and then pass it to a function to print it out.

```
#include <stdio.h>

void print_matrix(int h, int w, int m[h][w])
{
    for (int row = 0; row < h; row++) {
        for (int col = 0; col < w; col++)
            printf("%2d ", m[row][col]);
        printf("\n");
    }
}

int main(void)
{
    int rows = 4;
    int cols = 7;

    int matrix[rows][cols];

    for (int row = 0; row < rows; row++)
        for (int col = 0; col < cols; col++)
            matrix[row][col] = row * col;

    print_matrix(rows, cols, matrix);
}
```

30.5.1 Partial Multidimensional VLAs

You can have some of the dimensions fixed and some variable. Let's say we have a record length fixed at 5 elements, but we don't know how many records there are.

```c
#include <stdio.h>

void print_records(int count, int record[count][5])
{
    for (int i = 0; i < count; i++) {
        for (int j = 0; j < 5; j++)
            printf("%2d ", record[i][j]);
        printf("\n");
    }
}

int main(void)
{
    int rec_count = 3;
    int records[rec_count][5];

    // Fill with some dummy data
    for (int i = 0; i < rec_count; i++)
        for (int j = 0; j < 5; j++)
            records[i][j] = (i+1)*(j+2);

    print_records(rec_count, records);
}
```

30.6 Compatibility with Regular Arrays

Because VLAs are just like regular arrays in memory, it's perfectly permissible to pass them interchangeably… as long as the dimensions match.

For example, if we have a function that specifically wants a 3×5 array, we can still pass a VLA into it.

```c
int foo(int m[5][3]) {...}

\\ ...

int w = 3, h = 5;
int matrix[h][w];

foo(matrix);   // OK!
```

Likewise, if you have a VLA function, you can pass a regular array into it:

```c
int foo(int h, int w, int m[h][w]) {...}

\\ ...

int matrix[3][5];

foo(3, 5, matrix);   // OK!
```

Beware, though: if your dimensions mismatch, you're going to have some undefined behavior going on,

likely.

30.7 typedef and VLAs

You can `typedef` a VLA, but the behavior might not be as you expect.

Basically, `typedef` makes a new type with the values as they existed the moment the `typedef` was executed.

So it's not a `typedef` of a VLA so much as a new fixed size array type of the dimensions at the time.

```c
#include <stdio.h>

int main(void)
{
    int w = 10;

    typedef int goat[w];

    // goat is an array of 10 ints
    goat x;

    // Init with squares of numbers
    for (int i = 0; i < w; i++)
        x[i] = i*i;

    // Print them
    for (int i = 0; i < w; i++)
        printf("%d\n", x[i]);

    // Now let's change w...

    w = 20;

    // But goat is STILL an array of 10 ints, because that was the
    // value of w when the typedef executed.
}
```

So it acts like an array of fixed size.

But you still can't use an initializer list on it.

30.8 Jumping Pitfalls

You have to watch out when using `goto` near VLAs because a lot of things aren't legal.

And when you're using `longjmp()` there's a case where you could leak memory with VLAs.

But both of these things we'll cover in their respective chapters.

30.9 General Issues

VLAs have been banned from the Linux kernel for a few reasons:

- Lots of places they were used should have just been fixed-size.
- The code behind VLAs is slower (to a degree that most people wouldn't notice, but makes a difference in an operating system).

- VLAs are not supported to the same degree by all C compilers.
- Stack size is limited, and VLAs go on the stack. If some code accidentally (or maliciously) passes a large value into a kernel function that allocates a VLA, *Bad Things*™ could happen.

Other folks online point out that there's no way to detect a VLA's failure to allocate, and programs that suffered such problems would likely just crash. While fixed-size arrays also have the same issue, it's far more likely that someone accidentally make a *VLA Of Unusual Size* than somehow accidentally declare a fixed-size, say, 30 megabyte array.

Chapter 31

goto

The goto statement is universally revered and can be here presented without contest.

Just kidding! Over the years, there has been a lot of back-and-forth over whether or not (often not) goto is considered harmful[1].

In this programmer's opinion, you should use whichever constructs leads to the *best* code, factoring in maintainability and speed. And sometimes this might be goto!

In this chapter, we'll see how goto works in C, and then check out some of the common cases where it is used[2].

31.1 A Simple Example

In this example, we're going to use goto to skip a line of code and jump to a *label*. The label is the identifier that can be a goto target—it ends with a colon (:).

```
#include <stdio.h>

int main(void)
{
    printf("One\n");
    printf("Two\n");

    goto skip_3;

    printf("Three\n");

skip_3:

    printf("Five!\n");
}
```

The output is:

```
One
Two
Five!
```

[1] https://en.wikipedia.org/wiki/Goto#Criticism

[2] I'd like to point out that using goto in all these cases is avoidable. You can use variables and loops intead. It's just that some people think goto produces the *best* code in those circumstances.

www.dbooks.org

`goto` sends execution jumping to the specified label, skipping everything in between.

You can jump forward or backward with `goto`.

```
infinite_loop:
    print("Hello, world!\n");
    goto infinite_loop;
```

Labels are skipped over during execution. The following will print all three numbers in order just as if the labels weren't there:

```
    printf("Zero\n");
label_1:
label_2:
    printf("One\n");
label_3:
    printf("Two\n");
label_4:
    printf("Three\n");
```

As you've noticed, it's common convention to justify the labels all the way on the left. This increases readability because a reader can quickly scan to find the destination.

Labels have *function scope*. That is, no matter how many levels deep in blocks they appear, you can still `goto` them from anywhere in the function.

It also means you can only `goto` labels that are in the same function as the `goto` itself. Labels in other functions are out of scope from `goto`'s perspective. And it means you can use the same label name in two functions—just not the same label name in the same function.

31.2 Labeled `continue`

In some languages, you can actually specify a label for a `continue` statement. C doesn't allow it, but you can easily use `goto` instead.

To show the issue, check out `continue` in this nested loop:

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d, %d\n", i, j);
        continue;   // Always goes to next j
    }
}
```

As we see, that `continue`, like all `continues`, goes to the next iteration of the nearest enclosing loop. What if we want to `continue` in the next loop out, the loop with `i`?

Well, we can `break` to get back to the outer loop, right?

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d, %d\n", i, j);
        break;     // Gets us to the next iteration of i
    }
}
```

That gets us two levels of nested loop. But then if we nest another loop, we're out of options. What about this, where we don't have any statement that will get us out to the next iteration of `i`?

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        for (int k = 0; k < 3; k++) {
            printf("%d, %d, %d\n", i, j, k);

            continue;  // Gets us to the next iteration of k
            break;     // Gets us to the next iteration of j
            ????;      // Gets us to the next iteration of i???

        }
    }
}
```

The `goto` statement offers us a way!

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k < 3; k++) {
                printf("%d, %d, %d\n", i, j, k);

                goto continue_i;   // Now continuing the i loop!!
            }
        }
continue_i: ;
    }
```

We have a `;` at the end there—that's because you can't have a label pointing to the plain end of a compound statement (or before a variable declaration).

31.3 Bailing Out

When you're super nested in the middle of some code, you can use `goto` to get out of it in a manner that's often cleaner than nesting more `if`s and using flag variables.

```
    // Pseudocode

    for(...) {
        for (...) {
            while (...) {
                do {
                    if (some_error_condition)
                        goto bail;

                } while(...);
            }
        }
    }

bail:
    // Cleanup here
```

Without `goto`, you'd have to check an error condition flag in all of the loops to get all the way out.

31.4 Labeled break

This is a very similar situation to how `continue` only continues the innermost loop. `break` also only breaks out of the innermost loop.

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d, %d\n", i, j);
            break;   // Only breaks out of the j loop
        }
    }

    printf("Done!\n");
```

But we can use `goto` to break farther:

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d, %d\n", i, j);
            goto break_i;   // Now breaking out of the i loop!
        }
    }

break_i:

    printf("Done!\n");
```

31.5 Multi-level Cleanup

If you're calling multiple functions to initialize multiple systems and one of them fails, you should only de-initialize the ones that you've gotten to so far.

Let's do a fake example where we start initializing systems and checking to see if any returns an error (we'll use `-1` to indicate an error). If one of them does, we have to shutdown only the systems we've initialized so far.

```
    if (init_system_1() == -1)
        goto shutdown;

    if (init_system_2() == -1)
        goto shutdown_1;

    if (init_system_3() == -1)
        goto shutdown_2;

    if (init_system_4() == -1)
        goto shutdown_3;

    do_main_thing();   // Run our program

    shutdown_system4();

shutdown_3:
    shutdown_system3();
```

```
shutdown_2:
    shutdown_system2();

shutdown_1:
    shutdown_system1();

shutdown:
    print("All subsystems shut down.\n");
```

Note that we're shutting down in the reverse order that we initialized the subsystems. So if subsystem 4 fails to start up, it will shut down 3, 2, then 1 in that order.

31.6 Tail Call Optimization

Kinda. For recursive functions only.

If you're unfamiliar, Tail Call Optimization (TCO)[3] is a way to not waste stack space when calling other functions under very specific circumstances. Unfortunately the details are beyond the scope of this guide.

But if you have a recursive function you know can be optimized in this way, you can make use of this technique. (Note that you can't tail call other functions due to the function scope of labels.)

Let's do a straightforward example, factorial.

Here's a recursive version that's not TCO, but it can be!

```
#include <stdio.h>
#include <complex.h>

int factorial(int n, int a)
{
    if (n == 0)
        return a;

    return factorial(n - 1, a * n);
}

int main(void)
{
    for (int i = 0; i < 8; i++)
        printf("%d! == %ld\n", i, factorial(i, 1));
}
```

To make it happen, you can replace the call with two steps:

1. Set the values of the parameters to what they'd be on the next call.
2. `goto` a label on the first line of the function.

Let's try it:

```
#include <stdio.h>

int factorial(int n, int a)
{
tco:  // add this

```

[3] https://en.wikipedia.org/wiki/Tail_call

www.dbooks.org

```
    if (n == 0)
        return a;

    // replace return by setting new parameter values and
    // goto-ing the beginning of the function

    //return factorial(n - 1, a * n);

    int next_n = n - 1;  // See how these match up with
    int next_a = a * n;  // the recursive arguments, above?

    n = next_n;   // Set the parameters to the new values
    a = next_a;

    goto tco;   // And repeat!
}

int main(void)
{
    for (int i = 0; i < 8; i++)
        printf("%d! == %d\n", i, factorial(i, 1));
}
```

I used temporary variables up there to set the next values of the parameters before jumping to the start of the function. See how they correspond to the recursive arguments that were in the recursive call?

Now, why use temp variables? I could have done this instead:

```
    a *= n;
    n -= 1;

    goto tco;
```

and that actually works just fine. But if I carelessly reverse those two lines of code:

```
    n -= 1;  // BAD NEWS
    a *= n;
```

—now we're in trouble. We modified `n` before using it to modify `a`. That's Bad because that's not how it works when you call recursively. Using the temporary variables avoids this problem even if you're not looking out for it. And the compiler likely optimizes them out, anyway.

31.7 Restarting Interrupted System Calls

This is outside the spec, but commonly seen in Unix-like systems.

Certain long-lived system calls might return an error if they're interrupted by a signal, and `errno` will be set to `EINTR` to indicate the syscall was doing fine; it was just interrupted.

In those cases, it's really common for the programmer to want to restart the call and try it again.

```
retry:
    byte_count = read(0, buf, sizeof(buf) - 1);  // Unix read() syscall

    if (byte_count == -1) {            // An error occurred...
        if (errno == EINTR) {          // But it was just interrupted
            printf("Restarting...\n");
```

```
            goto retry;
        }
```

Many Unix-likes have an `SA_RESTART` flag you can pass to `sigaction()` to request the OS automatically restart any slow syscalls instead of failing with `EINTR`.

Again, this is Unix-specific and is outside the C standard.

That said, it's possible to use a similar technique any time any function should be restarted.

31.8 goto and Variable Scope

We've already seen that labels have function scope, but weird things can happen if we jump past some variable initialization.

Look at this example where we jump from a place where the variable `x` is out of scope into the middle of its scope (in the block).

```
    goto label;

    {
        int x = 12345;

label:
        printf("%d\n", x);
    }
```

This will compile and run, but gives me a warning:

```
warning: 'x' is used uninitialized in this function
```

And then it prints out `0` when I run it (your mileage may vary).

Basically what has happened is that we jumped into `x`'s scope (so it was OK to reference it in the `printf()`) but we jumped over the line that actually initialized it to `12345`. So the value was indeterminate.

The fix is, of course, to get the initialization *after* the label one way or another.

```
    goto label;

    {
        int x;

label:
        x = 12345;
        printf("%d\n", x);
    }
```

Let's look at one more example.

```
    {
        int x = 10;

label:

        printf("%d\n", x);
    }

    goto label;
```

What happens here?

The first time through the block, we're good. `x` is `10` and that's what prints.

But after the `goto`, we're jumping into the scope of `x`, but past its initialization. Which means we can still print it, but the value is indeterminate (since it hasn't been reinitialized).

On my machine, it prints `10` again (to infinity), but that's just luck. It could print any value after the `goto` since `x` is uninitialized.

31.9 goto and Variable-Length Arrays

When it comes to VLAs and `goto`, there's one rule: you can't jump from outside the scope of a VLA into the scope of that VLA.

If I try to do this:

```
    int x = 10;

    goto label;

    {
        int v[x];

label:

        printf("Hi!\n");
    }
```

I get an error:

```
error: jump into scope of identifier with variably modified type
```

You can jump in ahead of the VLA declaration, like this:

```
    int x = 10;

    goto label;

    {
label:  ;
        int v[x];

        printf("Hi!\n");
    }
```

Because that way the VLA gets allocated properly before its inevitable deallocation once it falls out of scope.

Chapter 32

Types Part V: Compound Literals and Generic Selections

This is the final chapter for types! We're going to talk about two things:

- How to have "anonymous" unnamed objects and how that's useful.
- How to generate type-dependent code.

They're not particularly related, but don't really each warrant their own chapters. So I crammed them in here like a rebel!

32.1 Compound Literals

This is a neat feature of the language that allows you to create an object of some type on the fly without ever assigning it to a variable. You can make simple types, arrays, `structs`, you name it.

One of the main uses for this is passing complex arguments to functions when you don't want to make a temporary variable to hold the value.

The way you create a compound literal is to put the type name in parentheses, and then put an initializer list after. For example, an unnamed array of `ints`, might look like this:

```
(int []){1,2,3,4}
```

Now, that line of code doesn't do anything on its own. It creates an unnamed array of 4 `ints`, and then throws them away without using them.

We could use a pointer to store a reference to the array…

```
int *p = (int []){1 ,2 ,3 ,4};

printf("%d\n", p[1]);  // 2
```

But that seems a little like a long-winded way to have an array. I mean, we could have just done this[1]:

```
int p[] = {1, 2, 3, 4};

printf("%d\n", p[1]);  // 2
```

So let's take a look at a more useful example.

[1] Which isn't quite the same, since it's an array, not a pointer to an `int`.

www.dbooks.org

32.1.1 Passing Unnamed Objects to Functions

Let's say we have a function to sum an array of `ints`:

```
int sum(int p[], int count)
{
    int total = 0;

    for (int i = 0; i < count; i++)
        total += p[i];

    return total;
}
```

If we wanted to call it, we'd normally have to do something like this, declaring an array and storing values in it to pass to the function:

```
int a[] = {1, 2, 3, 4};

int s = sum(a, 4);
```

But unnamed objects give us a way to skip the variable by passing it directly in (parameter names listed above). Check it out—we're going to replace the variable `a` with an unnamed array that we pass in as the second argument:

```
//                    p[]         count
//            |-----------------|  |
int s = sum((int []){1, 2, 3, 4}, 4);
```

Pretty slick!

32.1.2 Unnamed `structs`

We can do something similar with `structs`.

First, let's do things without unnamed objects. We'll define a `struct` to hold some `x`/`y` coordinates. Then we'll define one, passing in values into its initializer. Finally, we'll pass it to a function to print the values:

```
#include <stdio.h>

struct coord {
    int x, y;
};

void print_coord(struct coord c)
{
    printf("%d, %d\n", c.x, c.y);
}

int main(void)
{
    struct coord t = {.x=10, .y=20};

    print_coord(t);   // prints "10, 20"
}
```

Straightforward enough?

Let's modify it to use an unnamed object instead of the variable `t` we're passing to `print_coord()`.

We'll just take `t` out of there and replace it with an unnamed `struct`:

```
    //struct coord t = {.x=10, .y=20};

    print_coord((struct coord){.x=10, .y=20});   // prints "10, 20"
```

Still works!

32.1.3 Pointers to Unnamed Objects

You might have noticed in the last example that even through we were using a `struct`, we were passing a copy of the `struct` to `print_coord()` as opposed to passing a pointer to the `struct`.

Turns out, we can just take the address of an unnamed object with `&` like always.

This is because, in general, if an operator would have worked on a variable of that type, you can use that operator on an unnamed object of that type.

Let's modify the above code so that we pass a pointer to an unnamed object

```
#include <stdio.h>

struct coord {
    int x, y;
};

void print_coord(struct coord *c)
{
    printf("%d, %d\n", c->x, c->y);
}

int main(void)
{
    //     Note the &
    //          |
    print_coord(&(struct coord){.x=10, .y=20});   // prints "10, 20"
}
```

Additionally, this can be a nice way to pass even pointers to simple objects:

```
// Pass a pointer to an int with value 3490
foo(&(int){3490});
```

Easy as that.

32.1.4 Unnamed Objects and Scope

The lifetime of an unnamed object ends at the end of its scope. The biggest way this could bite you is if you make a new unnamed object, get a pointer to it, and then leave the object's scope. In that case, the pointer will refer to a dead object.

So this is undefined behavior:

```
int *p;

{
    p = &(int){10};
}
```

```
printf("%d\n", *p);  // INVALID: The (int){10} fell out of scope
```

Likewise, you can't return a pointer to an unnamed object from a function. The object is deallocated when it falls out of scope:

```
#include <stdio.h>

int *get3490(void)
{
    // Don't do this
    return &(int){3490};
}

int main(void)
{
    printf("%d\n", *get3490());  // INVALID: (int){3490} fell out of scope
}
```

Just think of their scope like that of an ordinary local variable. You can't return a pointer to a local variable, either.

32.1.5 Silly Unnamed Object Example

You can put any type in there and make an unnamed object.

For example, these are effectively equivalent:

```
int x = 3490;

printf("%d\n", x);              // 3490 (variable)
printf("%d\n", 3490);           // 3490 (constant)
printf("%d\n", (int){3490});    // 3490 (unnamed object)
```

That last one is unnamed, but it's silly. Might as well do the simple one on the line before.

But hopefully that provides a little more clarity on the syntax.

32.2 Generic Selections

This is an expression that allows you to select different pieces of code depending on the *type* of the first argument to the expression.

We'll look at an example in just a second, but it's important to know this is processed at compile time, *not at runtime*. There's no runtime analysis going on here.

The expression begins with `_Generic`, works kinda like a `switch`, and it takes at least two arguments.

The first argument is an expression (or variable[2]) that has a *type*. All expressions have a type. The remaining arguments to `_Generic` are the cases of what to substitute in for the result of the expression if the first argument is that type.

Wat?

Let's try it out and see.

[2] A variable used here *is* an expression.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i;
6      float f;
7      char c;
8
9      char *s = _Generic(i,
10                         int: "that variable is an int",
11                         float: "that variable is a float",
12                         default: "that variable is some type"
13                     );
14
15     printf("%s\n", s);
16 }
```

Check out the `_Generic` expression starting on line 9.

When the compiler sees it, it look at the type of the first argument. (In this example, the type of the variable `i`.) It then looks through the cases for something of that type. And then it substitutes the argument in place of the entire `_Generic` expression.

In this case, `i` is an `int`, so it matches that case. Then the string is substituted in for the expression. So the line turns into this when the compiler sees it:

```
    char *s = "that variable is an int";
```

If the compiler can't find a type match in the `_Generic`, it looks for the optional `default` case and uses that.

If it can't find a type match and there's no `default`, you'll get a compile error. The first expression **must** match one of the types or `default`.

Because it's inconvenient to write `_Generic` over and over, it's often used to make the body of a macro that can be easily repeatedly reused.

Let's make a macro `TYPESTR(x)` that takes an argument and returns a string with the type of the argument.

So `TYPESTR(1)` will return the string `"int"`, for example.

Here we go:

```
#include <stdio.h>

#define TYPESTR(x) _Generic((x), \
                        int: "int", \
                        long: "long", \
                        float: "float", \
                        double: "double", \
                        default: "something else")

int main(void)
{
    int i;
    long l;
    float f;
    double d;
    char c;
```

```
    printf("i is type %s\n", TYPESTR(i));
    printf("l is type %s\n", TYPESTR(l));
    printf("f is type %s\n", TYPESTR(f));
    printf("d is type %s\n", TYPESTR(d));
    printf("c is type %s\n", TYPESTR(c));
}
```

This outputs:

```
i is type int
l is type long
f is type float
d is type double
c is type something else
```

Which should be no surprise, because, like we said, that code in `main()` is replaced with the following when it is compiled:

```
    printf("i is type %s\n", "int");
    printf("l is type %s\n", "long");
    printf("f is type %s\n", "float");
    printf("d is type %s\n", "double");
    printf("c is type %s\n", "something else");
```

And that's exactly the output we see.

Let's do one more. I've included some macros here so that when you run:

```
int i = 10;
char *s = "Foo!";

PRINT_VAL(i);
PRINT_VAL(s);
```

you get the output:

```
i = 10
s = Foo!
```

We'll have to make use of some macro magic to do that.

```
#include <stdio.h>
#include <string.h>

// Macro that gives back a format specifier for a type
#define FMTSPEC(x) _Generic((x), \
                        int: "%d", \
                        long: "%ld", \
                        float: "%f", \
                        double: "%f", \
                        char *: "%s")
                        // TODO: add more types

// Macro that prints a variable in the form "name = value"
#define PRINT_VAL(x) do { \
    char fmt[512]; \
    snprintf(fmt, sizeof fmt, #x " = %s\n", FMTSPEC(x)); \
    printf(fmt, (x)); \
```

```
} while(0)

int main(void)
{
    int i = 10;
    float f = 3.14159;
    char *s = "Hello, world!";

    PRINT_VAL(i);
    PRINT_VAL(f);
    PRINT_VAL(s);
}
```

for the output:

```
i = 10
f = 3.141590
s = Hello, world!
```

We could have crammed that all in one big macro, but I broke it into two to prevent eye bleeding.

Chapter 33

Arrays Part II

We're going to go over a few extra misc things this chapter concerning arrays.

- Type qualifiers with array parameters
- The `static` keyword with array parameters
- Partial multi-dimensional array initializers

They're not super-commonly seen, but we'll peek at them since they're part of the newer spec.

33.1 Type Qualifiers for Arrays in Parameter Lists

If you recall from earlier, these two things are equivalent in function parameter lists:

```
int func(int *p) {...}
int func(int p[]) {...}
```

And you might also recall that you can add type qualifiers to a pointer variable like so:

```
int *const p;
int *volatile p;
int *const volatile p;
// etc.
```

But how can we do that when we're using array notation in your parameter list?

Turns out it goes in the brackets. And you can put the optional count after. The two following lines are equivalent:

```
int func(int *const volatile p) {...}
int func(int p[const volatile]) {...}
int func(int p[const volatile 10]) {...}
```

If you have a multidimensional array, you need to put the type qualifiers in the first set of brackets.

33.2 `static` for Arrays in Parameter Lists

Similarly, you can use the keyword static in the array in a parameter list.

This is something I've never seen in the wild. It is **always** followed by a dimension:

```
int func(int p[static 4]) {...}
```

What this means, in the above example, is the compiler is going to assume that any array you pass to the function will be *at least* 4 elements.

Anything else is undefined behavior.

```
int func(int p[static 4]) {...}

int main(void)
{
    int a[] = {11, 22, 33, 44};
    int b[] = {11, 22, 33, 44, 55};
    int c[] = {11, 22};

    func(a);  // OK! a is 4 elements, the minimum
    func(b);  // OK! b is at least 4 elements
    func(c);  // Undefined behavior! c is under 4 elements!
}
```

This basically sets the minimum size array you can have.

Important note: there is nothing in the compiler that prohibits you from passing in a smaller array. The compiler probably won't warn you, and it won't detect it at runtime.

By putting `static` in there, you're saying, "I double secret PROMISE that I will never pass in a smaller array than this." And the compiler says, "Yeah, fine," and trusts you to not do it.

And then the compiler can make certain code optimizations, safe in the knowledge that you, the programmer, will always do the right thing.

33.3 Equivalent Initializers

C is a little bit, shall we say, *flexible* when it comes to array initializers.

We've already seen some of this, where any missing values are replaced with zero.

For example, we can initialize a 5 element array to `1,2,0,0,0` with this:

```
int a[5] = {1, 2};
```

Or set an array entirely to zero with:

```
int a[5] = {0};
```

But things get interesting when initializing multidimensional arrays.

Let's make an array of 3 rows and 2 columns:

```
int a[3][2];
```

Let's write some code to initialize it and print the result:

```
#include <stdio.h>

int main(void)
{
    int a[3][2] = {
        {1, 2},
        {3, 4},
        {5, 6}
    };
```

```
    for (int row = 0; row < 3; row++) {
        for (int col = 0; col < 2; col++)
            printf("%d ", a[row][col]);
        printf("\n");
    }
}
```

And when we run it, we get the expected:

```
1 2
3 4
5 6
```

Let's leave off some of the initializer elements and see they get set to zero:

```
    int a[3][2] = {
        {1, 2},
        {3},    // Left off the 4!
        {5, 6}
    };
```

which produces:

```
1 2
3 0
5 6
```

Now let's leave off the entire last middle element:

```
    int a[3][2] = {
        {1, 2},
        // {3, 4},   // Just cut this whole thing out
        {5, 6}
    };
```

And now we get this, which might not be what you expect:

```
1 2
5 6
0 0
```

But if you stop to think about it, we only provided enough initializers for two rows, so they got used for the first two rows. And the remaining elements were initialized to zero.

So far so good. Generally, if we leave off parts of the initializer, the compiler sets the corresponding elements to `0`.

But let's get *crazy*.

```
    int a[3][2] = { 1, 2, 3, 4, 5, 6 };
```

What—? That's a 2D array, but it only has a 1D initializer!

Turns out that's legal (though GCC will warn about it with the proper warnings turned on).

Basically, what it does is starts filling in elements in row 0, then row 1, then row 2 from left to right.

So when we print, it prints in order:

```
1 2
3 4
5 6
```

If we leave some off:

```
int a[3][2] = { 1, 2, 3 };
```

they fill with `0`:

```
1 2
3 0
0 0
```

So if you want to fill the whole array with `0`, then go ahead and:

```
int a[3][2] = {0};
```

But my recommendation is if you have a 2D array, use a 2D initializer. It just makes the code more readable. (Except for initializing the whole array with `0`, in which case it's idiomatic to use `{0}` no matter the dimension of the array.)

www.dbooks.org

Chapter 34

Long Jumps with `setjmp`, `longjmp`

We've already seen `goto`, which jumps in function scope. But `longjmp()` allows you to jump back to an earlier point in execution, back to a function that called this one.

There are a lot of limitations and caveats, but this can be a useful function for bailing out from deep in the call stack back up to an earlier state.

In my experience, this is very rarely-used functionality.

34.1 Using `setjmp` and `longjmp`

The dance we're going to do here is to basically put a bookmark in execution with `setjmp()`. Later on, we'll call `longjmp()` and it'll jump back to the earlier point in execution where we set the bookmark with `setjmp()`.

And it can do this even if you've called subfunctions.

Here's a quick demo where we call into functions a couple levels deep and then bail out of it.

We're going to use a file scope variable `env` to keep the *state* of things when we call `setjmp()` so we can restore them when we call `longjmp()` later. This is the variable in which we remember our "place".

The variable `env` is of type `jmp_buf`, an opaque type declared in `<setjmp.h>`.

```c
#include <stdio.h>
#include <setjmp.h>

jmp_buf env;

void depth2(void)
{
    printf("Entering depth 2\n");
    longjmp(env, 3490);           // Bail out
    printf("Leaving depth 2\n");  // This won't happen
}

void depth1(void)
{
    printf("Entering depth 1\n");
    depth2();
    printf("Leaving depth 1\n");  // This won't happen
```

```
}

int main(void)
{
    switch (setjmp(env)) {
      case 0:
          printf("Calling into functions, setjmp() returned 0\n");
          depth1();
          printf("Returned from functions\n");  // This won't happen
          break;

      case 3490:
          printf("Bailed back to main, setjmp() returned 3490\n");
          break;
    }
}
```

When run, this outputs:

```
Calling into functions, setjmp() returned 0
Entering depth 1
Entering depth 2
Bailed back to main, setjmp() returned 3490
```

If you try to take that output and match it up with the code, it's clear there's some really *funky* stuff going on.

One of the most notable things is that `setjmp()` returns *twice*. What the actual frank? What is this sorcery?!

So here's the deal: if `setjmp()` returns `0`, it means that you've successfully set the "bookmark" at that point.

If it returns non-zero, it means you've just returned to the "bookmark" set earlier. (And the value returned is the one you pass to `longjmp()`.)

This way you can tell the difference between setting the bookmark and returning to it later.

So when the code, above, calls `setjmp()` the first time, `setjmp()` *stores* the state in the `env` variable and returns `0`. Later when we call `longjmp()` with that same `env`, it restores the state and `setjmp()` returns the value `longjmp()` was passed.

34.2 Pitfalls

Under the hood, this is pretty straightforward. Typically the *stack pointer* keeps track of the locations in memory that local variables are stored, and the *program counter* keeps track of the address of the currently-executing instruction[1].

So if we want to jump back to an earlier function, it's basically only a matter of restoring the stack pointer and program counter to the values kept in the `jmp_buf` variable, and making sure the return value is set correctly. And then execution will resume there.

But a variety of factors confound this, making a significant number of undefined behavior traps.

34.2.1 The Values of Local Variables

If you want the values of automatic (non-`static` and non-`extern`) local variables to persist in the function that called `setjmp()` after a `longjmp()` happens, you must declare those variables to be `volatile`.

[1] Both "stack pointer" and "program counter" are related to the underlying architecture and C implementation, and are not part of the spec.

www.dbooks.org

Technically, they only have to be `volatile` if they change between the time `setjmp()` is called and `longjmp()` is called[2].

For example, if we run this code:

```
int x = 20;

if (setjmp(env) == 0) {
    x = 30;
}
```

and then later `longjmp()` back, the value of `x` will be indeterminate.

If we want to fix this, `x` must be `volatile`:

```
volatile int x = 20;

if (setjmp(env) == 0) {
    x = 30;
}
```

Now the value will be the correct `30` after a `longjmp()` returns us to this point.

34.2.2 How Much State is Saved?

When you `longjmp()`, execution resumes at the point of the corresponding `setjmp()`. And that's it.

The spec points out that it's just as if you'd jumped back into the function at that point with local variables set to whatever values they had when the `longjmp()` call was made.

Things that aren't restored include, paraphrasing the spec:

- Floating point status flags
- Open files
- Any other component of the abstract machine

34.2.3 You Can't Name Anything `setjmp`

You can't have any `extern` identifiers with the name `setjmp`. Or, if `setjmp` is a macro, you can't undefine it.

Both are undefined behavior.

34.2.4 You Can't `setjmp()` in a Larger Expression

That is, you can't do something like this:

```
if (x == 12 && setjmp(env) == 0) { ... }
```

That's too complex to be allowed by the spec due to the machinations that must occur when unrolling the stack and all that. We can't `longjmp()` back into some complex expression that's only been partially executed.

So there are limits on the complexity of that expression.

- It can be the entire controlling expression of the conditional.

  ```
  if (setjmp(env)) {...}
  ```

[2] The rationale here is that the program might store a value temporarily in a *CPU register* while it's doing work on it. In that timeframe, the register holds the correct value, and the value on the stack might be out of date. Then later the register values would get overwritten and the changes to the variable lost.

```
switch (setjmp(env)) {...}
```

- It can be part of a relational or equality expression, as long as the other operand is an integer constant. And the whole thing is the controlling expression of the conditional.

```
if (setjmp(env) == 0) {...}
```

- The operand to a logical NOT (`!`) operation, being the entire controlling expression.

```
if (!setjmp(env)) {...}
```

- A standalone expression, possibly cast to `void`.

```
setjmp(env);
```

```
(void)setjmp(env);
```

34.2.5 When Can't You `longjmp()`?

It's undefined behavior if:

- You didn't call `setjmp()` earlier
- You called `setjmp()` from another thread
- You called `setjmp()` in the scope of a variable length array (VLA), and execution left the scope of that VLA before `longjmp()` was called.
- The function containing the `setjmp()` exited before `longjmp()` was called.

On that last one, "exited" includes normal returns from the function, as well as the case if another `longjmp()` jumped back to "earlier" in the call stack than the function in question.

34.2.6 You Can't Pass `0` to `longjmp()`

If you try to pass the value `0` to `longjmp()`, it will silently change that value to `1`.

Since `setjmp()` ultimately returns this value, and having `setjmp()` return `0` has special meaning, returning `0` is prohibited.

34.2.7 `longjmp()` and Variable Length Arrays

If you are in scope of a VLA and `longjmp()` out there, the memory allocated to the VLA could leak[3].

Same thing happens if you `longjmp()` back over any earlier functions that had VLAs still in scope.

This is one thing that really bugged me able VLAs—that you could write perfectly legitimate C code that squandered memory. But, hey—I'm not in charge of the spec.

[3] That is, remain allocated until the program ends with no way to free it.

www.dbooks.org

Chapter 35

Incomplete Types

It might surprise you to learn that this builds without error:

```
extern int a[];

int main(void)
{
    struct foo *x;
    union bar *y;
    enum baz *z;
}
```

We never gave a size for `a`. And we have pointers to `structs` `foo`, `bar`, and `baz` that never seem to be declared anywhere.

And the only warnings I get are that `x`, `y`, and `z` are unused.

These are examples of *incomplete types*.

An incomplete type is a type the size (i.e. the size you'd get back from `sizeof`) for which is not known. Another way to think of it is a type that you haven't finished declaring.

You can have a pointer to an incomplete type, but you can't dereference it or use pointer arithmetic on it. And you can't `sizeof` it.

So what can you do with it?

35.1 Use Case: Self-Referential Structures

I only know of one real use case: forward references to `structs` or `unions` with self-referential or co-dependent structures. (I'm going to use `struct` for the rest of these examples, but they all apply equally to `unions`, as well.)

Let's do the classic example first.

But before I do, know this! As you declare a `struct`, the `struct` is incomplete until the closing brace is reached!

```
struct antelope {               // struct antelope is incomplete here
    int leg_count;              // Still incomplete
    float stomach_fullness;     // Still incomplete
    float top_speed;            // Still incomplete
```

```
    char *nickname;            // Still incomplete
};                             // NOW it's complete.
```

So what? Seems sane enough.

But what if we're doing a linked list? Each linked list node needs to have a reference to another node. But how can we create a reference to another node if we haven't finished even declaring the node yet?

C's allowance for incomplete types makes it possible. We can't declare a node, but we *can* declare a pointer to one, even if it's incomplete!

```
struct node {
    int val;
    struct node *next;  // struct node is incomplete, but that's OK!
};
```

Even though the `struct node` is incomplete on line 3, we can still declare a pointer to one[1].

We can do the same thing if we have two different `structs` that refer to each other:

```
struct a {
    struct b *x;  // Refers to a `struct b`
};

struct b {
    struct a *x;  // Refers to a `struct a`
};
```

We'd never be able to make that pair of structures without the relaxed rules for incomplete types.

35.2 Incomplete Type Error Messages

Are you getting errors like these?

```
invalid application of 'sizeof' to incomplete type

invalid use of undefined type

dereferencing pointer to incomplete type
```

Most likely culprit: you probably forgot to `#include` the header file that declares the type.

35.3 Other Incomplete Types

Declaring a `struct` or `union` with no body makes an incomplete type, e.g. `struct foo;`.

`enums` are incomplete until the closing brace.

`void` is an incomplete type.

Arrays declared `extern` with no size are incomplete, e.g.:

```
extern int a[];
```

If it's a non-`extern` array with no size followed by an initializer, it's incomplete until the closing brace of the initializer.

[1] This works because in C, pointers are the same size regardless of the type of data they point to. So the compiler doesn't need to know the size of the `struct node` at this point; it just needs to know the size of a pointer.

35.4 Use Case: Arrays in Header Files

It can be useful to declare incomplete array types in header files. In those cases, the actual storage (where the complete array is declared) should be in a single `.c` file. If you put it in the `.h` file, it will be duplicated every time the header file is included.

So what you can do is make a header file with an incomplete type that refers to the array, like so:

```
// File: bar.h

#ifndef BAR_H
#define BAR_H

extern int my_array[];  // Incomplete type

#endif
```

And the in the `.c` file, actually define the array:

```
// File: bar.c

int my_array[1024];     // Complete type!
```

Then you can include the header from as many places as you'd like, and every one of those places will refer to the same underlying `my_array`.

```
// File: foo.c

#include <stdio.h>
#include "bar.h"    // includes the incomplete type for my_array

int main(void)
{
    my_array[0] = 10;

    printf("%d\n", my_array[0]);
}
```

When compiling multiple files, remember to specify all the `.c` files to the compiler, but not the `.h` files, e.g.:

```
gcc -o foo foo.c bar.c
```

35.5 Completing Incomplete Types

If you have an incomplete type, you can complete it by defining the complete `struct`, `union`, `enum`, or array in the same scope.

```
struct foo;         // incomplete type

struct foo *p;      // pointer, no problem

// struct foo f;    // Error: incomplete type!

struct foo {
    int x, y, z;
};                  // Now the struct foo is complete!
```

```
struct foo f;      // Success!
```

Note that though `void` is an incomplete type, there's no way to complete it. Not that anyone ever thinks of doing that weird thing. But it does explain why you can do this:

```
void *p;             // OK: pointer to incomplete type
```

and not either of these:

```
void v;              // Error: declare variable of incomplete type

printf("%d\n", *p);  // Error: dereference incomplete type
```

The more you know…

Chapter 36

Complex Numbers

A tiny primer on Complex numbers[1] stolen directly from Wikipedia:

> A **complex number** is a number that can be expressed in the form $a + bi$, where a and b are real numbers [i.e. floating point types in C], and i represents the imaginary unit, satisfying the equation $i^2 = -1$. Because no real number satisfies this equation, i is called an imaginary number. For the complex number $a + bi$, a is called the **real part**, and b is called the **imaginary part**.

But that's as far as I'm going to go. We'll assume that if you're reading this chapter, you know what a complex number is and what you want to do with them.

And all we need to cover is C's faculties for doing so.

Turns out, though, that complex number support in a compiler is an *optional* feature. Not all compliant compilers can do it. And the ones that do, might do it to various degrees of completeness.

You can test if your system supports complex numbers with:

```
#ifdef __STDC_NO_COMPLEX__
#error Complex numbers not supported!
#endif
```

Furthermore, there is a macro that indicates adherence to the ISO 60559 (IEEE 754) standard for floating point math with complex numbers, as well as the presence of the `_Imaginary` type.

```
#if __STDC_IEC_559_COMPLEX__ != 1
#error Need IEC 60559 complex support!
#endif
```

More details on that are spelled out in Annex G in the C11 spec.

36.1 Complex Types

To use complex numbers, `#include <complex.h>`.

With that, you get at least two types:

```
_Complex
complex
```

Those both mean the same thing, so you might as well use the prettier `complex`.

[1] https://en.wikipedia.org/wiki/Complex_number

You also get some types for imaginary numbers if you implementation is IEC 60559-compliant:

```
_Imaginary
imaginary
```

These also both mean the same thing, so you might as well use the prettier `imaginary`.

You also get values for the imaginary number i, itself:

```
I
_Complex_I
_Imaginary_I
```

The macro `I` is set to `_Imaginary_I` (if available), or `_Complex_I`. So just use `I` for the imaginary number.

One aside: I've said that if a compiler has `__STDC_IEC_559_COMPLEX__` set to `1`, it must support `_Imaginary` types to be compliant. That's my read of the spec. However, I don't know of a single compiler that actually supports `_Imaginary` even though they have `__STDC_IEC_559_COMPLEX__` set. So I'm going to write some code with that type in here I have no way of testing. Sorry!

OK, so now we know there's a `complex` type, how can we use it?

36.2 Assigning Complex Numbers

Since the complex number has a real and imaginary part, but both of them rely on floating point numbers to store values, we need to also tell C what precision to use for those parts of the complex number.

We do that by just pinning a `float`, `double`, or `long double` to the `complex`, either before or after it.

Let's define a complex number that uses `float` for its components:

```
float complex c;   // Spec prefers this way
complex float c;   // Same thing--order doesn't matter
```

So that's great for declarations, but how do we initialize them or assign to them?

Turns out we get to use some pretty natural notation. Example!

```
double complex x = 5 + 2*I;
double complex y = 10 + 3*I;
```

For $5 + 2i$ and $10 + 3i$, respectively.

36.3 Constructing, Deconstructing, and Printing

We're getting there…

We've already seen one way to write a complex number:

```
double complex x = 5 + 2*I;
```

There's also no problem using other floating point numbers to build it:

```
double a = 5;
double b = 2;
double complex x = a + b*I;
```

There is also a set of macros to help build these. The above code could be written using the `CMPLX()` macro, like so:

```
double complex x = CMPLX(5, 2);
```

As far as I can tell in my research, these are *almost* equivalent:

```
double complex x = 5 + 2*I;
double complex x = CMPLX(5, 2);
```

But the `CMPLX()` macro will handle negative zeros in the imaginary part correctly every time, whereas the other way might convert them to positive zeros. I *think*[2] This seems to imply that if there's a chance the imaginary part will be zero, you should use the macro… but someone should correct me on this if I'm mistaken!

The `CMPLX()` macro works on `double` types. There are two other macros for `float` and `long double`: `CMPLXF()` and `CMPLXL()`. (These "f" and "l" suffixes appear in virtually all the complex-number-related functions.)

Now let's try the reverse: if we have a complex number, how do we break it apart into its real and imaginary parts?

Here we have a couple functions that will extract the real and imaginary parts from the number: `creal()` and `cimag()`:

```
double complex x = 5 + 2*I;
double complex y = 10 + 3*I;

printf("x = %f + %fi\n", creal(x), cimag(x));
printf("y = %f + %fi\n", creal(y), cimag(y));
```

for the output:

```
x = 5.000000 + 2.000000i
y = 10.000000 + 3.000000i
```

Note that the `i` I have in the `printf()` format string is a literal `i` that gets printed—it's not part of the format specifier. Both return values from `creal()` and `cimag()` are `double`.

And as usual, there are `float` and `long double` variants of these functions: `crealf()`, `cimagf()`, `creall()`, and `cimagl()`.

36.4 Complex Arithmetic and Comparisons

Arithmetic can be performed on complex numbers, though how this works mathematically is beyond the scope of the guide.

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 1 + 2*I;
    double complex y = 3 + 4*I;
    double complex z;

    z = x + y;
```

[2]This was a harder one to research, and I'll take any more information anyone can give me. `I` could be defined as `_Complex_I` or `_Imaginary_I`, if the latter exists. `_Imaginary_I` will handle signed zeros, but `_Complex_I` *might* not. This has implications with branch cuts and other complex-numbery-mathy things. Maybe. Can you tell I'm really getting out of my element here? In any case, the `CMPLX()` macros behave as if `I` were defined as `_Imaginary_I`, with signed zeros, even if `_Imaginary_I` doesn't exist on the system.

```
    printf("x + y = %f + %fi\n", creal(z), cimag(z));

    z = x - y;
    printf("x - y = %f + %fi\n", creal(z), cimag(z));

    z = x * y;
    printf("x * y = %f + %fi\n", creal(z), cimag(z));

    z = x / y;
    printf("x / y = %f + %fi\n", creal(z), cimag(z));
}
```

for a result of:

```
x + y = 4.000000 + 6.000000i
x - y = -2.000000 + -2.000000i
x * y = -5.000000 + 10.000000i
x / y = 0.440000 + 0.080000i
```

You can also compare two complex numbers for equality (or inequality):

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 1 + 2*I;
    double complex y = 3 + 4*I;

    printf("x == y = %d\n", x == y);  // 0
    printf("x != y = %d\n", x != y);  // 1
}
```

with the output:

```
x == y = 0
x != y = 1
```

They are equal if both components test equal. Note that as with all floating point, they could be equal if they're close enough due to rounding error[3].

36.5 Complex Math

But wait! There's more than just simple complex arithmetic!

Here's a summary table of all the math functions available to you with complex numbers.

I'm only going to list the `double` version of each function, but for all of them there is a `float` version that you can get by appending `f` to the function name, and a `long double` version that you can get by appending `l`.

For example, the `cabs()` function for computing the absolute value of a complex number also has `cabsf()` and `cabsl()` variants. I'm omitting them for brevity.

[3]The simplicity of this statement doesn't do justice to the incredible amount of work that goes into simply understanding how floating point actually functions. https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

www.dbooks.org

36.5.1 Trigonometry Functions

Function	Description
`ccos()`	Cosine
`csin()`	Sine
`ctan()`	Tangent
`cacos()`	Arc cosine
`casin()`	Arc sine
`catan()`	Play *Settlers of Catan*
`ccosh()`	Hyperbolic cosine
`csinh()`	Hyperbolic sine
`ctanh()`	Hyperbolic tangent
`cacosh()`	Arc hyperbolic cosine
`casinh()`	Arc hyperbolic sine
`catanh()`	Arc hyperbolic tangent

36.5.2 Exponential and Logarithmic Functions

Function	Description
`cexp()`	Base-e exponential
`clog()`	Natural (base-e) logarithm

36.5.3 Power and Absolute Value Functions

Function	Description
`cabs()`	Absolute value
`cpow()`	Power
`csqrt()`	Square root

36.5.4 Manipulation Functions

Function	Description
`creal()`	Return real part
`cimag()`	Return imaginary part
`CMPLX()`	Construct a complex number
`carg()`	Argument/phase angle
`conj()`	Conjugate[4]
`cproj()`	Projection on Riemann sphere

[4] This is the only one that doesn't begin with an extra leading `c`, strangely.

Chapter 37

Fixed Width Integer Types

C has all those small, bigger, and biggest integer types like `int` and `long` and all that. And you can look in the section on limits to see what the largest int is with `INT_MAX` and so on.

How big are those types? That is, how many bytes do they take up? We could use `sizeof` to get that answer.

But what if I wanted to go the other way? What if I needed a type that was exactly 32 bits (4 bytes) or at least 16 bits or somesuch?

How can we declare a type that's a certain size?

The header `<stdint.h>` gives us a way.

37.1 The Bit-Sized Types

For both signed and unsigned integers, we can specify a type that is a certain number of bits, with some caveats, of course.

And there are three main classes of these types (in these examples, the `N` would be replaced by a certain number of bits):

- Integers of exactly a certain size (`intN_t`)
- Integers that are at least a certain size (`int_leastN_t`)
- Integers that are at least a certain size and are as fast as possible (`int_fastN_t`)[1]

How much faster is `fast`? Definitely maybe some amount faster. Probably. The spec doesn't say how much faster, just that they'll be the fastest on this architecture. Most C compilers are pretty good, though, so you'll probably only see this used in places where the most possible speed needs to be guaranteed (rather than just hoping the compiler is producing pretty-dang-fast code, which it is).

Finally, these unsigned number types have a leading `u` to differentiate them.

For example, these types have the corresponding listed meaning:

```
int32_t w;        // x is exactly 32 bits, signed
uint16_t x;       // y is exactly 16 bits, unsigned

int_least8_t y;   // y is at least 8 bits, signed

uint_fast64_t z;  // z is the fastest representation at least 64 bits, unsigned
```

[1] Some architectures have different sized data that the CPU and RAM can operate with at a faster rate than others. In those cases, if you need the fastest 8-bit number, it might give you have a 16- or 32-bit type instead because that's just faster. So with this, you won't know how big the type is, but it will be least as big as you say.

www.dbooks.org

The following types are guaranteed to be defined:

```
int_least8_t      uint_least8_t
int_least16_t     uint_least16_t
int_least32_t     uint_least32_t
int_least64_t     uint_least64_t

int_fast8_t       uint_fast8_t
int_fast16_t      uint_fast16_t
int_fast32_t      uint_fast32_t
int_fast64_t      uint_fast64_t
```

There might be others of different widths, as well, but those are optional.

Hey! Where are the fixed types like `int16_t`? Turns out those are entirely optional…unless certain conditions are met[2]. And if you have an average run-of-the-mill modern computer system, those conditions probably are met. And if they are, you'll have these types:

```
int8_t      uint8_t
int16_t     uint16_t
int32_t     uint32_t
int64_t     uint64_t
```

Other variants with different widths might be defined, but they're optional.

37.2 Maximum Integer Size Type

There's a type you can use that holds the largest representable integers available on the system, both signed and unsigned:

```
intmax_t
uintmax_t
```

Use these types when you want to go as big as possible.

Obviously values from any other integer types of the same sign will fit in this type, necessarily.

37.3 Using Fixed Size Constants

If you have a constant that you want to have fit in a certain number of bits, you can use these macros to automatically append the proper suffix onto the number (e.g. `22L` or `3490ULL`).

```
INT8_C(x)     UINT8_C(x)
INT16_C(x)    UINT16_C(x)
INT32_C(x)    UINT32_C(x)
INT64_C(x)    UINT64_C(x)
INTMAX_C(x)   UINTMAX_C(x)
```

Again, these work only with constant integer values.

For example, we can use one of these to assign constant values like so:

```
uint16_t x = UINT16_C(12);
intmax_t y = INTMAX_C(3490);
```

[2] Namely, the system has 8, 16, 32, or 64 bit integers with no padding that use two's complement representation, in which case the `intN_t` variant for that particular number of bits *must* be defined.

37.4 Limits of Fixed Size Integers

We also have some limits defined so you can get the maximum and minimum values for these types:

```
INT8_MAX            INT8_MIN            UINT8_MAX
INT16_MAX           INT16_MIN           UINT16_MAX
INT32_MAX           INT32_MIN           UINT32_MAX
INT64_MAX           INT64_MIN           UINT64_MAX

INT_LEAST8_MAX      INT_LEAST8_MIN      UINT_LEAST8_MAX
INT_LEAST16_MAX     INT_LEAST16_MIN     UINT_LEAST16_MAX
INT_LEAST32_MAX     INT_LEAST32_MIN     UINT_LEAST32_MAX
INT_LEAST64_MAX     INT_LEAST64_MIN     UINT_LEAST64_MAX

INT_FAST8_MAX       INT_FAST8_MIN       UINT_FAST8_MAX
INT_FAST16_MAX      INT_FAST16_MIN      UINT_FAST16_MAX
INT_FAST32_MAX      INT_FAST32_MIN      UINT_FAST32_MAX
INT_FAST64_MAX      INT_FAST64_MIN      UINT_FAST64_MAX

INTMAX_MAX          INTMAX_MIN          UINTMAX_MAX
```

Note the `MIN` for all the unsigned types is `0`, so, as such, there's no macro for it.

37.5 Format Specifiers

In order to print these types, you need to send the right format specifier to `printf()`. (And the same issue for getting input with `scanf()`.)

But how are you going to know what size the types are under the hood? Luckily, once again, C provides some macros to help with this.

All this can be found in `<inttypes.h>`.

Now, we have a bunch of macros. Like a complexity explosion of macros. So I'm going to stop listing out every one and just put the lowercase letter `n` in the place where you should put `8`, `16`, `32`, or `64` depending on your needs.

Let's look at the macros for printing signed integers:

```
PRIdn    PRIdLEASTn    PRIdFASTn    PRIdMAX
PRIin    PRIiLEASTn    PRIiFASTn    PRIiMAX
```

Look for the patterns there. You can see there are variants for the fixed, least, fast, and max types.

And you also have a lowercase `d` and a lowercase `i`. Those correspond to the `printf()` format specifiers `%d` and `%i`.

So if I have something of type:

```
int_least16_t x = 3490;
```

I can print that with the equivalent format specifier for `%d` by using `PRId16`.

But how? How do we use that macro?

First of all, that macro specifies a string containing the letter or letters `printf()` needs to use to print that type. Like, for example, it could be `"d"` or `"ld"`.

So all we need to do is embed that in our format string to the `printf()` call.

To do this, we can take advantage of a fact about C that you might have forgotten: adjacent string literals are automatically concatenated to a single string. E.g.:

```
printf("Hello, " "world!\n");   // Prints "Hello, world!"
```

And since these macros are string literals, we can use them like so:

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

int main(void)
{
    int_least16_t x = 3490;

    printf("The value is %" PRIdLEAST16 "!\n", x);
}
```

We also have a pile of macros for printing unsigned types:

```
PRIon    PRIoLEASTn    PRIoFASTn    PRIoMAX
PRIun    PRIuLEASTn    PRIuFASTn    PRIuMAX
PRIxn    PRIxLEASTn    PRIxFASTn    PRIxMAX
PRIXn    PRIXLEASTn    PRIXFASTn    PRIXMAX
```

In this case, `o`, `u`, `x`, and `X` correspond to the documented format specifiers in `printf()`.

And, as before, the lowercase `n` should be substituted with `8`, `16`, `32`, or `64`.

But just when you think you had enough of the macros, it turns out we have a complete complementary set of them for `scanf()`!

```
SCNdn    SCNdLEASTn    SCNdFASTn    SCNdMAX
SCNin    SCNiLEASTn    SCNiFASTn    SCNiMAX
SCNon    SCNoLEASTn    SCNoFASTn    SCNoMAX
SCNun    SCNuLEASTn    SCNuFASTn    SCNuMAX
SCNxn    SCNxLEASTn    SCNxFASTn    SCNxMAX
```

Remember: when you want to print out a fixed size integer type with `printf()` or `scanf()`, grab the correct corresponding format specifer from `<inttypes.h>`.

Chapter 38

Date and Time Functionality

> "Time is an illusion. Lunchtime doubly so."
> —Ford Prefect, The Hitchhikers Guide to the Galaxy

This isn't too complex, but it can be a little intimidating at first, both with the different types available and the way we can convert between them.

Mix in GMT (UTC) and local time and we have all the *Usual Fun*™ one gets with times and dates.

And of course never forget the golden rule of dates and times: *Never attempt to write your own date and time functionality. Only use what the library gives you.*

Time is too complex for mere mortal programmers to handle correctly. Seriously, we all owe a point to everyone who worked on any date and time library, so put that in your budget.

38.1 Quick Terminology and Information

Just a couple quick terms in case you don't have them down.

- **UTC**: Coordinated Universal Time is a universally[1] agreed upon, absolute time. Everyone on the planet thinks it's the same time right now in UTC… even though they have different local times.
- **GMT**: Greenwich Mean Time, effectively the same as UTC[2]. You probably want to say UTC, or "universal time". If you're talking specifically about the GMT time zone, say GMT. Confusingly, many of C's UTC functions predate UTC and still refer to Greenwich Mean Time. When you see that, know that C means UTC.
- **Local time**: what time it is where the computer running the program is located. This is described as an offset from UTC. Although there are many time zones in the world, most computers do work in either local time or UTC.

As a general rule, if you are describing an event that happens one time, like a log entry, or a rocket launch, or when pointers finally clicked for you, use UTC.

On the other hand, if it's something that happens the same time *in every time zone*, like New Year's Eve or dinner time, use local time.

Since a lot of languages are only good at converting between UTC and local time, you can cause yourself a lot of pain by choosing to store your dates in the wrong form. (Ask me how I know.)

[1] On Earth, anyway. Who know what crazy systems they use *out there*…

[2] OK, don't murder me! GMT is technically a time zone while UTC is a global time system. Also some countries might adjust GMT for daylight saving time, whereas UTC is never adjusted for daylight saving time.

www.dbooks.org

38.2 Date Types

There are two[3] main types in C when it comes to dates: `time_t` and `struct tm`.

The spec doesn't actually say much about them:

- `time_t`: a real type capable of holding a time. So by the spec, this could be a floating type or integer type. In POSIX (Unix-likes), it's an integer. This holds *calendar time*. Which you can think of as UTC time.
- `struct tm`: holds the components of a calendar time. This is a *broken-down time*, i.e. the components of the time, like hour, minute, second, day, month, year, etc.

On a lot of systems, `time_t` represents the number of seconds since *Epoch*[4]. Epoch is in some ways the start of time from the computer's perspective, which is commonly January 1, 1970 UTC. `time_t` can go negative to represent times before Epoch. Windows behaves the same way as Unix from what I can tell.

And what's in a `struct tm`? The following fields:

```
struct tm {
    int tm_sec;    // seconds after the minute -- [0, 60]
    int tm_min;    // minutes after the hour -- [0, 59]
    int tm_hour;   // hours since midnight -- [0, 23]
    int tm_mday;   // day of the month -- [1, 31]
    int tm_mon;    // months since January -- [0, 11]
    int tm_year;   // years since 1900
    int tm_wday;   // days since Sunday -- [0, 6]
    int tm_yday;   // days since January 1 -- [0, 365]
    int tm_isdst;  // Daylight Saving Time flag
};
```

Note that everything is zero-based except the day of the month.

It's important to know that you can put any values in these types you want. There are functions to help get the time *now*, but the types hold *a* time, not *the* time.

So the question becomes: "How do you initialize data of these types, and how do you convert between them?"

38.3 Initialization and Conversion Between Types

First, you can get the current time and store it in a `time_t` with the `time()` function.

```
time_t now;  // Variable to hold the time now

now = time(NULL);  // You can get it like this...

time(&now);        // ...or this. Same as the previous line.
```

Great! You have a variable that gets you the time now.

Amusingly, there's only one portable way to print out what's in a `time_t`, and that's the rarely-used `ctime()` function that prints the value in local time:

```
now = time(NULL);
printf("%s", ctime(&now));
```

This returns a string with a very specific form that includes a newline at the end:

[3] Admittedly, there are more than two.
[4] https://en.wikipedia.org/wiki/Unix_time

```
Sun Feb 28 18:47:25 2021
```

So that's kind of inflexible. If you want more control, you should convert that `time_t` into a `struct tm`.

38.3.1 Converting `time_t` to `struct tm`

There are two amazing ways to do this conversion:

- `localtime()`: this function converts a `time_t` to a `struct tm` in local time.
- `gmtime()`: this function converts a `time_t` to a `struct tm` in UTC. (See ye olde GMT creeping into that function name?)

Let's see what time it is now by printing out a `struct tm` with the `asctime()` function:

```
printf("Local: %s", asctime(localtime(&now)));
printf("  UTC: %s", asctime(gmtime(&now)));
```

Output (I'm in the Pacific Standard Time zone):

```
Local: Sun Feb 28 20:15:27 2021
  UTC: Mon Mar  1 04:15:27 2021
```

Once you have your `time_t` in a `struct tm`, it opens all kinds of doors. You can print out the time in a variety of ways, figure out which day of the week a date is, and so on. Or convert it back into a `time_t`.

More on that soon!

38.3.2 Converting `struct tm` to `time_t`

If you want to go the other way, you can use `mktime()` to get that information.

`mktime()` sets the values of `tm_wday` and `tm_yday` for you, so don't bother filling them out because they'll just be overwritten.

Also, you can set `tm_isdst` to `-1` to have it make the determination for you. Or you can manually set it to true or false.

```
struct tm some_time = {
    .tm_year=82,   // years since 1900
    .tm_mon=3,     // months since January -- [0, 11]
    .tm_mday=12,   // day of the month -- [1, 31]
    .tm_hour=12,   // hours since midnight -- [0, 23]
    .tm_min=00,    // minutes after the hour -- [0, 59]
    .tm_sec=04,    // seconds after the minute -- [0, 60]
    .tm_isdst=-1,  // Daylight Saving Time flag
};

time_t some_time_epoch;

some_time_epoch = mktime(&some_time);

printf("%s", ctime(&some_time_epoch));
printf("Is DST: %d\n", some_time.tm_isdst);
```

Output:

```
Mon Apr 12 12:00:04 1982
Is DST: 0
```

When you manually load a `struct tm` like that, it should be in local time. `mktime()` will convert that local time into a `time_t` calendar time.

Weirdly, however, the standard doesn't give us a way to load up a `struct tm` with a UTC time and convert that to a `time_t`. If you want to do that with Unix-likes, try the non-standard `timegm()`. On Windows, `_mkgmtime()`.

38.4 Formatted Date Output

We've already seen a couple ways to print formatted date output to the screen. With `time_t` we can use `ctime()`, and with `struct tm` we can use `asctime()`.

```
time_t now = time(NULL);
struct tm *local = localtime(&now);
struct tm *utc = gmtime(&now);

printf("Local time: %s", ctime(&now));     // Local time with time_t
printf("Local time: %s", asctime(local));  // Local time with struct tm
printf("UTC       : %s", asctime(utc));    // UTC with a struct tm
```

But what if I told you, dear reader, that there's a way to have much more control over how the date was printed?

Sure, we could fish individual fields out of the `struct tm`, but there's a great function called `strftime()` that will do a lot of the hard work for you. It's like `printf()`, except for dates!

Let's see some examples. In each of these, we pass in a destination buffer, a maximum number of characters to write, and then a format string (in the style of—but not the same as—`printf()`) which tells `strftime()` which components of a `struct tm` to print and how.

You can add other constant characters to include in the output in the format string, as well, just like with `printf()`.

We get a `struct tm` in this case from `localtime()`, but any source works fine.

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    char s[128];
    time_t now = time(NULL);

    // %c: print date as per current locale
    strftime(s, sizeof s, "%c", localtime(&now));
    puts(s);   // Sun Feb 28 22:29:00 2021

    // %A: full weekday name
    // %B: full month name
    // %d: day of the month
    strftime(s, sizeof s, "%A, %B %d", localtime(&now));
    puts(s);   // Sunday, February 28

    // %I: hour (12 hour clock)
    // %M: minute
    // %S: second
    // %p: AM or PM
```

```
    strftime(s, sizeof s, "It's %I:%M:%S %p", localtime(&now));
    puts(s);   // It's 10:29:00 PM

    // %F: ISO 8601 yyyy-mm-dd
    // %T: ISO 8601 hh:mm:ss
    // %z: ISO 8601 time zone offset
    strftime(s, sizeof s, "ISO 8601: %FT%T%z", localtime(&now));
    puts(s);   // ISO 8601: 2021-02-28T22:29:00-0800
}
```

There are a *ton* of date printing format specifiers for `strftime()`, so be sure to check them out in the `strftime()` reference page.

38.5 More Resolution with `timespec_get()`

You can get the number of seconds and nanoseconds since Epoch with `timespec_get()`.

Maybe.

Implementations might not have nanosecond resolution (that's one billionth of a second) so who knows how many significant places you'll get, but give it a shot and see.

`timespec_get()` takes two arguments. One is a pointer to a `struct timespec` to hold the time information. And the other is the `base`, which the spec lets you set to `TIME_UTC` indicating that you're interested in seconds since Epoch. (Other implementations might give you more options for the `base`.)

And the structure itself has two fields:

```
struct timespec {
    time_t tv_sec;   // Seconds
    long   tv_nsec;  // Nanoseconds (billionths of a second)
};
```

Here's an example where we get the time and print it out both as integer values and also a floating value:

```
struct timespec ts;

timespec_get(&ts, TIME_UTC);

printf("%ld s, %ld ns\n", ts.tv_sec, ts.tv_nsec);

double float_time = ts.tv_sec + ts.tv_nsec/1000000000.0;
printf("%f seconds since epoch\n", float_time);
```

Example output:

```
1614581530 s, 806325800 ns
1614581530.806326 seconds since epoch
```

`struct timespec` also makes an appearance in a number of the threading functions that need to be able to specify time with that resolution.

38.6 Differences Between Times

One quick note about getting the difference between two `time_ts`: since the spec doesn't dictate how that type represents a time, you might not be able to simply subtract two `time_ts` and get anything sensible[5].

[5] You will on POSIX, where `time_t` is definitely an integer. Unfortunately the entire world isn't POSIX, so there we are.

Luckily you can use `difftime()` to compute the difference in seconds between two dates.

In the following example, we have two events that occur some time apart, and we use `difftime()` to compute the difference.

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    struct tm time_a = {
        .tm_year=82,   // years since 1900
        .tm_mon=3,     // months since January -- [0, 11]
        .tm_mday=12,   // day of the month -- [1, 31]
        .tm_hour=4,    // hours since midnight -- [0, 23]
        .tm_min=00,    // minutes after the hour -- [0, 59]
        .tm_sec=04,    // seconds after the minute -- [0, 60]
        .tm_isdst=-1,  // Daylight Saving Time flag
    };

    struct tm time_b = {
        .tm_year=120,  // years since 1900
        .tm_mon=10,    // months since January -- [0, 11]
        .tm_mday=15,   // day of the month -- [1, 31]
        .tm_hour=16,   // hours since midnight -- [0, 23]
        .tm_min=27,    // minutes after the hour -- [0, 59]
        .tm_sec=00,    // seconds after the minute -- [0, 60]
        .tm_isdst=-1,  // Daylight Saving Time flag
    };

    time_t cal_a = mktime(&time_a);
    time_t cal_b = mktime(&time_b);

    double diff = difftime(cal_b, cal_a);

    double years = diff / 60 / 60 / 24 / 365.2425;  // close enough

    printf("%f seconds (%f years) between events\n", diff, years);
}
```

Output:

```
1217996816.000000 seconds (38.596783 years) between events
```

And there you have it! Remember to use `difftime()` to take the time difference. Even though you can just subtract on a POSIX system, might as well stay portable.

>]

Chapter 39

Multithreading

C11 introduced, formally, multithreading to the C language. It's very eerily similar to POSIX threads[1], if you've ever used those.

And if you're not, no worries. We'll talk it through.

Do note, however, that I'm not intending this to be a full-blown classic multithreading how-to[2]; you'll have to pick up a different very thick book for that, specifically. Sorry!

Threading is an optional feature. If a C11+ compiler defines `__STDC_NO_THREADS__`, threads will **not** be present in the library. Why they decided to go with a negative sense in that macro is beyond me, but there we are.

You can test for it like this:

```
#ifdef __STDC_NO_THREADS__
#error I need threads to build this program!
#endif
```

Also, you might need to specify certain linker options when building. In the case of Unix-likes, try appending a `-lpthreads` to the end of the command line to link the `pthreads` library[3]:

```
gcc -std=c11 -o foo foo.c -lpthreads
```

If you're getting linker errors on your system, it could be because the appropriate library wasn't included.

39.1 Background

Threads are a way to have all those shiny CPU cores you paid for do work for you in the same program.

Normally, a C program just runs on a single CPU core. But if you know how to split up the work, you can give pieces of it to a number of threads and have them do the work simultaneously.

Though the spec doesn't say it, on your system it's very likely that C (or the OS at its behest) will attempt to balance the threads over all your CPU cores.

And if you have more threads than cores, that's OK. You just won't realize all those gains if they're all trying to compete for CPU time.

[1] https://en.wikipedia.org/wiki/POSIX_Threads

[2] I'm more a fan of shared-nothing, myself, and my skills with classic multithreading constructs are rusty, to say the least.

[3] Yes, `pthreads` with a "p". It's short for POSIX threads, a library that C11 borrowed liberally from for its threads implementation.

www.dbooks.org

39.2 Things You Can Do

You can create a thread. It will begin running the function you specify. The parent thread that spawned it will also continue to run.

And you can wait for the thread to complete. This is called *joining*.

Or if you don't care when the thread completes and don't want to wait, you can *detach it*.

A thread can explicitly *exit*, or it can implicitly call it quits by returning from its main function.

A thread can also *sleep* for a period of time, doing nothing while other threads run.

The `main()` program is a thread, as well.

Additionally, we have thread local storage, mutexes, and conditional variables. But more on those later. Let's just look at the basics for now.

39.3 Data Races and the Standard Library

Some of the functions in the standard library (e.g. `asctime()` and `strtok()`) return or use `static` data elements that aren't threadsafe. But in general unless it's said otherwise, the standard library makes an effort to be so[4].

But keep an eye out. If a standard library function is maintaining state between calls in a variable you don't own, or if a function is returning a pointer to a thing that you didn't pass in, it's not threadsafe.

39.4 Creating and Waiting for Threads

Let's hack something up!

We'll make some threads (create) and wait for them to complete (join).

We have a tiny bit to understand first, though.

Every single thread is identified by an opaque variable of type `thrd_t`. It's a unique identifier per thread in your program. When you create a thread, it's given a new ID.

Also when you make the thread, you have to give it a pointer to a function to run, and a pointer to an argument to pass to it (or `NULL` if you don't have anything to pass).

The thread will begin execution on the function you specify.

When you want to wait for a thread to complete, you have to specify its thread ID so C knows which one to wait for.

So the basic idea is:

1. Write a function to act as the thread's "`main`". It's not `main()`-proper, but analogous to it. The thread will start running there.
2. From the main thread, launch a new thread with `thrd_create()`, and pass it a pointer to the function to run.
3. In that function, have the thread do whatever it has to do.
4. Meantimes, the main thread can continue doing whatever *it* has to do.
5. When the main thread decides to, it can wait for the child thread to complete by calling `thrd_join()`. Generally you **must** `thrd_join()` the thread to clean up after it or else you'll leak memory[5]

[4] Per §7.1.4¶5.

[5] Unless you `thrd_detach()`. More on this later.

`thrd_create()` takes a pointer to the function to run, and it's of type `thrd_start_t`, which is `int (*)(void *)`. That's Greek for "a pointer to a function that takes an `void*` as an argument, and returns an `int`."

Let's make a thread! We'll launch it from the main thread with `thrd_create()` to run a function, do some other things, then wait for it to complete with `thrd_join()`. I've named the thread's main function `run()`, but you can name it anything as long as the types match `thrd_start_t`.

```c
#include <stdio.h>
#include <threads.h>

// This is the function the thread will run. It can be called anything.
//
// arg is the argument pointer passed to `thrd_create()`.
//
// The parent thread will get the return value back from `thrd_join()`'
// later.

int run(void *arg)
{
    int *a = arg;  // We'll pass in an int* from thrd_create()

    printf("THREAD: Running thread with arg %d\n", *a);

    return 12;  // Value to be picked up by thrd_join() (chose 12 at random)
}

int main(void)
{
    thrd_t t;  // t will hold the thread ID
    int arg = 3490;

    printf("Launching a thread\n");

    // Launch a thread to the run() function, passing a pointer to 3490
    // as an argument. Also stored the thread ID in t:

    thrd_create(&t, run, &arg);

    printf("Doing other things while the thread runs\n");

    printf("Waiting for thread to complete...\n");

    int res;  // Holds return value from the thread exit

    // Wait here for the thread to complete; store the return value
    // in res:

    thrd_join(t, &res);

    printf("Thread exited with return value %d\n", res);
}
```

See how we did the `thrd_create()` there to call the `run()` function? Then we did other things in `main()` and then stopped and waited for the thread to complete with `thrd_join()`.

Sample output (yours might vary):

```
Launching a thread
Doing other things while the thread runs
Waiting for thread to complete...
THREAD: Running thread with arg 3490
Thread exited with return value 12
```

The `arg` that you pass to the function has to have a lifetime long enough so that the thread can pick it up before it goes away. Also, it needs to not be overwritten by the main thread before the new thread can use it.

Let's look at an example that launches 5 threads. One thing to note here is how we use an array of `thrd_ts` to keep track of all the thread IDs.

```c
#include <stdio.h>
#include <threads.h>

int run(void *arg)
{
    int i = *(int*)arg;

    printf("THREAD %d: running!\n", i);

    return i;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    int i;

    printf("Launching threads...\n");
    for (i = 0; i < THREAD_COUNT; i++)

        // NOTE! In the following line, we pass a pointer to i,
        // but each thread sees the same pointer. So they'll
        // print out weird things as i changes value here in
        // the main thread! (More in the text, below.)

        thrd_create(t + i, run, &i);

    printf("Doing other things while the thread runs...\n");
    printf("Waiting for thread to complete...\n");

    for (int i = 0; i < THREAD_COUNT; i++) {
        int res;
        thrd_join(t[i], &res);

        printf("Thread %d complete!\n", res);
    }

    printf("All threads complete!\n");
}
```

When I run the threads, I count `i` up from 0 to 4. And pass a pointer to it to `thrd_create()`. This pointer ends up in the `run()` routine where we make a copy of it.

Simple enough? Here's the output:

```
Launching threads...
THREAD 2: running!
THREAD 3: running!
THREAD 4: running!
THREAD 2: running!
Doing other things while the thread runs...
Waiting for thread to complete...
Thread 2 complete!
Thread 2 complete!
THREAD 5: running!
Thread 3 complete!
Thread 4 complete!
Thread 5 complete!
All threads complete!
```

Whaaa—? Where's `THREAD 0`? And why do we have a `THREAD 5` when clearly `i` is never more than `4` when we call `thrd_create()`? And two `THREAD 2`s? Madness!

This is getting into the fun land of *race conditions*. The main thread is modifying `i` before the thread has a chance to copy it. Indeed, `i` makes it all the way to `5` and ends the loop before the last thread gets a chance to copy it.

We've got to have a per-thread variable that we can refer to so we can pass it in as the `arg`.

We could have a big array of them. Or we could `malloc()` space (and free it somewhere—maybe in the thread itself.)

Let's give that a shot:

```c
#include <stdio.h>
#include <stdlib.h>
#include <threads.h>

int run(void *arg)
{
    int i = *(int*)arg;  // Copy the arg

    free(arg);  // Done with this

    printf("THREAD %d: running!\n", i);

    return i;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    int i;

    printf("Launching threads...\n");
```

```
25      for (i = 0; i < THREAD_COUNT; i++) {
26
27          // Get some space for a per-thread argument:
28
29          int *arg = malloc(sizeof *arg);
30          *arg = i;
31
32          thrd_create(t + i, run, arg);
33      }
34
35      // ...
```

Notice on lines 27-30 we `malloc()` space for an `int` and copy the value of `i` into it. Each new thread gets its own freshly-`malloc()`d variable and we pass a pointer to that to the `run()` function.

Once `run()` makes its own copy of the `arg` on line 7, it `free()`s the `malloc()`d `int`. And now that it has its own copy, it can do with it what it pleases.

And a run shows the result:

```
Launching threads...
THREAD 0: running!
THREAD 1: running!
THREAD 2: running!
THREAD 3: running!
Doing other things while the thread runs...
Waiting for thread to complete...
Thread 0 complete!
Thread 1 complete!
Thread 2 complete!
Thread 3 complete!
THREAD 4: running!
Thread 4 complete!
All threads complete!
```

There we go! Threads 0-4 all in effect!

Your run might vary—how the threads get scheduled to run is beyond the C spec. We see in the above example that thread 4 didn't even begin until threads 0-1 had completed. Indeed, if I run this again, I likely get different output. We cannot guarantee a thread execution order.

39.5 Detaching Threads

If you want to fire-and-forget a thread (i.e. so you don't have to `thrd_join()` it later), you can do that with `thrd_detach()`.

This removes the parent thread's ability to get the return value from the child thread, but if you don't care about that and just want threads to clean up nicely on their own, this is the way to go.

Basically we're going to do this:

```
thrd_create(&t, run, NULL);
thrd_detach(t);
```

where the `thrd_detach()` call is the parent thread saying, "Hey, I'm not going to wait for this child thread to complete with `thrd_join()`. So go ahead and clean it up on your own when it completes."

```c
#include <stdio.h>
#include <threads.h>

int run(void *arg)
{
    (void)arg;

    //printf("Thread running! %lu\n", thrd_current()); // non-portable!
    printf("Thread running!\n");

    return 0;
}

#define THREAD_COUNT 10

int main(void)
{
    thrd_t t;

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_create(&t, run, NULL);
        thrd_detach(t);               // <-- DETACH!
    }

    // Sleep for a second to let all the threads finish
    thrd_sleep(&(struct timespec){.tv_sec=1}, NULL);
}
```

Note that in this code, we put the main thread to sleep for 1 second with `thrd_sleep()`—more on that later.

Also in the `run()` function, I have a commented-out line in there that prints out the thread ID as an `unsigned long`. This is non-portable, because the spec doesn't say what type a `thrd_t` is under the hood—it could be a `struct` for all we know. But that line works on my system.

Something interesting I saw when I ran the code, above, and printed out the thread IDs was that some threads had duplicate IDs! This seems like it should be impossible, but C is allowed to *reuse* thread IDs after the corresponding thread has exited. So what I was seeing was that some threads completed their run before other threads were launched.

39.6 Thread Local Data

Threads are interesting because they don't have their own memory beyond local variables. If you want a `static` variable or file scope variable, all threads will see that same variable.

This can lead to race conditions, where you get *Weird Things*™ happening.

Check out this example. We have a `static` variable `foo` in block scope in `run()`. This variable will be visible to all threads that pass through the `run()` function. And the various threads can effectively step on each others toes.

Each thread copies `foo` into a local variable `x` (which is not shared between threads—all the threads have their own call stacks). So they *should* be the same, right?

And the first time we print them, they are[6]. But then right after that, we check to make sure they're still the

[6]Though I don't think they have to be. It's just that the threads don't seem to get rescheduled until some system call like might happen with a `printf()`… which is why I have the `printf()` in there.

www.dbooks.org

same.

And they *usually* are. But not always!

```c
#include <stdio.h>
#include <stdlib.h>
#include <threads.h>

int run(void *arg)
{
    int n = *(int*)arg;  // Thread number for humans to differentiate

    free(arg);

    static int foo = 10;  // Static value shared between threads

    int x = foo;  // Automatic local variable--each thread has its own

    // We just assigned x from foo, so they'd better be equal here.
    // (In all my test runs, they were, but even this isn't guaranteed!)

    printf("Thread %d: x = %d, foo = %d\n", n, x, foo);

    // And they should be equal here, but they're not always!
    // (Sometimes they were, sometimes they weren't!)

    // What happens is another thread gets in and increments foo
    // right now, but this thread's x remains what it was before!

    if (x != foo) {
        printf("Thread %d: Craziness! x != foo! %d != %d\n", n, x, foo);
    }

    foo++;  // Increment shared value

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Holds a thread serial number
        *n = i;
        thrd_create(t + i, run, n);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }
}
```

Here's an example output (though this varies from run to run):

```
Thread 0: x = 10, foo = 10
Thread 1: x = 10, foo = 10
Thread 1: Craziness! x != foo! 10 != 11
Thread 2: x = 12, foo = 12
Thread 4: x = 13, foo = 13
Thread 3: x = 14, foo = 14
```

In thread 1, between the two `printf()`s, the value of `foo` somehow changed from `10` to `11`, even though clearly there's no increment between the `printf()`s!

It was another thread that got in there (probably thread 0, from the look of it) and incremented the value of `foo` behind thread 1's back!

Let's solve this problem two different ways. (If you want all the threads to share the variable *and* not step on each other's toes, you'll have to read on to the mutex section.)

39.6.1 `_Thread_local` Storage-Class

First things first, let's just look at the easy way around this: the `_Thread_local` storage-class.

Basically we're just going to slap this on the front of our block scope `static` variable and things will work! It tells C that every thread should have its own version of this variable, so none of them step on each other's toes.

The `<threads.h>` header defines `thread_local` as an alias to `_Thread_local` so your code doesn't have to look so ugly.

Let's take the previous example and make `foo` into a `thread_local` variable so that we don't share that data.

```
int run(void *arg)
{
    int n = *(int*)arg;  // Thread number for humans to differentiate

    free(arg);

    thread_local static int foo = 10;  // <-- No longer shared!!
```

And running we get:

```
Thread 0: x = 10, foo = 10
Thread 1: x = 10, foo = 10
Thread 2: x = 10, foo = 10
Thread 4: x = 10, foo = 10
Thread 3: x = 10, foo = 10
```

No more weird problems!

One thing: if a `thread_local` variable is block scope, it **must** be `static`. Them's the rules. (But this is OK because non-`static` variables are per-thread already since each thread has it's own non-`static` variables.)

A bit of a lie there: block scope `thread_local` variables can also be `extern`.

39.6.2 Another Option: Thread-Specific Storage

Thread-specific storage (TSS) is another way of getting per-thread data.

One additional feature is that these functions allow you to specify a destructor that will be called on the data when the TSS variable is deleted. Commonly this destructor is `free()` to automatically clean up `malloc()`d per-thread data. Or `NULL` if you don't need to destroy anything.

The destructor is type `tss_dtor_t` which is a pointer to a function that returns `void` and takes a `void*` as an argument (the `void*` points to the data stored in the variable). In other words, it's a `void (*)(void*)`, if that clears it up. Which I admit it probably doesn't. Check out the example, below.

Generally, `thread_local` is probably your go-to, but if you like the destructor idea, then you can make use of that.

The usage is a bit weird in that we need a variable of type `tss_t` to be alive to represent the value on a per thread basis. Then we initialize it with `tss_create()`. Eventually we get rid of it with `tss_delete()`. Note that calling `tss_delete()` doesn't run all the destructors—it's `thrd_exit()` (or returning from the run function) that does that. `tss_delete()` just releases any memory allocated by `tss_create()`.

In the middle, threads can call `tss_set()` and `tss_get()` to set and get the value.

In the following code, we set up the TSS variable before creating the threads, then clean up after the threads.

In the `run()` function, the threads `malloc()` some space for a string and store that pointer in the TSS variable.

When the thread exits, the destructor function (`free()` in this case) is called for *all* the threads.

```c
#include <stdio.h>
#include <stdlib.h>
#include <threads.h>

tss_t str;

void some_function(void)
{
    // Retrieve the per-thread value of this string
    char *tss_string = tss_get(str);

    // And print it
    printf("TSS string: %s\n", tss_string);
}

int run(void *arg)
{
    int serial = *(int*)arg;  // Get this thread's serial number
    free(arg);

    // malloc() space to hold the data for this thread
    char *s = malloc(64);
    sprintf(s, "thread %d! :)", serial);  // Happy little string

    // Set this TSS variable to point at the string
    tss_set(str, s);

    // Call a function that will get the variable
    some_function();

    return 0;   // Equivalent to thrd_exit(0)
}

#define THREAD_COUNT 15
```

```

int main(void)
{
    thrd_t t[THREAD_COUNT];

    // Make a new TSS variable, the free() function is the destructor
    tss_create(&str, free);

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Holds a thread serial number
        *n = i;
        thrd_create(t + i, run, n);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }

    // All threads are done, so we're done with this
    tss_delete(str);
}
```

Again, this is kind of a painful way of doing things compared to `thread_local`, so unless you really need that destructor functionality, I'd use that instead.

39.7 Mutexes

If you want to only allow a single thread into a critical section of code at a time, you can protect that section with a mutex[7].

For example, if we had a `static` variable and we wanted to be able to get and set it in two operations without another thread jumping in the middle and corrupting it, we could use a mutex for that.

You can acquire a mutex or release it. If you attempt to acquire the mutex and succeed, you may continue execution. If you attempt and fail (because someone else holds it), you will *block*[8] until the mutex is released.

If multiple threads are blocked waiting for a mutex to be released, one of them will be chosen to run (at random, from our perspective), and the others will continue to sleep.

The gameplan is that first we'll initialize a mutex variable to make it ready to use with `mtx_init()`.

Then subsequent threads can call `mtx_lock()` and `mtx_unlock()` to get and release the mutex.

When we're completely done with the mutex, we can destroy it with `mtx_destroy()`, the logical opposite of `mtx_init()`.

First, let's look at some code that does *not* use a mutex, and endeavors to print out a shared (`static`) serial number and then increment it. Because we're not using a mutex over the getting of the value (to print it) and the setting (to increment it), threads might get in each other's way in that critical section.

```
#include <stdio.h>
#include <threads.h>

int run(void *arg)
{
```

[7] Short for "mutual exclusion", AKA a "lock" on a section of code that only one thread is permitted to execute.
[8] That is, your process will go to sleep.

www.dbooks.org

```
    (void)arg;

    static int serial = 0;   // Shared static variable!

    printf("Thread running! %d\n", serial);

    serial++;

    return 0;
}

#define THREAD_COUNT 10

int main(void)
{
    thrd_t t[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_create(t + i, run, NULL);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }
}
```

When I run this, I get something that looks like this:

```
Thread running! 0
Thread running! 0
Thread running! 0
Thread running! 3
Thread running! 4
Thread running! 5
Thread running! 6
Thread running! 7
Thread running! 8
Thread running! 9
```

Clearly multiple threads are getting in there and running the `printf()` before anyone gets a change to update the `serial` variable.

What we want to do is wrap the getting of the variable and setting of it into a single mutex-protected stretch of code.

We'll add a new variable to represent the mutex of type `mtx_t` in file scope, initialize it, and then the threads can lock and unlock it in the `run()` function.

```
#include <stdio.h>
#include <threads.h>

mtx_t serial_mtx;     // <-- MUTEX VARIABLE

int run(void *arg)
{
    (void)arg;
```

```
9
10      static int serial = 0;   // Shared static variable!
11
12      // Acquire the mutex--all threads will block on this call until
13      // they get the lock:
14
15      mtx_lock(&serial_mtx);           // <-- ACQUIRE MUTEX
16
17      printf("Thread running! %d\n", serial);
18
19      serial++;
20
21      // Done getting and setting the data, so free the lock. This will
22      // unblock threads on the mtx_lock() call:
23
24      mtx_unlock(&serial_mtx);         // <-- RELEASE MUTEX
25
26      return 0;
27  }
28
29  #define THREAD_COUNT 10
30
31  int main(void)
32  {
33      thrd_t t[THREAD_COUNT];
34
35      // Initialize the mutex variable, indicating this is a normal
36      // no-frills, mutex:
37
38      mtx_init(&serial_mtx, mtx_plain);        // <-- CREATE MUTEX
39
40      for (int i = 0; i < THREAD_COUNT; i++) {
41          thrd_create(t + i, run, NULL);
42      }
43
44      for (int i = 0; i < THREAD_COUNT; i++) {
45          thrd_join(t[i], NULL);
46      }
47
48      // Done with the mutex, destroy it:
49
50      mtx_destroy(&serial_mtx);                // <-- DESTROY MUTEX
51  }
```

See how on lines 38 and 50 of `main()` we initialize and destroy the mutex.

But each individual thread acquires the mutex on line 15 and releases it on line 24.

In between the `mtx_lock()` and `mtx_unlock()` is the *critical section*, the area of code where we don't want multiple threads mucking about at the same time.

And now we get proper output!

```
Thread running! 0
Thread running! 1
Thread running! 2
```

```
Thread running! 3
Thread running! 4
Thread running! 5
Thread running! 6
Thread running! 7
Thread running! 8
Thread running! 9
```

If you need multiple mutexes, no problem: just have multiple mutex variables.

And always remember the Number One Rule of Multiple Mutexes: *Unlock mutexes in the opposite order in which you lock them!*

39.7.1 Different Mutex Types

As hinted earlier, we have a few mutex types that you can create with `mtx_init()`. (Some of these types are the result of a bitwise-OR operation, as noted in the table.)

Type	Description
`mtx_plain`	Regular ol' mutex
`mtx_timed`	Mutex that supports timeouts
`mtx_plain|mtx_recursive`	Recursive mutex
`mtx_timed|mtx_recursive`	Recursive mutex that supports timeouts

"Recursive" means that the holder of a lock can call `mtx_lock()` multiple times on the same lock. (They have to unlock it an equal number of times before anyone else can take the mutex.) This might ease coding from time to time, especially if you call a function that needs to lock the mutex when you already hold the mutex.

And the timeout gives a thread a chance to *try* to get the lock for a while, but then bail out if it can't get it in that timeframe.

For a timeout mutex, be sure to create it with `mtx_timed`:

```
mtx_init(&serial_mtx, mtx_timed);
```

And then when you wait for it, you have to specify a time in UTC when it will unlock[9].

The function `timespec_get()` from `<time.h>` can be of assistance here. It'll get you the current time in UTC in a `struct timespec` which is just what we need. In fact, it seems to exist merely for this purpose.

It has two fields: `tv_sec` has the current time in seconds since epoch, and `tv_nsec` has the nanoseconds (billionths of a second) as the "fractional" part.

So you can load that up with the current time, and then add to it to get a specific timeout.

Then call `mtx_timedlock()` instead of `mtx_lock()`. If it returns the value `thrd_timedout`, it timed out.

```
struct timespec timeout;

timespec_get(&timeout, TIME_UTC);  // Get current time
timeout.tv_sec += 1;               // Timeout 1 second after now

int result = mtx_timedlock(&serial_mtx, &timeout));
```

[9] You might have expected it to be "time from now", but you'd just like to think that, wouldn't you!

```
if (result == thrd_timedout) {
    printf("Mutex lock timed out!\n");
}
```

Other than that, timed locks are the same as regular locks.

39.8 Condition Variables

Condition Variables are the last piece of the puzzle we need to make performant multithreaded applications and to compose more complex multithreaded structures.

A condition variable provides a way for threads to go to sleep until some event on another thread occurs.

In other words, we might have a number of threads that are rearing to go, but they have to wait until some event is true before they continue. Basically they're being told "wait for it!" until they get notified.

And this works hand-in-hand with mutexes since what we're going to wait on generally depends on the value of some data, and that data generally needs to be protected by a mutex.

It's important to note that the condition variable itself isn't the holder of any particular data from our perspective. It's merely the variable by which C keeps track of the waiting/not-waiting status of a particular thread or group of threads.

Let's write a contrived program that reads in groups of 5 numbers from the main thread one at a time. Then, when 5 numbers have been entered, the child thread wakes up, sums up those 5 numbers, and prints the result.

The numbers will be stored in a global, shared array, as will the index into the array of the about-to-be-entered number.

Since these are shared values, we at least have to hide them behind a mutex for both the main and child threads. (The main will be writing data to them and the child will be reading data from them.)

But that's not enough. The child thread needs to block ("sleep") until 5 numbers have been read into the array. And then the parent thread needs to wake up the child thread so it can do its work.

And when it wakes up, it needs to be holding that mutex. And it will! When a thread waits on a condition variable, it also acquires a mutex when it wakes up.

All this takes place around an additional variable of type `cnd_t` that is the *condition variable*. We create this variable with the `cnd_init()` function and destroy it when we're done with it with the `cnd_destroy()` function.

But how's this all work? Let's look at the outline of what the child thread will do:

1. Lock the mutex with `mtx_lock()`
2. If we haven't entered all the numbers, wait on the condition variable with `cnd_wait()`
3. Do the work that needs doing
4. Unlock the mutex with `mtx_unlock()`

Meanwhile the main thread will be doing this:

1. Lock the mutex with `mtx_lock()`
2. Store the recently-read number into the array
3. If the array is full, signal the child to wake up with `cnd_signal()`
4. Unlock the mutex with `mtx_unlock()`

If you didn't skim that too hard (it's OK—I'm not offended), you might notice something weird: how can the main thread hold the mutex lock and signal the child, if the child has to hold the mutex lock to wait for the signal? They can't both hold the lock!

www.dbooks.org

And indeed they don't! There's some behind-the-scenes magic with condition variables: when you `cnd_wait()`, it releases the mutex that you specify and the thread goes to sleep. And when someone signals that thread to wake up, it reacquires the lock as if nothing had happened.

It's a little different on the `cnd_signal()` side of things. This doesn't do anything with the mutex. The signaling thread still must manually release the mutex before the waiting threads can wake up.

One more thing on the `cnd_wait()`. You'll probably be calling `cnd_wait()` if some condition[10] is not yet met (e.g. in this case, if not all the numbers have yet been entered). Here's the deal: this condition should be in a `while` loop, not an `if` statement. Why?

It's because of a mysterious phenomenon called a *spurious wakeup*. Sometimes, in some implementations, a thread can be woken up out of a `cnd_wait()` sleep for seemingly *no reason*. *[X-Files music]*[11]. And so we have to check to see that the condition we need is still actually met when we wake up. And if it's not, back to sleep with us!

So let's do this thing! Starting with the main thread:

- The main thread will set up the mutex and condition variable, and will launch the child thread.
- Then it will, in an infinite loop, get numbers as input from the console.
- It will also acquire the mutex to store the inputted number into a global array.
- When the array has 5 numbers in it, the main thread will signal the child thread that it's time to wake up and do its work.
- Then the main thread will unlock the mutex and go back to reading the next number from the console.

Meanwhile, the child thread has been up to its own shenanigans:

- The child thread grabs the mutex
- While the condition is not met (i.e. while the shared array doesn't yet have 5 numbers in it), the child thread sleeps by waiting on the condition variable. When it waits, it implicitly unlocks the mutex.
- Once the main thread signals the child thread to wake up, it wakes up to do the work and gets the mutex lock back.
- The child thread sums the numbers and resets the variable that is the index into the array.
- It then releases the mutex and runs again in an infinite loop.

And here's the code! Give it some study so you can see where all the above pieces are being handled:

```
#include <stdio.h>
#include <threads.h>

#define VALUE_COUNT_MAX 5

int value[VALUE_COUNT_MAX];  // Shared global
int value_count = 0;   // Shared global, too

mtx_t value_mtx;   // Mutex around value
cnd_t value_cnd;   // Condition variable on value

int run(void *arg)
{
    (void)arg;
```

[10] And that's why they're called *condition variables*!

[11] I'm not saying it's aliens… but it's aliens. OK, really more likely another thread might have been woken up and gotten to the work first.

```

    for (;;) {
        mtx_lock(&value_mtx);      // <-- GRAB THE MUTEX

        while (value_count < VALUE_COUNT_MAX) {
            printf("Thread: is waiting\n");
            cnd_wait(&value_cnd, &value_mtx);  // <-- CONDITION WAIT
        }

        printf("Thread: is awake!\n");

        int t = 0;

        // Add everything up
        for (int i = 0; i < VALUE_COUNT_MAX; i++)
            t += value[i];

        printf("Thread: total is %d\n", t);

        // Reset input index for main thread
        value_count = 0;

        mtx_unlock(&value_mtx);   // <-- MUTEX UNLOCK
    }

    return 0;
}

int main(void)
{
    thrd_t t;

    // Spawn a new thread

    thrd_create(&t, run, NULL);
    thrd_detach(t);

    // Set up the mutex and condition variable

    mtx_init(&value_mtx, mtx_plain);
    cnd_init(&value_cnd);

    for (;;) {
        int n;

        scanf("%d", &n);

        mtx_lock(&value_mtx);    // <-- LOCK MUTEX

        value[value_count++] = n;

        if (value_count == VALUE_COUNT_MAX) {
            printf("Main: signaling thread\n");
            cnd_signal(&value_cnd);  // <-- SIGNAL CONDITION
```

```
        }

        mtx_unlock(&value_mtx);  // <-- UNLOCK MUTEX
    }

    // Clean up (I know that's an infinite loop above here, but I
    // want to at least pretend to be proper):

    mtx_destroy(&value_mtx);
    cnd_destroy(&value_cnd);
}
```

And here's some sample output (individual numbers on lines are my input):

```
Thread: is waiting
1
1
1
1
1
Main: signaling thread
Thread: is awake!
Thread: total is 5
Thread: is waiting
2
8
5
9
0
Main: signaling thread
Thread: is awake!
Thread: total is 24
Thread: is waiting
```

It's a common use of condition variables in producer-consumer situations like this. If we didn't have a way to put the child thread to sleep while it waited for some condition to be met, it would be force to poll which is a big waste of CPU.

39.8.1 Timed Condition Wait

There's a variant of `cnd_wait()` that allows you to specify a timeout so you can stop waiting.

Since the child thread must relock the mutex, this doesn't necessarily mean that you'll be popping back to life the instant the timeout occurs; you still must wait for any other threads to release the mutex.

But it does mean that you won't be waiting until the `cnd_signal()` happens.

To make this work, call `cnd_timedwait()` instead of `cnd_wait()`. If it returns the value `thrd_timedout`, it timed out.

The timestamp is an absolute time in UTC, not a time-from-now. Thankfully the `timespec_get()` function in `<time.h>` seems custom-made for exactly this case.

```
struct timespec timeout;

timespec_get(&timeout, TIME_UTC);  // Get current time
timeout.tv_sec += 1;               // Timeout 1 second after now
```

```
int result = cnd_timedwait(&condition, &mutex, &timeout));

if (result == thrd_timedout) {
    printf("Condition variable timed out!\n");
}
```

39.8.2 Broadcast: Wake Up All Waiting Threads

`cnd_signal()` function]] `cnd_signal()` only wakes up one thread to continue working. Depending on how you have your logic done, it might make sense to wake up more than one thread to continue once the condition is met.

Of course only one of them can grab the mutex, but if you have a situation where:

- The newly-awoken thread is responsible for waking up the next one, and—
- There's a chance the spurious-wakeup loop condition will prevent it from doing so, then—

you'll want to broadcast the wake up so that you're sure to get at least one of the threads out of that loop to launch the next one.

How, you ask?

Simply use `cnd_broadcast()` instead of `cnd_signal()`. Exact same usage, except `cnd_broadcast()` wakes up **all** the sleeping threads that were waiting on that condition variable.

39.9 Running a Function One Time

Let's say you have a function that *could* be run by many threads, but you don't know when, and it's not work trying to write all that logic.

There's a way around it: use `call_once()`. Tons of threads could try to run the function, but only the first one counts[12]

To work with this, you need a special flag variable you declare to keep track of whether or not the thing's been run. And you need a function to run, which takes no parameters and returns no value.

```
once_flag of = ONCE_FLAG_INIT;  // Initialize it like this

void run_once_function(void)
{
    printf("I'll only run once!\n");
}

int run(void *arg)
{
    (void)arg;

    call_once(&of, run_once_function);

    // ...
```

In this example, no matter how many threads get to the `run()` function, the `run_once_function()` will only be called a single time.

[12] Survival of the fittest! Right? I admit it's actually nothing like that.

www.dbooks.org

Chapter 40

Atomics

"They tried and failed, all of them?"
"Oh, no." She shook her head. "They tried and died."

—Paul Atreides and The Reverend Mother Gaius Helen Mohiam, *Dune*

This is one of the more challenging aspects of multithreading with C. But we'll try to take it easy.

Basically, I'll talk about the more straightforward uses of atomic variables, what they are, and how they work, etc. And I'll mention some of the more insanely-complex paths that are available to you.

But I won't go down those paths. Not only am I barely qualified to even write about them, but I figure if you know you need them, you already know more than I do.

But there are some weird things out here even in the basics. So buckle your seatbelts, everyone, 'cause Kansas is goin' bye-bye.

40.1 Testing for Atomic Support

Atomics are an optional feature. There's a macro `__STDC_NO_ATOMICS__` that's `1` if you *don't* have atomics.

That macro might not exist pre-C11, so we should test the language version with `__STDC_VERSION__`[1].

```
#if __STDC_VERSION__ < 201112L || __STDC_NO_ATOMICS__ == 1
#define HAS_ATOMICS 0
#else
#define HAS_ATOMICS 1
#endif
```

If those tests pass, then you can safely include `<stdatomic.h>`, the header on which the rest of this chapter is based. But if there is no atomic support, that header might not even exist.

On some systems, you might need to add `-latomic` to the end of your compilation command line to use any functions in the header file.

40.2 Atomic Variables

Here's *part* of how atomic variables work:

If you have a shared atomic variable and you write to it from one thread, that write will be *all-or-nothing* in a different thread.

[1] The `__STDC_VERSION__` macro didn't exist in early C89, so if you're worried about that, check it with `#ifdef`.

That is, the other thread will see the entire write of, say, a 32-bit value. Not half of it. There's no way for one thread to interrupt another that is in the *middle* of an atomic multi-byte write.

It's almost like there's a little lock around the getting and setting of that one variable. (And there *might* be! See Lock-Free Atomic Variables, below.)

And on that note, you can get away with never using atomics if you use mutexes to lock your critical sections. It's just that there are a class of *lock-free data structures* that always allow other threads to make progress instead of being blocked by a mutex… but these are tough to create correctly from scratch, and are one of the things that are beyond the scope of the guide, sadly.

That's only part of the story. But it's the part we'll start with.

Before we go further, how do you declare a variable to be atomic?

First, include `<stdatomic.h>`.

This gives us types such as `atomic_int`.

And then we can simply declare variables to be of that type.

But let's do a demo where we have two threads. The first runs for a while and then sets a variable to a specific value, then exits. The other runs until it sees that value get set, and then it exits.

```
#include <stdio.h>
#include <threads.h>
#include <stdatomic.h>

atomic_int x;   // THE POWER OF ATOMICS! BWHAHAHA!

int thread1(void *arg)
{
    (void)arg;

    printf("Thread 1: Sleeping for 1.5 seconds\n");
    thrd_sleep(&(struct timespec){.tv_sec=1, .tv_nsec=500000000}, NULL);

    printf("Thread 1: Setting x to 3490\n");
    x = 3490;

    printf("Thread 1: Exiting\n");
    return 0;
}

int thread2(void *arg)
{
    (void)arg;

    printf("Thread 2: Waiting for 3490\n");
    while (x != 3490) {}  // spin here

    printf("Thread 2: Got 3490--exiting!\n");
    return 0;
}

int main(void)
{
    x = 0;
```

```

    thrd_t t1, t2;

    thrd_create(&t1, thread1, NULL);
    thrd_create(&t2, thread2, NULL);

    thrd_join(t1, NULL);
    thrd_join(t2, NULL);

    printf("Main    : Threads are done, so x better be 3490\n");
    printf("Main    : And indeed, x == %d\n", x);
}
```

The second thread spins in place, looking at the flag and waiting for it to get set to the value `3490`. And the first one does that.

And I get this output:

```
Thread 1: Sleeping for 1.5 seconds
Thread 2: Waiting for 3490
Thread 1: Setting x to 3490
Thread 1: Exiting
Thread 2: Got 3490--exiting!
Main    : Threads are done, so x better be 3490
Main    : And indeed, x == 3490
```

Look, ma! We're accessing a variable from different threads and not using a mutex! And that'll work every time thanks to the atomic nature of atomic variables.

You might be wondering what happens if that's a regular non-atomic `int`, instead. Well, on my system it still works… unless I do an optimized build in which case it hangs on thread 2 waiting to see the 3490 to get set[2].

But that's just the beginning of the story. The next part is going to require more brain power and has to do with something called *synchronization*.

40.3 Synchronization

The next part of our story is all about when certain memory writes in one thread become visible to those in another thread.

You might think, it's right away, right? But it's not. A number of things can go wrong. Weirdly wrong.

The compiler might have rearranged memory accesses so that when you think you set a value relative to another might not be true. And even if the compiler didn't, your CPU might have done it on the fly. Or maybe there's something else about this architecture that causes writes on one CPU to be delayed before they're visible on another.

The good news is that we can condense all these potential troubles into one: unsynchronized memory accesses can appear out of order depending on which thread is doing the observing, as if the lines of code themselves had been rearranged.

By way of example, which happens first in the following code, the write to `x` or the write to `y`?

[2]The reason for this is when optimized, my compiler has put the value of `x` in a register to make the `while` loop fast. But the register has no way of knowing that the variable was updated in another thread, so it never sees the `3490`. This isn't really related to the *all-or-nothing* part of atomicity, but is more related to the synchronization aspects in the next section.

```
1  int x, y;  // global
2
3  // ...
4
5  x = 2;
6  y = 3;
7
8  printf("%d %d\n", x, y);
```

Answer: we don't know. The compiler or CPU could silently reverse lines 5 and 6 and we'd be none-the-wiser. The code would run single-threaded *as-if* it were executed in code order.

In a multithreaded scenario, we might have something like this pseudocode:

```
int x = 0, y = 0;

thread1() {
    x = 2;
    y = 3;
}

thread2() {
    while (y != 3) {}  // spin
    printf("x is now %d\n", x);  // 2? ...or 0?
}
```

What is the output from thread 2?

Well, if `x` gets assigned `2` *before* `y` is assigned `3`, then I'd expect the output to be the very sensible:

```
x is now 2
```

But something sneaky could rearrange lines 4 and 5 causing us to see the value of `0` for `x` when we print it.

In other words, all bets are off unless we can somehow say, "As of this point, I expect all previous writes in another thread to be visible in this thread."

Two threads *synchronize* when they agree on the state of shared memory. As we've seen, they're not always in agreement with the code. So how do they agree?

Using atomic variables can force the agreement[3]. If a thread writes to an atomic variable, it's saying "anyone who reads this atomic variable in the future will also see all the changes I made to memory (atomic or not) up to and including the atomic variable".

Or, in more human terms, let's sit around the conference table and make sure we're on the same page as to which pieces of shared memory hold what values. You agree that the memory changes that you'd made up-to-and-including the atomic store will be visible to me after I do a load of the same atomic variable.

So we can easily fix our example:

```
int x = 0;
atomic int y = 0;  // Make y atomic

thread1() {
    x = 2;
    y = 3;              // Synchronize on write
}

```

[3]Until I say otherwise, I'm speaking generally about *sequentially consistent* operations. More on what that means soon.

```
thread2() {
    while (y != 3) {}  // Synchronize on read
    printf("x is now %d\n", x);  // 2, period.
}
```

Because the threads synchronize across `y`, all writes in thread 1 that happened *before* the write to `y` are visible in thread 2 *after* the read from `y` (in the `while` loop).

It's important to note a couple things here:

1. Nothing sleeps. The synchronization is not a blocking operation. Both threads are running full bore until they exit. Even the one stuck in the spin loop isn't blocking anyone else from running.
2. The synchronization happens when one thread reads an atomic variable another thread wrote. So when thread 2 reads `y`, all previous memory writes in thread 1 (namely setting `x`) will be visible in thread 2.
3. Notice that `x` isn't atomic. That's OK because we're not synchronizing over `x`, and the synchronization over `y` when we write it in thread 1 means that all previous writes—including `x`—in thread 1 will become visible to other threads… if those other threads read `y` to synchronize.

Forcing this synchronization is inefficient and can be a lot slower than just using a regular variable. This is why we don't use atomics unless we have to for a particular application.

So that's the basics. Let's look deeper.

40.4 Acquire and Release

More terminology! It'll pay off to learn this now.

When a thread reads an atomic variable, it is said to be an *acquire* operation.

When a thread writes an atomic variable, it is said to be a *release* operation.

What are these? Let's line them up with terms you already know when it comes to atomic variables:

Read = Load = Acquire. Like when you compare an atomic variable or read it to copy it to another value.

Write = Store = Release. Like when you assign a value into an atomic variable.

When using atomic variables with these acquire/release semantics, C spells out what can happen when.

Acquire/release form the basis for the synchronization we just talked about.

When a thread acquires an atomic variable, it can see values set in another thread that released that same variable.

In other words:

When a thread reads an atomic variable, it can see values set in another thread that wrote to that same variable.

The synchronization happens across the acquire/release pair.

More details:

With read/load/acquire of a particular atomic variable:

- All writes (atomic or non-atomic) in another thread that happened before that other thread wrote/stored/released this atomic variable are now visible in this thread.
- The new value of the atomic variable set by the other thread is also visible in this thread.
- No reads or writes of any variables/memory in the current thread can be reordered to happen before this acquire.

- The acquire acts as a one-way barrier when it comes to code reordering; reads and writes in the current thread can be moved down from *after* the release to *before* it. But, more importantly for synchronization, nothing can move down from *after* the acquire to *before* it.

With write/store/release of a particular atomic variable:

- All writes (atomic or non-atomic) in the current thread that happened before this release become visible to other threads that have read/loaded/acquired the same atomic variable.
- The value written to this atomic variable by this thread is also visible to other threads.
- No reads or writes of any variables/memory in the current thread can be reordered to happen after this release.
- The release acts as a one-way barrier when it comes to code reordering: reads and writes in the current thread can be moved up from *after* the release to *before* it. But, more importantly for synchronization, nothing can move down from *before* the release to *after* it.

Again, the upshot is synchronization of memory from one thread to another. The second thread can be sure that variables and memory are written in the order the programmer intended.

```
int x, y, z = 0;
atomic_int a = 0;

thread1() {
    x = 10;
    y = 20;
    a = 999;  // Release
    z = 30;
}

thread2()
{
    while (a != 999) { } // Acquire

    assert(x == 10);  // never asserts, x is always 10
    assert(y == 20);  // never asserts, y is always 20

    assert(z == 0);  // might assert!!
}
```

In the above example, `thread2` can be sure of the values in `x` and `y` after it acquires `a` because they were set before `thread1` released the atomic `a`.

But `thread2` can't be sure of `z`'s value because it happened after the release. Maybe the assignment to `z` got moved before the assignment to `a`.

An important note: releasing one atomic variable has no effect on acquires of different atomic variables. Each variable is isolated from the others.

40.5 Sequential Consistency

You hanging in there? We're through the meat of the simpler usage of atomics. And since we're not even going to talk about the more complex uses here, you can relax a bit.

Sequential consistency is what's called a *memory ordering*. There are many memory orderings, but sequential consistency is the sanest[4] C has to offer. It is also the default. You have to go out of your way to use other

[4] Sanest from a programmer perspective.

www.dbooks.org

memory orderings.

All the stuff we've been talking about so far has happened within the realm of sequential consistency.

We've talked about how the compiler or CPU can rearrange memory reads and writes in a single thread as long as it follows the *as-if* rule.

And we've seen how we can put the brakes on this behavior by synchronizing over atomic variables.

Let's formalize just a little more.

If operations are *sequentially consistent*, it means at the end of the day, when all is said and done, all the threads can kick up their feet, open their beverage of choice, and all agree on the order in which memory changes occurred during the run. And that order is the one specified by the code.

One won't say, "But didn't *B* happen before *A*?" if the rest of them say, "*A* definitely happened before *B*". They're all friends, here.

In particular, within a thread, none of the acquires and releases can be reordered with respect to one another. This is in addition to the rules about what other memory accesses can be reordered around them.

This rule gives an additional level of sanity to the progression of atomic loads/acquires and stores/releases.

Every other memory order in C involves a relaxation of the reordering rules, either for acquires/releases or other memory accesses, atomic or otherwise. You'd do that if you *really* knew what you were doing and needed the speed boost. *Here be armies of dragons…*

More on that later, but for now, let's stick to the safe and practical.

40.6 Atomic Assignments and Operators

Certain operators on atomic variables are atomic. And others aren't.

Let's start with a counter-example:

```
atomic_int x = 0;

thread1() {
    x = x + 3;  // NOT atomic!
}
```

Since there's a read of `x` on the right hand side of the assignment and a write effectively on the left, these are two operations. Another thread could sneak in the middle and make you unhappy.

But you *can* use the shorthand `+=` to get an atomic operation:

```
atomic_int x = 0;

thread1() {
    x += 3;   // ATOMIC!
}
```

In that case, `x` will be atomically incremented by `3`—no other thread can jump in the middle.

In particular, the following operators are atomic read-modify-write operations with sequential consistency, so use them with gleeful abandon. (In the example, `a` is atomic.)

```
a++       a--       --a       ++a
a += b    a -= b    a *= b    a /= b    a %= b
a &= b    a |= b    a ^= b    a >>= b   a <<= b
```

40.7 Library Functions that Automatically Synchronize

So far we've talked about how you can synchronize with atomic variables, but it turns out there are a few library functions that do some limited behind-the-scenes synchronization, themselves.

```
call_once()      thrd_create()      thrd_join()
mtx_lock()       mtx_timedlock()    mtx_trylock()
malloc()         calloc()           realloc()
aligned_alloc()
```

`call_once()`: synchronizes with all subsequent calls to `call_once()` for a particular flag. This way subsequent calls can rest assured that in another thread set the flag, they will see it.

`thrd_create()`: synchronizes with the beginning of the new thread. The new thread can be sure it will see all shared memory writes from the parent thread from before the `thrd_create()` call.

`thrd_join()`: when a thread dies, it synchronizes with this function. The thread that has called `thrd_join()` can be assured that it can see all the late thread's shared writes.

`mtx_lock()`: earlier calls to `mtx_unlock()` on the same mutex synchronize on this call. This is the case that most mirrors the acquire/release process we've already talked about. `mtx_unlock()` performs a release on the mutex variable, assuring any subsequent thread that makes an acquire with `mtx_lock()` can see all the shared memory changes in the critical section.

`mtx_timedlock()` and **`mtx_trylock()`**: similar to the situation with `mtx_lock()`, if this call succeeds, earlier calls to `mtx_unlock()` synchronize with this one.

Dynamic Memory Functions: if you allocate memory, it synchronizes with the previous deallocation of that same memory. And allocations and deallocations of that particular memory region happen in a single total order that all threads can agree upon. I *think* the idea here is that the deallocation can wipe the region if it chooses, and we want to be sure that a subsequent allocation doesn't see the non-wiped data. Someone let me know if there's more to it.

40.8 Atomic Type Specifier, Qualifier

Let's take it down a notch and see what types we have available, and how we can even make new atomic types.

First things first, let's look at the built-in atomic types and what they are `typedef`'d to. (Spoiler: `_Atomic` is a type qualifier!)

Atomic type	Longhand equivalent
`atomic_bool`	`_Atomic _Bool`
`atomic_char`	`_Atomic char`
`atomic_schar`	`_Atomic signed char`
`atomic_uchar`	`_Atomic unsigned char`
`atomic_short`	`_Atomic short`
`atomic_ushort`	`_Atomic unsigned short`
`atomic_int`	`_Atomic int`
`atomic_uint`	`_Atomic unsigned int`
`atomic_long`	`_Atomic long`
`atomic_ulong`	`_Atomic unsigned long`
`atomic_llong`	`_Atomic long long`
`atomic_ullong`	`_Atomic unsigned long long`
`atomic_char16_t`	`_Atomic char16_t`
`atomic_char32_t`	`_Atomic char32_t`

www.dbooks.org

Atomic type	Longhand equivalent
`atomic_wchar_t`	`_Atomic wchar_t`
`atomic_int_least8_t`	`_Atomic int_least8_t`
`atomic_uint_least8_t`	`_Atomic uint_least8_t`
`atomic_int_least16_t`	`_Atomic int_least16_t`
`atomic_uint_least16_t`	`_Atomic uint_least16_t`
`atomic_int_least32_t`	`_Atomic int_least32_t`
`atomic_uint_least32_t`	`_Atomic uint_least32_t`
`atomic_int_least64_t`	`_Atomic int_least64_t`
`atomic_uint_least64_t`	`_Atomic uint_least64_t`
`atomic_int_fast8_t`	`_Atomic int_fast8_t`
`atomic_uint_fast8_t`	`_Atomic uint_fast8_t`
`atomic_int_fast16_t`	`_Atomic int_fast16_t`
`atomic_uint_fast16_t`	`_Atomic uint_fast16_t`
`atomic_int_fast32_t`	`_Atomic int_fast32_t`
`atomic_uint_fast32_t`	`_Atomic uint_fast32_t`
`atomic_int_fast64_t`	`_Atomic int_fast64_t`
`atomic_uint_fast64_t`	`_Atomic uint_fast64_t`
`atomic_intptr_t`	`_Atomic intptr_t`
`atomic_uintptr_t`	`_Atomic uintptr_t`
`atomic_size_t`	`_Atomic size_t`
`atomic_ptrdiff_t`	`_Atomic ptrdiff_t`
`atomic_intmax_t`	`_Atomic intmax_t`
`atomic_uintmax_t`	`_Atomic uintmax_t`

Use those at will! They're consistent with the atomic aliases found in C++, if that helps.

But what if you want more?

You can do it either with a type qualifier or type specifier.

First, specifier! It's the keyword `_Atomic` with a type in parens after[5]—suitable for use with `typedef`:

```
typedef _Atomic(double) atomic_double;

atomic_double f;
```

Restrictions on the specifier: the type you're making atomic can't be of type array or function, nor can it be atomic or otherwise qualified.

Next, qualifier! It's the keyword `_Atomic` *without* a type in parens.

So these do similar things[6]:

```
_Atomic(int) i;   // type specifier
_Atomic int  j;   // type qualifier
```

The thing is, you can include other type qualifiers with the latter:

```
_Atomic volatile int k;   // qualified atomic variable
```

Restrictions on the qualifier: the type you're making atomic can't be of type array or function.

[5] Apparently C++23 is adding this as a macro.
[6] The spec notes that they might differ in size, representation, and alignment.

40.9 Lock-Free Atomic Variables

Hardware architectures are limited in the amount of data they can atomically read and write. It depends on how it's wired together. And it varies.

If you use an atomic type, you can be assured that accesses to that type will be atomic… but there's a catch: if the hardware can't do it, it's done with a lock, instead.

So the atomic access becomes lock-access-unlock, which is rather slower and has some implications for signal handlers.

Atomic flags, below, is the only atomic type that is guaranteed to be lock-free in all conforming implementations. In typical desktop/laptop computer world, other larger types are likely lock-free.

Luckily, we have a couple ways to determine if a particular type is a lock-free atomic or not.

First of all, some macros—you can use these at compile time with `#if`. They apply to both signed and unsigned types.

Atomic Type	Lock Free Macro
`atomic_bool`	`ATOMIC_BOOL_LOCK_FREE`
`atomic_char`	`ATOMIC_CHAR_LOCK_FREE`
`atomic_char16_t`	`ATOMIC_CHAR16_T_LOCK_FREE`
`atomic_char32_t`	`ATOMIC_CHAR32_T_LOCK_FREE`
`atomic_wchar_t`	`ATOMIC_WCHAR_T_LOCK_FREE`
`atomic_short`	`ATOMIC_SHORT_LOCK_FREE`
`atomic_int`	`ATOMIC_INT_LOCK_FREE`
`atomic_long`	`ATOMIC_LONG_LOCK_FREE`
`atomic_llong`	`ATOMIC_LLONG_LOCK_FREE`
`atomic_intptr_t`	`ATOMIC_POINTER_LOCK_FREE`

These macros can interestingly have *three* different values:

Value	Meaning
`0`	Never lock-free.
`1`	*Sometimes* lock-free.
`2`	Always lock-free.

Wait—how can something be *sometimes* lock-free? This just means the answer isn't known at compile-time, but could later be known at runtime. Maybe the answer varies depending on whether or not you're running this code on Genuine Intel or AMD, or something like that[7].

But you can always test at runtime with the `atomic_is_lock_free()` function. This function returns true or false if the particular type is atomic right now.

So why do we care?

Lock-free is faster, so maybe there's a speed concern that you'd code around another way. Or maybe you need to use an atomic variable in a signal handler.

[7] I just pulled that example out of nowhere. Maybe it doesn't matter on Intel/AMD, but it could matter somewhere, dangit!

www.dbooks.org

40.9.1 Signal Handlers and Lock-Free Atomics

If you read or write a shared variable (static storage duration or `_Thread_Local`) in a signal handler, it's undefined behavior [gasp!]… Unless you do one of the following:

1. Write to a variable of type `volatile sig_atomic_t`.
2. Read or write a lock-free atomic variable.

As far as I can tell, lock-free atomic variables are one of the few ways you get portably get information out of a signal handler.

The spec is a bit vague, in my read, about the memory order when it comes to acquiring or releasing atomic variables in the signal handler. C++ says, and it makes sense, that such accesses are unsequenced with respect to the rest of the program[8]. The signal can be raised, after all, at any time. So I'm assuming C's behavior is similar.

40.10 Atomic Flags

There's only one type the standard guarantees will be a lock-free atomic: `atomic_flag`. This is an opaque type for test-and-set[9] operations.

It can be either *set* or *clear*. You can initialize it to clear with:

```
atomic_flag f = ATOMIC_FLAG_INIT;
```

You can set the flag atomically with `atomic_flag_test_and_set()`, which will set the flag and return its previous status as a `_Bool` (true for set).

You can clear the flag atomically with `atomic_flag_clear()`.

Here's an example where we init the flag to clear, set it twice, then clear it again.

```
#include <stdio.h>
#include <stdbool.h>
#include <stdatomic.h>

atomic_flag f = ATOMIC_FLAG_INIT;

int main(void)
{
    bool r = atomic_flag_test_and_set(&f);
    printf("Value was: %d\n", r);           // 0

    r = atomic_flag_test_and_set(&f);
    printf("Value was: %d\n", r);           // 1

    atomic_flag_clear(&f);
    r = atomic_flag_test_and_set(&f);
    printf("Value was: %d\n", r);           // 0
}
```

40.11 Atomic `structs` and `unions`

Using the `_Atomic` qualifier or specifier, you can make atomic `structs` or `unions`! Pretty astounding.

[8] C++ elaborates that if the signal is the result of a call to `raise()`, it is sequenced *after* the `raise()`.
[9] https://en.wikipedia.org/wiki/Test-and-set

If there's not a lot of data in there (i.e. a handful of bytes), the resulting atomic type might be lock-free. Test it with `atomic_is_lock_free()`.

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    struct point {
        float x, y;
    };

    _Atomic(struct point) p;

    printf("Is lock free: %d\n", atomic_is_lock_free(&p));
}
```

Here's the catch: you can't access fields of an atomic `struct` or `union`... so what's the point? Well, you can atomically *copy* the entire `struct` into a non-atomic variable and then used it. You can atomically copy the other way, too.

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    struct point {
        float x, y;
    };

    _Atomic(struct point) p;
    struct point t;

    p = (struct point){1, 2};  // Atomic copy

    //printf("%f\n", p.x);  // Error

    t = p;   // Atomic copy

    printf("%f\n", t.x);  // OK!
}
```

You can also declare a `struct` where individual fields are atomic. It is implementation defined if atomic types are allowed on bitfields.

40.12 Atomic Pointers

Just a note here about placement of `_Atomic` when it comes to pointers.

First, pointers to atomics (i.e. the pointer value is not atomic, but the thing it points to is):

```
_Atomic int x;
_Atomic int *p;  // p is a pointer to an atomic int

p = &x;  // OK!
```

Second, atomic pointers to non-atomic values (i.e. the pointer value itself is atomic, but the thing it points to is not):

```
int x;
int * _Atomic p;  // p is an atomic pointer to an int

p = &x;  // OK!
```

Lastly, atomic pointers to atomic values (i.e. the pointer and the thing it points to are both atomic):

```
_Atomic int x;
_Atomic int * _Atomic p;  // p is an atomic pointer to an atomic int

p = &x;  // OK!
```

40.13 Memory Order

We've already talked about sequential consistency, which is the sensible one of the bunch. But there are a number of other ones:

memory_order	Description
memory_order_seq_cst	Sequential Consistency
memory_order_acq_rel	Acquire/Release
memory_order_release	Release
memory_order_acquire	Acquire
memory_order_consume	Consume
memory_order_relaxed	Relaxed

You can specify other ones with certain library functions. For example, you can add a value to an atomic variable like this:

```
atomic_int x = 0;

x += 5;  // Sequential consistency, the default
```

or you can do the same with this library function:

```
atomic_int x = 0;

atomic_fetch_add(&x, 5);  // Sequential consistency, the default
```

or you can do the same thing with an explicit memory ordering:

```
atomic_int x = 0;

atomic_fetch_add_explicit(&x, 5, memory_order_seq_cst);
```

But what if we didn't want sequential consistency? And you wanted acquire/release instead for whatever reason? Just name it:

```
atomic_int x = 0;

atomic_fetch_add_explicit(&x, 5, memory_order_acq_rel);
```

We'll do a breakdown of the different memory orders, below. Don't mess with anything other than sequential consistency unless you know what you're doing. It's really easy to make mistakes that will cause rare, hard-to-repro failures.

40.13.1 Sequential Consistency

- Load operations acquire (see below).
- Store operations release (see below).
- Read-modify-write operations acquire then release.

Also, in order to maintain the total order of acquires and releases, no acquires or releases will be reordered with respect to each other. (The acquire/release rules do not forbid reordering a release followed by an acquire. But the sequentially consistent rules do.)

40.13.2 Acquire

This is what happens on a load/read operation on an atomic variable.

- If another thread released this atomic variable, all the writes that thread did are now visible in this thread.
- Memory accesses in this thread that happen after this load can't be reordered before it.

40.13.3 Release

This is what happens on a store/write of an atomic variable.

- If another thread later acquires this atomic variable, all memory writes in this thread before its atomic write become visible to that other thread.
- Memory accesses in this thread that happen before the release can't be reordered after it.

40.13.4 Consume

This is an odd one, similar to a less-strict version of acquire. It affects memory accesses that are *data dependent* on the atomic variable.

Being "data dependent" vaguely means that the atomic variable is used in a calculation.

That is, if a thread consumes an atomic variable then all the operations in that thread that go on to use that atomic variable will be able to see the memory writes in the releasing thread.

Compare to acquire where memory writes in the releasing thread will be visible to *all* operations in the current thread, not just the data-dependent ones.

Also like acquire, there is a restriction on which operations can be reordered *before* the consume. With acquire, you couldn't reorder anything before it. With consume, you can't reorder anything that depends on the loaded atomic value before it.

40.13.5 Acquire/Release

This only applies to read-modify-write operations. It's an acquire and release bundled into one.

- An acquire happens for the read.
- A release happens for the write.

40.13.6 Relaxed

No rules; it's anarchy! Everyone can reorder everything everywhere! Dogs and cats living together—mass hysteria!

Actually, there is a rule. Atomic reads and writes are still all-or-nothing. But the operations can be reordered whimsically and there is zero synchronization between threads.

www.dbooks.org

There are a few use cases for this memory order, which you can find with a tiny bit of searching, e.g. simple counters.

And you can use a fence to force synchronization after a bunch of relaxed writes.

40.14 Fences

You know how the releases and acquires of atomic variables occur as you read and write them?

Well, it's possible to do a release or acquire *without* an atomic variable, as well.

This is called a *fence*. So if you want all the writes in a thread to be visible elsewhere, you can put up a release fence in one thread and an acquire fence in another, just like with how atomic variables work.

Since a consume operation doesn't really make sense on a fence[10], `memory_order_consume` is treated as an acquire.

You can put up a fence with any specified order:

```
atomic_thread_fence(memory_order_release);
```

There's also a light version of a fence for use with signal handlers, called `atomic_signal_fence()`.

It works just the same way as `atomic_thread_fence()`, except:

- It only deals with visibility of values within the same thread; there is no synchronization with other threads.
- No hardware fence instructions are emitted.

If you want to be sure the side effects of non-atomic operations (and relaxed atomic operations) are visible in the signal handler, you can use this fence.

The idea is that the signal handler is executing in *this* thread, not another, so this is a lighter-weight way of making sure changes outside the signal handler are visible within it (i.e. they haven't been reordered).

40.15 References

If you want to learn more about this stuff, here are some of the things that helped me plow through it:

- Herb Sutter's *`atomic<>` Weapons* talk:
  - Part 1[11]
  - part 2[12]
- Jeff Preshing's materials[13], in particular:
  - An Introduction to Lock-Free Programming[14]
  - Acquire and Release Semantics[15]
  - The *Happens-Before* Relation[16]
  - The *Synchronizes-With* Relation[17]
  - The Purpose of `memory_order_consume` in C++11[18]

[10] Because consume is all about the operations that are dependent on the value of the acquired atomic variable, and there is no atomic variable with a fence.

[11] https://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-1-of-2

[12] https://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-2-of-2

[13] https://preshing.com/archives/

[14] https://preshing.com/20120612/an-introduction-to-lock-free-programming/

[15] https://preshing.com/20120913/acquire-and-release-semantics/

[16] https://preshing.com/20130702/the-happens-before-relation/

[17] https://preshing.com/20130823/the-synchronizes-with-relation/

[18] https://preshing.com/20140709/the-purpose-of-memory_order_consume-in-cpp11/

  - You Can Do Any Kind of Atomic Read-Modify-Write Operation[19]
- CPPReference:
  - Memory Order[20]
  - Atomic Types[21]
- Bruce Dawson's Lockless Programming Considerations[22]
- The helpful and knowledgeable folks on r/C_Programming[23]

[19] https://preshing.com/20150402/you-can-do-any-kind-of-atomic-read-modify-write-operation/
[20] https://en.cppreference.com/w/c/atomic/memory_order
[21] https://en.cppreference.com/w/c/language/atomic
[22] https://docs.microsoft.com/en-us/windows/win32/dxtecharts/lockless-programming
[23] https://www.reddit.com/r/C_Programming/

www.dbooks.org

Chapter 41

Function Specifiers, Alignment Specifiers/Operators

These don't see a heck of a lot of use in my experience, but we'll cover them here for the sake of completeness.

41.1 Function Specifiers

When you declare a function, you can give the compiler a couple tips about how the functions could or will be used. This enables or encourages the compiler to make certain optimizations.

41.1.1 `inline` for Speed—Maybe

You can declare a function to be inline like this:

```
static inline int add(int x, int y) {
    return x + y;
}
```

This is meant to encourage the compiler to make this function call as fast as possible. And, historically, one way to do this was *inlining*, which means that the body of the function would be embedded in its entirety where the call was made. This would avoid all the overhead of setting up the function call and tearing it down at the expense of larger code size as the function was copied all over the place instead of being reused.

That would seem to be the end of the story, but it's not. `inline` comes with a whole pile of rules that make for *interesting times*. I'm not sure I even understand them all, and behavior seems to vary from compiler to compiler.

The short answer is define the `inline` function as `static` in the file that you need it. And then use it in that one file. And you never have to worry about the rest of it.

But if you're wondering, here are more fun times.

Let's try leaving the `static` off.

```
#include <stdio.h>

inline int add(int x, int y)
{
    return x + y;
}

```

```
int main(void)
{
    printf("%d\n", add(1, 2));
}
```

gcc gives a linker error on add()... unless you compile with optimizations on (probably)!

See, a compiler can choose to inline or not, but if it chooses not to, you're left with no function at all. gcc doesn't inline unless you're doing an optimized build.

One way around this is to define a non-inline external linkage version of the function elsewhere, and that one will be used when the inline one isn't. But you as the programmer can't determine which, portably. If both are available, it's unspecified which one the compiler chooses. With gcc the inline function will be used if you're compiling with optimizations, and the non-inline one will be used otherwise. Even if the bodies of these functions are completely different. Zany!

Another way is to declare the function as extern inline. This will attempt to inline in this file, but will also create a version with external linkage. And so gcc will use one or the other depending on optimizations, but at least they're the same function.

Unless, of course, you have another source file with an inline function of the same name; it will use its inline function or the one with external linkage depending on optimizations.

But let's say you're doing a build where the compiler *is* inlining the function. In that case, you can just use a plain inline in the definition. However, there are now additional restrictions.

You can't refer to any static globals:

```
static int b = 13;

inline int add(int x, int y)
{
    return x + y + b;  // BAD -- can't refer to b
}
```

And you can't define any non-const static local variables:

```
inline int add(int x, int y)
{
    static int b = 13;  // BAD -- can't define static

    return x + y + b;
}
```

But making it const is OK:

```
inline int add(int x, int y)
{
    static const int b = 13;  // OK -- static const

    return x + y + b;
}
```

Now, you know the functions are extern by default, so we should be able to call add() from another file. You'd like to think that, wouldn't you!

But you can't! If it's just a plain inline, it's similar to static: it's only visible in that file.

Okay, so what if you throw an extern on there? Now we're coming full circle to when we discussed having inline mixed with functions with external linkage.

If both are visible, the compiler can choose which to use.

Let's do a demo of this behavior. We'll have two files, `foo.c` and `bar.c`. They'll both call `func()` which is `inline` in `foo.c` and external linkage in `bar.c`.

Here's `foo.c` with the `inline`.

```c
// foo.c

#include <stdio.h>

inline char *func(void)
{
    return "foo's function";
}

int main(void)
{
    printf("foo.c: %s\n", func());

    void bar(void);
    bar();
}
```

Recall that unless we're doing an optimized build with `gcc`. `func()` will vanish and we'll get a linker error. Unless, or course, we have a version with external linkage defined elsewhere.

And we do. In `bar.c`.

```c
// bar.c

#include <stdio.h>

char *func(void)
{
    return "bar's function";
}

void bar(void)
{
    printf("bar.c: %s\n", func());
}
```

So the question is, what is the output?

Seems like when we call `func()` from `foo.c`, it should print "`foo's function`". And from `bar.c`, that `func()` should print "`bar's function`".

And if I compile with `gcc` with optimizations[1] it will use inline functions, and we'll get the expected:

```
foo.c: foo's function
bar.c: bar's function
```

Great!

But if we compile in `gcc` without optimizations, it ignores the inline function and uses the external linkage `func()` from `bar.c`! And we get this:

[1] You can do this with `-O` on the command line.

```
foo.c: bar's function
bar.c: bar's function
```

In short, the rules are surprisingly complex. I give myself a good 30% chance of having described them correctly.

41.1.2 `noreturn` and `_Noreturn`

This indicates to the compiler that a particular function will not ever return to its caller, i.e. the program will exit by some mechanism before the function returns.

It allows the compiler to perhaps perform some optimizations around the function call.

It also allows you to indicate to other devs that some program logic depends on a function *not* returning.

You'll likely never need to use this, but you'll see it on some library calls like `exit()` and `abort()`.

The built-in keyword is `_Noreturn`, but if it doesn't break your existing code, everyone would recommend including `<stdnoreturn.h>` and using the easier-to-read `noreturn` instead.

It's undefined behavior if a function specified as `noreturn` actually does return. It's computationally dishonest, see.

Here's an example of using `noreturn` correctly:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void foo(void) // This function should never return!
{
    printf("Happy days\n");

    exit(1);            // And it doesn't return--it exits here!
}

int main(void)
{
    foo();
}
```

If the compiler detects that a `noreturn` function could return, it might warn you, helpfully.

Replacing the `foo()` function with this:

```c
noreturn void foo(void)
{
    printf("Breakin' the law\n");
}
```

gets me a warning:

```
foo.c:7:1: warning: function declared 'noreturn' should not return
```

41.2 Alignment Specifiers and Operators

Alignment[2] is all about multiples of addresses on which objects can be stored. Can you store this at any

[2] https://en.wikipedia.org/wiki/Data_structure_alignment

www.dbooks.org

address? Or must it be a starting address that's divisible by 2? Or 8? Or 16?

If you're coding up something low-level like a memory allocator that interfaces with your OS, you might need to consider this. Most devs go their careers without using this functionality in C.

41.2.1 `alignas` and `_Alignas`

This isn't a function. Rather, it's an *alignment specifier* that you can use with a variable declaration.

The built-in specifier is `_Alignas`, but the header `<stdalign.h>` defines it as `alignas` for something better looking.

If you need your `char` to be aligned like an `int`, you can force it like this when you declare it:

```
char alignas(int) c;
```

You can also pass a constant value or expression in for the alignment. This has to be something supported by the system, but the spec stops short of dictating what values you can put in there. Small powers of 2 (1, 2, 4, 8, and 16) are generally safe bets.

```
char alignas(8) c;   // align on 8-byte boundaries
```

If you want to align at the maximum used alignment by your system, include `<stddef.h>` and use the type `max_align_t`, like so:

```
char alignas(max_align_t) c;
```

You could potentially *over-align* by specifying an alignment more than that of `max_align_t`, but whether or not such things are allowed is system dependent.

41.2.2 `alignof` and `_Alignof`

This operator will return the address multiple a particular type uses for alignment on this system. For example, maybe `char`s are aligned every 1 address, and `int`s are aligned every 4 addresses.

The built-in operator is `_Alignof`, but the header `<stdalign.h>` defines it as `alignof` if you want to look cooler.

Here's a program that will print out the alignments of a variety of different types. Again, these will vary from system to system. Note that the type `max_align_t` will give you the maximum alignment used by the system.

```
#include <stdalign.h>
#include <stdio.h>     // for printf()
#include <stddef.h>    // for max_align_t

struct t {
    int a;
    char b;
    float c;
};

int main(void)
{
    printf("char       : %zu\n", alignof(char));
    printf("short      : %zu\n", alignof(short));
    printf("int        : %zu\n", alignof(int));
    printf("long       : %zu\n", alignof(long));
    printf("long long  : %zu\n", alignof(long long));
```

```
    printf("double     : %zu\n", alignof(double));
    printf("long double: %zu\n", alignof(long double));
    printf("struct t   : %zu\n", alignof(struct t));
    printf("max_align_t: %zu\n", alignof(max_align_t));
}
```

Output on my system:

```
char       : 1
short      : 2
int        : 4
long       : 8
long long  : 8
double     : 8
long double: 16
struct t   : 16
max_align_t: 16
```

And there you have it. Alignment!

Chapter 42

<assert.h> Runtime and Compile-time Diagnostics

Macro	Description
`assert()`	Runtime assertion
`static_assert()`	Compile-time assertion

This functionality has to do with things that Should Never Happen™. If you have something that should never be true and you want your program to bomb out because it happened, this is the header file for you.

There are two types of assertions: compile-time assertions (called "static assertions") and runtime assertions. If the assertion *fails* (i.e. the thing that you need to be true is not true) then the program will bomb out either at compile-time or runtime.

42.1 Macros

If you define the macro `NDEBUG` **before** you include `<assert.h>`, then the `assert()` macro will have no effect. You can define `NDEBUG` to be anything, but `1` seems like a good value.

Since `assert()` causes your program to bomb out at runtime, you might not desire this behavior when you go into production. Defining `NDEBUG` causes `assert()` to be ignored.

`NDEBUG` has no effect on `static_assert()`.

42.2 `assert()`

Bomb out at runtime if a condition fails

Synopsis

```
#include <assert.h>

void assert(scalar expression);
```

Description

You pass in an expression to this macro. If it evaluates to false, the program will crash with an assertion failure (by calling the `abort()` function).

Basically, you're saying, "Hey, I'm assuming this condition is true, and if it's not, I don't want to continue running."

This is used while debugging to make sure no unexpected conditions arise. And if you find during development that the condition does arise, maybe you should modify the code to handle it before going to production.

If you've defined the macro `NDEBUG` to any value before `<assert.h>` was included, the `assert()` macro is ignored. This is a good idea before production.

Unlike `static_assert()`, this macro doesn't allow you to print an arbitrary message. If you want to do this, check out the example in the Preprocessor chapter.

Return Value

This macro doesn't return (since it calls `abort()` which never returns).

If `NDEBUG` is set, the macro evaluates to `((void)0)`, which does nothing.

Example

Here's a function that divides the size of our goat herd. But we're assuming we'll never get a `0` passed to us.

So we assert that `amount != 0`… and if it is, the program aborts/

```
//#define NDEBUG 1   // uncomment this to disable the assert

#include <stdio.h>
#include <assert.h>

int goat_count = 10;

void divide_goat_herd_by(int amount)
{
    assert(amount != 0);

    goat_count /= amount;
}

int main(void)
{
    divide_goat_herd_by(2);  // OK

    divide_goat_herd_by(0);  // Causes the assert to fire
}
```

When I run this and pass `0` to the function, I get the following on my system (the exact output may vary):

```
assert: assert.c:10: divide_goat_herd_by: Assertion `amount != 0' failed.
```

See Also

`static_assert()`, `abort()`

42.3 static_assert()

Bomb out at compile-time if a condition fails

Synopsis

```
#include <assert.h>

static_assert(constant-expression, string-literal);
```

Description

This macro prevents your program from even compiling if a condition isn't true.

And it prints the string literal you give it.

Basically if `constant-expression` is false, then compilation will cease and the `string-literal` will be printed.

The constant expression must be truly constant–just values, no variables. And the same is true for the string literal: no variables, just a literal string in double quotes. (It has to be this way since the program's not running at this point.)

Return Value

Not applicable, as this is a compile-time feature.

Example

Here's a partial example with an algorithm that presumably has poor performance or memory issues if the size of the local array is too large. We prevent that eventuality at compile-time by catching it with the `static_assert()`.

```
#include <stdio.h>
#include <assert.h>

#define ARRAY_SIZE 16

int main(void)
{
    static_assert(ARRAY_SIZE > 32, "ARRAY_SIZE too small");

    int a[ARRAY_SIZE];

    a[32] = 10;

    printf("%d\n", a[32]);
}
```

On my system, when I try to compile it, this prints (your output may vary):

```
In file included from static_assert.c:2:
static_assert.c: In function ‘main’:
static_assert.c:8:5: error: static assertion failed: "ARRAY_SIZE too small"
    8 |     static_assert(ARRAY_SIZE > 32, "ARRAY_SIZE too small");
      |     ^~~~~~~~~~~~~
```

See Also

`assert()`

www.dbooks.org

Chapter 43

<complex.h> Complex Number Functionality

The complex functions in this reference section come in three flavors each: `double complex`, `float complex`, and `long double complex`.

The `float` variants end with `f` and the `long double` variants end with `l`, e.g. for complex cosine:

```
ccos()   double complex
ccosf()  float complex
ccosl()  long double complex
```

The table below only lists the `double complex` version for brevity.

Function	Description
`cabs()`	Compute the complex absolute value
`cacos()`	Compute the complex arc-cosine
`cacosh()`	Compute the complex arc hyperbolic cosine
`carg()`	Compute the complex argument
`casin()`	Compute the complex arc-sine
`casinh()`	Compute the complex arc hyperbolic sine
`catan()`	Compute the complex arc-tangent
`catanh()`	Compute the complex arc hyperbolic tangent
`ccos()`	Compute the complex cosine
`ccosh()`	Compute the complex hyperbolic cosine
`cexp()`	Compute the complex base-e exponential
`cimag()`	Returns the imaginary part of a complex number
`clog()`	Compute the complex logarithm
`CMPLX()`	Build a complex value from real and imaginary types
`conj()`	Compute the conjugate of a complex number
`cproj()`	Compute the projection of a complex number
`creal()`	Returns the real part of a complex number
`csin()`	Compute the complex sine
`csinh()`	Compute the complex hyperbolic sine
`csqrt()`	Compute the complex square root
`ctan()`	Compute the complex tangent
`ctanh()`	Compute the complex hyperbolic tangent

You can test for complex number support by looking at the `__STDC_NO_COMPLEX__` macro. If it's defined, complex numbers aren't available.

There are possibly two types of numbers defined: *complex* and *imaginary*. No system I'm currently aware of implements imaginary types.

The complex types, which are a real value plus a multiple of i, are:

```
float complex
double complex
long double complex
```

The imaginary types, which hold a multiple of i, are:

```
float imaginary
double imaginary
long double imaginary
```

The mathematical value $i = \sqrt{-1}$ is represented by the symbol `_Complex_I` or `_Imaginary_I`, if it exists.

The The macro `I` will be preferentially set to `_Imaginary_I` (if it exists), or to `_Complex_I` otherwise.

You can write imaginary literals (if supported) using this notation:

```
double imaginary x = 3.4 * I;
```

You can write complex literals using regular complex notation:

```
double complex x = 1.2 + 3.4 * I;
```

or build them with the `CMPLX()` macro:

```
double complex x = CMPLX(1.2, 3.4);  // Like 1.2 + 3.4 * I
```

The latter has the advantage of handing special cases of complex numbers correctly (like those involving infinity or signed zeroes) as if `_Imaginary_I` were present, even if it's not.

All angular values are in radians.

Some functions have discontinuities called *branch cuts*. Now, I'm no mathematician so I can't really talk sensibly about this, but if you're here, I like to think you know what you're doing when it comes to this side of things.

If you system has signed zeroes, you can tell which side of the cut you're on by the sign. And you can't if you don't. The spec elaborates:

> Implementations that do not support a signed zero [...] cannot distinguish the sides of branch cuts. These implementations shall map a cut so the function is continuous as the cut is approached coming around the finite endpoint of the cut in a counter clockwise direction. (Branch cuts for the functions specified here have just one finite endpoint.) For example, for the square root function, coming counter clockwise around the finite endpoint of the cut along the negative real axis approaches the cut from above, so the cut maps to the positive imaginary axis.

Finally, there's a pragma called `CX_LIMITED_RANGE` that can be turned on and off (default is off). You can turn it on with:

```
#pragma STDC CX_LIMITED_RANGE ON
```

It allows for certain intermediate operations to underflow, overflow, or deal badly with infinity, presumably for a tradeoff in speed. If you're sure these types of errors won't occur with the numbers you're using AND you're trying to get as much speed out as you can, you could turn this macro on.

The spec also elaborates here:

The purpose of the pragma is to allow the implementation to use the formulas:

$(x + iy) \times (u + iv) = (xu - yv) + i(yu + xv)$

$(x + iy)/(u + iv) = [(xu + yv) + i(yu - xv)]/(u^2 + v^2)$

$|x + iy| = \sqrt{x^2 + y^2}$

where the programmer can determine they are safe.

43.1 cacos() cacosf() cacosl()

Compute the complex arc-cosine

Synopsis

```
#include <complex.h>

double complex cacos(double complex z);

float complex cacosf(float complex z);

long double complex cacosl(long double complex z);
```

Description

Computes the complex arc-cosine of a complex number.

The complex number z will have an imaginary component in the range $[0, \pi]$, and the real component is unbounded.

There are branch cuts outside the interval $[-1, +1]$ on the real axis.

Return Value

Returns the complex arc-cosine of z.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = cacos(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 0.195321 + -2.788006i
```

See Also

ccos(), casin(), catan()

43.2 casin() casinf() casinl()

Compute the complex arc-sine

Synopsis

```
#include <complex.h>

double complex casin(double complex z);

float complex casinf(float complex z);

long double complex casinl(long double complex z);
```

Description

Computes the complex arc-sine of a complex number.

The complex number z will have an imaginary component in the range $[-\pi/2, +\pi/2]$, and the real component is unbounded.

There are branch cuts outside the interval $[-1, +1]$ on the real axis.

Return Value

Returns the complex arc-sine of z.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = casin(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 1.375476 + 2.788006i
```

See Also

csin(), cacos(), catan()

43.3 catan() catanf() catanl()

Compute the complex arc-tangent

Synopsis

```
#include <complex.h>

double complex catan(double complex z);

float complex catanf(float complex z);

long double complex catanl(long double complex z);
```

Description

Computes the complex arc-tangent of a complex number.

The complex number z will have an real component in the range $[-\pi/2, +\pi/2]$, and the imaginary component is unbounded.

There are branch cuts outside the interval $[-i, +i]$ on the imaginary axis.

Return Value

Returns the complex arc-tangent of z.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double wheat = 8;
    double sheep = 1.5708;

    double complex x = wheat + sheep * I;

    double complex y = catan(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 1.450947 + 0.023299i
```

See Also

ctan(), cacos(), casin()

43.4 ccos() ccosf() ccosl()

Compute the complex cosine

Synopsis

```
#include <complex.h>

double complex ccos(double complex z);

float complex ccosf(float complex z);

long double complex ccosl(long double complex z);
```

Description

Computes the complex cosine of a complex number.

Return Value

Returns the complex cosine of z.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = ccos(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: -0.365087 + -2.276818i
```

See Also

csin(), ctan(), cacos()

43.5 csin() csinf() csinl()

Compute the complex sine

Synopsis

```
#include <complex.h>

double complex csin(double complex z);

float complex csinf(float complex z);

long double complex csinl(long double complex z);
```

Description

Computes the complex sine of a complex number.

Return Value

Returns the complex sine of z.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = csin(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 2.482485 + -0.334840i
```

See Also

ccos(), ctan(), casin()

43.6 ctan() ctanf() ctanl()

Compute the complex tangent

Synopsis

```
#include <complex.h>

double complex ctan(double complex z);

float complex ctanf(float complex z);

long double complex ctanl(long double complex z);
```

Description

Computes the complex tangent of a complex number.

Return Value

Returns the complex tangent of `z`.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = ctan(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: -0.027073 + 1.085990i
```

See Also

`ccos()`, `csin()`, `catan()`

43.7 `cacosh()` `cacoshf()` `cacoshl()`

Compute the complex arc hyperbolic cosine

Synopsis

```
#include <complex.h>

double complex cacosh(double complex z);

float complex cacoshf(float complex z);

long double complex cacoshl(long double complex z);
```

Description

Computes the complex arc hyperbolic cosine of a complex number.

There is a branch cut at values less than 1 on the real axis.

The return value will be non-negative on the real number axis, and in the range $[-i\pi, +i\pi]$ on the imaginary axis.

Return Value

Returns the complex arc hyperbolic cosine of `z`.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = cacosh(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 2.788006 + 0.195321i
```

See Also

`casinh()`, `catanh()`, `acosh()`

43.8 `casinh()` `casinhf()` `casinhl()`

Compute the complex arc hyperbolic sine

Synopsis

```
#include <complex.h>

double complex casinh(double complex z);

float complex casinhf(float complex z);

long double complex casinhl(long double complex z);
```

Description

Computes the complex arc hyperbolic sine of a complex number.

There are branch cuts outside $[-i, +i]$ on the imaginary axis.

The return value will be unbounded on the real number axis, and in the range $[-i\pi/2, +i\pi/2]$ on the imaginary axis.

Return Value

Returns the complex arc hyperbolic sine of z.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = casinh(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 2.794970 + 0.192476i
```

See Also

cacosh(), catanh(), asinh()

43.9 catanh() catanhf() catanhl()

Compute the complex arc hyperbolic tangent

Synopsis

```
#include <complex.h>

double complex catanh(double complex z);

float complex catanhf(float complex z);

long double complex catanhl(long double complex z);
```

Description

Computes the complex arc hyperbolic tangent of a complex number.

There are branch cuts outside $[-1, +1]$ on the real axis.

The return value will be unbounded on the real number axis, and in the range $[-i\pi/2, +i\pi/2]$ on the imaginary axis.

Return Value

Returns the complex arc hyperbolic tangent of z.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = catanh(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 0.120877 + 1.546821i
```

See Also

cacosh(), casinh(), atanh()

43.10 ccosh() ccoshf() ccoshl()

Compute the complex hyperbolic cosine

Synopsis

```
#include <complex.h>

double complex ccosh(double complex z);

float complex ccoshf(float complex z);

long double complex ccoshl(long double complex z);
```

Description

Computes the complex hyperbolic cosine of a complex number.

Return Value

Returns the complex hyperbolic cosine of z.

Example

```c
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = ccosh(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: -0.005475 + 1490.478826i
```

See Also

`csinh()`, `ctanh()`, `ccos()`

43.11 `csinh()` `csinhf()` `csinhl()`

Compute the complex hyperbolic sine

Synopsis

```c
#include <complex.h>

double complex csinh(double complex z);

float complex csinhf(float complex z);

long double complex csinhl(long double complex z);
```

Description

Computes the complex hyperbolic sine of a complex number.

Return Value

Returns the complex hyperbolic sine of `z`.

Example

```c
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 8 + 1.5708 * I;

```

```
    double complex y = csinh(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: -0.005475 + 1490.479161i
```

See Also

`ccosh()`, `ctanh()`, `csin()`

43.12 `ctanh()` `ctanhf()` `ctanhl()`

Compute the complex hyperbolic tangent

Synopsis

```
#include <complex.h>

double complex ctanh(double complex z);

float complex ctanhf(float complex z);

long double complex ctanhl(long double complex z);
```

Description

Computes the complex hyperbolic tangent of a complex number.

Return Value

Returns the complex hyperbolic tangent of `z`.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = ctanh(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 1.000000 + -0.000000i
```

See Also

ccosh(), csinh(), ctan()

43.13 cexp() cexpf() cexpl()

Compute the complex base-e exponential

Synopsis

```
#include <complex.h>

double complex cexp(double complex z);

float complex cexpf(float complex z);

long double complex cexpl(long double complex z);
```

Description

Computes the complex base-e exponential of z.

Return Value

Returns the complex base-e exponential of z.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 1 + 2 * I;

    double complex y = cexp(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: -1.131204 + 2.471727i
```

See Also

cpow(), clog(), exp()

43.14 clog() clogf() clogl()

Compute the complex logarithm

Synopsis

```
#include <complex.h>

double complex clog(double complex z);

float complex clogf(float complex z);

long double complex clogl(long double complex z);
```

Description

Compute the base-e complex logarithm of z. There is a branch cut on the negative real axis.

The returns value is unbounded on the real axis and in the range $[-i\pi, +i\pi]$ on the imaginary axis.

Return Value

Returns the base-e complex logarithm of z.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 1 + 2 * I;

    double complex y = clog(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 0.804719 + 1.107149i
```

See Also

cexp(), log()

43.15 cabs() cabsf() cabsl()

Compute the complex absolute value

Synopsis

```
#include <complex.h>

double cabs(double complex z);

float cabsf(float complex z);

long double cabsl(long double complex z);
```

Description

Computes the complex absolute value of z.

Return Value

Returns the complex absolute value of z.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 1 + 2 * I;

    double complex y = cabs(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 2.236068 + 0.000000i
```

See Also

fabs(), abs()

43.16 cpow() cpowf() cpowl()

Compute complex power

Synopsis

```
#include <complex.h>

double complex cpow(double complex x, double complex y);

float complex cpowf(float complex x, float complex y);
```

```
long double complex cpowl(long double complex x,
                          long double complex y);
```

Description

Computes the complex x^y.

There is a branch cut for x along the negative real axis.

Return Value

Returns the complex x^y.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 1 + 2 * I;
    double complex y = 3 + 4 * I;

    double r = cpow(x, y);

    printf("Result: %f + %fi\n", creal(r), cimag(r));
}
```

Result:

```
Result: 0.129010 + 0.000000i
```

See Also

`csqrt()`, `cexp()`

43.17 `csqrt()` `csqrtf()` `csqrtl()`

Compute the complex square root

Synopsis

```
#include <complex.h>

double complex csqrt(double complex z);

float complex csqrtf(float complex z);

long double complex csqrtl(long double complex z);
```

Description

Computes the complex square root of z.

There is a branch cut along the negative real axis.

The return value is in the right half of the complex plane and includes the imaginary axis.

Return Value

Returns the complex square root of z.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 1 + 2 * I;

    double complex y = csqrt(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 1.272020 + 0.786151i
```

See Also

cpow(), sqrt()

43.18 carg() cargf() cargl()

Compute the complex argument

Synopsis

```
#include <complex.h>

double carg(double complex z);

float cargf(float complex z);

long double cargl(long double complex z);
```

Description

Computes the complex argument (AKA phase angle) of z.

There is a branch cut along the negative real axis.

Returns a value in the range $[-\pi, +\pi]$.

Return Value

Returns the complex argument of z.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 1 + 2 * I;

    double y = carg(x);

    printf("Result: %f\n", y);
}
```

Output:

```
Result: 1.107149
```

43.19 cimag() cimagf() cimagl()

Returns the imaginary part of a complex number

Synopsis

```
#include <complex.h>

double cimag(double complex z);

float cimagf(float complex z);

long double cimagl(long double complex z);
```

Description

Returns the imaginary part of z.

As a footnote, the spec points out that any complex number x is part of the following equivalency:

```
x == creal(x) + cimag(x) * I;
```

Return Value

Returns the imaginary part of z.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 1 + 2 * I;

    double y = cimag(x);

    printf("Result: %f\n", y);
}
```

Output—just the imaginary part:

```
Result: 2.000000
```

See Also

`creal()`

43.20 `CMPLX()` `CMPLXF()` `CMPLXL()`

Build a complex value from real and imaginary types

Synopsis

```
#include <complex.h>

double complex CMPLX(double x, double y);

float complex CMPLXF(float x, float y);

long double complex CMPLXL(long double x, long double y);
```

Description

These macros build a complex value from real and imaginary types.

Now I know what you're thinking. "But I can already build a complex value from real and imaginary types using the `I` macro, like in the example you're about to give us."

```
double complex x = 1 + 2 * I;
```

And that's true.

But the reality of the matter is weird and complex.

Maybe `I` got undefined, or maybe you redefined it.

Or maybe `I` was defined as `_Complex_I` which doesn't necessarily preserve the sign of a zero value.

As the spec points out, these macros build complex numbers as if `_Imaginary_I` were defined (thus preserving your zero sign) even if it's not. That is, they are defined equivalently to:

```
#define CMPLX(x, y)  ((double complex)((double)(x) + \
                      _Imaginary_I * (double)(y)))

#define CMPLXF(x, y) ((float complex)((float)(x) + \
                      _Imaginary_I * (float)(y)))

#define CMPLXL(x, y) ((long double complex)((long double)(x) + \
                      _Imaginary_I * (long double)(y)))
```

Return Value

Returns the complex number for the given real `x` and imaginary `y` components.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = CMPLX(1, 2);  // Like 1 + 2 * I

    printf("Result: %f + %fi\n", creal(x), cimag(x));
}
```

Output:

```
Result: 1.000000 + 2.000000i
```

See Also

`creal()`, `cimag()`

43.21 `conj()` `conjf()` `conjl()`

Compute the conjugate of a complex number

Synopsis

```
#include <complex.h>

double complex conj(double complex z);

float complex conjf(float complex z);

long double complex conjl(long double complex z);
```

Description

This function computes the complex conjugate[1] of z. Apparently it does this by reversing the sign of the imaginary part, but dammit, I'm a programmer not a mathematician, Jim!

Return Value

Returns the complex conjugate of z

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 1 + 2 * I;

    double complex y = conj(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 1.000000 + -2.000000i
```

43.22 cproj() cproj() cproj()

Compute the projection of a complex number

Synopsis

```
#include <complex.h>

double complex cproj(double complex z);

float complex cprojf(float complex z);

long double complex cprojl(long double complex z);
```

Description

Computes the projection of z onto a Riemann sphere[2].

Now we're *really* outside my expertise. The spec has this to say, which I'm quoting verbatim because I'm not knowledgable enough to rewrite it sensibly. Hopefully it makes sense to anyone who would need to use this function.

[1] https://en.wikipedia.org/wiki/Complex_conjugate
[2] https://en.wikipedia.org/wiki/Riemann_sphere

z projects to z except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If z has an infinite part, then cproj(z) is equivalent to

```
INFINITY + I * copysign(0.0, cimag(z))
```

So there you have it.

Return Value

Returns the projection of z onto a Riemann sphere.

Example

Fingers crossed this is a remotely sane example…

```
#include <stdio.h>
#include <complex.h>
#include <math.h>

int main(void)
{
    double complex x = 1 + 2 * I;

    double complex y = cproj(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));

    x = INFINITY + 2 * I;
    y = cproj(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 1.000000 + 2.000000i
Result: inf + 0.000000i
```

43.23 creal() crealf() creall()

Returns the real part of a complex number

Synopsis

```
#include <complex.h>

double creal(double complex z);

float crealf(float complex z);

long double creall(long double complex z);
```

Description

Returns the real part of `z`.

As a footnote, the spec points out that any complex number `x` is part of the following equivalency:

```
x == creal(x) + cimag(x) * I;
```

Return Value

Returns the real part of `z`.

Example

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 1 + 2 * I;

    double y = creal(x);

    printf("Result: %f\n", y);
}
```

Output—just the real part:

```
Result: 1.000000
```

See Also

`cimag()`

Chapter 44

<ctype.h> Character Classification and Conversion

Function	Description
`isalnum()`	Tests if a character is alphabetic or is a digit
`isalpha()`	Returns true if a character is alphabetic
`isblank()`	Tests if a character is word-separating whitespace
`iscntrl()`	Test if a character is a control character
`isdigit()`	Tests if a character is a digit
`isgraph()`	Tests if the character is printable and not a space
`islower()`	Tests if a character is lowercase
`isprint()`	Tests if a character is printable
`ispunct()`	Test if a character is punctuation
`isspace()`	Test if a character is whitespace
`isupper()`	Tests if a character is uppercase
`isxdigit()`	Tests if a character is a hexadecimal digit
`tolower()`	Convert a letter to lowercase
`toupper()`	Convert a letter to uppercase

This collection of macros is good for testing characters to see if they're of a certain class, such as alphabetic, numeric, control characters, etc.

Surprisingly, they take `int` arguments instead of some kind of `char`. This is so you can feed `EOF` in for convenience if you have an integer representation of that. If not `EOF`, the value passed in has to be representable in an `unsigned char`. Otherwise it's (dun dun DUUNNNN) undefined behavior. So you can forget about passing in your UTF-8 multibyte characters.

You can portably avoid this undefined behavior by casting the arguments to these functions to `(unsigned char)`. This is irksome and ugly, admittedly. The values in the basic character set are all safe to use since they're positive values that fit into an `unsigned char`.

Also, the behavior of these functions varies based on locale.

In many of the pages in this section, I give some examples. These are from the "C" locale, and might vary if you've set a different locale.

Note that wide characters have their own set of classification functions, so don't try to use these on `wchar_t`s. Or *else*!

44.1 isalnum()

Tests if a character is alphabetic or is a digit

Synopsis

```
#include <ctype.h>

int isalnum(int c);
```

Description

Tests if a character is alphabetic (A-Z or a-z) or a digit (0-9).

Is equivalent to:

```
isalpha(c) || isdigit(c)
```

Return Value

Returns true if a character is alphabetic (A-Z or a-z) or a digit (0-9).

Example

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    //              testing this char
    //                       v
    printf("%s\n", isalnum('a')? "yes": "no");  // yes
    printf("%s\n", isalnum('B')? "yes": "no");  // yes
    printf("%s\n", isalnum('5')? "yes": "no");  // yes
    printf("%s\n", isalnum('?')? "yes": "no");  // no
}
```

See Also

isalpha(), isdigit()

44.2 isalpha()

Returns true if a character is alphabetic

Synopsis

```
#include <ctype.h>

int isalpha(int c);
```

Description

Returns true for alphabetic characters (A-Z or a-z).

Technically (and in the "C" locale) equivalent to:

```
isupper(c) || islower(c)
```

Extra super technically, because I know you're dying for this to be extra unnecessarily complex, it can also include some locale-specific characters for which this is true:

```
!iscntrl(c) && !isdigit(c) && !ispunct(c) && !isspace(c)
```

and this is true:

```
isupper(c) || islower(c)
```

Return Value

Returns true for alphabetic characters (A-Z or a-z).

Or for any of the other crazy stuff in the description, above.

Example

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    //             testing this char
    //                      v
    printf("%s\n", isalpha('a')? "yes": "no");  // yes
    printf("%s\n", isalpha('B')? "yes": "no");  // yes
    printf("%s\n", isalpha('5')? "yes": "no");  // no
    printf("%s\n", isalpha('?')? "yes": "no");  // no
}
```

See Also

isalnum()

44.3 isblank()

Tests if a character is word-separating whitespace

Synopsis

```
#include <ctype.h>

int isblank(int c);
```

Description

True if the character is a whitespace character used to separate words in a single line.

For example, space (`' '`) or horizontal tab (`'\t'`). Other locales might define other blank characters.

Return Value

Returns true if the character is a whitespace character used to separate words in a single line.

Example

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    //              testing this char
    //                       v
    printf("%s\n", isblank(' ')? "yes": "no");   // yes
    printf("%s\n", isblank('\t')? "yes": "no");  // yes
    printf("%s\n", isblank('\n')? "yes": "no");  // no
    printf("%s\n", isblank('a')? "yes": "no");   // no
    printf("%s\n", isblank('?')? "yes": "no");   // no
}
```

See Also

`isspace()`

44.4 `iscntrl()`

Test if a character is a control character

Synopsis

```
#include <ctype.h>

int iscntrl(int c);
```

Description

A *control character* is a locale-specific non-printing character.

For the "C" locale, this means control characters are in the range 0x00 to 0x1F (the character right before SPACE) and 0x7F (the DEL character).

Basically if it's not an ASCII (or Unicode less than 128) printable character, it's a control character in the "C" locale.

Return Value

Returns true if `c` is a control character.

Example

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    //              testing this char
    //                       v
    printf("%s\n", iscntrl('\t')? "yes": "no");  // yes (tab)
    printf("%s\n", iscntrl('\n')? "yes": "no");  // yes (newline)
    printf("%s\n", iscntrl('\r')? "yes": "no");  // yes (return)
    printf("%s\n", iscntrl('\a')? "yes": "no");  // yes (bell)
    printf("%s\n", iscntrl(' ')? "yes": "no");   // no
    printf("%s\n", iscntrl('a')? "yes": "no");   // no
    printf("%s\n", iscntrl('?')? "yes": "no");   // no
}
```

See Also

`isgraph()`, `isprint()`

44.5 `isdigit()`

Tests if a character is a digit

Synopsis

```
#include <ctype.h>

int isdigit(int c);
```

Description

Tests if `c` is a digit in the range `0-9`.

Return Value

Returns true if the character is a digit, unsurprisingly.

Example

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    //              testing this char
    //                       v
    printf("%s\n", isdigit('0')? "yes": "no");   // yes
```

```
    printf("%s\n", isdigit('5')? "yes": "no");   // yes
    printf("%s\n", isdigit('a')? "yes": "no");   // no
    printf("%s\n", isdigit('B')? "yes": "no");   // no
    printf("%s\n", isdigit('?')? "yes": "no");   // no
}
```

See Also

`isalnum()`, `isxdigit()`

44.6 `isgraph()`

Tests if the character is printable and not a space

Synopsis

```
#include <ctype.h>

int isgraph(int c);
```

Description

Tests if `c` is any printable character that isn't a space (`' '`).

Return Value

Returns true if `c` is any printable character that isn't a space (`' '`).

Example

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    //              testing this char
    //                       v
    printf("%s\n", isgraph('0')? "yes": "no");   // yes
    printf("%s\n", isgraph('a')? "yes": "no");   // yes
    printf("%s\n", isgraph('B')? "yes": "no");   // yes
    printf("%s\n", isgraph('?')? "yes": "no");   // yes
    printf("%s\n", isgraph(' ')? "yes": "no");   // no
    printf("%s\n", isgraph('\n')? "yes": "no");  // no
}
```

See Also

`iscntrl()`, `isprint()`

44.7 islower()

Tests if a character is lowercase

Synopsis

```
#include <ctype.h>

int islower(int c);
```

Description

Tests if a character is lowercase, in the range a-z.

In other locales, there could be other lowercase characters. In all cases, to be lowercase, the following must be true:

```
!iscntrl(c) && !isdigit(c) && !ispunct(c) && !isspace(c)
```

Return Value

Returns true if the character is lowercase.

Example

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    //              testing this char
    //                       v
    printf("%s\n", islower('c')? "yes": "no");   // yes
    printf("%s\n", islower('0')? "yes": "no");   // no
    printf("%s\n", islower('B')? "yes": "no");   // no
    printf("%s\n", islower('?')? "yes": "no");   // no
    printf("%s\n", islower(' ')? "yes": "no");   // no
}
```

See Also

isupper(), isalpha(), toupper(), tolower()

44.8 isprint()

Tests if a character is printable

Synopsis

```
#include <ctype.h>

int isprint(int c);
```

Description

Tests if a character is printable, including space (`' '`). So like `isgraph()`, except space isn't left out in the cold.

Return Value

Returns true if the character is printable, including space (`' '`).

Example

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    //              testing this char
    //                       v
    printf("%s\n", isprint('c')? "yes": "no");   // yes
    printf("%s\n", isprint('0')? "yes": "no");   // yes
    printf("%s\n", isprint(' ')? "yes": "no");   // yes
    printf("%s\n", isprint('\r')? "yes": "no");  // no
}
```

See Also

`isgraph()`, `iscntrl()`

44.9 `ispunct()`

Test if a character is punctuation

Synopsis

```
#include <ctype.h>

int ispunct(int c);
```

Description

Tests if a character is punctuation.

In the "C" locale, this means:

```
!isspace(c) && !isalnum(c)
```

In other locales, there could be other punctuation characters (but they also can't be space or alphanumeric).

Return Value

True if the character is punctuation.

Example

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    //              testing this char
    //                       v
    printf("%s\n", ispunct(',')? "yes": "no");   // yes
    printf("%s\n", ispunct('!')? "yes": "no");   // yes
    printf("%s\n", ispunct('c')? "yes": "no");   // no
    printf("%s\n", ispunct('0')? "yes": "no");   // no
    printf("%s\n", ispunct(' ')? "yes": "no");   // no
    printf("%s\n", ispunct('\n')? "yes": "no");  // no
}
```

See Also

`isspace()`, `isalnum()`

44.10 `isspace()`

Test if a character is whitespace

Synopsis

```
#include <ctype.h>

int isspace(int c);
```

Description

Tests if `c` is a whitespace character. These are:

- Space (`' '`)
- Formfeed (`'\f'`)
- Newline (`'\n'`)
- Carriage Return (`'\r'`)
- Horizontal Tab (`'\t'`)
- Vertical Tab (`'\v'`)

Other locales might specify other whitespace characters. `isalnum()` is false for all whitespace characters.

Return Value

True if the character is whitespace.

Example

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    //              testing this char
    //                       v
    printf("%s\n", isspace(' ')? "yes": "no");   // yes
    printf("%s\n", isspace('\n')? "yes": "no");  // yes
    printf("%s\n", isspace('\t')? "yes": "no");  // yes
    printf("%s\n", isspace(',')? "yes": "no");   // no
    printf("%s\n", isspace('!')? "yes": "no");   // no
    printf("%s\n", isspace('c')? "yes": "no");   // no
}
```

See Also

`isblank()`

44.11 `isupper()`

Tests if a character is uppercase

Synopsis

```
#include <ctype.h>

int isupper(int c);
```

Description

Tests if a character is uppercase, in the range `A-Z`.

In other locales, there could be other uppercase characters. In all cases, to be uppercase, the following must be true:

```
!iscntrl(c) && !isdigit(c) && !ispunct(c) && !isspace(c)
```

Return Value

Returns true if the character is uppercase.

Example

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
```

```
    //              testing this char
    //                       v
    printf("%s\n", isupper('B')? "yes": "no");   // yes
    printf("%s\n", isupper('c')? "yes": "no");   // no
    printf("%s\n", isupper('0')? "yes": "no");   // no
    printf("%s\n", isupper('?')? "yes": "no");   // no
    printf("%s\n", isupper(' ')? "yes": "no");   // no
}
```

See Also

`islower()`, `isalpha()`, `toupper()`, `tolower()`

44.12 `isxdigit()`

Tests if a character is a hexadecimal digit

Synopsis

```
#include <ctype.h>

int isxdigit(int c);
```

Description

Returns true if the character is a hexadecimal digit. Namely if it's `0-9`, `a-f`, or `A-F`.

Return Value

True if the character is a hexadecimal digit.

Example

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    //              testing this char
    //                       v
    printf("%s\n", isxdigit('B')? "yes": "no");   // yes
    printf("%s\n", isxdigit('c')? "yes": "no");   // yes
    printf("%s\n", isxdigit('2')? "yes": "no");   // yes
    printf("%s\n", isxdigit('G')? "yes": "no");   // no
    printf("%s\n", isxdigit('?')? "yes": "no");   // no
}
```

See Also

`isdigit()`

44.13 tolower()

Convert a letter to lowercase

Synopsis

```
#include <ctype.h>

int tolower(int c);
```

Description

If the character is uppercase (i.e. `isupper(c)` is true), this function returns the corresponding lowercase letter.

Different locales might have different upper- and lowercase letters.

Return Value

Returns the lowercase value for an uppercase letter. If the letter isn't uppercase, returns it unchanged.

Example

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    //             changing this char
    //                       v
    printf("%c\n", tolower('B'));  // b (made lowercase!)
    printf("%c\n", tolower('e'));  // e (unchanged)
    printf("%c\n", tolower('!'));  // ! (unchanged)
}
```

See Also

`toupper()`, `islower()`, `isupper()`

44.14 toupper()

Convert a letter to uppercase

Synopsis

```
#include <ctype.h>

int toupper(int c);
```

Description

If the character is lower (i.e. `islower(c)` is true), this function returns the corresponding uppercase letter.

Different locales might have different upper- and lowercase letters.

Return Value

Returns the uppercase value for a lowercase letter. If the letter isn't lowercase, returns it unchanged.

Example

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    //             changing this char
    //                      v
    printf("%c\n", toupper('B'));  // B (unchanged)
    printf("%c\n", toupper('e'));  // E (made uppercase!)
    printf("%c\n", toupper('!'));  // ! (unchanged)
}
```

See Also

`tolower()`, `islower()`, `isupper()`

Chapter 45

<errno.h> Error Information

Variable	Description
`errno`	Holds the error status of the last call

This header defines a single variable[1], `errno`, that can be checked to see if an error has occurred.

`errno` is set to `0` on startup, but no library function sets it to `0`. If you're going to use solely it to check for errors, set it to `0` before the call and then check it after. Not only that, but if there's no error, all library functions will leave the value of `errno` unchanged.

Often, though, you'll get some error indication from the function you're calling then check `errno` to see what went wrong.

This is commonly used in conjunction with `perror()` to get a human-readable error message that corresponds to the specific error.

Important Safety Tip: You should never make your own variable called `errno`—that's undefined behavior.

Note that the C Spec defines less than a handful of values `errno` can take on. Unix defines a bunch more[2], as does Windows[3].

45.1 `errno`

Holds the error status of the last call

Synopsis

```
errno   // Type is undefined, but it's assignable
```

Description

Indicates the error status of the last call (note that not all calls will set this value).

[1] Really it's just required to be a modifiable lvalue, so not necessarily a variable. But you can treat it as such.
[2] https://man.archlinux.org/man/errno.3.en
[3] https://docs.microsoft.com/en-us/cpp/c-runtime-library/errno-constants?view=msvc-160

www.dbooks.org

Value	Description
`0`	No error
`EDOM`	Domain error (from math)
`EILSEQ`	Encoding error (from character conversion)
`ERANGE`	Range error (from math)

If you're doing a number of math functions, you might come across `EDOM` or `ERANGE`.

With multibyte/wide character conversion functions, you might see `EILSEQ`.

And your system might define any other number of values that `errno` could be set to, all of which will begin with the letter `E`.

Fun Fact: you can use `EDOM`, `EILSEQ`, and `ERANGE` with preprocessor directives such as `#ifdef`. But, frankly, I'm not sure why you'd do that other than to test their existence.

Example

The following prints an error message, since passing `2.0` to `acos()` is outside the function's domain.

```c
#include <stdio.h>
#include <math.h>
#include <errno.h>

int main(void)
{
    double x;

    errno = 0;       // Make sure this is clear before the call

    x = acos(2.0);   // Invalid argument to acos()

    if (errno == EDOM)
        perror("acos");
    else
        printf("Answer is %f\n", x);

    return 0;
}
```

Output:

```
acos: Numerical argument out of domain
```

The following prints an error message (on my system), since passing `1e+30` to `exp()` produces a result that's outside the range of a `double`.

```c
#include <stdio.h>
#include <math.h>
#include <errno.h>

int main(void)
{
    double x;

    errno = 0;       // Make sure this is clear before the call
```

```

    x = exp(1e+30);  // Pass in some too-huge number

    if (errno == ERANGE)
        perror("exp");
    else
        printf("Answer is %f\n", x);

    return 0;
}
```

Output:

```
exp: Numerical result out of range
```

This example tries to convert an invalid character into a wide character, failing. This sets `errno` to `EILSEQ`. We then use `perror()` to print an error message.

```
#include <stdio.h>
#include <string.h>
#include <wchar.h>
#include <errno.h>
#include <locale.h>

int main(void)
{
    setlocale(LC_ALL, "");

    char *bad_str = "\xff";  // Probably invalid char in this locale
    wchar_t wc;
    size_t result;
    mbstate_t ps;

    memset(&ps, 0, sizeof ps);

    result = mbrtowc(&wc, bad_str, 1, &ps);

    if (result == (size_t)(-1))
        perror("mbrtowc");  // mbrtowc: Illegal byte sequence
    else
        printf("Converted to L'%lc'\n", wc);

    return 0;
}
```

Output:

```
mbrtowc: Invalid or incomplete multibyte or wide character
```

See Also

`perror()`, `mbrtoc16()`, `c16rtomb()`, `mbrtoc32()`, `c32rtomb()`, `fgetwc()`, `fputwc()`, `mbrtowc()`, `wcrtomb()`, `mbsrtowcs()`, `wcsrtombs()`, `<math.h>`,

Chapter 46

<fenv.h> Floating Point Exceptions and Environment

Function	Description
`feclearexcept()`	Clear floating point exceptions
`fegetexceptflag()`	Save the floating point exception flags
`fesetexceptflag()`	Restore the floating point exception flags
`feraiseexcept()`	Raise a floating point exception through software
`fetestexcept()`	Test to see if an exception has occurred
`fegetround()`	Get the rounding direction
`fesetround()`	Set the rounding direction
`fegetenv()`	Save the entire floating point environment
`fesetenv()`	Restore the entire floating point environment
`feholdexcept()`	Save floating point state and install non-stop mode
`feupdateenv()`	Restore floating point environment and apply recent exceptions

46.1 Types and Macros

There are two types defined in this header:

Type	Description
`fenv_t`	The entire floating point environment
`fexcept_t`	A set of floating point exceptions

The "environment" can be thought of as the status at this moment of the floating point processing system: this includes the exceptions, rounding, etc. It's an opaque type, so you won't be able to access it directly, and it must be done through the proper functions.

If the functions in question exist on your system (they might not be!), then you'll also have these macros defined to represent different exceptions:

Macro	Description
`FE_DIVBYZERO`	Division by zero
`FE_INEXACT`	Result was not exact, was rounded

Macro	Description
FE_INVALID	Domain error
FE_OVERFLOW	Numeric overflow
FE_UNDERFLOW	Numeric underflow
FE_ALL_EXCEPT	All of the above combined

The idea is that you can bitwise-OR these together to represent multiple exceptions, e.g. FE_INVALID|FE_OVERFLOW.

The functions, below, that have an excepts parameter will take these values.

See <math.h> for which functions raise which exceptions and when.

46.2 Pragmas

Normally C is free to optimize all kinds of stuff that might cause the flags to not look like you might expect. So if you're going to use this stuff, be sure to set this pragma:

```
#pragma STDC FENV_ACCESS ON
```

If you do this at global scope, it remains in effect until you turn it off:

```
#pragma STDC FENV_ACCESS OFF
```

If you do it in block scope, it has to come before any statements or declarations. In this case, it has effect until the block ends (or until it is explicitly turned off.)

A caveat: this program isn't supported on either of the compilers I have (gcc and clang) as of this writing, so though I have built the code, below, it's not particularly well-tested.

46.3 feclearexcept()

Clear floating point exceptions

Synopsis

```
#include <fenv.h>

int feclearexcept(int excepts);
```

Description

If a floating point exception has occurred, this function can clear it.

Set excepts to a bitwise-OR list of exceptions to clear.

Passing 0 has no effect.

Return Value

Returns 0 on success and non-zero on failure.

Example

```
#include <stdio.h>
#include <math.h>
#include <fenv.h>

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    double f = sqrt(-1);

    int r = feclearexcept(FE_INVALID);

    printf("%d %f\n", r, f);
}
```

See Also

`feraiseexcept()`, `fetestexcept()`

46.4 `fegetexceptflag()` `fesetexceptflag()`

Save or restore the floating point exception flags

Synopsis

```
#include <fenv.h>

int fegetexceptflag(fexcept_t *flagp, int excepts);

int fesetexceptflag(fexcept_t *flagp, int excepts);
```

Description

Use these functions to save or restore the current floating point environment in a variable.

Set `excepts` to the set of exceptions you want to save or restore the state of. Setting it to `FE_ALL_EXCEPT` will save or restore the entire state.

Note that `fexcept_t` is an opaque type—you don't know what's in it.

`excepts` can be set to zero for no effect.

Return Value

Returns `0` on success or if `excepts` is zero.

Returns non-zero on failure.

Example

This program saves the state (before any error has happened), then deliberately causes a domain error by trying to take $\sqrt{-1}$.

After that, it restores the floating point state to before the error had occurred, thereby clearing it.

```
#include <stdio.h>
#include <math.h>
#include <fenv.h>

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    fexcept_t flag;

    fegetexceptflag(&flag, FE_ALL_EXCEPT);  // Save state

    double f = sqrt(-1);                    // I imagine this won't work
    printf("%f\n", f);                      // "nan"

    if (fetestexcept(FE_INVALID))
        printf("1: Domain error\n");        // This prints!
    else
        printf("1: No domain error\n");

    fesetexceptflag(&flag, FE_ALL_EXCEPT);  // Restore to before error

    if (fetestexcept(FE_INVALID))
        printf("2: Domain error\n");
    else
        printf("2: No domain error\n");     // This prints!
}
```

46.5 feraiseexcept()

Raise a floating point exception through software

Synopsis

```
#include <fenv.h>

int feraiseexcept(int excepts);
```

Description

This attempts to raise a floating point exception as if it had happened.

You can specify multiple exceptions to raise.

If either `FE_UNDERFLOW` or `FE_OVERFLOW` is raised, C *might* also raise `FE_INEXACT`.

If either `FE_UNDERFLOW` or `FE_OVERFLOW` is raised at the same time as `FE_INEXACT`, then `FE_UNDERFLOW` or `FE_OVERFLOW` will be raised *before* `FE_INEXACT` behind the scenes.

The order the other exceptions are raised is undefined.

Return Value

Returns `0` if all the exceptions were raised or if `excepts` is `0`.

Returns non-zero otherwise.

Example

This code deliberately raises a division-by-zero exception and then detects it.

```
#include <stdio.h>
#include <math.h>
#include <fenv.h>

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    feraiseexcept(FE_DIVBYZERO);

    if (fetestexcept(FE_DIVBYZERO) == FE_DIVBYZERO)
        printf("Detected division by zero\n");  // This prints!!
    else
        printf("This is fine.\n");
}
```

See Also

`feclearexcept()`, `fetestexcept()`

46.6 `fetestexcept()`

Test to see if an exception has occurred

Synopsis

```
#include <fenv.h>

int fetestexcept(int excepts);
```

Description

Put the exceptions you want to test in `excepts`, bitwise-ORing them together.

Return Value

Returns the bitwise-OR of the exceptions that have been raised.

Example

This code deliberately raises a division-by-zero exception and then detects it.

```
#include <stdio.h>
#include <math.h>
#include <fenv.h>

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    feraiseexcept(FE_DIVBYZERO);

    if (fetestexcept(FE_DIVBYZERO) == FE_DIVBYZERO)
        printf("Detected division by zero\n");  // This prints!!
    else
        printf("This is fine.\n");
}
```

See Also

`feclearexcept()`, `feraiseexcept()`

46.7 `fegetround()` `fesetround()`

Get or set the rounding direction

Synopsis

```
#include <fenv.h>

int fegetround(void);

int fesetround(int round);
```

Description

Use these to get or set the rounding direction used by a variety of math functions.

Basically when a function "rounds" a number, it wants to know how to do it. By default, it does it how we tend to expect: if the fractional part is less than 0.5, it rounds down closer to zero, otherwise up farther from zero.

Macro	Description
`FE_TONEAREST`	Round to the nearest whole number, the default
`FE_TOWARDZERO`	Round toward zero always
`FE_DOWNWARD`	Round toward the next lesser whole number
`FE_UPWARD`	Round toward the next greater whole number

Some implementations don't support rounding. If it does, the above macros will be defined.

Note that the `round()` function is always "to-nearest" and doesn't pay attention to the rounding mode.

Return Value

`fegetround()` returns the current rounding direction, or a negative value on error.

`fesetround()` returns zero on success, or non-zero on failure.

Example

This rounds some numbers

```c
#include <stdio.h>
#include <math.h>
#include <fenv.h>

// Helper function to print the rounding mode
const char *rounding_mode_str(int mode)
{
    switch (mode) {
        case FE_TONEAREST:  return "FE_TONEAREST";
        case FE_TOWARDZERO: return "FE_TOWARDZERO";
        case FE_DOWNWARD:   return "FE_DOWNWARD";
        case FE_UPWARD:     return "FE_UPWARD";
    }

    return "Unknown";
}

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    int rm;

    rm = fegetround();

    printf("%s\n", rounding_mode_str(rm));    // Print current mode
    printf("%f %f\n", rint(2.1), rint(2.7));  // Try rounding

    fesetround(FE_TOWARDZERO);                // Set the mode

    rm = fegetround();

    printf("%s\n", rounding_mode_str(rm));    // Print it
    printf("%f %f\n", rint(2.1), rint(2.7));  // Try it now!
}
```

Output:

```
FE_TONEAREST
2.000000 3.000000
FE_TOWARDZERO
2.000000 2.000000
```

See Also

nearbyint(), nearbyintf(), nearbyintl(), rint(), rintf(), rintl(), lrint(), lrintf(), lrintl(), llrint(), llrintf(), llrintl()

46.8 fegetenv() fesetenv()

Save or restore the entire floating point environment

Synopsis

```
#include <fenv.h>

int fegetenv(fenv_t *envp);
int fesetenv(const fenv_t *envp);
```

Description

You can save the environment (exceptions, rounding direction, etc.) by calling fegetenv() and restore it with fesetenv().

Use this if you want to restore the state after a function call, i.e. hide from the caller that some floating point exceptions or changes occurred.

Return Value

fegetenv() and fesetenv() return 0 on success, and non-zero otherwise.

Example

This example saves the environment, messes with the rounding and exceptions, then restores it. After the environment is restored, we see that the rounding is back to default and the exception is cleared.

```
#include <stdio.h>
#include <math.h>
#include <fenv.h>

void show_status(void)
{
    printf("Rounding is FE_TOWARDZERO: %d\n",
           fegetround() == FE_TOWARDZERO);

    printf("FE_DIVBYZERO is set: %d\n",
           fetestexcept(FE_DIVBYZERO) != 0);
}

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    fenv_t env;

    fegetenv(&env);  // Save the environment
```

```

    fesetround(FE_TOWARDZERO);    // Change rounding
    feraiseexcept(FE_DIVBYZERO);  // Raise an exception

    show_status();

    fesetenv(&env);  // Restore the environment

    show_status();
}
```

Output:

```
Rounding is FE_TOWARDZERO: 1
FE_DIVBYZERO is set: 1
Rounding is FE_TOWARDZERO: 0
FE_DIVBYZERO is set: 0
```

See Also

`feholdexcept()`, `feupdateenv()`

46.9 `feholdexcept()`

Save floating point state and install non-stop mode

Synopsis

```
#include <fenv.h>

int feholdexcept(fenv_t *envp);
```

Description

This is just like `fegetenv()` except that it updates the current environment to be in *non-stop* mode, namely it won't halt on any exceptions.

It remains in this state until you restore the state with `fesetenv()` or `feupdateenv()`.

Return Value

Example

This example saves the environment and goes into non-stop mode, messes with the rounding and exceptions, then restores it. After the environment is restored, we see that the rounding is back to default and the exception is cleared. We'll also be out of non-stop mode.

```
#include <stdio.h>
#include <math.h>
#include <fenv.h>

void show_status(void)
{
```

```
    printf("Rounding is FE_TOWARDZERO: %d\n",
           fegetround() == FE_TOWARDZERO);

    printf("FE_DIVBYZERO is set: %d\n",
           fetestexcept(FE_DIVBYZERO) != 0);
}

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    fenv_t env;

    // Save the environment and don't stop on exceptions
    feholdexcept(&env);

    fesetround(FE_TOWARDZERO);    // Change rounding
    feraiseexcept(FE_DIVBYZERO);  // Raise an exception

    show_status();

    fesetenv(&env);  // Restore the environment

    show_status();
}
```

See Also

`fegetenv()`, `fesetenv()`, `feupdateenv()`

46.10 `feupdateenv()`

Restore floating point environment and apply recent exceptions

Synopsis

```
#include <fenv.h>

int feupdateenv(const fenv_t *envp);
```

Description

This is like `fesetenv()` except that it modifies the passed-in environment so that it is updated with exceptions that have happened in the meantime.

So let's say you had a function that might raise exceptions, but you wanted to hide those in the caller. One option might be to:

1. Save the environment with `fegetenv()` or `feholdexcept()`.
2. Do whatever you do that might raise exceptions.
3. Restore the environment with `fesetenv()`, thereby hiding the exceptions that happened in step 2.

But that hides *all* exceptions. What if you just wanted to hide some of them? You could use `feupdateenv()` like this:

1. Save the environment with `fegetenv()` or `feholdexcept()`.
2. Do whatever you do that might raise exceptions.
3. Call `feclearexcept()` to clear the exceptions you want to hide from the caller.
4. Call `feupdateenv()` to restore the previous environment and update it with the other exceptions that have occurred.

So it's like a more capable way of restoring the environment than simply `fegetenv()`/`fesetenv()`.

Return Value

Returns `0` on success, non-zero otherwise.

Example

This program saves state, raises some exceptions, then clears one of the exceptions, then restores and updates the state.

```c
#include <stdio.h>
#include <math.h>
#include <fenv.h>

void show_status(void)
{
    printf("FE_DIVBYZERO: %d\n", fetestexcept(FE_DIVBYZERO) != 0);
    printf("FE_INVALID  : %d\n", fetestexcept(FE_INVALID) != 0);
    printf("FE_OVERFLOW : %d\n\n", fetestexcept(FE_OVERFLOW) != 0);
}

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    fenv_t env;

    feholdexcept(&env);  // Save the environment

    // Pretend some bad math happened here:
    feraiseexcept(FE_DIVBYZERO);  // Raise an exception
    feraiseexcept(FE_INVALID);    // Raise an exception
    feraiseexcept(FE_OVERFLOW);   // Raise an exception

    show_status();

    feclearexcept(FE_INVALID);

    feupdateenv(&env);  // Restore the environment

    show_status();
}
```

In the output, at first we have no exceptions. Then we have the three we raised. Then after we restore/update the environment, we see the one we cleared (`FE_INVALID`) hasn't been applied:

```
FE_DIVBYZERO: 0
FE_INVALID  : 0
FE_OVERFLOW : 0

FE_DIVBYZERO: 1
FE_INVALID  : 1
FE_OVERFLOW : 1

FE_DIVBYZERO: 1
FE_INVALID  : 0
FE_OVERFLOW : 1
```

See Also

`fegetenv()`, `fesetenv()`, `feholdexcept()`, `feclearexcept()`

www.dbooks.org

Chapter 47

<float.h> Floating Point Limits

Macro	Minimum Magnitude	Description
FLT_ROUNDS		Current rounding mode
FLT_EVAL_METHOD		Types used for evaluation
FLT_HAS_SUBNORM		Subnormal support for float
DBL_HAS_SUBNORM		Subnormal support for double
LDBL_HAS_SUBNORM		Subnormal support for long double
FLT_RADIX	2	Floating point radix (base)
FLT_MANT_DIG		Number of base FLT_RADIX digits in a float
DBL_MANT_DIG		Number of base FLT_RADIX digits in a double
LDBL_MANT_DIG		Number of base FLT_RADIX digits in a long double
FLT_DECIMAL_DIG	6	Number of decimal digits required to encode a float
DBL_DECIMAL_DIG	10	Number of decimal digits required to encode a double
LDBL_DECIMAL_DIG	10	Number of decimal digits required to encode a long double
DECIMAL_DIG	10	Number of decimal digits required to encode the the widest floating point number supported
FLT_DIG	6	Number of decimal digits that can be safely stored in a float
DBL_DIG	10	Number of decimal digits that can be safely stored in a double
LDBL_DIG	10	Number of decimal digits that can be safely stored in a long double
FLT_MIN_EXP		FLT_RADIX to the FLT_MIN_EXP-1 power is the smallest normalized float
DBL_MIN_EXP		FLT_RADIX to the DBL_MIN_EXP-1 power is the smallest normalized double
LDBL_MIN_EXP		FLT_RADIX to the LDBL_MIN_EXP-1 power is the smallest normalized long double
FLT_MIN_10_EXP	-37	Minimum exponent such that 10 to this number is a normalized float
DBL_MIN_10_EXP	-37	Minimum exponent such that 10 to this number is a normalized double
LDBL_MIN_10_EXP	-37	Minimum exponent such that 10 to this number is a normalized long_double
FLT_MAX_EXP		FLT_RADIX to the FLT_MAX_EXP-1 power is the largest finite float

Macro	Minimum Magnitude	Description
DBL_MAX_EXP		FLT_RADIX to the DBL_MAX_EXP-1 power is the largest finite double
LDBL_MAX_EXP		FLT_RADIX to the LDBL_MAX_EXP-1 power is the largest finite long double
FLT_MAX_10_EXP	-37	Minimum exponent such that 10 to this number is a finite float
DBL_MAX_10_EXP	-37	Minimum exponent such that 10 to this number is a finite double
LDBL_MAX_10_EXP	-37	Minimum exponent such that 10 to this number is a finite long_double
FLT_MAX	1E+37	Largest finite float
DBL_MAX	1E+37	Largest finite double
LDBL_MAX	1E+37	Largest finite long double

Macro	Maximum Value	Description
FLT_EPSILON	1E-5	Difference between 1 and the next biggest representable float
DBL_EPSILON	1E-9	Difference between 1 and the next biggest representable double
LDBL_EPSILON	1E-9	Difference between 1 and the next biggest representable long double
FLT_MIN	1E-37	Minimum positive normalized float
DBL_MIN	1E-37	Minimum positive normalized double
LDBL_MIN	1E-37	Minimum positive normalized long double
FLT_TRUE_MIN	1E-37	Minimum positive float
DBL_TRUE_MIN	1E-37	Minimum positive double
LDBL_TRUE_MIN	1E-37	Minimum positive long double

The minimum and maximum values here are from the spec—they should what you can at least expect across all platforms. Your super dooper machine might do better, still!

47.1 Background

The spec allows a lot of leeway when it comes to how C represents floating point numbers. This header file spells out the limits on those numbers.

It gives a model that can describe any floating point number that I *know* you're going to absolutely love. It looks like this:

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}, e_{min} \leq e \leq e_{max}$$

where:

Variable	Meaning
s	Sign, -1 or 1
b	Base (radix), probably 2 on your system
e	Exponent
p	Precision: how many base-b digits in the number

www.dbooks.org

Variable	Meaning
f_k	The individual digits of the number, the significand

But let's blissfully ignore all that for a second.

Let's assume your computer uses base 2 for it's floating point (it probably does). And that in the example below the 1s-and-0s numbers are in binary, and the rest are in decimal.

The short of it is you could have floating point numbers like shown in this example:

$-0.10100101 \times 2^5 = -10100.101 = -20.625$

That's your fractional part multiplied by the base to the exponent's power. The exponent controls where the decimal point is. It "floats" around!

47.2 `FLT_ROUNDS` Details

This tells you the rounding mode. It can be changed with a call to `fesetround()`.

Mode	Description
`-1`	Indeterminable
`0`	Toward zero
`1`	To nearest
`2`	Toward positive infinity
`3`	Toward negative infinity… and beyond!

Unlike every other macro in this here header, `FLT_ROUNDS` might not be a constant expression.

47.3 `FLT_EVAL_METHOD` Details

This basically tells you how floating point values are promoted to different types in expressions.

Method	Description
`-1`	Indeterminable
`0`	Evaluate all operations and constants to the precision of their respective types
`1`	Evaluate `float` and `double` operations as `double` and `long double` ops as `long double`
`2`	Evaluate all operations and constants as `long double`

47.4 Subnormal Numbers

The macros `FLT_HAS_SUBNORM`, `DBL_HAS_SUBNORM`, and `LDBL_HAS_SUBNORM` all let you know if those types support subnormal numbers[1].

Value	Description
`-1`	Indeterminable
`0`	Subnormals not supported for this type

[1] https://en.wikipedia.org/wiki/Subnormal_number

Value	Description
1	Subnormals supported for this type

47.5 How Many Decimal Places Can I Use?

It depends on what you want to do.

The safe thing is if you never use more than FLT_DIG base-10 digits in your float, you're good. (Same for DBL_DIG and LDBL_DIG for their types.)

And by "use" I mean print out, have in code, read from the keyboard, etc.

You can print out that many decimal places with printf() and the %g format specifier:

```
#include <stdio.h>
#include <float.h>

int main(void)
{
    float pi = 3.14159265358979323846264338327950288419 71;

    // With %g or %G, the precision refers to the number of significant
    // digits:

    printf("%.*g\n", FLT_DIG, pi);  // For me: 3.14159

    // But %f prints too many, since the precision is the number of
    // digits to the right of the decimal--it doesn't count the digits
    // to the left of it:

    printf("%.*f\n", FLT_DIG, pi);  // For me: 3.14159... 3 ???
}
```

That's the end, but stay tuned for the exciting conclusion of "How Many Decimal Places Can I Use?"

Because base 10 and base 2 (your typical FLT_RADIX) don't mix very well, you can actually have more than FLT_DIG in your float; the bits of storage go out a little farther. But these might round in a way you don't expect.

But if you want to convert a floating point number to base 10 and then be able to convert it back again to the exact same floating point number, you'll need FLT_DECIMAL_DIG digits from your float to make sure you get those extra bits of storage represented.

Here's some example output that shows how the value stored might have some extra decimal places at the end.

```
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include <float.h>

int main(void)
{
    printf("FLT_DIG = %d\n", FLT_DIG);
    printf("FLT_DECIMAL_DIG = %d\n\n", FLT_DECIMAL_DIG);

```

```
    assert(FLT_DIG == 6);  // Code below assumes this

    for (float x = 0.123456; x < 0.12346; x += 0.000001) {
        printf("As written: %.*g\n", FLT_DIG, x);
        printf("As stored:  %.*g\n\n", FLT_DECIMAL_DIG, x);
    }
}
```

And the output on my machine, starting at `0.123456` and incrementing by `0.000001` each time:

```
FLT_DIG = 6
FLT_DECIMAL_DIG = 9

As written: 0.123456
As stored:  0.123456001

As written: 0.123457
As stored:  0.123457

As written: 0.123458
As stored:  0.123457998

As written: 0.123459
As stored:  0.123458996

As written: 0.12346
As stored:  0.123459995
```

You can see that the value stored isn't always the value we're expecting since base-2 can't represent all base-10 fractions exactly. The best it can do is store more places and then round.

Also notice that even though we tried to stop the `for` loop *before* `0.123460`, it actually ran including that value since the stored version of that number was `0.123459995`, which is still less than `0.123460`.

Aren't floating point numbers fun?

47.6 Comprehensive Example

Here's a program that prints out the details for a particular machine:

```
#include <stdio.h>
#include <float.h>

int main(void)
{
    printf("FLT_RADIX: %d\n", FLT_RADIX);
    printf("FLT_ROUNDS: %d\n", FLT_ROUNDS);
    printf("FLT_EVAL_METHOD: %d\n", FLT_EVAL_METHOD);
    printf("DECIMAL_DIG: %d\n\n", DECIMAL_DIG);

    printf("FLT_HAS_SUBNORM: %d\n", FLT_HAS_SUBNORM);
    printf("FLT_MANT_DIG: %d\n", FLT_MANT_DIG);
    printf("FLT_DECIMAL_DIG: %d\n", FLT_DECIMAL_DIG);
    printf("FLT_DIG: %d\n", FLT_DIG);
    printf("FLT_MIN_EXP: %d\n", FLT_MIN_EXP);
    printf("FLT_MIN_10_EXP: %d\n", FLT_MIN_10_EXP);
```

```
    printf("FLT_MAX_EXP: %d\n", FLT_MAX_EXP);
    printf("FLT_MAX_10_EXP: %d\n", FLT_MAX_10_EXP);
    printf("FLT_MIN: %.*e\n", FLT_DECIMAL_DIG, FLT_MIN);
    printf("FLT_MAX: %.*e\n", FLT_DECIMAL_DIG, FLT_MAX);
    printf("FLT_EPSILON: %.*e\n", FLT_DECIMAL_DIG, FLT_EPSILON);
    printf("FLT_TRUE_MIN: %.*e\n\n", FLT_DECIMAL_DIG, FLT_TRUE_MIN);

    printf("DBL_HAS_SUBNORM: %d\n", DBL_HAS_SUBNORM);
    printf("DBL_MANT_DIG: %d\n", DBL_MANT_DIG);
    printf("DBL_DECIMAL_DIG: %d\n", DBL_DECIMAL_DIG);
    printf("DBL_DIG: %d\n", DBL_DIG);
    printf("DBL_MIN_EXP: %d\n", DBL_MIN_EXP);
    printf("DBL_MIN_10_EXP: %d\n", DBL_MIN_10_EXP);
    printf("DBL_MAX_EXP: %d\n", DBL_MAX_EXP);
    printf("DBL_MAX_10_EXP: %d\n", DBL_MAX_10_EXP);
    printf("DBL_MIN: %.*e\n", DBL_DECIMAL_DIG, DBL_MIN);
    printf("DBL_MAX: %.*e\n", DBL_DECIMAL_DIG, DBL_MAX);
    printf("DBL_EPSILON: %.*e\n", DBL_DECIMAL_DIG, DBL_EPSILON);
    printf("DBL_TRUE_MIN: %.*e\n\n", DBL_DECIMAL_DIG, DBL_TRUE_MIN);

    printf("LDBL_HAS_SUBNORM: %d\n", LDBL_HAS_SUBNORM);
    printf("LDBL_MANT_DIG: %d\n", LDBL_MANT_DIG);
    printf("LDBL_DECIMAL_DIG: %d\n", LDBL_DECIMAL_DIG);
    printf("LDBL_DIG: %d\n", LDBL_DIG);
    printf("LDBL_MIN_EXP: %d\n", LDBL_MIN_EXP);
    printf("LDBL_MIN_10_EXP: %d\n", LDBL_MIN_10_EXP);
    printf("LDBL_MAX_EXP: %d\n", LDBL_MAX_EXP);
    printf("LDBL_MAX_10_EXP: %d\n", LDBL_MAX_10_EXP);
    printf("LDBL_MIN: %.*Le\n", LDBL_DECIMAL_DIG, LDBL_MIN);
    printf("LDBL_MAX: %.*Le\n", LDBL_DECIMAL_DIG, LDBL_MAX);
    printf("LDBL_EPSILON: %.*Le\n", LDBL_DECIMAL_DIG, LDBL_EPSILON);
    printf("LDBL_TRUE_MIN: %.*Le\n\n", LDBL_DECIMAL_DIG, LDBL_TRUE_MIN);

    printf("sizeof(float): %zu\n", sizeof(float));
    printf("sizeof(double): %zu\n", sizeof(double));
    printf("sizeof(long double): %zu\n", sizeof(long double));
}
```

And here's the output on my machine:

```
FLT_RADIX: 2
FLT_ROUNDS: 1
FLT_EVAL_METHOD: 0
DECIMAL_DIG: 21

FLT_HAS_SUBNORM: 1
FLT_MANT_DIG: 24
FLT_DECIMAL_DIG: 9
FLT_DIG: 6
FLT_MIN_EXP: -125
FLT_MIN_10_EXP: -37
FLT_MAX_EXP: 128
FLT_MAX_10_EXP: 38
FLT_MIN: 1.175494351e-38
```

```
FLT_MAX: 3.402823466e+38
FLT_EPSILON: 1.192092896e-07
FLT_TRUE_MIN: 1.401298464e-45

DBL_HAS_SUBNORM: 1
DBL_MANT_DIG: 53
DBL_DECIMAL_DIG: 17
DBL_DIG: 15
DBL_MIN_EXP: -1021
DBL_MIN_10_EXP: -307
DBL_MAX_EXP: 1024
DBL_MAX_10_EXP: 308
DBL_MIN: 2.22507385850720138e-308
DBL_MAX: 1.79769313486231571e+308
DBL_EPSILON: 2.22044604925031308e-16
DBL_TRUE_MIN: 4.94065645841246544e-324

LDBL_HAS_SUBNORM: 1
LDBL_MANT_DIG: 64
LDBL_DECIMAL_DIG: 21
LDBL_DIG: 18
LDBL_MIN_EXP: -16381
LDBL_MIN_10_EXP: -4931
LDBL_MAX_EXP: 16384
LDBL_MAX_10_EXP: 4932
LDBL_MIN: 3.36210314311209350626 3e-4932
LDBL_MAX: 1.18973149535723176502 1e+4932
LDBL_EPSILON: 1.08420217248550443400 7e-19
LDBL_TRUE_MIN: 3.64519953188247460252 8e-4951

sizeof(float): 4
sizeof(double): 8
sizeof(long double): 16
```

Chapter 48

<inttypes.h> More Integer Conversions

Function	Description
imaxabs()	Compute the absolute value of an intmax_t
imaxdiv()	Compute the quotient and remainder of intmax_ts
strtoimax()	Convert strings to type intmax_t
strtoumax()	Convert strings to type uintmax_t
wcstoimax()	Convert wide strings to type intmax_t
wcstoumax()	Convert wide strings to type uintmax_t

This header does conversions to maximum sized integers, division with maximum sized integers, and also provides format specifiers for printf() and scanf() for a variety of types defined in <stdint.h>.

The header <stdint.h> is included by this one.

48.1 Macros

These are to help with printf() and scanf() when you use a type such as int_least16_t... what format specifiers do you use?

Let's start with printf()—all these macros start with PRI and then are followed by the format specifier you'd typically use for that type. Lastly, the number of bits is added on.

For example, the format specifier for a 64-bit integer is PRId64—the d is because you usually print integers with "%d".

An unsigned 16-bit integer could be printed with PRIu16.

These macros expand to string literals. We can take advantage of the fact that C automatically concatenates neighboring string literals and use these specifiers like this:

```
#include <stdio.h>     // for printf()
#include <inttypes.h>

int main(void)
{
    int16_t x = 32;

```


www.dbooks.org

```
8     printf("The value is %" PRId16 "!\n", x);
9 }
```

There's nothing magical happening on line 8, above. Indeed, if I print out the value of the macro:

```
printf("%s\n", PRId16);
```

we get this on my system:

```
hd
```

which is a `printf()` format specifier meaning "short signed integer" .

So back to line 8, after string literal concatenation, it's just as if I'd typed:

```
8     printf("The value is %hd!\n", x);
```

Here's a table of all the macros you can use for `printf()` format specifiers… substitute the number of bits for *N*, usually 8, 16, 32, or 64.

PRId*N*	PRIdLEAST*N*	PRIdFAST*N*	PRIdMAX	PRIdPTR
PRIi*N*	PRIiLEAST*N*	PRIiFAST*N*	PRIiMAX	PRIiPTR
PRIo*N*	PRIoLEAST*N*	PRIoFAST*N*	PRIoMAX	PRIoPTR
PRIu*N*	PRIuLEAST*N*	PRIuFAST*N*	PRIuMAX	PRIuPTR
PRIx*N*	PRIxLEAST*N*	PRIxFAST*N*	PRIxMAX	PRIxPTR
PRIX*N*	PRIXLEAST*N*	PRIXFAST*N*	PRIXMAX	PRIXPTR

Note again how the lowercase center letter represents the usual format specifiers you'd pass to `printf()`: `d`, `i`, `o`, `u`, `x`, and `X`.

And we have a similar set of macros for `scanf()` for reading in these various types:

SCNd*N*	SCNdLEAST*N*	SCNdFAST*N*	SCNdMAX	SCNdPTR
SCNi*N*	SCNiLEAST*N*	SCNiFAST*N*	SCNiMAX	SCNiPTR
SCNo*N*	SCNoLEAST*N*	SCNoFAST*N*	SCNoMAX	SCNoPTR
SCNu*N*	SCNuLEAST*N*	SCNuFAST*N*	SCNuMAX	SCNuPTR
SCNx*N*	SCNxLEAST*N*	SCNxFAST*N*	SCNxMAX	SCNxPTR

The rule is that for each type defined in `<stdint.h>` there will be corresponding `printf()` and `scanf()` macros defined here.

48.2 imaxabs()

Compute the absolute value of an `intmax_t`

Synopsis

```
#include <inttypes.h>

intmax_t imaxabs(intmax_t j);
```

Description

When you need the absolute value of the biggest integer type on the system, this is the function for you.

The spec notes that if the absolute value of the number cannot be represented, the behavior is undefined. This would happen if you tried to take the absolute value of the smallest possible negative number in a two's-complement system.

Return Value

Returns the absolute value of the input, $|j|$.

Example

```
#include <stdio.h>
#include <inttypes.h>

int main(void)
{
    intmax_t j = -3490;

    printf("%jd\n", imaxabs(j));    // 3490
}
```

See Also

`fabs()`

48.3 `imaxdiv()`

Compute the quotient and remainder of `intmax_t`s

Synopsis

```
#include <inttypes.h>

imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

Description

When you want to do integer division and remainder in a single operation, this function will do it for you.

It computes `numer/denom` and `numer%denom` and returns the result in a structure of type `imaxdiv_t`.

This structure has two `imaxdiv_t` fields, `quot` and `rem`, that you use to retrieve the sought-after values.

Return Value

Returns an `imaxdiv_t` containing the quotient and remainder of the operation.

Example

```
#include <stdio.h>
#include <inttypes.h>

int main(void)
{
    intmax_t numer = INTMAX_C(3490);
    intmax_t denom = INTMAX_C(17);

    imaxdiv_t r = imaxdiv(numer, denom);

    printf("Quotient: %jd, remainder: %jd\n", r.quot, r.rem);
}
```

Output:

```
Quotient: 205, remainder: 5
```

See Also

`remquo()`

48.4 `strtoimax()` `strtoumax()`

Convert strings to types `intmax_t` and `uintmax_t`

Synopsis

```
#include <inttypes.h>

intmax_t strtoimax(const char * restrict nptr, char ** restrict endptr,
                   int base);

uintmax_t strtoumax(const char * restrict nptr, char ** restrict endptr,
                   int base);
```

Description

These work just like the `strtol()` family of functions, except they return an `intmax_t` or `uintmax_t`.

See the `strtol()` reference page for details.

Return Value

Returns the converted string as an `intmax_t` or `uintmax_t`.

If the result is out of range, the returned value will be `INTMAX_MAX`, `INTMAX_MIN`, or `UINTMAX_MAX`, as appropriate. And the `errno` variable will be set to `ERANGE`.

Example

The following example converts a base-10 number to an `intmax_t`. Then it attempts to convert an invalid base-2 number, catching the error.

```
#include <stdio.h>
#include <inttypes.h>

int main(void)
{
    intmax_t r;
    char *endptr;

    // Valid base-10 number
    r = strtoimax("123456789012345", &endptr, 10);

    if (*endptr != '\0')
        printf("Invalid digit: %c\n", *endptr);
    else
        printf("Value is %jd\n", r);

    // The following binary number contains an invalid digit
    r = strtoimax("0100102010101101", &endptr, 2);

    if (*endptr != '\0')
        printf("Invalid digit: %c\n", *endptr);
    else
        printf("Value is %jd\n", r);
}
```

Output:

```
Value is 123456789012345
Invalid digit: 2
```

See Also

`strtol()`, `errno`

48.5 `wcstoimax()` `wcstoumax()`

Convert wide strings to types `intmax_t` and `uintmax_t`

Synopsis

```
#include <stddef.h> // for wchar_t
#include <inttypes.h>

intmax_t wcstoimax(const wchar_t * restrict nptr,
                   wchar_t ** restrict endptr, int base);

uintmax_t wcstoumax(const wchar_t * restrict nptr,
                    wchar_t ** restrict endptr, int base);
```

Description

These work just like the `wcstol()` family of functions, except they return an `intmax_t` or `uintmax_t`.

See the `wcstol()` reference page for details.

Return Value

Returns the converted wide string as an `intmax_t` or `uintmax_t`.

If the result is out of range, the returned value will be `INTMAX_MAX`, `INTMAX_MIN`, or `UINTMAX_MAX`, as appropriate. And the `errno` variable will be set to `ERANGE`.

Example

The following example converts a base-10 number to an `intmax_t`. Then it attempts to convert an invalid base-2 number, catching the error.

```c
#include <wchar.h>
#include <inttypes.h>

int main(void)
{
    intmax_t r;
    wchar_t *endptr;

    // Valid base-10 number
    r = wcstoimax(L"123456789012345", &endptr, 10);

    if (*endptr != '\0')
        wprintf(L"Invalid digit: %lc\n", *endptr);
    else
        wprintf(L"Value is %jd\n", r);

    // The following binary number contains an invalid digit
    r = wcstoimax(L"0100102010101101", &endptr, 2);

    if (*endptr != '\0')
        wprintf(L"Invalid digit: %lc\n", *endptr);
    else
        wprintf(L"Value is %jd\n", r);
}
```

```
Value is 123456789012345
Invalid digit: 2
```

See Also

`wcstol()`, `errno`

Chapter 49

<iso646.h> Alternative Operator Spellings

ISO-646 is a character encoding standard that's very similar to ASCII. But it's missing a few notable characters, like `|`, `^`, and `~`.

Since these are operators or parts of operators in C, this header file defines a number of macros you can use in case those characters aren't found on your keyboard. (And also C++ can use these same alternate spellings.)

Operator	<iso646.h> equivalent
&&	and
&=	and_eq
&	bitand
|	bitor
~	compl
!	not
!=	not_eq
||	or
|=	or_eq
^	xor
^=	xor_eq

Interestingly, there is no `eq` for `==`, and `&` and `!` are included despite being in ISO-646.

Example usage:

```
#include <stdio.h>
#include <iso646.h>

int main(void)
{
    int x = 12;
    int y = 30;

    if (x == 12 and y not_eq 40)
        printf("Now we know.\n");
}
```

I've personally never seen this file included, but I'm sure it gets used from time to time.

www.dbooks.org

Chapter 50

<limits.h> Numeric Limits

Important note: the "minimum magnitude" in the table below is the minimum allowed by the spec. It's very likely that the values on your bad-ass system exceed those, below.

Macro	Minimum Magnitude	Description
CHAR_BIT	8	Number of bits in a byte
SCHAR_MIN	-127	Minimum value of a signed char
SCHAR_MAX	127	Maximum value of a signed char
UCHAR_MAX	255	Maximum value of an unsigned char[1]
CHAR_MIN	0 or SCHAR_MIN	More detail below
CHAR_MAX	SCHAR_MAX or UCHAR_MAX	More detail below
MB_LEN_MAX	1	Maximum number of bytes in a multibyte character on any locale
SHRT_MIN	-32767	Minimum value of a short
SHRT_MAX	32767	Maximum value of a short
USHRT_MAX	65535	Maximum value of an unsigned short
INT_MIN	-32767	Minimum vale of an int
INT_MAX	32767	Maximum value of an int
UINT_MAX	65535	Maximum value of an unsigned int
LONG_MIN	-2147483647	Minimum value of a long
LONG_MAX	2147483647	Maximum value of a long
ULONG_MAX	4294967295	Maximum value of an unsigned long
LLONG_MIN	-9223372036854775807	Minimum value of a long long
LLONG_MAX	9223372036854775807	Maximum value of a long long
ULLONG_MAX	18446744073709551615	Maximum value of an unsigned long long

50.1 CHAR_MIN and CHAR_MAX

When it comes to the CHAR_MIN and CHAR_MAX macros, it all depends on if your char type is signed or unsigned by default. Remember that C leaves that up to the implementation? No? Well, it does.

So if it's signed, the values of CHAR_MIN and CHAR_MAX are the same as SCHAR_MIN and SCHAR_MAX.

And if it's unsigned, the values of CHAR_MIN and CHAR_MAX are the same as 0 and UCHAR_MAX.

[1] The minimum value of an unsigned char is 0. Same fo an unsigned short and unsigned long. Or any unsigned type, for that matter.

Side benefit: you can tell at runtime if the system has signed or unsigned chars by checking to see if `CHAR_MIN` is `0`.

```
#include <stdio.h>
#include <limits.h>

int main(void)
{
    printf("chars are %ssigned\n", CHAR_MIN == 0? "un": "");
}
```

On my system, `chars` are signed.

50.2 Choosing the Correct Type

If you want to be super portable, choose a type you know will be at least as big as you need by the table, above.

That said, a lot of code, for better or (likely) worse, assumes `ints` are 32-bits, when in actuality it's only guaranteed to be 16.

If you need a guaranteed bit size, check out the `int_leastN_t` types in `<stdint.h>`.

50.3 Whither Two's Complement?

If you were looking closely and have *a priori* knowledge of the matter, you might have thought I erred in the minimum values of the macros, above.

"`short` goes from `32767` to `-32767`? Shouldn't it go to `-32768`?"

No, I have it right. The spec list the minimum magnitudes for those macros, and some old-timey systems might have used a different encoding for their signed values that could only go that far.

Virtually every modern system uses Two's Complement[2] for signed numbers, and those would go from `32767` to `-32768` for a `short`. Your system probably does, too.

50.4 Demo Program

Here's a program to print out the values of the macros:

```
#include <stdio.h>
#include <limits.h>

int main(void)
{
    printf("CHAR_BIT = %d\n", CHAR_BIT);
    printf("SCHAR_MIN = %d\n", SCHAR_MIN);
    printf("SCHAR_MAX = %d\n", SCHAR_MAX);
    printf("UCHAR_MAX = %d\n", UCHAR_MAX);
    printf("CHAR_MIN = %d\n", CHAR_MIN);
    printf("CHAR_MAX = %d\n", CHAR_MAX);
    printf("MB_LEN_MAX = %d\n", MB_LEN_MAX);
    printf("SHRT_MIN = %d\n", SHRT_MIN);
    printf("SHRT_MAX = %d\n", SHRT_MAX);
```

[2] https://en.wikipedia.org/wiki/Two%27s_complement

www.dbooks.org

```
    printf("USHRT_MAX = %u\n", USHRT_MAX);
    printf("INT_MIN = %d\n", INT_MIN);
    printf("INT_MAX = %d\n", INT_MAX);
    printf("UINT_MAX = %u\n", UINT_MAX);
    printf("LONG_MIN = %ld\n", LONG_MIN);
    printf("LONG_MAX = %ld\n", LONG_MAX);
    printf("ULONG_MAX = %lu\n", ULONG_MAX);
    printf("LLONG_MIN = %lld\n", LLONG_MIN);
    printf("LLONG_MAX = %lld\n", LLONG_MAX);
    printf("ULLONG_MAX = %llu\n", ULLONG_MAX);
}
```

On my 64-bit Intel system with clang, this outputs:

```
CHAR_BIT = 8
SCHAR_MIN = -128
SCHAR_MAX = 127
UCHAR_MAX = 255
CHAR_MIN = -128
CHAR_MAX = 127
MB_LEN_MAX = 6
SHRT_MIN = -32768
SHRT_MAX = 32767
USHRT_MAX = 65535
INT_MIN = -2147483648
INT_MAX = 2147483647
UINT_MAX = 4294967295
LONG_MIN = -9223372036854775808
LONG_MAX = 9223372036854775807
ULONG_MAX = 18446744073709551615
LLONG_MIN = -9223372036854775808
LLONG_MAX = 9223372036854775807
ULLONG_MAX = 18446744073709551615
```

Looks like my system probably uses two's-complement encoding for signed numbers, my `chars` are signed, and my `ints` are 32-bit.

Chapter 51

<locale.h> locale handling

Function	Description
`setlocale()`	Set the locale
`localeconv()`	Get information about the current locale

The "locale" is the details of how the program should run given its physical location on the planet.

For example, in one locale, a unit of money might be printed as `$123`, and in another `€123`.

Or one locale might use ASCII encoding and another UTF-8 encoding.

By default, the program runs in the "C" locale. It has a basic set of characters with a single-byte encoding. If you try to print UTF-8 characters in the C locale, nothing will print. You have to switch to a proper locale.

51.1 setlocale()

Set the locale

Synopsis

```
#include <locale.h>

char *setlocale(int category, const char *locale);
```

Description

Sets the `locale` for the given `category`.

Category is one of the following:

Category	Description
`LC_ALL`	All of the following categories
`LC_COLLATE`	Affects the `strcoll()` and `strxfrm()` functions
`LC_CTYPE`	Affects the functions in `<ctype.h>`

www.dbooks.org

Category	Description
`LC_MONETARY`	Affects the monetary information returned from `localeconv()`
`LC_NUMERIC`	Affects the decimal point for formatted I/O and formatted string functions, and the monetary information returned from `localeconv()`
`LC_TIME`	Affects the `strftime()` and `wcsftime()` functions

And there are three portable things you can pass in for `locale`; any other string passed in is implementation-defined and non-portable.

Locale	Description
`"C"`	Set the program to the C locale
`""`	(Empty string) Set the program to the native locale of this system
`NULL`	Change nothing; just return the current locale
Other	Set the program to an implementation-defined locale

The most common call, I'd wager, is this:

```
// Set all locale settings to the local, native locale

setlocale(LC_ALL, "");
```

Handily, `setlocale()` returns the locale that was just set, so you could see what the actual locale is on your system.

Return Value

On success, returns a pointer to the string representing the current locale. You may not modify this string, and it might be changed by subsequent calls to `setlocale()`.

On failure, returns `NULL`.

Example

Here we get the current locale. Then we set it to the native locale, and print out what that is.

```
#include <stdio.h>
#include <locale.h>

int main(void)
{
    char *loc;

    // Get the current locale
    loc = setlocale(LC_ALL, NULL);

    printf("Starting locale: %s\n", loc);

    // Set (and get) the locale to native locale
    loc = setlocale(LC_ALL, "");
```

```

    printf("Native locale: %s\n", loc);
}
```

Output on my system:

```
Starting locale: C
Native locale: en_US.UTF-8
```

Note that my native locale (on a Linux box) might be different from what you see.

Nevertheless, I can explicitly set it on my system without a problem, or to any other locale I have installed:

```
    loc = setlocale(LC_ALL, "en_US.UTF-8");   // Non-portable
```

But again, your system might have different locales defined.

See Also

`localeconv()`, `strcoll()`, `strxfrm()`, `strftime()`, `wcsftime()`, `printf()`, `scanf()`, `<ctype.h>`

51.2 `localeconv()`

Get information about the current locale

Synopsis

```
#include <locale.h>

struct lconv *localeconv(void);
```

Description

This function just returns a pointer to a `struct lconv`, but is still a bit of a powerhouse.

The returned structure contains *tons* of information about the locale. Here are the fields of `struct lconv` and their meanings.

First, some conventions. In the field names, below, a _p_ means "positive", and _n_ means "negative", and `int_` means "international". Though a lot of these are type `char` or `char*`, most (or the strings they point to) are actually treated as integers[1].

Before we go further, know that `CHAR_MAX` (from `<limits.h>`) is the maximum value that can be held in a `char`. And that many of the following `char` values use that to indicate the value isn't available in the given locale.

Field	Description
`char *mon_decimal_point`	Decimal pointer character for money, e.g. `".."`.
`char *mon_thousands_sep`	Thousands separator character for money, e.g. `","`.
`char *mon_grouping`	Grouping description for money (see below).
`char *positive_sign`	Positive sign for money, e.g. `"+"` or `""`.
`char *negative_sign`	Negative sign for money, e.g. `"-"`.
`char *currency_symbol`	Currency symbol, e.g. `"$"`.

[1] Remember that `char` is just a byte-sized integer.

www.dbooks.org

Field	Description
`char frac_digits`	When printing monetary amounts, how many digits to print past the decimal point, e.g. `2`.
`char p_cs_precedes`	`1` if the `currency_symbol` comes before the value for a non-negative monetary amount, `0` if after.
`char n_cs_precedes`	`1` if the `currency_symbol` comes before the value for a negative monetary amount, `0` if after.
`char p_sep_by_space`	Determines the separation of the `currency symbol` from the value for non-negative amounts (see below).
`char n_sep_by_space`	Determines the separation of the `currency symbol` from the value for negative amounts (see below).
`char p_sign_posn`	Determines the `positive_sign` position for non-negative values.
`char p_sign_posn`	Determines the `positive_sign` position for negative values.
`char *int_curr_symbol`	International currency symbol, e.g. `"USD "`.
`char int_frac_digits`	International value for `frac_digits`.
`char int_p_cs_precedes`	International value for `p_cs_precedes`.
`char int_n_cs_precedes`	International value for `n_cs_precedes`.
`char int_p_sep_by_space`	International value for `p_sep_by_space`.
`char int_n_sep_by_space`	International value for `n_sep_by_space`.
`char int_p_sign_posn`	International value for `p_sign_posn`.
`char int_n_sign_posn`	International value for `n_sign_posn`.

Even though many of these have `char` type, the value stored within is meant to be accessed as an integer.

All the `sep_by_space` variants deal with spacing around the currency sign. Valid values are:

Value	Description
`0`	No space between currency symbol and value.
`1`	Separate the currency symbol (and sign, if any) from the value with a space.
`2`	Separate the sign symbol from the currency symbol (if adjacent) with a space, otherwise separate the sign symbol from the value with a space.

The `sign_posn` variants are determined by the following values:

Value	Description
`0`	Put parens around the value and the currency symbol.
`1`	Put the sign string in front of the currency symbol and value.
`2`	Put the sign string after the currency symbol and value.
`3`	Put the sign string directly in front of the currency symbol.
`4`	Put the sign string directly behind the currency symbol.

For more information on the `mon_grouping` field, see “Monetary Digit Grouping” in the “Locale and Internationalization” chapter.

Return Value

Returns a pointer to the structure containing the locale information.

The program may not modify this structure.

Subsequent calls to `localeconv()` may overwrite this structure, as might calls to `setlocale()` with `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC`.

Example

Here's a program to print the locale information for the native locale.

```
#include <stdio.h>
#include <locale.h>
#include <limits.h>  // for CHAR_MAX

void print_grouping(char *mg)
{
    int done = 0;

    while (!done) {
        if (*mg == CHAR_MAX)
            printf("CHAR_MAX ");
        else
            printf("%c ", *mg + '0');
        done = *mg == CHAR_MAX || *mg == 0;
        mg++;
    }
}

int main(void)
{
    setlocale(LC_ALL, "");

    struct lconv *lc = localeconv();

    printf("mon_decimal_point : %s\n", lc->mon_decimal_point);
    printf("mon_thousands_sep : %s\n", lc->mon_thousands_sep);
    printf("mon_grouping      : ");
    print_grouping(lc->mon_grouping);
    printf("\n");
    printf("positive_sign     : %s\n", lc->positive_sign);
    printf("negative_sign     : %s\n", lc->negative_sign);
    printf("currency_symbol   : %s\n", lc->currency_symbol);
    printf("frac_digits       : %c\n", lc->frac_digits);
    printf("p_cs_precedes     : %c\n", lc->p_cs_precedes);
    printf("n_cs_precedes     : %c\n", lc->n_cs_precedes);
    printf("p_sep_by_space    : %c\n", lc->p_sep_by_space);
    printf("n_sep_by_space    : %c\n", lc->n_sep_by_space);
    printf("p_sign_posn       : %c\n", lc->p_sign_posn);
    printf("p_sign_posn       : %c\n", lc->p_sign_posn);
    printf("int_curr_symbol   : %s\n", lc->int_curr_symbol);
    printf("int_frac_digits   : %c\n", lc->int_frac_digits);
    printf("int_p_cs_precedes : %c\n", lc->int_p_cs_precedes);
    printf("int_n_cs_precedes : %c\n", lc->int_n_cs_precedes);
    printf("int_p_sep_by_space: %c\n", lc->int_p_sep_by_space);
    printf("int_n_sep_by_space: %c\n", lc->int_n_sep_by_space);
    printf("int_p_sign_posn   : %c\n", lc->int_p_sign_posn);
    printf("int_n_sign_posn   : %c\n", lc->int_n_sign_posn);
}
```

Output on my system:

```
mon_decimal_point : .
mon_thousands_sep : ,
mon_grouping      : 3 3 0
positive_sign     :
negative_sign     : -
currency_symbol   : $
frac_digits       : 2
p_cs_precedes     : 1
n_cs_precedes     : 1
p_sep_by_space    : 0
n_sep_by_space    : 0
p_sign_posn       : 1
p_sign_posn       : 1
int_curr_symbol   : USD
int_frac_digits   : 2
int_p_cs_precedes : 1
int_n_cs_precedes : 1
int_p_sep_by_space: 1
int_n_sep_by_space: 1
int_p_sign_posn   : 1
int_n_sign_posn   : 1
```

See Also

`setlocale()`

Chapter 52

<math.h> Mathematics

Many of the following functions have `float` and `long double` variants as described below (e.g. `pow()`, `powf()`, `powl()`). The `float` and `long double` variants are omitted from the following table to keep your eyeballs from melting out.

Function	Description
`acos()`	Calculate the arc cosine of a number.
`acosh()`	Compute arc hyperbolic cosine.
`asin()`	Calculate the arc sine of a number.
`asinh()`	Compute arc hyperbolic sine.
`atan()`, `atan2()`	Calculate the arc tangent of a number.
`atanh()`	Compute the arc hyperbolic tangent.
`cbrt()`	Compute the cube root.
`ceil()`	Ceiling—return the next whole number not smaller than the given number.
`copysign()`	Copy the sign of one value into another.
`cos()`	Calculate the cosine of a number.
`cosh()`	Compute the hyperbolic cosine.
`erf()`	Compute the error function of the given value.
`erfc()`	Compute the complementary error function of a value.
`exp()`	Compute e raised to a power.
`exp2()`	Compute 2 to a power.
`expm1()`	Compute $e^x - 1$.
`fabs()`	Compute the absolute value.
`fdim()`	Return the positive difference between two numbers clamped at 0.
`floor()`	Compute the largest whole number not larger than the given value.
`fma()`	Floating (AKA "Fast") multiply and add.
`fmax()`, `fmin()`	Return the maximum or minimum of two numbers.
`fmod()`	Compute the floating point remainder.
`fpclassify()`	Return the classification of a given floating point number.
`frexp()`	Break a number into its fraction part and exponent (as a power of 2).
`hypot()`	Compute the length of the hypotenuse of a triangle.
`ilogb()`	Return the exponent of a floating point number.
`isfinite()`	True if the number is not infinite or NaN.
`isgreater()`	True if one argument is greater than another.
`isgreatereequal()`	True if one argument is greater than or equal to another.
`isinf()`	True if the number is infinite.
`isless()`	True if one argument is less than another.
`islesseequal()`	True if one argument is less than or equal to another.

www.dbooks.org

Function	Description
`islessgreater()`	Test if a floating point number is less than or greater than another.
`isnan()`	True if the number is Not-a-Number.
`isnormal()`	True if the number is normal.
`isunordered()`	Macro returns true if either floating point argument is NaN.
`ldexp()`	Multiply a number by an integral power of 2.
`lgamma()`	Compute the natural logarithm of the absolute value of $\Gamma(x)$.
`log()`	Compute the natural logarithm.
`log10()`	Compute the log-base-10 of a number.
`log2()`	Compute the base-2 logarithm of a number.
`logb()`	Extract the exponent of a number given `FLT_RADIX`.
`log1p()`	Compute the natural logarithm of a number plus 1.
`lrint()`	Returns `x` rounded in the current rounding direction as an integer.
`lround()`, `llround()`	Round a number in the good old-fashioned way, returning an integer.
`modf()`	Extract the integral and fractional parts of a number.
`nan()`	Return `NAN`.
`nearbyint()`	Rounds a value in the current rounding direction.
`nextafter()`	Get the next (or previous) representable floating point value.
`nexttoward()`	Get the next (or previous) representable floating point value.
`pow()`	Compute a value raised to a power.
`remainder()`	Compute the remainder IEC 60559-style.
`remquo()`	Compute the remainder and (some of the) quotient.
`rint()`	Rounds a value in the current rounding direction.
`round()`	Round a number in the good old-fashioned way.
`scalbn()`, `scalbln()`	Efficiently compute $x \times r^n$, where r is `FLT_RADIX`.
`signbit()`	Return the sign of a number.
`sin()`	Calculate the sine of a number.
`sqrt()`	Calculate the square root of a number.
`tan()`	Calculate the tangent of a number.
`tanh()`	Compute the hyperbolic tangent.
`tgamma()`	Compute the gamma function, $\Gamma(x)$.
`trunc()`	Truncate the fractional part off a floating point value.

It's your favorite subject: Mathematics! Hello, I'm Doctor Math, and I'll be making math FUN and EASY!

[vomiting sounds]

Ok, I know math isn't the grandest thing for some of you out there, but these are merely functions that quickly and easily do math you either know, want, or just don't care about. That pretty much covers it.

52.1 Math Function Idioms

Many of these math functions exist in three forms, each corresponding to the argument and/or return types the function uses, `float`, `double`, or `long double`.

The alternate form for `float` is made by appending `f` to the end of the function name.

The alternate form for `long double` is made by appending `l` to the end of the function name.

For example, the `pow()` function, which computes x^y, exists in these forms:

```
double      pow(double x, double y);             // double
float       powf(float x, float y);              // float
long double powl(long double x, long double y);  // long double
```

Remember that parameters are given values as if you assigned into them. So if you pass a `double` to `powf()`, it'll choose the closest `float` it can to hold the double. If the `double` doesn't fit, undefined behavior happens.

52.2 Math Types

We have two exciting new types in `<math.h>`:

- `float_t`
- `double_t`

The `float_t` type is at least as accurate as a `float`, and the `double_t` type is at least as accurate as a `double`.

The idea with these types is they can represent the most efficient way of storing numbers for maximum speed.

Their actual types vary by implementation, but can be determined by the value of the `FLT_EVAL_METHOD` macro.

`FLT_EVAL_METHOD`	`float_t` type	`double_t` type
`0`	`float`	`double`
`1`	`double`	`double`
`2`	`long double`	`long double`
Other	Implementation-defined	Implementation-defined

For all defined values of `FLT_EVAL_METHOD`, `float_t` is the least-precise type used for all floating calculations.

52.3 Math Macros

There are actually a number of these defined, but we'll cover most of them in their relevant reference sections, below.

But here are a couple:

`NAN` represents Not-A-Number.

Defined in `<float.h>` is `FLT_RADIX`: the number base used by floating point numbers. This is commonly `2`, but could be anything.

52.4 Math Errors

As we know, nothing can ever go wrong with math… except *everything*!

So there are just a couple errors that might occur when using some of these functions.

- **Range errors** mean that some result is beyond what can be stored in the result type.
- **Domain errors** mean that you've passed in an argument that doesn't have a defined result for this function.
- **Pole errors** mean that the limit of the function as x approaches the given argument is infinite.
- **Overflow errors** are when the result is really large, but can't be stored without incurring large roundoff error.
- **Underflow errors** are like overflow errors, except with very small numbers.

www.dbooks.org

Now, the C math library can do a couple things when these errors occur:

- Set `errno` to some value, or…
- Raise a floating point exception.

Your system might vary on what happens. You can check it by looking at the value of the variable `math_errhandling`. It will be equivalent to one of the following[1]:

`math_errhandling`	Description
`MATH_ERRNO`	The system uses `errno` for math errors.
`MATH_ERREXCEPT`	The system uses exceptions for math errors.
`MATH_ERRNO | MATH_ERREXCEPT`	The system does both! (That's a bitwise-OR!)

You are not allowed to change `math_errhandling`.

For a fuller description on how exceptions work and their meanings, see the `<fenv.h>` section.

52.5 Math Pragmas

In case you don't remember, you can brush up on pragmas back in the C Preprocessor section.

But in a nutshell, they offer various ways to control the compiler's behavior.

In this case, we have a pragma `FP_CONTRACT` that can be turned off and on.

What does it mean?

First of all, keep in mind that any operation in an expression can cause rounding error. So each step of the expression can introduce more rounding error.

But what if the compiler knows a *double secret* way of taking the expression you wrote and converting it to a single instruction that reduced the number of steps such that the intermediate rounding error didn't occur?

Could it use it? I mean, the results would be different than if you let the rounding error settle each step of the way…

Because the results would be different, you can tell the compiler if you want to allow it to do this or not.

If you want to allow it:

```
#pragma STDC FP_CONTRACT ON
```

and to disallow it:

```
#pragma STDC FP_CONTRACT OFF
```

If you do this at global scope, it stays at whatever state you set it to until you change it.

If you do it at block scope, it reverts to the value outside the block when the block ends.

The initial value of the `FP_CONTRACT` pragma varies from system to system.

52.6 `fpclassify()`

Return the classification of a given floating point number.

[1] Though the system defines `MATH_ERRNO` as `1` and `MATH_ERREXCEPT` as `2`, it's best to always use their symbolic names. Just in case.

Synopsis

```
#include <math.h>

int fpclassify(any_floating_type x);
```

Description

What kind of entity does this floating point number represent? What are the options?

We're used to floating point numbers being regular old things like `3.14` or `3490.0001`.

But floating point numbers can also represent things like infinity. Or Not-A-Number (NAN). This function will let you know which type of floating point number the argument is.

This is a macro, so you can use it with `float`, `double`, `long double` or anything similar.

Return Value

Returns one of these macros depending on the argument's classification:

Classification	Description
`FP_INFINITE`	Number is infinite.
`FP_NAN`	Number is Not-A-Number (NAN).
`FP_NORMAL`	Just a regular number.
`FP_SUBNORMAL`	Number is a sub-normal number.
`FP_ZERO`	Number is zero.

A discussion of subnormal numbers is beyond the scope of the guide, and is something that most devs go their whole lives without dealing with. In a nutshell, it's a way to represent really small numbers that might normally round down to zero. If you want to know more, see the Wikipedia page on denormal numbers[2].

Example

Print various number classifications.

```
#include <stdio.h>
#include <math.h>

const char *get_classification(double n)
{
    switch (fpclassify(n)) {
        case FP_INFINITE: return "infinity";
        case FP_NAN: return "not a number";
        case FP_NORMAL: return "normal";
        case FP_SUBNORMAL: return "subnormal";
        case FP_ZERO: return "zero";
    }

    return "unknown";
}

int main(void)
```

[2] https://en.wikipedia.org/wiki/Denormal_number

www.dbooks.org

```
{
    printf("    1.23: %s\n", get_classification(1.23));
    printf("     0.0: %s\n", get_classification(0.0));
    printf("sqrt(-1): %s\n", get_classification(sqrt(-1)));
    printf("1/tan(0): %s\n", get_classification(1/tan(0)));
    printf("  1e-310: %s\n", get_classification(1e-310));  // very small!
}
```

Output[3]:

```
    1.23: normal
     0.0: zero
sqrt(-1): not a number
1/tan(0): infinity
  1e-310: subnormal
```

See Also

`isfinite()`, `isinf()`, `isnan()`, `isnormal()`, `signbit()`

52.7 `isfinite()`, `isinf()`, `isnan()`, `isnormal()`

Return true if a number matches a classification.

Synopsis

```
#include <math.h>

int isfinite(any_floating_type x);

int isinf(any_floating_type x);

int isnan(any_floating_type x);

int isnormal(any_floating_type x);
```

Description

These are helper macros to `fpclassify()`. Bring macros, they work on any floating point type.

Macro	Description
`isfinite()`	True if the number is not infinite or NaN.
`isinf()`	True if the number is infinite.
`isnan()`	True if the number is Not-a-Number.
`isnormal()`	True if the number is normal.

For more superficial discussion on normal and subnormal numbers, see `fpclassify()`.

[3] This is on my system. Some systems will have different points at which numbers become subnormal, or they might not support subnormal values at all.

Return Value

Returns non-zero for true, and zero for false.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("  isfinite(1.23): %d\n", isfinite(1.23));    // 1
    printf(" isinf(1/tan(0)): %d\n", isinf(1/tan(0)));   // 1
    printf(" isnan(sqrt(-1)): %d\n", isnan(sqrt(-1)));   // 1
    printf("isnormal(1e-310): %d\n", isnormal(1e-310));  // 0
}
```

See Also

`fpclassify()`, `signbit()`,

52.8 `signbit()`

Return the sign of a number.

Synopsis

```
#include <math.h>

int signbit(any_floating_type x);
```

Description

This macro takes any floating point number and returns a value indicating the sign of the number, positive or negative.

Return Value

Returns `1` if the sign is negative, otherwise `0`.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("%d\n", signbit(3490.0));  // 0
    printf("%d\n", signbit(-37.0));   // 1
}
```

See Also

`fpclassify()`, `isfinite()`, `isinf()`, `isnan()`, `isnormal()`, `copysign()`

52.9 acos(), acosf(), acosl()

Calculate the arc cosine of a number.

Synopsis

```
#include <math.h>

double acos(double x);
float acosf(float x);
long double acosl(long double x);
```

Description

Calculates the arc cosine of a number in radians. (That is, the value whose cosine is `x`.) The number must be in the range -1.0 to 1.0.

For those of you who don't remember, radians are another way of measuring an angle, just like degrees. To convert from degrees to radians or the other way around, use the following code:

```
pi = 3.14159265358979;
degrees = radians * 180 / pi;
radians = degrees * pi / 180;
```

Return Value

Returns the arc cosine of `x`, unless `x` is out of range. In that case, `errno` will be set to EDOM and the return value will be NaN. The variants return different types.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    double acosx;
    long double ldacosx;

    acosx = acos(0.2);
    ldacosx = acosl(0.3L);

    printf("%f\n", acosx);
    printf("%Lf\n", ldacosx);
}
```

See Also

`asin()`, `atan()`, `atan2()`, `cos()`

52.10 `asin()`, `asinf()`, `asinl()`

Calculate the arc sine of a number.

Synopsis

```
#include <math.h>

double asin(double x);
float asinf(float x);
long double asinl(long double x);
```

Description

Calculates the arc sine of a number in radians. (That is, the value whose sine is `x`.) The number must be in the range -1.0 to 1.0.

For those of you who don't remember, radians are another way of measuring an angle, just like degrees. To convert from degrees to radians or the other way around, use the following code:

```
pi = 3.14159265358979;
degrees = radians * 180 / pi;
radians = degrees * pi / 180;
```

Return Value

Returns the arc sine of `x`, unless `x` is out of range. In that case, `errno` will be set to EDOM and the return value will be NaN. The variants return different types.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    double asinx;
    long double ldasinx;

    asinx = asin(0.2);
    ldasinx = asinl(0.3L);

    printf("%f\n", asinx);
    printf("%Lf\n", ldasinx);
}
```

See Also

acos(), atan(), atan2(), sin()

52.11 atan(), atanf(), atanl(), atan2(), atan2f(), atan2l()

Calculate the arc tangent of a number.

Synopsis

```
#include <math.h>

double atan(double x);
float atanf(float x);
long double atanl(long double x);

double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);
```

Description

Calculates the arc tangent of a number in radians. (That is, the value whose tangent is x.)

The atan2() variants are pretty much the same as using atan() with y/x as the argument…except that atan2() will use those values to determine the correct quadrant of the result.

For those of you who don't remember, radians are another way of measuring an angle, just like degrees. To convert from degrees to radians or the other way around, use the following code:

```
pi = 3.14159265358979;
degrees = radians * 180 / pi;
radians = degrees * pi / 180;
```

Return Value

The atan() functions return the arc tangent of x, which will be between PI/2 and -PI/2. The atan2() functions return an angle between PI and -PI.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    double atanx;
    long double ldatanx;

    atanx = atan(0.7);
    ldatanx = atanl(0.3L);

```

```
    printf("%f\n", atanx);
    printf("%Lf\n", ldatanx);

    atanx = atan2(7, 10);
    ldatanx = atan2l(3L, 10L);

    printf("%f\n", atanx);
    printf("%Lf\n", ldatanx);
}
```

See Also

tan(), asin(), atan()

52.12 cos(), cosf(), cosl()

Calculate the cosine of a number.

Synopsis

```
#include <math.h>

double cos(double x)
float cosf(float x)
long double cosl(long double x)
```

Description

Calculates the cosine of the value x, where x is in radians.

For those of you who don't remember, radians are another way of measuring an angle, just like degrees. To convert from degrees to radians or the other way around, use the following code:

```
pi = 3.14159265358979;
degrees = radians * 180 / pi;
radians = degrees * pi / 180;
```

Return Value

Returns the cosine of x. The variants return different types.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    double cosx;
    long double ldcosx;

```

```
    cosx = cos(3490.0); // round and round we go!
    ldcosx = cosl(3.490L);

    printf("%f\n", cosx);
    printf("%Lf\n", ldcosx);
}
```

See Also

sin(), tan(), acos()

52.13 sin(), sinf(), sinl()

Calculate the sine of a number.

Synopsis

```
#include <math.h>

double sin(double x);
float sinf(float x);
long double sinl(long double x);
```

Description

Calculates the sine of the value x, where x is in radians.

For those of you who don't remember, radians are another way of measuring an angle, just like degrees. To convert from degrees to radians or the other way around, use the following code:

```
pi = 3.14159265358979;
degrees = radians * 180 / pi;
radians = degrees * pi / 180;
```

Return Value

Returns the sine of x. The variants return different types.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    double sinx;
    long double ldsinx;

    sinx = sin(3490.0); // round and round we go!
    ldsinx = sinl(3.490L);

```

```
    printf("%f\n", sinx);
    printf("%Lf\n", ldsinx);
}
```

See Also

cos(), tan(), asin()

52.14 tan(), tanf(), tanl()

Calculate the tangent of a number.

Synopsis

```
#include <math.h>

double tan(double x)
float tanf(float x)
long double tanl(long double x)
```

Description

Calculates the tangent of the value x, where x is in radians.

For those of you who don't remember, radians are another way of measuring an angle, just like degrees. To convert from degrees to radians or the other way around, use the following code:

```
pi = 3.14159265358979;
degrees = radians * 180 / pi;
radians = degrees * pi / 180;
```

Return Value

Returns the tangent of x. The variants return different types.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    double tanx;
    long double ldtanx;

    tanx = tan(3490.0); // round and round we go!
    ldtanx = tanl(3.490L);

    printf("%f\n", tanx);
    printf("%Lf\n", ldtanx);
}
```

See Also

`sin()`, `cos()`, `atan()`, `atan2()`

52.15 acosh(), acoshf(), acoshl()

Compute arc hyperbolic cosine.

Synopsis

```
#include <math.h>

double acosh(double x);

float acoshf(float x);

long double acoshl(long double x);
```

Description

Trig lovers can rejoice! C has arc hyperbolic cosine!

These functions return the nonnegative acosh of `x`, which must be greater than or equal to `1`.

Return Value

Returns the arc hyperbolic cosince in the range $[0, +\infty]$.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("acosh 1.8 = %f\n", acosh(1.8));  // 1.192911
}
```

See Also

`asinh()`

52.16 asinh(), asinhf(), asinhl()

Compute arc hyperbolic sine.

Synopsis

```
#include <math.h>

double asinh(double x);

float asinhf(float x);

long double asinhl(long double x);
```

Description

Trig lovers can rejoice! C has arc hyperbolic sine!

These functions return the asinh of x.

Return Value

Returns the arc hyperbolic sine.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("asinh 1.8 = %f\n", asinh(1.8));  // 1.350441
}
```

See Also

acosh()

52.17 atanh(), atanhf(), atanhl()

Compute the arc hyperbolic tangent.

Synopsis

```
#include <math.h>

double atanh(double x);

float atanhf(float x);

long double atanhl(long double x);
```

Description

These functions compute the arc hyperbolic tangent of x, which must be in the range $[-1, +1]$. Passing exactly -1 or $+1$ might result in a pole error.

Return Value

Returns the arc hyperbolic tangent of x.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("atanh 0.5 = %f\n", atanh(0.5));  // 0.549306
}
```

See Also

acosh(), asinh()

52.18 cosh(), coshf(), coshl()

Compute the hyperbolic cosine.

Synopsis

```
#include <math.h>

double cosh(double x);

float coshf(float x);

long double coshl(long double x);
```

Description

These functions predictably compute the hyperbolic cosine of x. A range error might occur if x is too large.

Return Value

Returns the hyperbolic cosine of x.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("cosh 0.5 = %f\n", cosh(0.5));  // 1.127626
}
```

See Also

sinh(), tanh()

52.19 sinh(), sinhf(), sinhl()

Compute the hyperbolic sine.

Synopsis

```
#include <math.h>

double sinh(double x);

float sinhf(float x);

long double sinhl(long double x);
```

Description

These functions predictably compute the hyperbolic sine of x. A range error might occur if x is too large.

Return Value

Returns the hyperbolic sine of x.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("sinh 0.5 = %f\n", sinh(0.5));  // 0.521095
}
```

See Also

sinh(), tanh()

52.20 tanh(), tanhf(), tanhl()

Compute the hyperbolic tangent.

Synopsis

```
#include <math.h>
```

```
double tanh(double x);

float tanhf(float x);

long double tanhl(long double x);
```

Description

These functions predictably compute the hyperbolic tangent of `x`.

Mercifully, this is the last trig-related man page I'm going to write.

Return Value

Returns the hyperbolic tangent of `x`.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("tanh 0.5 = %f\n", tanh(0.5));  // 0.462117
}
```

See Also

`cosh()`, `sinh()`

52.21 `exp()`, `expf()`, `expl()`

Compute e raised to a power.

Synopsis

```
#include <math.h>

double exp(double x);

float expf(float x);

long double expl(long double x);
```

Description

Compute e^x where e is Euler's number[4].

The number e is named after Leonard Euler, born April 15, 1707, who is responsible, among other things, for making this reference page longer than it needed to be.

[4] https://en.wikipedia.org/wiki/E_(mathematical_constant)

Return Value

Returns e^x.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("exp(1) = %f\n", exp(1));  // 2.718282
    printf("exp(2) = %f\n", exp(2));  // 7.389056
}
```

See Also

`exp2()`, `expm1()`, `pow()`, `log()`

52.22 `exp2()`, `exp2f()`, `exp2l()`

Compute 2 to a power.

Synopsis

```
#include <math.h>

double exp2(double x);

float exp2f(float x);

long double exp2l(long double x);
```

Description

These functions raise 2 to a power. Very exciting, since computers are all about twos-to-powers!

These are likely to be faster than using `pow()` to do the same thing.

They support fractional exponents, as well.

A range error occurs if `x` is too large.

Return Value

`exp2()` returns 2^x.

Example

```
#include <stdio.h>
#include <math.h>

```

```
int main(void)
{
    printf("2^3 = %f\n", exp2(3));      // 2^3 = 8.000000
    printf("2^8 = %f\n", exp2(8));      // 2^8 = 256.000000
    printf("2^0.5 = %f\n", exp2(0.5));  // 2^0.5 = 1.414214
}
```

See Also

`exp()`, `pow()`

52.23 `expm1()`, `expm1f()`, `expm1l()`

Compute $e^x - 1$.

Synopsis

```
#include <math.h>

double expm1(double x);

float expm1f(float x);

long double expm1l(long double x);
```

Description

This is just like `exp()` except—*plot twist!*–it computes that result minus one.

For more discussion about what e is, see the `exp()` man page.

If `x` is giant, a range error might occur.

For small values of `x` near zero, `expm1(x)` might be more accurate than computing `exp(x)-1`.

Return Value

Returns $e^x - 1$.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("%f\n", expm1(2.34));  // 9.381237
}
```

See Also

exp()

52.24 frexp(), frexpf(), frexpl()

Break a number into its fraction part and exponent (as a power of 2).

Synopsis

```
#include <math.h>

double frexp(double value, int *exp);

float frexpf(float value, int *exp);

long double frexpl(long double value, int *exp);
```

Description

If you have a floating point number, you can break it into its fractional part and exponent part (as a power of 2).

For example, if you have the number 1234.56, this can be represented as a multiple of a power of 2 like so:

$1234.56 = 0.6028125 \times 2^{11}$

And you can use this function to get the 0.6028125 and 11 parts of that equation.

As for why, I have a simple answer: I don't know. I can't find a use. K&R2 and everyone else I can find just says *how* to use it, but not *why* you might want to.

The C99 Rationale document says:

> The functions frexp, ldexp, and modf are primitives used by the remainder of the library.
>
> There was some sentiment for dropping them for the same reasons that ecvt, fcvt, and gcvt were dropped, but their adherents rescued them for general use. Their use is problematic: on non-binary architectures, ldexp may lose precision and frexp may be inefficient.

So there you have it. If you need it.

Return Value

frexp() returns the fractional part of value in the range 0.5 (inclusive) to 1 (exclusive), or 0. And it stores the exponent power-of-2 in the variable pointed to by exp.

If you pass in zero, the return value and the variable exp points to are both zero.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
```

```
    double frac;
    int expt;

    frac = frexp(1234.56, &expt);
    printf("1234.56 = %.7f x 2^%d\n", frac, expt);
}
```

Output:

```
1234.56 = 0.6028125 x 2^11
```

See Also

`ldexp()`, `ilogb()`, `modf()`

52.25 `ilogb()`, `ilogbf()`, `ilogbl()`

Return the exponent of a floating point number.

Synopsis

```
#include <math.h>

int ilogb(double x);

int ilogbf(float x);

int ilogbl(long double x);
```

Description

This gives you the exponent of the given number… it's a little weird, because the exponent depends on the value of `FLT_RADIX`. Now, this is very often `2`—but no guarantees!

It actually returns $\log_r |x|$ where r is `FLT_RADIX`.

Domain or range errors might occur for invalid values of `x`, or for return values that are outside the range of the return type.

Return Value

The exponent of the absolute value of the given number, depending on `FLT_RADIX`.

Specifically $\log_r |x|$ where r is `FLT_RADIX`.

If you pass in `0`, it'll return `FP_ILOGB0`.

If you pass in infinity, it'll return `INT_MAX`.

If you pass in NaN, it'll return `FP_ILOGBNAN`.

The spec goes on to say that the value of `FP_ILOGB0` will be either `INT_MIN` or `-INT_MAX`. And the value of `FP_ILOGBNAN` shall be either `INT_MAX` or `INT_MIN`, if that's useful in any way.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("%d\n", ilogb(257));  // 8
    printf("%d\n", ilogb(256));  // 8
    printf("%d\n", ilogb(255));  // 7
}
```

See Also

`frexp()`, `logb()`

52.26 `ldexp()`, `ldexpf()`, `ldexpl()`

Multiply a number by an integral power of 2.

Synopsis

```
#include <math.h>

double ldexp(double x, int exp);

float ldexpf(float x, int exp);

long double ldexpl(long double x, int exp);
```

Description

These functions multiply the given number `x` by 2 raised to the `exp` power.

Return Value

Returns $x \times 2^{exp}$.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("1 x 2^10 = %f\n", ldexp(1, 10));
    printf("5.67 x 2^7 = %f\n", ldexp(5.67, 7));
}
```

Output:

```
1 x 2^10 = 1024.000000
5.67 x 2^7 = 725.760000
```

See Also

`exp()`

52.27 `log()`, `logf()`, `logl()`

Compute the natural logarithm.

Synopsis

```c
#include <math.h>

double log(double x);

float logf(float x);

long double logl(long double x);
```

Description

Natural logarithms! And there was much rejoycing.

These compute the base-e logarithm of a number, $\log_e x$, $\ln x$.

In other words, for a given x, solves $x = e^y$ for y.

Return Value

The base-e logarithm of the given value, $\log_e x$, $\ln x$.

Example

```c
#include <stdio.h>
#include <math.h>

int main(void)
{
    const double e = 2.718281828459045;

    printf("%f\n", log(3490.2));  // 8.157714
    printf("%f\n", log(e));       // 1.000000
}
```

See Also

`exp()`, `log10()`, `log1p()`

52.28 log10(), log10f(), log10l()

Compute the log-base-10 of a number.

Synopsis

```
#include <math.h>

double log10(double x);

float log10f(float x);

long double log10l(long double x);
```

Description

Just when you thought you might have to use Laws of Logarithms to compute this, here's a function coming out of the blue to save you.

These compute the base-10 logarithm of a number, $\log_{10} x$.

In other words, for a given x, solves $x = 10^y$ for y.

Return Value

Returns the log base-10 of x, $\log_{10} x$.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("%f\n", log10(3490.2));   // 3.542850
    printf("%f\n", log10(10));       // 1.000000
}
```

See Also

pow(), log()

52.29 log1p(), log1pf(), log1pl()

Compute the natural logarithm of a number plus 1.

Synopsis

```
#include <math.h>

double log1p(double x);
```

```
float log1pf(float x);

long double log1pl(long double x);
```

Description

This computes $\log_e(1+x)$, $\ln(1+x)$.

This works just like calling:

```
log(1 + x)
```

except it could be more accurate for small values of `x`.

So if your `x` is small magnitude, use this.

Return Value

Returns $\log_e(1+x)$, $\ln(1+x)$.

Example

Compute some big and small logarithm values to see the difference between `log1p()` and `log()`:

```
#include <stdio.h>
#include <float.h> // for LDBL_DECIMAL_DIG
#include <math.h>

int main(void)
{
    printf("Big log1p()  : %.*Lf\n", LDBL_DECIMAL_DIG-1, log1pl(9));
    printf("Big log()    : %.*Lf\n", LDBL_DECIMAL_DIG-1, logl(1 + 9));

    printf("Small log1p(): %.*Lf\n", LDBL_DECIMAL_DIG-1, log1pl(0.01));
    printf("Small log()  : %.*Lf\n", LDBL_DECIMAL_DIG-1, logl(1 + 0.01));
}
```

Output on my system:

```
Big log1p()  : 2.30258509299404568403
Big log()    : 2.30258509299404568403
Small log1p(): 0.00995033085316808305
Small log()  : 0.00995033085316809164
```

See Also

`log()`

52.30 `log2()`, `log2f()`, `log2l()`

Compute the base-2 logarithm of a number.

Synopsis

```
#include <math.h>

double log2(double x);

float log2f(float x);

long double log2l(long double x);
```

Description

Wow! Were you thinking we were done with the logarithm functions? We're only getting started!

This one computes $\log_2 x$. That is, computes y that satisfies $x = 2^y$.

Love me those powers of 2!

Return Value

Returns the base-2 logarithm of the given value, $\log_2 x$.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("%f\n", log2(3490.2));  // 11.769094
    printf("%f\n", log2(256));     // 8.000000
}
```

See Also

`log()`

52.31 `logb()`, `logbf()`, `logbl()`

Extract the exponent of a number given `FLT_RADIX`.

Synopsis

```
#include <math.h>

double logb(double x);

float logbf(float x);

long double logbl(long double x);
```

Description

This function returns the whole number portion of the exponent of the number with radix `FLT_RADIX`, namely the whole number portion $\log_r |x|$ where r is `FLT_RADIX`. Fractional numbers are truncated.

If the number is subnormal[5], `logb()` treats it as if it were normalized.

If `x` is `0`, there could be a domain error or pole error.

Return Value

This function returns the whole number portion of $\log_r |x|$ where r is `FLT_RADIX`.

Example

```c
#include <stdio.h>
#include <float.h>  // For FLT_RADIX
#include <math.h>

int main(void)
{
    printf("FLT_RADIX = %d\n", FLT_RADIX);
    printf("%f\n", logb(3490.2));
    printf("%f\n", logb(256));
}
```

Output:

```
FLT_RADIX = 2
11.000000
8.000000
```

See Also

`ilogb()`

52.32 `modf()`, `modff()`, `modfl()`

Extract the integral and fractional parts of a number.

Synopsis

```c
#include <math.h>

double modf(double value, double *iptr);

float modff(float value, float *iptr);

long double modfl(long double value, long double *iptr);
```

[5] https://en.wikipedia.org/wiki/Denormal_number

Description

If you have a floating point number, like `123.456`, this function will extract the integral part (`123.0`) and the fractional part (`0.456`). It's total coincidence that this is exactly the plot for the latest Jason Statham action spectacular.

Both the integral part and fractional parts keep the sign of the passed in `value`.

The integral part is stored in the address pointed to by `iptr`.

See the note in `frexp()` regarding why this is in the library.

Return Value

These functions return the fractional part of the number. The integral part is stored in the address pointed to by `iptr`. Both the integral and fractional parts preserve the sign of the passed-in `value`.

Example

```c
#include <stdio.h>
#include <math.h>

void print_parts(double x)
{
    double i, f;

    f = modf(x, &i);

    printf("Entire number  : %f\n", x);
    printf("Integral part  : %f\n", i);
    printf("Fractional part: %f\n\n", f);
}

int main(void)
{
    print_parts(123.456);
    print_parts(-123.456);
}
```

Output:

```
Entire number  : 123.456000
Integral part  : 123.000000
Fractional part: 0.456000

Entire number  : -123.456000
Integral part  : -123.000000
Fractional part: -0.456000
```

See Also

`frexp()`

52.33 scalbn(), scalbnf(), scalbnl() scalbln(), scalblnf(), scalblnl()

Efficiently compute $x \times r^n$, where r is FLT_RADIX.

Synopsis

```
#include <math.h>

double scalbn(double x, int n);

float scalbnf(float x, int n);

long double scalbnl(long double x, int n);

double scalbln(double x, long int n);

float scalblnf(float x, long int n);

long double scalblnl(long double x, long int n);
```

Description

These functions efficiently compute $x \times r^n$, where r is FLT_RADIX.

If FLT_RADIX happens to be 2 (no guarantees!), then this works like exp2().

The name of this function should have an obvious meaning to you. Clearly they all start with the prefix "scalb" which means…

…OK, I confess! I have no idea what it means. My searches are futile!

But let's look at the suffixes:

Suffix	Meaning
n	scalbn()—exponent n is an int
nf	scalbnf()—float version of scalbn()
nl	scalbnl()—long double version of scalbn()
ln	scalbln()—exponent n is a long int
lnf	scalblnf()—float version of scalbln()
lnl	scalblnl()—long double version of scalbln()

So while I'm still in the dark about "scalb", at least I have that part down.

A range error might occur for large values.

Return Value

Returns $x \times r^n$, where r is FLT_RADIX.

Example

```c
#include <stdio.h>
#include <math.h>
#include <float.h>

int main(void)
{
    printf("FLT_RADIX = %d\n\n", FLT_RADIX);
    printf("scalbn(3, 8)      = %f\n", scalbn(2, 8));
    printf("scalbnf(10.2, 20) = %f\n", scalbnf(10.2, 20));
}
```

Output on my system:

```
FLT_RADIX = 2

scalbn(3, 8)       = 512.000000
scalbn(10.2, 20.7) = 10695475.200000
```

See Also

`exp2()`, `pow()`

52.34 `cbrt()`, `cbrtf()`, `cbrtl()`

Compute the cube root.

Synopsis

```c
#include <math.h>

double cbrt(double x);

float cbrtf(float x);

long double cbrtl(long double x);
```

Description

Computes the cube root of x, $x^{1/3}$, $\sqrt[3]{x}$.

Return Value

Returns the cube root of x, $x^{1/3}$, $\sqrt[3]{x}$.

Example

```c
#include <stdio.h>
#include <math.h>

int main(void)
{
```

```
    printf("cbrt(1729.03) = %f\n", cbrt(1729.03));
}
```

Output:

```
cbrt(1729.03) = 12.002384
```

See Also

sqrt(), pow()

52.35 fabs(), fabsf(), fabsl()

Compute the absolute value.

Synopsis

```
#include <math.h>

double fabs(double x);

float fabsf(float x);

long double fabsl(long double x);
```

Description

These functions straightforwardly return the absolute value of x, that is $|x|$.

If you're rusty on your absolute values, all it means is that the result will be positive, even if x is negative. It's just strips negative signs off.

Return Value

Returns the absolute value of x, $|x|$.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("fabs(3490.0)  = %f\n", fabs(3490.0));  // 3490.000000
    printf("fabs(-3490.0) = %f\n", fabs(3490.0));  // 3490.000000
}
```

See Also

abs(), copysign(), imaxabs()

52.36 hypot(), hypotf(), hypotl()

Compute the length of the hypotenuse of a triangle.

Synopsis

```
#include <math.h>

double hypot(double x, double y);

float hypotf(float x, float y);

long double hypotl(long double x, long double y);
```

Description

Pythagorean Theorem[6] fans rejoice! This is the function you've been waiting for!

If you know the lengths of the two sides of a right triangle, x and y, you can compute the length of the hypotenuse (the longest, diagonal side) with this function.

In particular, it computes the square root of the sum of the squares of the sides: $\sqrt{x^2 + y^2}$.

Return Value

Returns the lenght of the hypotenuse of a right triangle with side lengths x and y: $\sqrt{x^2 + y^2}$.

Example

```
printf("%f\n", hypot(3, 4));  // 5.000000
```

See Also

sqrt()

52.37 pow(), powf(), powl()

Compute a value raised to a power.

Synopsis

```
#include <math.h>

double pow(double x, double y);

float powf(float x, float y);

long double powl(long double x, long double y);
```

[6] https://en.wikipedia.org/wiki/Pythagorean_theorem

Description

Computes x raised to the yth power: x^y.

These arguments can be fractional.

Return Value

Returns x raised to the yth power: x^y.

A domain error can occur if:

- x is a finite negative number and y is a finite non-integer
- x is zero and y is zero.

A domain error or pole error can occur if x is zero and y is negative.

A range error can occur for large values.

Example

```
printf("%f\n", pow(3, 4));    // 3^4    = 81.000000
printf("%f\n", pow(2, 0.5));  // sqrt 2 = 1.414214
```

See Also

exp(), exp2(), sqrt(), cbrt()

52.38 sqrt()

Calculate the square root of a number.

Synopsis

```
#include <math.h>

double sqrt(double x);

float sqrtf(float x);

long double sqrtl(long double x);
```

Description

Computes the square root of a number: $\sqrt{x}$. To those of you who don't know what a square root is, I'm not going to explain. Suffice it to say, the square root of a number delivers a value that when squared (multiplied by itself) results in the original number.

Ok, fine—I did explain it after all, but only because I wanted to show off. It's not like I'm giving you examples or anything, such as the square root of nine is three, because when you multiply three by three you get nine, or anything like that. No examples. I hate examples!

And I suppose you wanted some actual practical information here as well. You can see the usual trio of functions here—they all compute square root, but they take different types as arguments. Pretty straightforward, really.

A domain error occurs if x is negative.

Return Value

Returns (and I know this must be something of a surprise to you) the square root of x: $\sqrt{x}$.

Example

```
// example usage of sqrt()

float something = 10;

double x1 = 8.2, y1 = -5.4;
double x2 = 3.8, y2 = 34.9;
double dx, dy;

printf("square root of 10 is %.2f\n", sqrtf(something));

dx = x2 - x1;
dy = y2 - y1;
printf("distance between points (x1, y1) and (x2, y2): %.2f\n",
    sqrt(dx*dx + dy*dy));
```

And the output is:

```
square root of 10 is 3.16
distance between points (x1, y1) and (x2, y2): 40.54
```

See Also

hypot(), pow()

52.39 erf(), erff(), erfl()

Compute the error function of the given value.

Synopsis

```
#include <math.h>

double erfc(double x);

float erfcf(float x);

long double erfcl(long double x);
```

Description

These functions compute the error function[7] of a value.

[7] https://en.wikipedia.org/wiki/Error_function

Return Value

Returns the error function of x:

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

Example

```
for (float i = -2; i <= 2; i += 0.5)
    printf("% .1f: %f\n", i, erf(i));
```

Output:

```
-2.0: -0.995322
-1.5: -0.966105
-1.0: -0.842701
-0.5: -0.520500
 0.0: 0.000000
 0.5: 0.520500
 1.0: 0.842701
 1.5: 0.966105
 2.0: 0.995322
```

See Also

erfc()

52.40 erfc(), erfcf(), erfcl()

Compute the complementary error function of a value.

Synopsis

```
#include <math.h>

double erfc(double x);

float erfcf(float x);

long double erfcl(long double x);
```

Description

These functions compute the complementary error function[8] of a value.

This is the same as:

```
1 - erf(x)
```

A range error can occur if x is too large.

[8] https://en.wikipedia.org/wiki/Error_function

Return Value

Returns `1 - erf(x)`, namely:

$$\frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\, dt$$

Example

```
for (float i = -2; i <= 2; i += 0.5)
    printf("% .1f: %f\n", i, erfc(i));
```

Output:

```
-2.0: 1.995322
-1.5: 1.966105
-1.0: 1.842701
-0.5: 1.520500
 0.0: 1.000000
 0.5: 0.479500
 1.0: 0.157299
 1.5: 0.033895
 2.0: 0.004678
```

See Also

`erf()`

52.41 `lgamma()`, `lgammaf()`, `lgammal()`

Compute the natural logarithm of the absolute value of $\Gamma(x)$.

Synopsis

```
#include <math.h>

double lgamma(double x);

float lgammaf(float x);

long double lgammal(long double x);
```

Description

Compute the natural log of the absolute value of gamma[9] `x`, $\log_e |\Gamma(x)|$.

A range error can occur if `x` is too large.

A pole error can occur is `x` is non-positive.

[9] https://en.wikipedia.org/wiki/Gamma_function

Return Value

Returns $\log_e |\Gamma(x)|$.

Example

```
for (float i = 0.5; i <= 4; i += 0.5)
    printf("%.1f: %f\n", i, lgamma(i));
```

Output:

```
0.5: 0.572365
1.0: 0.000000
1.5: -0.120782
2.0: 0.000000
2.5: 0.284683
3.0: 0.693147
3.5: 1.200974
4.0: 1.791759
```

See Also

`tgamma()`

52.42 `tgamma()`, `tgammaf()`, `tgammal()`

Compute the gamma function, $\Gamma(x)$.

Synopsis

```
#include <math.h>

double tgamma(double x);

float tgammaf(float x);

long double tgammal(long double x);
```

Description

Computes the gamma function[10] of `x`, $\Gamma(x)$.

A domain or pole error might occur if `x` is non-positive.

A range error might occur if `x` is too large or too small.

Return Value

Returns the gamma function of `x`, $\Gamma(x)$.

[10] https://en.wikipedia.org/wiki/Gamma_function

Example

```
for (float i = 0.5; i <= 4; i += 0.5)
    printf("%.1f: %f\n", i, tgamma(i));
```

Output:

```
0.5: 1.772454
1.0: 1.000000
1.5: 0.886227
2.0: 1.000000
2.5: 1.329340
3.0: 2.000000
3.5: 3.323351
4.0: 6.000000
```

See Also

`lgamma()`

52.43 `ceil()`, `ceilf()`, `ceill()`

Ceiling—return the next whole number not smaller than the given number.

Synopsis

```
#include <math.h>

double ceil(double x);

float ceilf(float x);

long double ceill(long double x);
```

Description

Returns the ceiling of the `x`: $\lceil x \rceil$.

This is the next whole number not smaller than `x`.

Beware this minor dragon: it's not just "rounding up". Well, it is for positive numbers, but negative numbers effectively round toward zero. (Because the ceiling function is headed for the next largest whole number and -4 is larger than -5.)

Return Value

Returns the next largest whole number larger than `x`.

Example

Notice for the negative numbers it heads toward zero, i.e. toward the next largest whole number—just like the positives head toward the next largest whole number.

```
printf("%f\n", ceil(4.0));   //  4.000000
printf("%f\n", ceil(4.1));   //  5.000000
printf("%f\n", ceil(-2.0));  // -2.000000
printf("%f\n", ceil(-2.1));  // -2.000000
printf("%f\n", ceil(-3.1));  // -3.000000
```

See Also

`floor()`, `round()`

52.44 `floor()`, `floorf()`, `floorl()`

Compute the largest whole number not larger than the given value.

Synopsis

```
#include <math.h>
double floor(double x);
float floorf(float x);
long double floorl(long double x);
```

Description

Returns the floor of the value: $\lfloor x \rfloor$. This is the opposite of `ceil()`.

This is the largest whole number that is not greater than `x`.

For positive numbers, this is like rounding down: `4.5` becomes `4.0`.

For negative numbers, it's like rounding up: `-3.6` becomes `-4.0`.

In both cases, those results are the largest whole number not bigger than the given number.

Return Value

Returns the largest whole number not greater than `x`: $\lfloor x \rfloor$.

Example

Note how the negative numbers effectively round away from zero, unlike the positives.

```
printf("%f\n", floor(4.0));   //  4.000000
printf("%f\n", floor(4.1));   //  4.000000
printf("%f\n", floor(-2.0));  // -2.000000
printf("%f\n", floor(-2.1));  // -3.000000
printf("%f\n", floor(-3.1));  // -4.000000
```

See Also

`ceil()`, `round()`

52.45 nearbyint(), nearbyintf(), nearbyintl()

Rounds a value in the current rounding direction.

Synopsis

```
#include <math.h>

double nearbyint(double x);

float nearbyintf(float x);

long double nearbyintl(long double x);
```

Description

This function rounds x to the nearest integer in the current rounding direction.

The rounding direction can be set with fesetround() in <fenv.h>.

nearbyint() won't raise the "inexact" floating point exception.

Return Value

Returns x rounded in the current rounding direction.

Example

```
#include <stdio.h>
#include <math.h>
#include <fenv.h>

int main(void)
{
    #pragma STDC FENV_ACCESS ON        // If supported

    fesetround(FE_TONEAREST);          // round to nearest

    printf("%f\n", nearbyint(3.14));   // 3.000000
    printf("%f\n", nearbyint(3.74));   // 4.000000

    fesetround(FE_TOWARDZERO);         // round toward zero

    printf("%f\n", nearbyint(1.99));   // 1.000000
    printf("%f\n", nearbyint(-1.99));  // -1.000000
}
```

See Also

rint(), lrint(), round(), fesetround(), fegetround()

52.46 rint(), rintf(), rintl()

Rounds a value in the current rounding direction.

Synopsis

```
#include <math.h>

double rint(double x);

float rintf(float x);

long double rintl(long double x);
```

Description

This works just like `nearbyint()` except that is can raise the "inexact" floating point exception.

Return Value

Returns `x` rounded in the current rounding direction.

Example

```
#include <stdio.h>
#include <math.h>
#include <fenv.h>

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    fesetround(FE_TONEAREST);

    printf("%f\n", rint(3.14));   // 3.000000
    printf("%f\n", rint(3.74));   // 4.000000

    fesetround(FE_TOWARDZERO);

    printf("%f\n", rint(1.99));   // 1.000000
    printf("%f\n", rint(-1.99));  // -1.000000
}
```

See Also

`nearbyint()`, `lrint()`, `round()`, `fesetround()`, `fegetround()`

52.47 lrint(), lrintf(), lrintl(), llrint(), llrintf(), llrintl()

Returns `x` rounded in the current rounding direction as an integer.

Synopsis

```
#include <math.h>

long int lrint(double x);
long int lrintf(float x);
long int lrintl(long double x);

long long int llrint(double x);
long long int llrintf(float x);
long long int llrintl(long double x);
```

Description

Round a floating point number in the current rounding direction, but this time return an integer intead of a float. You know, just to mix it up.

These come in two variants:

- `lrint()`—returns `long int`
- `llrint()`—returns `long long int`

If the result doesn't fit in the return type, a domain or range error might occur.

Return Value

The value of `x` rounded to an integer in the current rounding direction.

Example

```
#include <stdio.h>
#include <math.h>
#include <fenv.h>

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    fesetround(FE_TONEAREST);

    printf("%ld\n", lrint(3.14));   // 3
    printf("%ld\n", lrint(3.74));   // 4

    fesetround(FE_TOWARDZERO);

    printf("%ld\n", lrint(1.99));   // 1
    printf("%ld\n", lrint(-1.99));  // -1
}
```

See Also

`nearbyint()`, `rint()`, `round()`, `fesetround()`, `fegetround()`

52.48 round(), roundf(), roundl()

Round a number in the good old-fashioned way.

Synopsis

```
#include <math.h>

double round(double x);

float roundf(float x);

long double roundl(long double x);
```

Description

Rounds a number to the nearest whole value.

In case of halfsies, rounds away from zero (i.e. "round up" in magnitude).

The current rounding direction's Jedi mind tricks don't work on this function.

Return Value

The rounded value of `x`.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("%f\n", round(3.14));   // 3.000000
    printf("%f\n", round(3.5));    // 4.000000

    printf("%f\n", round(-1.5));   // -2.000000
    printf("%f\n", round(-1.14));  // -1.000000
}
```

See Also

`lround()`, `nearbyint()`, `rint()`, `lrint()`, `trunc()`

52.49 lround(), lroundf(), lroundl() llround(), llroundf(), llroundl()

Round a number in the good old-fashioned way, returning an integer.

Synopsis

```
#include <math.h>

long int lround(double x);
long int lroundf(float x);
long int lroundl(long double x);

long long int llround(double x);
long long int llroundf(float x);
long long int llroundl(long double x);
```

Description

These are just like `round()` except they return integers.

Halfway values round away from zero, e.g. 1.5 rounds to 2 and -1.5 rounds to -2.

The functions are grouped by return type:

- `lround()`—returns a `long int`
- `llround()`—returns a `long long int`

If the rounded value can't find in the return time, a domain or range error can occur.

Return Value

Returns the rounded value of `x` as an integer.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("%ld\n", lround(3.14));   // 3
    printf("%ld\n", lround(3.5));    // 4

    printf("%ld\n", lround(-1.5));   // -2
    printf("%ld\n", lround(-1.14));  // -1
}
```

See Also

`round()`, `nearbyint()`, `rint()`, `lrint()`, `trunc()`

52.50 `trunc()`, `truncf()`, `truncl()`

Truncate the fractional part off a floating point value.

Synopsis

```
#include <math.h>

double trunc(double x);

float truncf(float x);

long double truncl(long double x);
```

Description

These functions just drop the fractional part of a floating point number. Boom.

In other words, they always round toward zero.

Return Value

Returns the truncated floating point number.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("%f\n", trunc(3.14));   // 3.000000
    printf("%f\n", trunc(3.8));    // 3.000000

    printf("%f\n", trunc(-1.5));   // -1.000000
    printf("%f\n", trunc(-1.14));  // -1.000000
}
```

See Also

`round()`, `lround()`, `nearbyint()`, `rint()`, `lrint()`

52.51 `fmod()`, `fmodf()`, `fmodl()`

Compute the floating point remainder.

Synopsis

```
#include <math.h>

double fmod(double x, double y);

float fmodf(float x, float y);

long double fmodl(long double x, long double y);
```

Description

Returns the remainder of $\frac{x}{y}$. The result will have the same sign as `x`.

Under the hood, the computation performed is:

```
x - trunc(x / y) * y
```

But it might be easier just to think of the remainder.

Return Value

Returns the remainder of $\frac{x}{y}$ with the same sign as `x`.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("%f\n", fmod(-9.2, 5.1));  // -4.100000
    printf("%f\n", fmod(9.2, 5.1));   //  4.100000
}
```

See Also

`remainder()`

52.52 `remainder()`, `remainderf()`, `remainderl()`

Compute the remainder IEC 60559-style.

Synopsis

```
#include <math.h>

double remainder(double x, double y);

float remainderf(float x, float y);

long double remainderl(long double x, long double y);
```

Description

This is similar to `fmod()`, but not quite the same. `fmod()` is probably what you're after if you're expecting remainders to wrap around like an odometer.

The C spec quotes IEC 60559 on how this works:

> When $y \neq 0$, the remainder $r = x$ REM y is defined regardless of the rounding mode by the mathematical relation $r = x - ny$, where n is the integer nearest the exact value of x/y; whenever $|n - x/y| = 1/2$, then n is even. If $r = 0$, its sign shall be that of x.

Hope that clears it up!

OK, maybe not. Here's the upshot:

You know how if you `fmod()` something by, say `2.0` you get a result that is somewhere between `0.0` and `2.0`? And how if you just increase the number that you're modding by `2.0`, you can see the result climb up to `2.0` and then wrap around to `0.0` like your car's odometer?

`remainder()` works just like that, except if `y` is `2.0`, it wraps from `-1.0` to `1.0` instead of from `0.0` to `2.0`.

In other words, the range of the function runs from `-y/2` to `y/2`. Contrasted to `fmod()` that runs from `0.0` to `y`, `remainder()`'s output is just shifted down half a `y`.

And zero-remainder-anything is `0`.

Except if `y` is zero, the function might return zero or a domain error might occur.

Return Value

The IEC 60559 result of `x`-remainder-`y`.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("%f\n", remainder(3.7, 4));  // -0.300000
    printf("%f\n", remainder(4.3, 4));  //  0.300000
}
```

See Also

`fmod()`, `remquo()`

52.53 `remquo()`, `remquof()`, `remquol()`

Compute the remainder and (some of the) quotient.

Synopsis

```
#include <math.h>

double remquo(double x, double y, int *quo);

float remquof(float x, float y, int *quo);

long double remquol(long double x, long double y, int *quo);
```

Description

This is a funky little thing.

First of all, the return value is the remainder, the same as the `remainder()` function, so check that out.

And the quotient comes back in the `quo` pointer.

Or at least *some of it* does. You'll get at least 3 bits worth of the quotient.

But *why*?

So a couple things.

One is that the quotient of some very large floating point numbers can easily be far too gigantic to fit in even a `long long unsigned int`. So some of it might very well need to be lopped off, anyway.

But at 3 bits? How's that even useful? That only gets you from 0 to 7!

The C99 Rationale document states:

> The `remquo` functions are intended for implementing argument reductions which can exploit a few low-order bits of the quotient. Note that x may be so large in magnitude relative to y that an exact representation of the quotient is not practical.

So… implementing argument reductions… which can exploit a few low-order bits… Ooookay.

CPPReference has this to say[11] on the matter, which is spoken so well, I will quote wholesale:

> This function is useful when implementing periodic functions with the period exactly representable as a floating-point value: when calculating $\sin(\pi x)$ for a very large `x`, calling `sin` directly may result in a large error, but if the function argument is first reduced with `remquo`, the low-order bits of the quotient may be used to determine the sign and the octant of the result within the period, while the remainder may be used to calculate the value with high precision.

And there you have it. If you have another example that works for you… congratulations! :)

Return Value

Returns the same as `remainder`: The IEC 60559 result of `x`-remainder-`y`.

In addition, at least the lowest 3 bits of the quotient will be stored in `quo` with the same sign as `x/y`.

Example

There's a great `cos()` example at CPPReference[12] that covers a genuine use case.

But instead of stealing it, I'll just post a simple example here and you can visit their site for a real one.

```c
#include <stdio.h>
#include <math.h>

int main(void)
{
    int quo;
    double rem;

    rem = remquo(12.75, 2.25, &quo);

    printf("%d remainder %f\n", quo, rem);  // 6 remainder -0.750000
}
```

[11] https://en.cppreference.com/w/c/numeric/math/remquo
[12] https://en.cppreference.com/w/c/numeric/math/remquo

www.dbooks.org

See Also

remainder(), imaxdiv()

52.54 copysign(), copysignf(), copysignl()

Copy the sign of one value into another.

Synopsis

```
#include <math.h>

double copysign(double x, double y);

float copysignf(float x, float y);

long double copysignl(long double x, long double y);
```

Description

These functions return a number that has the magnitude of x and the sign of y. You can use them to coerce the sign to that of another value.

Neither x nor y are modified, of course. The return value holds the result.

Return Value

Returns a value with the magnitude of x and the sign of y.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    double x = 34.9;
    double y = -999.9;
    double z = 123.4;

    printf("%f\n", copysign(x, y)); // -34.900000
    printf("%f\n", copysign(x, z)); //  34.900000
}
```

See Also

signbit()

52.55 nan(), nanf(), nanl()

Return NAN.

Synopsis

```
#include <math.h>

double nan(const char *tagp);

float nanf(const char *tagp);

long double nanl(const char *tagp);
```

Description

These functions return a quiet NaN[13]. It is produced as if calling strtod() with "NAN" (or a variant thereof) as an argument.

tagp points to a string which could be several things, including empty. The contents of the string determine which variant of NaN might get returned depending on the implementation.

Which *version* of NaN? Did you even know it was possible to get this far into the weeds with something that wasn't a number?

Case 1 in which you pass in an empty string, in which case these are the same:

```
nan("");

strtod("NAN()", NULL);
```

Case 2 in which the string contains only digits 0-9, letters a-z, letters A-Z, and/or underscore:

```
nan("goats");

strtod("NAN(goats)", NULL);
```

And Case 3, in which the string contains anything else and is ignored:

```
nan("!");

strtod("NAN", NULL);
```

As for what strtod() does with those values in parens, see the [strtod()] reference page. Spoiler: it's implementation-defined.

Return Value

Returns the requested quiet NaN, or 0 if such things aren't supported by your system.

Example

```
#include <stdio.h>
#include <math.h>

```

[13] A *quiet NaN* is one that doesn't raise any exceptions.

```
int main(void)
{
    printf("%f\n", nan(""));        // nan
    printf("%f\n", nan("goats"));   // nan
    printf("%f\n", nan("!"));       // nan
}
```

See Also

strtod()

52.56 nextafter(), nextafterf(), nextafterl()

Get the next (or previous) representable floating point value.

Synopsis

```
#include <math.h>

double nextafter(double x, double y);

float nextafterf(float x, float y);

long double nextafterl(long double x, long double y);
```

Description

As you probably know, floating point numbers can't represent *every* possible real number. There are limits.

And, as such, there exists a "next" and "previous" number after or before any floating point number.

These functions return the next (or previous) representable number. That is, no floating point numbers exist between the given number and the next one.

The way it figures it out is it works from x in the direction of y, answering the question of "what is the next representable number from x as we head toward y.

Return Value

Returns the next representable floating point value from x in the direction of y.

If x equals y, returns y. And also x, I suppose.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("%.*f\n", DBL_DECIMAL_DIG, nextafter(0.5, 1.0));
```

```
    printf("%.*f\n", DBL_DECIMAL_DIG, nextafter(0.349, 0.0));
}
```

Output on my system:

```
0.50000000000000011
0.34899999999999992
```

See Also

`nexttoward()`

52.57 nexttoward(), nexttowardf(), nexttowardl()

Get the next (or previous) representable floating point value.

Synopsis

```
include <math.h>

double nexttoward(double x, long double y);

float nexttowardf(float x, long double y);

long double nexttowardl(long double x, long double y);
```

Description

These functions are the same as `nextafter()` except the second parameter is always `long double`.

Return Value

Returns the same as `nextafter()` except if `x` equals `y`, returns `y` cast to the function's return type.

Example

```
#include <stdio.h>
#include <float.h>
#include <math.h>

int main(void)
{
    printf("%.*f\n", DBL_DECIMAL_DIG, nexttoward(0.5, 1.0));
    printf("%.*f\n", DBL_DECIMAL_DIG, nexttoward(0.349, 0.0));
}
```

Output on my system:

```
0.50000000000000011
0.34899999999999992
```

See Also

`nextafter()`

52.58 fdim(), fdimf(), fdiml()

Return the positive difference between two numbers clamped at 0.

Synopsis

```
#include <math.h>

double fdim(double x, double y);

float fdimf(float x, float y);

long double fdiml(long double x, long double y);
```

Description

The positive difference between `x` and `y` is the difference… except if the difference is less than `0`, it's clamped to `0`.

These functions might throw a range error.

Return Value

Returns the difference of `x-y` if the difference is greater than `0`. Otherwise it returns `0`.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("%f\n", fdim(10.0, 3.0));   // 7.000000
    printf("%f\n", fdim(3.0, 10.0));   // 0.000000, clamped
}
```

52.59 fmax(), fmaxf(), fmaxl(), fmin(), fminf(), fminl()

Return the maximum or minimum of two numbers.

Synopsis

```
#include <math.h>

double fmax(double x, double y);
```

```
float fmaxf(float x, float y);

long double fmaxl(long double x, long double y);

double fmin(double x, double y);

float fminf(float x, float y);

long double fminl(long double x, long double y);
```

Description

Straightforwardly, these functions return the minimum or maximum of two given numbers.

If one of the numbers is NaN, the functions return the non-NaN number. If both arguments are NaN, the functions return NaN.

Return Value

Returns the minimum or maximum values, with NaN handled as mentioned above.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("%f\n", fmin(10.0, 3.0));   //  3.000000
    printf("%f\n", fmax(3.0, 10.0));   // 10.000000
}
```

52.60 fma(), fmaf(), fmal()

Floating (AKA "Fast") multiply and add.

Synopsis

```
#include <math.h>

double fma(double x, double y, double z);

float fmaf(float x, float y, float z);

long double fmal(long double x, long double y, long double z);
```

Description

This performs the operation $(x \times y) + z$, but does so in a nifty way. It does the computation as if it had infinite precision, and then rounds the final result to the final data type according to the current rounding

mode.

Contrast to if you'd do the math yourself, where it would have rounded each step of the way, potentially.

Also some architectures have a CPU instruction to do exactly this calculation, so it can do it super quick. (If it doesn't, it's considerably slower.)

You can tell if your CPU supports the fast version by checking that the macro `FP_FAST_FMA` is set to `1`. (The `float` and `long` variants of `fma()` can be tested with `FP_FAST_FMAF` and `FP_FAST_FMAL`, respectively.)

These functions might cause a range error to occur.

Return Value

Returns `(x * y) + z`.

Example

```
printf("%f\n", fma(1.0, 2.0, 3.0));  // 5.000000
```

52.61 `isgreater()`, `isgreaterequal()`, `isless()`, `islessequal()`

Floating point comparison macros.

Synopsis

```
#include <math.h>

int isgreater(any_floating_type x, any_floating_type y);

int isgreaterequal(any_floating_type x, any_floating_type y);

int isless(any_floating_type x, any_floating_type y);

int islessequal(any_floating_type x, any_floating_type y);
```

Description

These macros compare floating point numbers. Being macros, we can pass in any floating point type.

You might think you can already do that with just regular comparison operators—and you'd be right!

One one exception: the comparison operators raise the "invalid" floating exception if one or more of the operands is NaN. These macros do not.

Note that you must only pass floating point types into these functions. Passing an integer or any other type is undefined behavior.

Return Value

`isgreater()` returns the result of `x > y`.

`isgreaterequal()` returns the result of `x >= y`.

`isless()` returns the result of `x < y`.

`islessequal()` returns the result of `x <= y`.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("%d\n", isgreater(10.0, 3.0));        // 1
    printf("%d\n", isgreaterequal(10.0, 10.0));  // 1
    printf("%d\n", isless(10.0, 3.0));           // 0
    printf("%d\n", islessequal(10.0, 3.0));      // 0
}
```

See Also

`islessgreater()`, `isunordered()`

52.62 `islessgreater()`

Test if a floating point number is less than or greater than another.

Synopsis

```
#include <math.h>

int islessgreater(any_floating_type x, any_floating_type y);
```

Description

This macro is similar to `isgreater()` and all those, except it made the section name too long if I included it up there. So it gets its own spot.

This returns true if $x < y$ or $x > y$.

Even though it's a macro, we can rest assured that `x` and `y` are only evaluated once.

And even if `x` or `y` are NaN, this will not throw an "invalid" exception, unlike the normal comparison operators.

If you pass in a non-floating type, the behavior is undefined.

Return Value

Returns `(x < y) || (x > y)`.

Example

```
#include <stdio.h>
#include <math.h>

```

```
int main(void)
{
    printf("%d\n", islessgreater(10.0, 3.0));   // 1
    printf("%d\n", islessgreater(10.0, 30.0));  // 1
    printf("%d\n", islessgreater(10.0, 10.0));  // 0
}
```

See Also

`isgreater()`, `isgreaterequal()`, `isless()`, `islessequal()`, `isunordered()`

52.63 `isunordered()`

Macro returns true if either floating point argument is NaN.

Synopsis

```
#include <math.h>

int isunordered(any_floating_type x, any_floating_type y);
```

Description

The spec writes:

> The isunordered macro determines whether its arguments are unordered.

See? Told you C was easy!

It does also elaborate that the arguments are unordered if one or both of them are NaN.

Return Value

This macro returns true if one or both of the arguments are NaN.

Example

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("%d\n", isunordered(1.0, 2.0));       // 0
    printf("%d\n", isunordered(1.0, sqrt(-1)));  // 1
    printf("%d\n", isunordered(NAN, 30.0));      // 1
    printf("%d\n", isunordered(NAN, NAN));       // 1
}
```

See Also

`isgreater()`, `isgreaterequal()`, `isless()`, `islessequal()`, `islessgreater()`

Chapter 53

<setjmp.h> Non-local Goto

These functions enable you to rewind the call stack to an earlier point, with a bunch of gotchas. See the chapter on `setjmp()`/`longjmp()` for more info.

Function	Description
`longjmp()`	Return to the previously-placed bookmark
`setjmp()`	Bookmark this place to return to later

There's also a new opaque type, `jmp_buf`, that holds all the information needed to pull off this magic trick.

If you want your automatic local variables to be correct after a call to `longjmp()`. declare them as `volatile` where you called `setjmp()`.

53.1 setjmp()

Save this location as one to return to later

Synopsis

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

Description

This is how you save your position so you can `longjmp()` back it, later. Think of it as setting up a warp destination for later use.

Basically, you call this, giving it an `env` it can fill in with all the information it needs to come back here later. This `env` is one you'll pass to `longjmp()` later when you want to teleport back here.

And the really funky part is this can return two different ways:

1. It can return `0` from the call where you set up the jump destination.
2. If can return non-zero when you actually warp back here as the result of a call to `longjmp()`.

www.dbooks.org

What you can do is check the return value to see which case has occurred.

You're only allowed to call `setjmp()` in a limited number of circumstances.

1. As a standalone expression:

   ```
   setjmp(env);
   ```

   You can also cast it to `(void)` if you really wanted to do such a thing.

2. As the complete controlling expression in an `if` or `switch`.

   ```
   if (setjmp(env)) { ... }

   switch (setjmp(env)) { ... }
   ```

   But not this as it's not the complete controlling expression in this case:

   ```
   if (x == 2 && setjmp()) { ... }   // Undefined behavior
   ```

3. The same as (2), above, except with a comparison to an integer constant:

   ```
   if (setjmp(env) == 0) { ... }

   if (setjmp(env) > 2) { ... }
   ```

4. As the operand to the not (`!`) operator:

   ```
   if (!setjmp(env)) { ... }
   ```

Anything else is (you guessed it) undefined behavior!

This can be a macro or a function, but you'll treat it the same way in any case.

Return Value

This one is funky. It returns one of two things:

Returns `0` if this was the call to `setjmp()` to set it up.

Returns non-zero if being here was the result of a call to `longjmp()`. (Namely, it returns the value passed into the `longjmp()` function.)

Example

Here's a function that calls `setjmp()` to set things up (where it returns `0`), then calls a couple levels deep into functions, and finally short-circuits the return path by `longjmp()`ing back to the place where `setjmp()` was called, earlier. This time, it passes `3490` as a value, which `setjmp()` returns.

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf env;

void depth2(void)
{
    printf("Entering depth 2\n");
    longjmp(env, 3490);           // Jump back to setjmp()!!
    printf("Leaving depth 2\n");  // This won't happen
}

void depth1(void)
```

```
{
    printf("Entering depth 1\n");
    depth2();
    printf("Leaving depth 1\n");  // This won't happen
}

int main(void)
{
    switch (setjmp(env)) {
        case 0:
            printf("Calling into functions, setjmp() returned 0\n");
            depth1();
            printf("Returned from functions\n");  // This won't happen
            break;

        case 3490:
            printf("Bailed back to main, setjmp() returned 3490\n");
            break;
    }
}
```

When run, this outputs:

```
Calling into functions, setjmp() returned 0
Entering depth 1
Entering depth 2
Bailed back to main, setjmp() returned 3490
```

Notice that the second `printf()` in `case 0` didn't run; it got jumped over by `longjmp()`!

See Also

`longjmp()`

53.2 `longjmp()`

Return to the previous `setjmp()` location

Synopsis

```
#include <setjmp.h>

_Noreturn void longjmp(jmp_buf env, int val);
```

Description

This returns to a previous call to `setjmp()` back in the call history. `setjmp()` will return the `val` passed into `longjmp()`.

The `env` passed to `setjmp()` should be the same one you pass into `longjmp()`.

There are a bunch of potential issues with doing this, so you'll want to be careful that you avoid undefined behavior by not doing the following:

1. Don't call `longjmp()` if the corresponding `setjmp()` was in a different thread.
2. Don't call `longjmp()` if you didn't call `setjmp()` first.
3. Don't call `longjmp()` if the function that called `setjmp()` has completed.
4. Don't call `longjmp()` if the call to `setjmp()` had a variable length array (VLA) in scope and the scope has ended.
5. Don't call `longjmp()` if there are any VLAs in any active scopes between the `setjmp()` and the `longjmp()`. A good rule of thumb here is to not mix VLAs and `longjmp()`.

Though `longjmp()` attempts to restore the machine to the state at the `setjmp()`, including local variables, there are some things that aren't brought back to life:

- Non-volatile local variables that might have changed
- Floating point status flags
- Open files
- Any other component of the abstract machine

Return Value

This one is also funky in that it is one of the few functions in C that never returns!

Example

Here's a function that calls `setjmp()` to set things up (where it returns `0`), then calls a couple levels deep into functions, and finally short-circuits the return path by `longjmp()`ing back to the place where `setjmp()` was called, earlier. This time, it passes `3490` as a value, which `setjmp()` returns.

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf env;

void depth2(void)
{
    printf("Entering depth 2\n");
    longjmp(env, 3490);           // Jump back to setjmp()!!
    printf("Leaving depth 2\n");  // This won't happen
}

void depth1(void)
{
    printf("Entering depth 1\n");
    depth2();
    printf("Leaving depth 1\n");  // This won't happen
}

int main(void)
{
    switch (setjmp(env)) {
        case 0:
            printf("Calling into functions, setjmp() returned 0\n");
            depth1();
            printf("Returned from functions\n");  // This won't happen
            break;
```

```

        case 3490:
            printf("Bailed back to main, setjmp() returned 3490\n");
            break;
    }
}
```

When run, this outputs:

```
Calling into functions, setjmp() returned 0
Entering depth 1
Entering depth 2
Bailed back to main, setjmp() returned 3490
```

Notice that the second `printf()` in `case 0` didn't run; it got jumped over by `longjmp()`!

See Also

`setjmp()`

Chapter 54

<signal.h> signal handling

Function	Description
signal()	Set a signal handler for a given signal
raise()	Cause a signal to be raised

Handle signals in a portable way, kind of!

These signals get raised for a variety of reasons such as CTRL-C being hit, requests to terminate for external programs, memory access violations, and so on.

Your OS likely defines a plethora of other signals, as well.

This system is pretty limited, as seen below. If you're on Unix, it's almost certain your OS has far superior signal handling capabilities than the C standard library. Check out sigaction[1].

54.1 signal()

Set a signal handler for a given signal

Synopsis

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

Description

How's *that* for a function declaration?

Let's ignore it for a moment and just talk about what this function does.

When a signal is raised, *something* is going to happen. This function lets you decide to do one of these things when the signal is raised:

- Ignore the signal

[1] https://man.archlinux.org/man/sigaction.2.en

- Perform the default action
- Have a specific function called

The `signal()` function takes two arguments. The first, `sig`, is the name of the signal to handle.

Signal	Description
`SIGABRT`	Raised when `abort()` is called
`SIGFPE`	Floating-point arithmetic exception
`SIGILL`	CPU tried to execute an illegal instruction
`SIGINT`	Interrupt signal, as if `CTRL-C` were pressed
`SIGSEGV`	Segmention Violation: attempted to access restricted memory
`SIGTERM`	Termination request[2]

So that's the first bit when you call `signal()`—tell it the signal in question:

```
signal(SIGINT, ...
```

But what's that `func` parameter?

For spoilers, it's a pointer to a function that takes an `int` argument and returns `void`. We can use this to call an arbitrary function when the signal occurs.

Before we do that, though, let's look at the easy ones: telling the system to ignore the signal or perform the default action (which it does by default if you never call `signal()`).

You can set `func` to one of two special values to make this happen:

`func`	Description
`SIG_DFL`	Perform the default action on this signal
`SIG_IGN`	Ignore this signal

For example:

```
signal(SIGTERM, SIG_DFL);  // Default action on SIGTERM
signal(SIGINT, SIG_IGN);   // Ignore SIGINT
```

But what if you want to have your own handler do something instead of the default or ignoring it? You can pass in your own function to be called. That's what the crazy function signature is partially about. It's saying that the argument can be a pointer to a function that takes an `int` argument and returns `void`.

So if you wanted to call your handler, you could have code like this:

```
int handler(int sig)
{
    // Handle the signal
}

int main(void)
{
    signal(SIGINT, handler);
```

What can you do in the signal handler? Not much.

If the signal is due to `abort()` or `raise()`, the handler can't call `raise()`.

[2] As if might be sent from Unix's `kill` command.]

www.dbooks.org

If the signal is **not** due to `abort()` or `raise()`, you're only allowed to call these functions from the standard library (though the spec doesn't prohibit calling other non-library functions):

- `abort()`
- `_Exit()`
- `quick_exit()`
- Functions in `<stdatomic.h>` when the atomic arguments are lock-free
- `signal()` with a first argument equivalent to the argument that was passed into the handler

In addition, if the signal was **not** due to `abort()` or `raise()`, the handler can't access any object with static or thread-storage duration unless it's lock-free.

An exception is that you can assign to (but not read from!) a variable of type `volatile sig_atomic_t`.

It's up to the implementation, but the signal handler might be reset to `SIG_DFL` just before the handler is called.

It's undefined behavior to call `signal()` in a multithreaded program.

It's undefined behavior to return from the handler for `SIGFPE`, `SIGILL`, `SIGSEGV`, or any implementation-defined value. You must exit.

The implementation might or might not prevent other signals from arising while in the signal handler.

Return Value

On success, `signal()` returns a pointer to the previous signal handler set by a call to `signal()` for that particular signal number. If you haven't called it set, returns `SIG_DFL`.

On failure, `SIG_ERR` is returned and `errno` is set to a positive value.

Example

Here's a program that causes `SIGINT` to be ignored. Commonly you trigger this signal by hitting `CTRL-C`.

```c
#include <stdio.h>
#include <signal.h>

int main(void)
{
    signal(SIGINT, SIG_IGN);

    printf("You can't hit CTRL-C to exit this program. Try it!\n\n");
    printf("Press return to exit, instead.");
    fflush(stdout);
    getchar();
}
```

Output:

```
You can't hit CTRL-C to exit this program. Try it!

Press return to exit, instead.^C^C^C^C^C^C^C^C^C^C^C
```

This program sets the signal handler, then raises the signal. The signal handler fires.

```c
#include <stdio.h>
#include <signal.h>

void handler(int sig)
```

```
{
    // Undefined behavior to call printf() if this handler was not
    // as the result of a raise(), i.e. if you hit CTRL-C.

    printf("Got signal %d!\n", sig);

    // Common to reset the handler just in case the implementation set
    // it to SIG_DFL when the signal occurred.

    signal(sig, handler);
}

int main(void)
{
    signal(SIGINT, handler);

    raise(SIGINT);
    raise(SIGINT);
    raise(SIGINT);
}
```

Output:

```
Got signal 2!
Got signal 2!
Got signal 2!
```

This example catches `SIGINT` but then sets a flag to `1`. Then the main loop sees the flag and exits.

```
#include <stdio.h>
#include <signal.h>

volatile sig_atomic_t x;

void handler(int sig)
{
    x = 1;
}

int main(void)
{
    signal(SIGINT, handler);

    printf("Hit CTRL-C to exit\n");
    while (x != 1);
}
```

See Also

`raise()`, `abort()`

54.2 raise()

Cause a signal to be raised

Synopsis

```
#include <signal.h>

int raise(int sig);
```

Description

Causes the signal handler for the signal `sig` to be called. If the handler is `SIG_DFL` or `SIG_IGN`, then the default action or no action happens.

`raise()` returns after the signal handler has finished running.

Interestingly, if you cause a signal to happen with `raise()`, you can call library functions from within the signal handler without causing undefined behavior. I'm not sure how this fact is practically useful, though.

Return Value

Returns `0` on success. Nonzero otherwise.

Example

This program sets the signal handler, then raises the signal. The signal handler fires.

```
#include <stdio.h>
#include <signal.h>

void handler(int sig)
{
    // Undefined behavior to call printf() if this handler was not
    // as the result of a raise(), i.e. if you hit CTRL-C.

    printf("Got signal %d!\n", sig);

    // Common to reset the handler just in case the implementation set
    // it to SIG_DFL when the signal occurred.

    signal(sig, handler);
}

int main(void)
{
    signal(SIGINT, handler);

    raise(SIGINT);
    raise(SIGINT);
    raise(SIGINT);
}
```

Output:

```
Got signal 2!
Got signal 2!
Got signal 2!
```

See Also

`signal()`

www.dbooks.org

Chapter 55

<stdalign.h> Macros for Alignment

If you're coding up something low-level like a memory allocator that interfaces with your OS, you might need this header file. But most C devs go their careers without using it.

Alignment[1] is all about multiples of addresses on which objects can be stored. Can you store this at any address? Or must it be a starting address that's divisible by 2? Or 8? Or 16?

Name	Description
`alignas()`	Specify alignment, expands to `_Alignas`
`alignof()`	Get alignment, expands to `_Alignof`

These two additional macros are defined to be `1`:

```
__alignas_is_defined
__alignof_is_defined
```

Quick note: alignments greater than that of `max_align_t` are known as *overalignments* and are implementation-defined.

55.1 `alignas()` `_Alignas()`

Force a variable to have a certain alignment

Synopsis

```
#include <stdalign.h>

alignas(type-name)
alignas(constant-expression)
```

```
_Alignas(type-name)
_Alignas(constant-expression)
```

[1] https://en.wikipedia.org/wiki/Data_structure_alignment

Description

Use this *alignment specifier* to force the alignment of particular variables. For instance, we can declare `c` to be `char`, but aligned as if it were an `int`:

```
char alignas(int) c;
```

You can put a constant integer expression in there, as well. The compiler will probably impose limits on what these values can be. Small powers of 2 (1, 2, 4, 8, and 16) are generally safe bets.

```
char alignas(8) c;   // align on 8-byte boundaries
```

For convenience, you can also specify `0` if you want the default alignment (as if you hadn't said `alignas()` at all):

```
char alignas(0) c;   // use default alignment for this type
```

Example

```
#include <stdalign.h>
#include <stdio.h>     // for printf()
#include <stddef.h>    // for max_align_t

int main(void)
{
    int i, j;
    char alignas(max_align_t) a, b;
    char alignas(int) c, d;
    char e, f;

    printf("i: %p\n", (void *)&i);
    printf("j: %p\n\n", (void *)&j);
    printf("a: %p\n", (void *)&a);
    printf("b: %p\n\n", (void *)&b);
    printf("c: %p\n", (void *)&c);
    printf("d: %p\n\n", (void *)&d);
    printf("e: %p\n", (void *)&e);
    printf("f: %p\n", (void *)&f);
}
```

Output on my system follows. Notice the difference between the pairs of values.

- `i` and `j`, both `int`s, are aligned on 4-byte boundaries.
- `a` and `b` have been forced to the boundary of the type `max_align_t`, which is every 16 bytes on my system.
- `c` and `d` have been forced to the same alignment as `int`, which is 4 bytes, just like with `i` and `j`.
- `e` and `f` do not have an alignment specified, so they were stored with their default alignment of 1 byte.

```
i: 0x7ffee7dfb4cc    <-- difference of 4 bytes
j: 0x7ffee7dfb4c8

a: 0x7ffee7dfb4c0    <-- difference of 16 bytes
b: 0x7ffee7dfb4b0

c: 0x7ffee7dfb4ac    <-- difference of 4 bytes
d: 0x7ffee7dfb4a8
```

```
e: 0x7ffee7dfb4a7    <-- difference of 1 byte
f: 0x7ffee7dfb4a6
```

See Also

alignof, max_align_t

55.2 alignof() _Alignof()

Get the alignment of a type

Synopsis

```
#include <stdalign.h>

alignof(type-name)
```

```
_Alignof(type-name)
```

Description

This evaluates to a value of type size_t that gives the alignment of a particular type on your system.

Return Value

Returns the alignment value, i.e. the address of the beginning of the given type of object must begin on an address boundary divisible by this number.

Example

Print out the alignments of a variety of different types.

```
#include <stdalign.h>
#include <stdio.h>     // for printf()
#include <stddef.h>    // for max_align_t

struct t {
    int a;
    char b;
    float c;
};

int main(void)
{
    printf("char        : %zu\n", alignof(char));
    printf("short       : %zu\n", alignof(short));
    printf("int         : %zu\n", alignof(int));
    printf("long        : %zu\n", alignof(long));
    printf("long long   : %zu\n", alignof(long long));
    printf("double      : %zu\n", alignof(double));
    printf("long double: %zu\n", alignof(long double));
```

```
    printf("struct t    : %zu\n", alignof(struct t));
    printf("max_align_t: %zu\n", alignof(max_align_t));
}
```

Output on my system:

```
char        : 1
short       : 2
int         : 4
long        : 8
long long   : 8
double      : 8
long double: 16
struct t    : 16
max_align_t: 16
```

See Also

`alignas`, `max_align_t`

Chapter 56

<stdarg.h> Variable Arguments

Macro	Description
va_arg()	Get the next variable argument
va_copy()	Copy a va_list and the work done so far
va_end()	Signify we're done processing variable arguments
va_start()	Initialize a va_list to start variable argument processing

This header file is what allows you to write functions that take a variable number of arguments.

In addition to the macros, you get a new type that helps C keep track of where it is in the variable-number-of-arguments-processing: va_list. This type is opaque, and you'll be passing it around to the various macros to help get at the arguments.

Note that every variadic function requires at least one non-variable parameter. You need this to kick off processing with va_start().

56.1 va_arg()

Get the next variable argument

Synopsis

```
#include <stdarg.h>

type va_arg(va_list ap, type);
```

Description

If you have a variable argument list you've initialized with va_start(), pass it to this one along with the type of argument you're trying to get, e.g.

```
int   x = va_arg(args, int);
float y = va_arg(args, float);
```

Return Value

Evaluates to the value and type of the next variable argument.

Example

Here's a demo that adds together an arbitrary number of integers. The first argument is the number of integers to add together. We'll make use of that to figure out how many times we have to call `va_arg()`.

```
#include <stdio.h>
#include <stdarg.h>

int add(int count, ...)
{
    int total = 0;
    va_list va;

    va_start(va, count);   // Start with arguments after "count"

    for (int i = 0; i < count; i++) {
        int n = va_arg(va, int);   // Get the next int

        total += n;
    }

    va_end(va);  // All done

    return total;
}

int main(void)
{
    printf("%d\n", add(4, 6, 2, -4, 17));  // 6 + 2 - 4 + 17 = 21
    printf("%d\n", add(2, 22, 44));        // 22 + 44 = 66
}
```

See Also

`va_start()`, `va_end()`

56.2 va_copy()

Copy a `va_list` and the work done so far

Synopsis

```
#include <stdarg.h>

void va_copy(va_list dest, va_list src);
```

Description

The main intended use of this is to save your state partway through processing variable arguments so you can scan ahead and then rewind back to the save point.

You pass in a `src` `va_list` and it copies it to `dest`.

If you've already called this once for a particular `dest`, you can't call it (or `va_start()`) again with the same `dest` unless you call `va_end()` on that `dest` first.

```
va_copy(dest, src);
va_copy(dest, src2);  // BAD!

va_copy(dest, src);
va_start(dest, var);  // BAD!

va_copy(dest, src);
va_end(dest);
va_copy(dest, src2);  // OK!

va_copy(dest, src);
va_end(dest);
va_start(dest, var);  // OK!
```

Return Value

Returns nothing.

Example

Here's an example where we're adding together all the variable arguments, but then we want to go back and add on all the numbers past the first two, for example if the arguments are:

```
10 20 30 40
```

First we add them all for `100`, and then we add on everything from the third number on, so add on `30+40` for a total of `170`.

We'll do this by saving our place in the variable argument processing with `va_copy` and then using that later to reprocess the trailing arguments.

(And yes, I know there's a mathematical way to do this without all the rewinding, but I'm having an heck of a time coming up with a good example!)

```
#include <stdio.h>
#include <stdarg.h>

// Add all the numbers together, but then add on all the numbers
// past the second one again.
int contrived_adder(int count, ...)
{
    if (count < 3) return 0; // OK, I'm being lazy. You got me.

    int total = 0;

    va_list args, mid_args;

    va_start(args, count);
```

```

    for (int i = 0; i < count; i++) {

        // If we're at the second number, save our place in
        // mid_args:

        if (i == 2)
            va_copy(mid_args, args);

        total += va_arg(args, int);
    }

    va_end(args); // Done with this

    // But now let's start with mid_args and add all those on:
    for (int i = 0; i < count - 2; i++)
        total += va_arg(mid_args, int);

    va_end(mid_args); // Done with this, too

    return total;
}

int main(void)
{
    // 10+20+30 + 30 == 90
    printf("%d\n", contrived_adder(3, 10, 20, 30));

    // 10+20+30+40+50 + 30+40+50 == 270
    printf("%d\n", contrived_adder(5, 10, 20, 30, 40, 50));
}
```

See Also

`va_start()`, `va_arg()`, `va_end()`

56.3 va_end()

Signify we're done processing variable arguments

Synopsis

```
#include <stdarg.h>

void va_end(va_list ap);
```

Description

After you've `va_start()`ed or `va_copy`'d a new `va_list`, you **must** call `va_end()` with it before it goes out of scope.

You also have to do this if you're going to call `va_start()` or `va_copy()` *again* on a variable you've already done that to.

Them's the rules if you want to avoid undefined behavior.

But just think of it as cleanup. You called `va_start()`, so you'll call `va_end()` when you're done.

Return Value

Returns nothing.

Example

Here's a demo that adds together an arbitrary number of integers. The first argument is the number of integers to add together. We'll make use of that to figure out how many times we have to call `va_arg()`.

```
#include <stdio.h>
#include <stdarg.h>

int add(int count, ...)
{
    int total = 0;
    va_list va;

    va_start(va, count);   // Start with arguments after "count"

    for (int i = 0; i < count; i++) {
        int n = va_arg(va, int);   // Get the next int

        total += n;
    }

    va_end(va);  // All done

    return total;
}

int main(void)
{
    printf("%d\n", add(4, 6, 2, -4, 17));  // 6 + 2 - 4 + 17 = 21
    printf("%d\n", add(2, 22, 44));        // 22 + 44 = 66
}
```

See Also

`va_start()`, `va_copy()`

56.4 `va_start()`

Initialize a `va_list` to start variable argument processing

Synopsis

```
#include <stdarg.h>

void va_start(va_list ap, parmN);
```

Description

You've declared a variable of type `va_list` to keep track of the variable argument processing… now how to initialize it so you can start calling `va_arg()` to get those arguments?

`va_start()` to the rescue!

What you do is pass in your `va_list`, here shown as parameter `ap`. Just pass the list, not a pointer to it.

Then for the second argument to `va_start()`, you give the name of the parameter that you want to start processing arguments *after*. This must be the parameter right before the `...` in the argument list.

If you've already called `va_start()` on a particular `va_list` and you want to call `va_start()` on it again, you **must** call `va_end()` first!

Return Value

Returns nothing!

Example

Here's a demo that adds together an arbitrary number of integers. The first argument is the number of integers to add together. We'll make use of that to figure out how many times we have to call `va_arg()`.

```
#include <stdio.h>
#include <stdarg.h>

int add(int count, ...)
{
    int total = 0;
    va_list va;

    va_start(va, count);   // Start with arguments after "count"

    for (int i = 0; i < count; i++) {
        int n = va_arg(va, int);   // Get the next int

        total += n;
    }

    va_end(va);  // All done

    return total;
}

int main(void)
{
    printf("%d\n", add(4, 6, 2, -4, 17));  // 6 + 2 - 4 + 17 = 21
    printf("%d\n", add(2, 22, 44));        // 22 + 44 = 66
}
```

See Also

`va_arg()`, `va_end()`

Chapter 57

<stdatomic.h> Atomic-Related Functions

Function	Description
atomic_compare_exchange_strong_explicit()	Atomic compare and exchange, strong, explicit
atomic_compare_exchange_strong()	Atomic compare and exchange, strong
atomic_compare_exchange_weak_explicit()	Atomic compare and exchange, weak, explicit
atomic_compare_exchange_weak()	Atomic compare and exchange, weak
atomic_exchange_explicit()	Replace a value in an atomic object, explicit
atomic_exchange()	Replace a value in an atomic object
atomic_fetch_add_explicit()	Atomically add to an atomic integer, explicit
atomic_fetch_add()	Atomically add to an atomic integer
atomic_fetch_and_explicit()	Atomically bitwise-AND an atomic integer, explicit
atomic_fetch_and()	Atomically bitwise-AND an atomic integer
atomic_fetch_or_explicit()	Atomically bitwise-OR an atomic integer, explicit
atomic_fetch_or()	Atomically bitwise-OR an atomic integer
atomic_fetch_sub_explicit()	Atomically subtract from an atomic integer, explicit
atomic_fetch_sub()	Atomically subtract from an atomic integer
atomic_fetch_xor_explicit()	Atomically bitwise-XOR an atomic integer, explicit
atomic_fetch_xor()	Atomically bitwise-XOR an atomic integer
atomic_flag_clear_explicit()	Clear an atomic flag, explicit
atomic_flag_clear()	Clear an atomic flag
atomic_flag_test_and_set_explicit()	Test and set an atomic flag, explicit
atomic_flag_test_and_set()	Test and set an atomic flag
atomic_init()	Initialize an atomic variable
atomic_is_lock_free()	Determine if an atomic type is lock free
atomic_load_explicit()	Return a value from an atomic variable, explicit
atomic_load()	Return a value from an atomic variable
atomic_signal_fence()	Fence for intra-thread signal handlers
atomic_store_explicit()	Store a value in an atomic variable, explicit
atomic_store()	Store a value in an atomic variable
atomic_thread_fence()	Set up a fence
ATOMIC_VAR_INIT()	Create an initializer for an atomic variable
kill_dependency()	End a dependency chain

You might need to add `-latomic` to your compilation command line on Unix-like operating systems.

www.dbooks.org

57.1 Atomic Types

A bunch of types are predefined by this header:

Atomic type	Longhand equivalent
`atomic_bool`	`_Atomic _Bool`
`atomic_char`	`_Atomic char`
`atomic_schar`	`_Atomic signed char`
`atomic_uchar`	`_Atomic unsigned char`
`atomic_short`	`_Atomic short`
`atomic_ushort`	`_Atomic unsigned short`
`atomic_int`	`_Atomic int`
`atomic_uint`	`_Atomic unsigned int`
`atomic_long`	`_Atomic long`
`atomic_ulong`	`_Atomic unsigned long`
`atomic_llong`	`_Atomic long long`
`atomic_ullong`	`_Atomic unsigned long long`
`atomic_char16_t`	`_Atomic char16_t`
`atomic_char32_t`	`_Atomic char32_t`
`atomic_wchar_t`	`_Atomic wchar_t`
`atomic_int_least8_t`	`_Atomic int_least8_t`
`atomic_uint_least8_t`	`_Atomic uint_least8_t`
`atomic_int_least16_t`	`_Atomic int_least16_t`
`atomic_uint_least16_t`	`_Atomic uint_least16_t`
`atomic_int_least32_t`	`_Atomic int_least32_t`
`atomic_uint_least32_t`	`_Atomic uint_least32_t`
`atomic_int_least64_t`	`_Atomic int_least64_t`
`atomic_uint_least64_t`	`_Atomic uint_least64_t`
`atomic_int_fast8_t`	`_Atomic int_fast8_t`
`atomic_uint_fast8_t`	`_Atomic uint_fast8_t`
`atomic_int_fast16_t`	`_Atomic int_fast16_t`
`atomic_uint_fast16_t`	`_Atomic uint_fast16_t`
`atomic_int_fast32_t`	`_Atomic int_fast32_t`
`atomic_uint_fast32_t`	`_Atomic uint_fast32_t`
`atomic_int_fast64_t`	`_Atomic int_fast64_t`
`atomic_uint_fast64_t`	`_Atomic uint_fast64_t`
`atomic_intptr_t`	`_Atomic intptr_t`
`atomic_uintptr_t`	`_Atomic uintptr_t`
`atomic_size_t`	`_Atomic size_t`
`atomic_ptrdiff_t`	`_Atomic ptrdiff_t`
`atomic_intmax_t`	`_Atomic intmax_t`
`atomic_uintmax_t`	`_Atomic uintmax_t`

You can make your own additional types with the `_Atomic` type qualifier:

```
_Atomic double x;
```

or the `_Atomic()` type specifier:

```
_Atomic(double) x;
```

57.2 Lock-free Macros

These macros let you know if a type is lock-free or not. Maybe.

They can be used at compile time with `#if`. They apply to both signed and unsigned types.

Atomic Type	Lock Free Macro
`atomic_bool`	`ATOMIC_BOOL_LOCK_FREE`
`atomic_char`	`ATOMIC_CHAR_LOCK_FREE`
`atomic_char16_t`	`ATOMIC_CHAR16_T_LOCK_FREE`
`atomic_char32_t`	`ATOMIC_CHAR32_T_LOCK_FREE`
`atomic_wchar_t`	`ATOMIC_WCHAR_T_LOCK_FREE`
`atomic_short`	`ATOMIC_SHORT_LOCK_FREE`
`atomic_int`	`ATOMIC_INT_LOCK_FREE`
`atomic_long`	`ATOMIC_LONG_LOCK_FREE`
`atomic_llong`	`ATOMIC_LLONG_LOCK_FREE`
`atomic_intptr_t`	`ATOMIC_POINTER_LOCK_FREE`

These macros can interestingly have *three* different values:

Value	Meaning
`0`	Never lock-free.
`1`	*Sometimes* lock-free[1].
`2`	Always lock-free.

57.3 Atomic Flag

The `atomic_flag` opaque type is the only time guaranteed to be lock-free. Though your PC implementation probably does a lot more.

It is accessed through the `atomic_flag_test_and_set()` and `atomic_flag_clear()` functions.

Before use, it can be initialized to a clear state with:

```
atomic_flag f = ATOMIC_FLAG_INIT;
```

57.4 Memory Order

This header introduces a new `enum` type called `memory_order`. This is used by a bunch of the functions to specify memory orders other than sequential consistency.

`memory_order`	Description
`memory_order_seq_cst`	Sequential Consistency
`memory_order_acq_rel`	Acquire/Release
`memory_order_release`	Release
`memory_order_acquire`	Acquire
`memory_order_consume`	Consume
`memory_order_relaxed`	Relaxed

You can feed these into atomic functions with the `_explicit` suffix.

The non-`_explcit` versions of the functions are the same as if you'd called the `_explicit` counterpart with `memory_order_seq_cst`.

[1] Maybe it depends on the run-time environment and can't be known at compile-time.

www.dbooks.org

57.5 ATOMIC_VAR_INIT()

Create an initializer for an atomic variable

Synopsis

```
#include <stdatomic.h>

#define ATOMIC_VAR_INIT(C value)   // Deprecated
```

Description

This macro expands to an initializer, so you can use it when a variable is defined.

The type of the `value` should be the base type of the atomic variable.

The initialization itself is *not* an atomic operation, ironically.

CPPReference says this is deprecated[2] and likely to be removed. Standards document p1138r0[3] elaborates that the macro is limited in that it can't properly initialize atomic `structs`, and its original *raison d'être* turned out to not be useful.

Just initialize the variable straight-up, instead.

Return Value

Expands to an initializer suitable for this atomic variable.

Example

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    atomic_int x = ATOMIC_VAR_INIT(3490);  // Deprecated
    printf("%d\n", x);
}
```

See Also

`atomic_init()`

57.6 atomic_init()

Initialize an atomic variable

[2] https://en.cppreference.com/w/cpp/atomic/ATOMIC_VAR_INIT
[3] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1138r0.pdf

Synopsis

```
#include <stdatomic.h>

void atomic_init(volatile A *obj, C value);
```

Description

You can use this to initialize an atomic variable.

The type of the `value` should be the base type of the atomic variable.

The initialization itself is *not* an atomic operation, ironically.

As far as I can tell, there's no difference between this and assigning directly to the atomic variable. The spec says it's there to allow the compiler to inject any additional initialization that needs doing, but everything seems fine without it. If anyone has more info, send it my way.

Return Value

Returns nothing!

Example

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    atomic_int x;

    atomic_init(&x, 3490);

    printf("%d\n", x);
}
```

See Also

`ATOMIC_VAR_INIT()`, `atomic_store()`, `atomic_store_explicit()`

57.7 `kill_dependency()`

End a dependency chain

Synopsis

```
#include <stdatomic.h>

type kill_dependency(type y);
```

Description

This is potentially useful for optimizing if you're using `memory_order_consume` anywhere.

And if you know what you're doing. If unsure, learn more before trying to use this.

Return Value

Returns the value passed in.

Example

In this example, `i` carries a dependency into `x`. And would do into `y`, except for the call to `kill_dependency()`.

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    atomic_int a;
    int i = 10, x, y;

    atomic_store_explicit(&a, 3490, memory_order_release);

    i = atomic_load_explicit(&a, memory_order_consume);
    x = i;
    y = kill_dependency(i);

    printf("%d %d\n", x, y);  // 3490 and either 3490 or 10
}
```

57.8 `atomic_thread_fence()`

Set up a fence

Synopsis

```
#include <stdatomic.h>

void atomic_thread_fence(memory_order order);
```

Description

This sets up a memory fence with the specified `order`.

`order`	Description
`memory_order_seq_cst`	Sequentially consistency acquire/release fence
`memory_order_acq_rel`	Acquire/release dence
`memory_order_release`	Release fence
`memory_order_acquire`	Acquire fence
`memory_order_consume`	Acquire fence (again)

order	Description
memory_order_relaxed	No fence at all—no point in calling with this

You might try to avoid using these and just stick with the different modes with atomic_store_explicit() and atomic_load_explicit(). Or not.

Return Value

Returns nothing!

Example

```
#include <stdio.h>
#include <threads.h>
#include <stdatomic.h>

atomic_int shared_1 = 1;
atomic_int shared_2 = 2;

int thread_1(void *arg)
{
    (void)arg;

    atomic_store_explicit(&shared_1, 10, memory_order_relaxed);

    atomic_thread_fence(memory_order_release);

    atomic_store_explicit(&shared_2, 20, memory_order_relaxed);

    return 0;
}

int thread_2(void *arg)
{
    (void)arg;

    // If this fence runs after the release fence, we're
    // guaranteed to see thread_1's changes to the shared
    // varaibles.

    atomic_thread_fence(memory_order_acquire);

    if (shared_2 == 20) {
        printf("Shared_1 better be 10 and it's %d\n", shared_1);
    } else {
        printf("Anything's possible: %d %d\n", shared_1, shared_2);
    }

    return 0;
}

int main(void)
```

```
{
    thrd_t t1, t2;

    thrd_create(&t2, thread_2, NULL);
    thrd_create(&t1, thread_1, NULL);

    thrd_join(t1, NULL);
    thrd_join(t2, NULL);
}
```

See Also

`atomic_store_explicit()`, `atomic_load_explicit()`, `atomic_signal_fence()`

57.9 atomic_signal_fence()

Fence for intra-thread signal handlers

Synopsis

```
#include <stdatomic.h>

void atomic_signal_fence(memory_order order);
```

Description

This works like `atomic_thread_fence()` except its purpose is within in a single thread; notably for use in a signal handler in that thread.

Since signals can happen at any time, we might need a way to be certain that any writes by the thread that happened before the signal handler be visible within that signal handler.

Return Value

Returns nothing!

Example

Partial demo. (Note that it's technically undefined behavior to call `printf()` in a signal handler.)

```
#include <stdio.h>
#include <signal.h>
#include <stdatomic.h>

int global;

void handler(int sig)
{
    (void)sig;

    // If this runs before the release, the handler will
```

```
    // potentially see global == 0.
    //
    // Otherwise, it will definitely see global == 10.

    atomic_signal_fence(memory_order_acquire);

    printf("%d\n", global);
}

int main(void)
{
    signal(SIGINT, handler);

    global = 10;

    atomic_signal_fence(memory_order_release);

    // If the signal handler runs after the release
    // it will definitely see the value 10 in global.
}
```

See Also

`atomic_thread_fence()`, `signal()`

57.10 atomic_is_lock_free()

Determine if an atomic type is lock free

Synopsis

```
#include <stdatomic.h>

_Bool atomic_is_lock_free(const volatile A *obj);
```

Description

Determines if the variable `obj` of type `A` is lock-free. Can be used with any type.

Unlike the lock-free macros which can be used at compile-time, this is strictly a run-time function. So in places where the macros say “maybe”, this function will definitely tell you one way or another if the atomic variable is lock-free.

This is useful when you’re defining your own atomic variables and want to know their lock-free status.

Return Value

True if the variable is lock-free, false otherwise.

Example

Test if a couple `structs` and an atomic `double` are lock-free. On my system, the larger `struct` is too big to be lock-free, but the other two are OK.

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    struct foo {
        int x, y;
    };

    struct bar {
        int x, y, z;
    };

    _Atomic(double) a;
    struct foo b;
    struct bar c;

    printf("a is lock-free: %d\n", atomic_is_lock_free(&a));
    printf("b is lock-free: %d\n", atomic_is_lock_free(&b));
    printf("c is lock-free: %d\n", atomic_is_lock_free(&c));
}
```

Output on my system (YMMV):

```
a is lock-free: 1
b is lock-free: 1
c is lock-free: 0
```

See Also

Lock-free Macros

57.11 `atomic_store()`

Store a value in an atomic variable

Synopsis

```
#include <stdatomic.h>

void atomic_store(volatile A *object, C desired);

void atomic_store_explicit(volatile A *object,
                           C desired, memory_order order);
```

Description

Store a value in an atomic variable, possible synchronized.

This is like a plain assignment, but with more flexibility.

These have the same storage effect for an `atomic_int x`:

```
x = 10;
atomic_store(&x, 10);
atomic_store_explicit(&x, 10, memory_order_seq_cst);
```

But the last function, `atomic_store_explicit()`, lets you specify the memory order.

Since this is a "release-y" operation, none of the "acquire-y" memory orders are legal. `order` can be only be `memory_order_seq_cst`, `memory_order_release`, or `memory_order_relaxed`.

`order` cannot be `memory_order_acq_rel`, `memory_order_acquire`, or `memory_order_consume`.

Return Value

Returns nothing!

Example

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    atomic_int x = 0;
    atomic_int y = 0;

    atomic_store(&x, 10);

    atomic_store_explicit(&y, 20, memory_order_relaxed);

    // Will print either "10 20" or "10 0":
    printf("%d %d\n", x, y);
}
```

See Also

`atomic_init()`, `atomic_load()`, `atomic_load_explicit()`, `atomic_exchange()`, `atomic_exchange_explicit()`, `atomic_compare_exchange_strong()`, `atomic_compare_exchange_strong_explicit()`, `atomic_compare_exchange_weak()`, `atomic_compare_exchange_weak_explicit()`, `atomic_fetch_*()`

57.12 atomic_load()

Return a value from an atomic variable

Synopsis

```
#include <stdatomic.h>

C atomic_load(const volatile A *object);
```

```
C atomic_load_explicit(const volatile A *object, memory_order order);
```

Description

For a pointer to an `object` of type `A`, atomically returns its value `C`. This is a generic function that can be used with any type.

The function `atomic_load_explicit()` lets you specify the memory order.

Since this is an "acquire-y" operation, none of the "release-y" memory orders are legal. `order` can be only be `memory_order_seq_cst`, `memory_order_acquire`, `memory_order_consume`, or `memory_order_relaxed`.

`order` cannot be `memory_order_acq_rel` or `memory_order_release`.

Return Value

Returns the value stored in `object`.

Example

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    atomic_int x = 10;

    int v = atomic_load(&x);

    printf("%d\n", v);  // 10
}
```

See Also

`atomic_store()`, `atomic_store_explicit()`

57.13 `atomic_exchange()`

Replace a value in an atomic object

Synopsis

```
#include <stdatomic.h>

C atomic_exchange(volatile A *object, C desired);

C atomic_exchange_explicit(volatile A *object, C desired,
                           memory_order order);
```

Description

Sets the value in `object` to `desired`.

`object` is type `A`, some atomic type.

`desired` is type `C`, the respective non-atomic type to `A`.

This is very similar to `atomic_store()`, except the previous value is atomically returned.

Return Value

Returns the previous value of `object`.

Example

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    atomic_int x = 10;

    int previous = atomic_exchange(&x, 20);

    printf("x is  %d\n", x);
    printf("x was %d\n", previous);
}
```

Output:

```
x is  20
x was 10
```

See Also

`atomic_init()`, `atomic_load()`, `atomic_load_explicit()`, `atomic_store()`, `atomic_store_explicit()` `atomic_compare_exchange_strong()`, `atomic_compare_exchange_strong_explicit()`, `atomic_compare_exchange_weak()`, `atomic_compare_exchange_weak_explicit()`

57.14 `atomic_compare_exchange_*()`

Atomic compare and exchange

Synopsis

```
#include <stdatomic.h>

_Bool atomic_compare_exchange_strong(volatile A *object,
                                     C *expected, C desired);

_Bool atomic_compare_exchange_strong_explicit(volatile A *object,
                                              C *expected, C desired,
```

```
                                                memory_order success,
                                                memory_order failure);

_Bool atomic_compare_exchange_weak(volatile A *object,
                                   C *expected, C desired);

_Bool atomic_compare_exchange_weak_explicit(volatile A *object,
                                            C *expected, C desired,
                                            memory_order success,
                                            memory_order failure);
```

Description

The venerable basis for some many things lock-free: compare and exchange.

In the above prototypes, `A` is the type of the atomic object, and `C` is the equivalent base type.

Ignoring the `_explicit` versions for a moment, what these do is:

- If the value pointed to by `object` is equal to the value pointed to by `expected`, then the value pointed to by `object` is set to `desired`. And the function returns `true` indicating the exchange did take place.
- Else the value pointed to by `expected` (yes, `expected`) is set to `desired` and the function returns `false` indicating the exchange did not take place.

Pseudocode for the exchange would look like this[4]:

```
bool compare_exchange(atomic_A *object, C *expected, C desired)
{
    if (*object is the same as *expected) {
        *object = desired
        return true
    }

    *expected = desired
    return false
}
```

The `_weak` variants might spontaneously fail, so even if `*object == *desired`, it might not change the value and will return `false`. So you'll want that in a loop if you use it[5].

The `_explicit` variants have two memory orders: `success` if `*object` is set to `desired`, and `failure` if it is not.

These are test-and-set functions, so you can use `memory_order_acq_rel` with the `_explicit` variants.

Return Value

Returns `true` if `*object` was `*expected`. Otherwise, `false`.

Example

A contrived example where multiple threads add `2` to a shared value in a lock-free way.

[4] This effectively does the same thing, but it's clearly not atomic.

[5] The spec says, "This spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g. load-locked store-conditional machines." And adds, "When a compare-and-exchange is in a loop, the weak version will yield better performance on some platforms. When a weak compare-and-exchange would require a loop and a strong one would not, the strong one is preferable."

(It would be better to use += 2 to get this done in real life unless you were using some _explicit wizardry.)

```
#include <stdio.h>
#include <threads.h>
#include <stdatomic.h>

#define LOOP_COUNT 10000

atomic_int value;

int run(void *arg)
{
    (void)arg;

    for(int i = 0; i < LOOP_COUNT; i++) {

        int cur = value;
        int next;

        do {
            next = cur + 2;
        } while (!atomic_compare_exchange_strong(&value, &cur, next));
    }

    return 0;
}

int main(void)
{
    thrd_t t1, t2;

    thrd_create(&t1, run, NULL);
    thrd_create(&t2, run, NULL);

    thrd_join(t1, NULL);
    thrd_join(t2, NULL);

    printf("%d should equal %d\n", value, LOOP_COUNT * 4);
}
```

Just replacing this with `value = value + 2` causes data trampling.

See Also

`atomic_load()`, `atomic_load_explicit()`, `atomic_store()`, `atomic_store_explicit()`, `atomic_exchange()`, `atomic_exchange_explicit()`, `atomic_fetch_*()`

57.15 atomic_fetch_*()

Atomically modify atomic variables

Synopsis

```
#include <stdatomic.h>

C atomic_fetch_KEY(volatile A *object, M operand);

C atomic_fetch_KEY_explicit(volatile A *object, M operand,
                            memory_order order);
```

Description

These are actually a group of 10 functions. You substitute one of the following for `KEY` to perform that operation:

- `add`
- `sub`
- `or`
- `xor`
- `and`

So these functions can add or subtract values to or from an atomic variable, or can perform bitwise-OR, XOR, or AND on them.

Use it with integer or pointer types. Though the spec is a little vague on the matter, other types make C unhappy. It goes out of its way to avoid undefined behavior with signed integers, as well:

C18 §7.17.7.5 ¶3:

> For signed integer types, arithmetic is defined to use two's complement representation with silent wrap-around on overflow; there are no undefined results.

In the synopsis, above, `A` is an atomic type, and `M` is the corresponding non-atomic type for `A` (or `ptrdiff_t` for atomic pointers), and `C` is the corresponding non-atomic type for `A`.

For example, here are some operations on an `atomic_int`.

```
atomic_fetch_add(&x, 20);
atomic_fetch_sub(&x, 37);
atomic_fetch_xor(&x, 3490);
```

They are the same as `+=`, `-=`, `|=`, `^=` and `&=`, except the return value is the *previous* value of the atomic object. (With the assignment operators, the value of the expression is that *after* its evaluation.)

```
atomic_int x = 10;
int prev = atomic_fetch_add(&x, 20);
printf("%d %d\n", prev, x);  // 10 30
```

versus:

```
atomic_int x = 10;
int prev = (x += 20);
printf("%d %d\n", prev, x);  // 30 30
```

And, of course, the `_explicit` version allows you to specify a memory order and all the assignment operators are `memory_order_seq_cst`.

Return Value

Returns the previous value of the atomic object before the modification.

Example

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    atomic_int x = 0;
    int prev;

    atomic_fetch_add(&x, 3490);
    atomic_fetch_sub(&x, 12);
    atomic_fetch_xor(&x, 444);
    atomic_fetch_or(&x, 12);
    prev = atomic_fetch_and(&x, 42);

    printf("%d %d\n", prev, x);   // 3118 42
}
```

See Also

`atomic_exchange()`, `atomic_exchange_explicit()`, `atomic_compare_exchange_strong()`, `atomic_compare_exchange_strong_explicit()`, `atomic_compare_exchange_weak()`, `atomic_compare_exchange_weak_explicit()`

57.16 atomic_flag_test_and_set()

Test and set an atomic flag

Synopsis

```
#include <stdatomic.h>

_Bool atomic_flag_test_and_set(volatile atomic_flag *object);

_Bool atomic_flag_test_and_set_explicit(volatile atomic_flag *object,
                                        memory_order order);
```

Description

One of the venerable old functions of lock-free programming, this function sets the given atomic flag in `object`, and returns the previous value of the flag.

As usual, the `_explicit` allows you to specify an alternate memory order.

Return Value

Returns `true` if the flag was set previously, and `false` if it wasn't.

Example

Using test-and-set to implement a spin lock[6]:

```
#include <stdio.h>
#include <threads.h>
#include <stdatomic.h>

// Shared non-atomic struct
struct {
    int x, y, z;
} s = {1, 2, 3};

atomic_flag f = ATOMIC_FLAG_INIT;

int run(void *arg)
{
    int tid = *(int*)arg;

    printf("Thread %d: waiting for lock...\n", tid);

    while (atomic_flag_test_and_set(&f));

    printf("Thread %d: got lock, s is {%d, %d, %d}\n", tid,
                                                        s.x, s.y, s.z);
    s.x = (tid + 1) * 5 + 0;
    s.y = (tid + 1) * 5 + 1;
    s.z = (tid + 1) * 5 + 2;
    printf("Thread %d: set s to {%d, %d, %d}\n", tid, s.x, s.y, s.z);

    printf("Thread %d: releasing lock...\n", tid);
    atomic_flag_clear(&f);

    return 0;
}

int main(void)
{
    thrd_t t1, t2;
    int tid[] = {0, 1};

    thrd_create(&t1, run, tid+0);
    thrd_create(&t2, run, tid+1);

    thrd_join(t1, NULL);
    thrd_join(t2, NULL);
}
```

Example output (varies run to run):

```
Thread 0: waiting for lock...
Thread 0: got lock, s is {1, 2, 3}
Thread 1: waiting for lock...
```

[6] Don't use this unless you know what you're doing—use the thread mutex functionality instead. It'll let your blocked thread sleep and stop chewing up CPU.

```
Thread 0: set s to {5, 6, 7}
Thread 0: releasing lock...
Thread 1: got lock, s is {5, 6, 7}
Thread 1: set s to {10, 11, 12}
Thread 1: releasing lock...
```

See Also

`atomic_flag_clear()`

57.17 atomic_flag_clear()

Clear an atomic flag

Synopsis

```
#include <stdatomic.h>

void atomic_flag_clear(volatile atomic_flag *object);

void atomic_flag_clear_explicit(volatile atomic_flag *object,
                                memory_order order);
```

Description

Clears an atomic flag.

As usual, the `_explicit` allows you to specify an alternate memory order.

Return Value

Returns nothing!

Example

Using test-and-set to implement a spin lock[7]:

```
#include <stdio.h>
#include <threads.h>
#include <stdatomic.h>

// Shared non-atomic struct
struct {
    int x, y, z;
} s = {1, 2, 3};

atomic_flag f = ATOMIC_FLAG_INIT;

int run(void *arg)
```

[7] Don't use this unless you know what you're doing—use the thread mutex functionality instead. It'll let your blocked thread sleep and stop chewing up CPU.

```
{
    int tid = *(int*)arg;

    printf("Thread %d: waiting for lock...\n", tid);

    while (atomic_flag_test_and_set(&f));

    printf("Thread %d: got lock, s is {%d, %d, %d}\n", tid,
                                                        s.x, s.y, s.z);
    s.x = (tid + 1) * 5 + 0;
    s.y = (tid + 1) * 5 + 1;
    s.z = (tid + 1) * 5 + 2;
    printf("Thread %d: set s to {%d, %d, %d}\n", tid, s.x, s.y, s.z);

    printf("Thread %d: releasing lock...\n", tid);
    atomic_flag_clear(&f);

    return 0;
}

int main(void)
{
    thrd_t t1, t2;
    int tid[] = {0, 1};

    thrd_create(&t1, run, tid+0);
    thrd_create(&t2, run, tid+1);

    thrd_join(t1, NULL);
    thrd_join(t2, NULL);
}
```

Example output (varies run to run):

```
Thread 0: waiting for lock...
Thread 0: got lock, s is {1, 2, 3}
Thread 1: waiting for lock...
Thread 0: set s to {5, 6, 7}
Thread 0: releasing lock...
Thread 1: got lock, s is {5, 6, 7}
Thread 1: set s to {10, 11, 12}
Thread 1: releasing lock...
```

See Also

`atomic_flag_test_and_set()`

Chapter 58

<stdbool.h> Boolean Types

This is a small header file that defines a number of convenient Boolean macros. If you really need that kind of thing.

Macro	Description
`bool`	Type for Boolean, expands to `_Bool`
`true`	True value, expands to `1`
`false`	False value, expands to `0`

There's on more macro that I'm not putting in the table because it's such a long name it'll blow up the table alignment:

```
__bool_true_false_are_defined
```

which expands to `1`.

58.1 Example

Here's a lame example that shows off these macros.

```c
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
    bool x;

    x = (3 > 2);

    if (x == true)
        printf("The universe still makes sense.\n");

    x = false;

    printf("x is now %d\n", x);  // 0
}
```

Output:

www.dbooks.org

```
The universe still makes sense.
x is now 0
```

58.2 _Bool?

What's the deal with _Bool? Why didn't they just make it bool?

Well, there was a lot of C code out there where people had defined their own bool type and adding an official bool would have broken those typedefs.

But C has already reserved all identifiers that start with an underscore followed by a capital letter, so it was clear to make up a new _Bool type and go with that.

And, if you know your code can handle it, you can include this header to get all this juicy syntax.

One more note on conversions: unlike converting to int, the *only* thing that converts to false in a _Bool is a scalar zero value. Anything at all that's not zero, like -3490, 0.12, or NaN, converts to true.

Chapter 59

<stddef.h> A Few Standard Definitions

Despite its name, I've haven't seen this frequently included.

It includes several types and macros.

Name	Description
`ptrdiff_t`	Signed integer difference between two pointers
`size_t`	Unsigned integer type returned by `sizeof`
`max_align_t`	Declare a type with the biggest possible alignment
`wchar_t`	Wide character type
`NULL`	`NULL` pointer, as defined a number of places
`offsetof`	Get the byte offsets of `struct` or `union` fields

59.1 ptrdiff_t

This holds the different between two pointers. You could store this in another type, but the result of a pointer subtraction is an implementation-defined type; you can be maximally portable by using `ptrdiff_t`.

```
#include <stdio.h>
#include <stddef.h>

int main(void)
{
    int cats[100];

    int *f = cats + 20;
    int *g = cats + 60;

    ptrdiff_t d = g - f;  // difference is 40
```

And you can print it by prefixing the integer format specifier with `t`:

```
    printf("%td\n", d);  // Print decimal: 40
    printf("%tX\n", d);  // Print hex:     28
}
```


www.dbooks.org

59.2 size_t

This is the type returned by `sizeof` and used in a few other places. It's an unsigned integer.

You can print it using the `z` prefix in `printf()`:

```
#include <stdio.h>
#include <uchar.h>
#include <string.h>
#include <stddef.h>

int main(void)
{
    size_t x;

    x = sizeof(int);

    printf("%zu\n", x);
```

Some functions return negative numbers cast to `size_t` as error values (such as `mbrtoc16()`). If you want to print these as negative values, you can do it with `%zd`:

```
    char16_t a;
    mbstate_t mbs;
    memset(&mbs, 0, sizeof mbs);

    x = mbrtoc16(&a, "b", 8, &mbs);

    printf("%zd\n", x);
}
```

59.3 max_align_t

As far as I can tell, this exists to allow the runtime computation of the maximum fundamental alignment[1] on the current platform. Someone please mail me if there's another use.

Maybe you need this if you're writing your own memory allocator or somesuch.

```
#include <stddef.h>
#include <stdio.h>      // For printf()
#include <stdalign.h>   // For alignof

int main(void)
{
    int max = alignof(max_align_t);

    printf("Maximum fundamental alignment: %d\n", max);
}
```

On my system, this prints:

```
Maximum fundamental alignment: 16
```

See also `alignas`, `alignof`.

[1] https://en.wikipedia.org/wiki/Data_structure_alignment

59.4 wchar_t

This is analogous to `char`, except it's for wide characters.

It's an integer type that has enough range to hold unique values for all characters in all supported locales.

The value `0` is the wide `NUL` character.

Finally, the values of character constants from the basic character set will be the same as their corresponding `wchar_t` values… unless `__STDC_MB_MIGHT_NEQ_WC__` is defined.

59.5 offsetof

If you have a `struct` or `union`, you can use this to get the byte offset of fields within that type.

Usage is:

```
offsetof(type, fieldname);
```

The resulting value has type `size_t`.

Here's an example that prints the field offsets of a `struct`:

```
#include <stdio.h>
#include <stddef.h>

struct foo {
    int a;
    char b;
    char c;
    float d;
};

int main(void)
{
    printf("a: %zu\n", offsetof(struct foo, a));
    printf("b: %zu\n", offsetof(struct foo, b));
    printf("c: %zu\n", offsetof(struct foo, c));
    printf("d: %zu\n", offsetof(struct foo, d));
}
```

On my system, this outputs:

```
a: 0
b: 4
c: 5
d: 8
```

And you can't use `offsetof` on a bitfield, so don't get your hopes up.

Chapter 60

<stdint.h> More Integer Types

This header gives us access to (potentially) types of a fixed number of bits, or, at the very least, types that are at least that many bits.

It also gives us handy macros to use.

60.1 Specific-Width Integers

There are three main classes of types defined here, signed and unsigned:

- Integers of exactly a certain size (`int`*N*`_t`, `uint`*N*`_t`)
- Integers that are at least a certain size (`int_least`*N*`_t`, `uint_least`*N*`_t`)
- Integers that are at least a certain size and are as fast as possible (`int_fast`*N*`_t`, `uint_fast`*N*`_t`)

Where the *N* occurs, you substitute the number of bits, commonly multiples of 8, e.g. `uint16_t`.

The following types are guaranteed to be defined:

```
int_least8_t      uint_least8_t
int_least16_t     uint_least16_t
int_least32_t     uint_least32_t
int_least64_t     uint_least64_t

int_fast8_t       uint_fast8_t
int_fast16_t      uint_fast16_t
int_fast32_t      uint_fast32_t
int_fast64_t      uint_fast64_t
```

Everything else is optional, but you'll probably also have the following, which are required when a system has integers of these sizes with no padding and two's-complement representation… which is the case for Macs and PCs and a lot of other systems. In short, you very likely have these:

```
int8_t      uint8_t
int16_t     uint16_t
int32_t     uint32_t
int64_t     uint64_t
```

Other numbers of bits can also be supported by an implementation if it wants to go all crazy with it.

Examples:

```
#include <stdint.h>

int main(void)
{
    int16_t x = 32;
    int_fast32_t y = 3490;

    // ...
```

60.2 Other Integer Types

There are a couple optional types that are integers capable of holding pointer types.

```
intptr_t
uintptr_t
```

You can convert a `void*` to one of these types, and back again. And the `void*`s will compare equal.

The use case is any place you need an integer that represents a pointer for some reason.

Also, there are a couple types that are just there to be the biggest possible integers your system supports:

```
intmax_t
uintmax_t
```

Fun fact: you can print these types with the `"%jd"` and `"%ju"` `printf()` format specifiers.

There are also a bunch of macros in `<inttypes.h>`(#inttypes) that you can use to print any of the types mentioned, above.

60.3 Macros

The following macros define the minimum and maximum values for these types:

```
INT8_MAX            INT8_MIN            UINT8_MAX
INT16_MAX           INT16_MIN           UINT16_MAX
INT32_MAX           INT32_MIN           UINT32_MAX
INT64_MAX           INT64_MIN           UINT64_MAX

INT_LEAST8_MAX      INT_LEAST8_MIN      UINT_LEAST8_MAX
INT_LEAST16_MAX     INT_LEAST16_MIN     UINT_LEAST16_MAX
INT_LEAST32_MAX     INT_LEAST32_MIN     UINT_LEAST32_MAX
INT_LEAST64_MAX     INT_LEAST64_MIN     UINT_LEAST64_MAX

INT_FAST8_MAX       INT_FAST8_MIN       UINT_FAST8_MAX
INT_FAST16_MAX      INT_FAST16_MIN      UINT_FAST16_MAX
INT_FAST32_MAX      INT_FAST32_MIN      UINT_FAST32_MAX
INT_FAST64_MAX      INT_FAST64_MIN      UINT_FAST64_MAX

INTMAX_MAX          INTMAX_MIN          UINTMAX_MAX

INTPTR_MAX          INTPTR_MIN          UINTPTR_MAX
```

For the exact-bit-size signed types, the minimum is exactly $-(2^{N-1})$ and the maximum is exactly $2^{N-1}-1$. And for the exact-bit-size unsigned types, the max is exactly $2^N - 1$.

For the signed "least" and "fast" variants, the magnitude and sign of the minimum is at least $-(2^{N-1}-1)$ and the maximum is at least $2^{N-1}-1$. And for unsigned it's at least 2^N-1.

`INTMAX_MAX` is at least $2^{63}-1$, `INTMAX_MIN` is at least $-(2^{63}-1)$ in sign and magnitude. And `UINTMAX_MAX` is at least $2^{64}-1$.

Finally, `INTPTR_MAX` is at least $2^{15}-1$, `INTPTR_MIN` is at least $-(2^{15}-1)$ in sign and magnitude. And `UINTPTR_MAX` is at least $2^{16}-1$.

60.4 Other Limits

There are a bunch of types in `<inttypes.h>`(#inttypes) that have their limits defined here. (`<inttypes.h>` includes `<stdint.h>`.)

Macro	Description
`PTRDIFF_MIN`	Minimum `ptrdiff_t` value
`PTRDIFF_MAX`	Maximum `ptrdiff_t` value
`SIG_ATOMIC_MIN`	Minimum `sig_atomic_t` value
`SIG_ATOMIC_MAX`	Maximum `sig_atomic_t` value
`SIZE_MAX`	Maximum `size_t` value
`WCHAR_MIN`	Minimum `wchar_t` value
`WCHAR_MAX`	Maximum `wchar_t` value
`WINT_MIN`	Minimum `wint_t` value
`WINT_MAX`	Maximum `wint_t` value

The spec says that `PTRDIFF_MIN` will be at least -65535 in magnitude. And `PTRDIFF_MAX` and `SIZE_MAX` will be at least 65535.

`SIG_ATOMIC_MIN` and `MAX` will be either -127 and 127 (if it's signed) or 0 and 255 (if it's unsigned).

Same for `WCHAR_MIN` and `MAX`.

`WINT_MIN` and `MAX` will be either -32767 and 32767 (if it's signed) or 0 and 65535 (if it's unsigned).

60.5 Macros for Declaring Constants

If you recall, you can specify a type for integer constants:

```
int x = 12;
long int y = 12L;
unsigned long long int z = 12ULL;
```

You can use the macros `INT`N`_C()` and `UINT`N`()` where N is `8`, `16`, `32` or `64`.

```
uint_least16_t x = INT16_C(3490);
uint_least64_t y = INT64_C(1122334455);
```

A variant on these is `INTMAX_C()` and `UINTMAX_C()`. They will make a constant suitable for storing in an `intmax_t` or `uintmax_t`.

```
intmax_t x = INTMAX_C(3490);
uintmax_t x = UINTMAX_C(1122334455);
```

Chapter 61

<stdio.h> Standard I/O Library

Function	Description
`clearerr()`	Clear the `feof` and `ferror` status flags
`fclose()`	Close an open file
`feof()`	Return the file end-of-file status
`ferror()`	Return the file error status
`fflush()`	Flush all buffered output to a file
`fgetc()`	Read a character in a file
`fgetpos()`	Get the file I/O position
`fgets()`	Read a line from a file
`fopen()`	Open a file
`fprintf()`	Print formatted output to a file
`fputc()`	Print a character to a file
`fputs()`	Print a string to a file
`fread()`	Read binary data from a file
`freopen()`	Change file associated with a stream
`fscanf()`	Read formatted input from a file
`fseek()`	Set the file I/O position
`fsetpos()`	Set the file I/O position
`ftell()`	Get the file I/O position
`fwrite()`	Write binary data to a file
`getc()`	Get a character from `stdin`
`getchar()`	Get a character from `stdin`
`gets()`	Get a string from `stdin` (removed in C11)
`perror()`	Print a human-formatted error message
`printf()`	Print formatted output to `stdout`
`putc()`	Print a character to `stdout`
`putchar()`	Print a character to `stdout`
`puts()`	Print a string to `stdout`
`remove()`	Delete a file from disk
`rename()`	Rename or move a file on disk
`rewind()`	Set the I/O position to the beginning of a file
`scanf()`	Read formatted input from `stdin`
`setbuf()`	Configure buffering for I/O operations
`setvbuf()`	Configure buffering for I/O operations
`snprintf()`	Print length-limited formatted output to a string
`sprintf()`	Print formatted output to a string

www.dbooks.org

Function	Description
`sscanf()`	Read formatted input from a string
`tmpfile()`	Create a temporary file
`tmpnam()`	Generate a unique name for a temporary file
`ungetc()`	Push a character back on the input stream
`vfprintf()`	Variadic print formatted output to a file
`vfscanf()`	Variadic read formatted input from a file
`vprintf()`	Variadic print formatted output to `stdout`
`vscanf()`	Variadic read formatted input from `stdin`
`vsnprintf()`	Variadic length-limited print formatted output to a string
`vsprintf()`	Variadic print formatted output to a string
`vsscanf()`	Variadic read formatted input to a string

The most basic of all libraries in the whole of the standard C library is the standard I/O library. It's used for reading from and writing to files. I can see you're very excited about this.

So I'll continue. It's also used for reading and writing to the console, as we've already often seen with the `printf()` function.

(A little secret here—many many things in various operating systems are secretly files deep down, and the console is no exception. "*Everything in Unix is a file!*" `:-)`)

You'll probably want some prototypes of the functions you can use, right? To get your grubby little mittens on those, you'll want to include `stdio.h`.

Anyway, so we can do all kinds of cool stuff in terms of file I/O. LIE DETECTED. Ok, ok. We can do all kinds of stuff in terms of file I/O. Basically, the strategy is this:

1. Use `fopen()` to get a pointer to a file structure of type `FILE*`. This pointer is what you'll be passing to many of the other file I/O calls.
2. Use some of the other file calls, like `fscanf()`, `fgets()`, `fprintf()`, or etc. using the `FILE*` returned from `fopen()`.
3. When done, call `fclose()` with the `FILE*`. This let's the operating system know that you're truly done with the file, no take-backs.

What's in the `FILE*`? Well, as you might guess, it points to a `struct` that contains all kinds of information about the current read and write position in the file, how the file was opened, and other stuff like that. But, honestly, who cares. No one, that's who. The `FILE` structure is *opaque* to you as a programmer; that is, you don't need to know what's in it, and you don't even *want* to know what's in it. You just pass it to the other standard I/O functions and they know what to do.

This is actually pretty important: try to not muck around in the `FILE` structure. It's not even the same from system to system, and you'll end up writing some really non-portable code.

One more thing to mention about the standard I/O library: a lot of the functions that operate on files use an "f" prefix on the function name. The same function that is operating on the console will leave the "f" off. For instance, if you want to print to the console, you use `printf()`, but if you want to print to a file, use `fprintf()`, see?

Wait a moment! If writing to the console is, deep down, just like writing to a file, since everything in Unix is a file, why are there two functions? Answer: it's more convenient. But, more importantly, is there a `FILE*` associated with the console that you can use? Answer: YES!

There are, in fact, *three* (count 'em!) special `FILE*`s you have at your disposal merely for just including `stdio.h`. There is one for input, and two for output.

That hardly seems fair—why does output get two files, and input only get one?

That's jumping the gun a bit—let's just look at them:

Stream	Description
stdin	Input from the console.
stdout	Output to the console.
stderr	Output to the console on the error file stream.

So standard input (stdin) is by default just what you type at the keyboard. You can use that in fscanf() if you want, just like this:

```
/* this line: */
scanf("%d", &x);

/* is just like this line: */
fscanf(stdin, "%d", &x);
```

And stdout works the same way:

```
printf("Hello, world!\n");
fprintf(stdout, "Hello, world!\n"); /* same as previous line! */
```

So what is this stderr thing? What happens when you output to that? Well, generally it goes to the console just like stdout, but people use it for error messages, specifically. Why? On many systems you can redirect the output from the program into a file from the command line…and sometimes you're interested in getting just the error output. So if the program is good and writes all its errors to stderr, a user can redirect just stderr into a file, and just see that. It's just a nice thing you, as a programmer, can do.

Finally, a lot of these functions return int where you might expect char. This is because the function can return a character *or* end-of-file (EOF), and EOF is potentially an integer. If you don't get EOF as a return value, you can safely store the result in a char.

61.1 remove()

Delete a file

Synopsis

```
#include <stdio.h>

int remove(const char *filename);
```

Description

Removes the specified file from the filesystem. It just deletes it. Nothing magical. Simply call this function and sacrifice a small chicken and the requested file will be deleted.

Return Value

Returns zero on success, and -1 on error, setting errno.

Example

```
#include <stdio.h>

int main(void)
{
    char *filename = "evidence.txt";

    remove(filename);
}
```

See Also

`rename()`

61.2 `rename()`

Renames a file and optionally moves it to a new location

Synopsis

```
#include <stdio.h>

int rename(const char *old, const char *new);
```

Description

Renames the file `old` to name `new`. Use this function if you're tired of the old name of the file, and you are ready for a change. Sometimes simply renaming your files makes them feel new again, and could save you money over just getting all new files!

One other cool thing you can do with this function is actually move a file from one directory to another by specifying a different path for the new name.

Return Value

Returns zero on success, and `-1` on error, setting `errno`.

Example

```
#include <stdio.h>

int main(void)
{
    // Rename a file
    rename("foo", "bar");

    // Rename and move to another directory:
    rename("/home/beej/evidence.txt", "/tmp/nothing.txt");
}
```

See Also

`remove()`

61.3 tmpfile()

Create a temporary file

Synopsis

```
#include <stdio.h>

FILE *tmpfile(void);
```

Description

This is a nifty little function that will create and open a temporary file for you, and will return a `FILE*` to it that you can use. The file is opened with mode "`r+b`", so it's suitable for reading, writing, and binary data.

By using a little magic, the temp file is automatically deleted when it is `close()`'d or when your program exits. (Specifically, in Unix terms, `tmpfile()` *unlinks*[1] the file right after it opens it. This means that it's primed to be deleted from disk, but still exists because your process still has it open. As soon as your process exits, all open files are closed, and the temp file vanishes into the ether.)

Return Value

This function returns an open `FILE*` on success, or `NULL` on failure.

Example

```
#include <stdio.h>

int main(void)
{
    FILE *temp;
    char s[128];

    temp = tmpfile();

    fprintf(temp, "What is the frequency, Alexander?\n");

    rewind(temp); // back to the beginning

    fscanf(temp, "%s", s); // read it back out

    fclose(temp); // close (and magically delete)
}
```

[1] https://man.archlinux.org/man/unlinkat.2.en#DESCRIPTION

See Also

`fopen()`, `fclose()`, `tmpnam()`

61.4 `tmpnam()`

Generate a unique name for a temporary file

Synopsis

```
#include <stdio.h>

char *tmpnam(char *s);
```

Description

This function takes a good hard look at the existing files on your system, and comes up with a unique name for a new file that is suitable for temporary file usage.

Let's say you have a program that needs to store off some data for a short time so you create a temporary file for the data, to be deleted when the program is done running. Now imagine that you called this file `foo.txt`. This is all well and good, except what if a user already has a file called `foo.txt` in the directory that you ran your program from? You'd overwrite their file, and they'd be unhappy and stalk you forever. And you wouldn't want that, now would you?

Ok, so you get wise, and you decide to put the file in `/tmp` so that it won't overwrite any important content. But wait! What if some other user is running your program at the same time and they both want to use that filename? Or what if some other program has already created that file?

See, all of these scary problems can be completely avoided if you just use `tmpnam()` to get a safe-ready-to-use filename.

So how do you use it? There are two amazing ways. One, you can declare an array (or `malloc()` it—whatever) that is big enough to hold the temporary file name. How big is that? Fortunately there has been a macro defined for you, `L_tmpnam`, which is how big the array must be.

And the second way: just pass `NULL` for the filename. `tmpnam()` will store the temporary name in a static array and return a pointer to that. Subsequent calls with a `NULL` argument will overwrite the static array, so be sure you're done using it before you call `tmpnam()` again.

Again, this function just makes a file name for you. It's up to you to later `fopen()` the file and use it.

One more note: some compilers warn against using `tmpnam()` since some systems have better functions (like the Unix function `mkstemp()`.) You might want to check your local documentation to see if there's a better option. Linux documentation goes so far as to say, "Never use this function. Use `mkstemp()` instead."

I, however, am going to be a jerk and not talk about `mkstemp()`[2] because it's not in the standard I'm writing about. Nyaah.

The macro `TMP_MAX` holds the number of unique filenames that can be generated by `tmpnam()`. Ironically, it is the *minimum* number of such filenames.

[2] https://man.archlinux.org/man/mkstemp.3.en

Return Value

Returns a pointer to the temporary file name. This is either a pointer to the string you passed in, or a pointer to internal static storage if you passed in `NULL`. On error (like it can't find any temporary name that is unique), `tmpnam()` returns `NULL`.

Example

```c
#include <stdio.h>

int main(void)
{
    char filename[L_tmpnam];
    char *another_filename;

    if (tmpnam(filename) != NULL)
        printf("We got a temp file name: \"%s\"\n", filename);
    else
        printf("Something went wrong, and we got nothing!\n");

    another_filename = tmpnam(NULL);

    printf("We got another temp file name: \"%s\"\n", another_filename);
    printf("And we didn't error check it because we're too lazy!\n");
}
```

On my Linux system, this generates the following output:

```
We got a temp file name: "/tmp/filew9PMuZ"
We got another temp file name: "/tmp/fileOwrgPO"
And we didn't error check it because we're too lazy!
```

See Also

`fopen()`, `tmpfile()`

61.5 `fclose()`

The opposite of `fopen()`—closes a file when you're done with it so that it frees system resources

Synopsis

```c
#include <stdio.h>

int fclose(FILE *stream);
```

Description

When you open a file, the system sets aside some resources to maintain information about that open file. Usually it can only open so many files at once. In any case, the Right Thing to do is to close your files when you're done using them so that the system resources are freed.

Also, you might not find that all the information that you've written to the file has actually been written to disk until the file is closed. (You can force this with a call to `fflush()`.)

When your program exits normally, it closes all open files for you. Lots of times, though, you'll have a long-running program, and it'd be better to close the files before then. In any case, not closing a file you've opened makes you look bad. So, remember to `fclose()` your file when you're done with it!

Return Value

On success, `0` is returned. Typically no one checks for this. On error `EOF` is returned. Typically no one checks for this, either.

Example

```
#include <stdio.h>

int main(void)
{
    FILE *fp;

    fp = fopen("spoon.txt", "r");

    if (fp == NULL) {
        printf("Error opening file\n");
    } else {
        printf("Opened file just fine!\n");
        fclose(fp);  // All done!
    }
}
```

See Also

`fopen()`

61.6 `fflush()`

Process all buffered I/O for a stream right now

Synopsis

```
#include <stdio.h>

int fflush(FILE *stream);
```

Description

When you do standard I/O, as mentioned in the section on the `setvbuf()` function, it is usually stored in a buffer until a line has been entered or the buffer is full or the file is closed. Sometimes, though, you really want the output to happen *right this second*, and not wait around in the buffer. You can force this to happen by calling `fflush()`.

The advantage to buffering is that the OS doesn't need to hit the disk every time you call `fprintf()`. The disadvantage is that if you look at the file on the disk after the `fprintf()` call, it might not have actually been written to yet. ("I called `fputs()`, but the file is still zero bytes long! Why?!") In virtually all circumstances, the advantages of buffering outweigh the disadvantages; for those other circumstances, however, use `fflush()`.

Note that `fflush()` is only designed to work on output streams according to the spec. What will happen if you try it on an input stream? Use your spooky voice: *who knoooooows!*

Return Value

On success, `fflush()` returns zero. If there's an error, it returns `EOF` and sets the error condition for the stream (see `ferror()`.)

Example

In this example, we're going to use the carriage return, which is `'\r'`. This is like newline (`'\n'`), except that it doesn't move to the next line. It just returns to the front of the current line.

What we're going to do is a little text-based status bar like so many command line programs implement. It'll do a countdown from 10 to 0 printing over itself on the same line.

What is the catch and what does this have to do with `fflush()`? The catch is that the terminal is most likely "line buffered" (see the section on `setvbuf()` for more info), meaning that it won't actually display anything until it prints a newline. But we're not printing newlines; we're just printing carriage returns, so we need a way to force the output to occur even though we're on the same line. Yes, it's `fflush()`!

```c
#include <stdio.h>
#include <threads.h>

void sleep_seconds(int s)
{
    thrd_sleep(&(struct timespec){.tv_sec=s}, NULL);
}

int main(void)
{
    int count;

    for(count = 10; count >= 0; count--) {
        printf("\rSeconds until launch: ");  // lead with a CR
        if (count > 0)
            printf("%2d", count);
        else
            printf("blastoff!\n");

        // force output now!!
        fflush(stdout);

        sleep_seconds(1);
    }
}
```

See Also

`setbuf()`, `setvbuf()`

61.7 fopen()

Opens a file for reading or writing

Synopsis

```
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);
```

Description

The `fopen()` opens a file for reading or writing.

Parameter `path` can be a relative or fully-qualified path and file name to the file in question.

Parameter `mode` tells `fopen()` how to open the file (reading, writing, or both), and whether or not it's a binary file. Possible modes are:

Mode	Description
`r`	Open the file for reading (read-only).
`w`	Open the file for writing (write-only). The file is created if it doesn't exist.
`r+`	Open the file for reading and writing. The file has to already exist.
`w+`	Open the file for writing and reading. The file is created if it doesn't already exist.
`a`	Open the file for append. This is just like opening a file for writing, but it positions the file pointer at the end of the file, so the next write appends to the end. The file is created if it doesn't exist.
`a+`	Open the file for reading and appending. The file is created if it doesn't exist.

Any of the modes can have the letter "`b`" appended to the end, as is "`wb`" ("write binary"), to signify that the file in question is a *binary* file. ("Binary" in this case generally means that the file contains non-alphanumeric characters that look like garbage to human eyes.) Many systems (like Unix) don't differentiate between binary and non-binary files, so the "`b`" is extraneous. But if your data is binary, it doesn't hurt to throw the "`b`" in there, and it might help someone who is trying to port your code to another system.

The macro `FOPEN_MAX` tells you how many streams (at least) you can have open at once.

The macro `FILENAME_MAX` tells you what the longest valid filename can be. Don't go crazy, now.

Return Value

`fopen()` returns a `FILE*` that can be used in subsequent file-related calls.

If something goes wrong (e.g. you tried to open a file for read that didn't exist), `fopen()` will return `NULL`.

Example

```
#include <stdio.h>

int main(void)
{
    FILE *fp;

    fp = fopen("spoon.txt", "r");

    if (fp == NULL) {
        printf("Error opening file\n");
    } else {
        printf("Opened file just fine!\n");
        fclose(fp);  // All done!
    }
}
```

See Also

`fclose()`, `freopen()`

61.8 `freopen()`

Reopen an existing `FILE*`, associating it with a new path

Synopsis

```
#include <stdio.h>

FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

Description

Let's say you have an existing `FILE*` stream that's already open, but you want it to suddenly use a different file than the one it's using. You can use `freopen()` to "re-open" the stream with a new file.

Why on Earth would you ever want to do that? Well, the most common reason would be if you had a program that normally would read from `stdin`, but instead you wanted it to read from a file. Instead of changing all your `scanf()`s to `fscanf()`s, you could simply reopen `stdin` on the file you wanted to read from.

Another usage that is allowed on some systems is that you can pass `NULL` for `filename`, and specify a new `mode` for `stream`. So you could change a file from "`r+`" (read and write) to just "`r`" (read), for instance. It's implementation dependent which modes can be changed.

When you call `freopen()`, the old `stream` is closed. Otherwise, the function behaves just like the standard `fopen()`.

Return Value

`freopen()` returns `stream` if all goes well.

If something goes wrong (e.g. you tried to open a file for read that didn't exist), `freopen()` will return `NULL`.

Example

```
#include <stdio.h>

int main(void)
{
    int i, i2;

    scanf("%d", &i); // read i from stdin

    // now change stdin to refer to a file instead of the keyboard
    freopen("someints.txt", "r", stdin);

    scanf("%d", &i2); // now this reads from the file "someints.txt"

    printf("Hello, world!\n"); // print to the screen

    // change stdout to go to a file instead of the terminal:
    freopen("output.txt", "w", stdout);

    printf("This goes to the file \"output.txt\"\n");

    // this is allowed on some systems--you can change the mode of a file:
    freopen(NULL, "wb", stdout); // change to "wb" instead of "w"
}
```

See Also

`fclose()`, `fopen()`

61.9 `setbuf()`, `setvbuf()`

Configure buffering for standard I/O operations

Synopsis

```
#include <stdio.h>

void setbuf(FILE *stream, char *buf);

int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Description

Now brace yourself because this might come as a bit of a surprise to you: when you `printf()` or `fprintf()` or use any I/O functions like that, *it does not normally work immediately*. For the sake of efficiency, and to irritate you, the I/O on a `FILE*` stream is buffered away safely until certain conditions are met, and only then is the actual I/O performed. The functions `setbuf()` and `setvbuf()` allow you to change those conditions and the buffering behavior.

So what are the different buffering behaviors? The biggest is called "full buffering", wherein all I/O is stored in a big buffer until it is full, and then it is dumped out to disk (or whatever the file is). The next biggest is

called "line buffering"; with line buffering, I/O is stored up a line at a time (until a newline (`'\n'`) character is encountered) and then that line is processed. Finally, we have "unbuffered", which means I/O is processed immediately with every standard I/O call.

You might have seen and wondered why you could call `putchar()` time and time again and not see any output until you called `putchar('\n')`; that's right—`stdout` is line-buffered!

Since `setbuf()` is just a simplified version of `setvbuf()`, we'll talk about `setvbuf()` first.

The `stream` is the `FILE*` you wish to modify. The standard says you *must* make your call to `setvbuf()` *before* any I/O operation is performed on the stream, or else by then it might be too late.

The next argument, `buf` allows you to make your own buffer space (using `malloc()` or just a `char` array) to use for buffering. If you don't care to do this, just set `buf` to `NULL`.

Now we get to the real meat of the function: `mode` allows you to choose what kind of buffering you want to use on this `stream`. Set it to one of the following:

Mode	Description
`_IOFBF`	`stream` will be fully buffered.
`_IOLBF`	`stream` will be line buffered.
`_IONBF`	`stream` will be unbuffered.

Finally, the `size` argument is the size of the array you passed in for `buf`…unless you passed `NULL` for `buf`, in which case it will resize the existing buffer to the size you specify.

Now what about this lesser function `setbuf()`? It's just like calling `setvbuf()` with some specific parameters, except `setbuf()` doesn't return a value. The following example shows the equivalency:

```
// these are the same:
setbuf(stream, buf);
setvbuf(stream, buf, _IOFBF, BUFSIZ); // fully buffered

// and these are the same:
setbuf(stream, NULL);
setvbuf(stream, NULL, _IONBF, BUFSIZ); // unbuffered
```

Return Value

`setvbuf()` returns zero on success, and nonzero on failure. `setbuf()` has no return value.

Example

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    char lineBuf[1024];

    fp = fopen("somefile.txt", "w");
    setvbuf(fp, lineBuf, _IOLBF, 1024);  // set to line buffering
    fprintf(fp, "You won't see this in the file yet. ");
    fprintf(fp, "But now you will because of this newline.\n");
    fclose(fp);
```

```

    fp = fopen("anotherfile.txt", "w");
    setbuf(fp, NULL); // set to unbuffered
    fprintf(fp, "You will see this in the file now.");
    fclose(fp);
}
```

See Also

`fflush()`

61.10 `printf()`, `fprintf()`, `sprintf()`, `snprintf()`

Print a formatted string to the console or to a file

Synopsis

```
#include <stdio.h>

int printf(const char *format, ...);

int fprintf(FILE *stream, const char *format, ...);

int sprintf(char * restrict s, const char * restrict format, ...);

int snprintf(char * restrict s, size_t n, const char * restrict format, ...);
```

Description

These functions print formatted output to a variety of destinations.

Function	Output Destination
`printf()`	Print to console (screen by default, typically).
`fprintf()`	Print to a file.
`sprintf()`	Print to a string.
`snprintf()`	Print to a string (safely).

The only differences between these is are the leading parameters that you pass to them before the `format` string.

Function	What you pass before `format`
`printf()`	Nothing comes before `format`.
`fprintf()`	Pass a `FILE*`.
`sprintf()`	Pass a `char*` to a buffer to print into.
`snprintf()`	Pass a `char*` to the buffer and a maximum buffer length.

The `printf()` function is legendary as being one of the most flexible outputting systems ever devised. It can also get a bit freaky here or there, most notably in the `format` string. We'll take it a step at a time here.

The easiest way to look at the format string is that it will print everything in the string as-is, *unless* a character has a percent sign (%) in front of it. That's when the magic happens: the next argument in the `printf()` argument list is printed in the way described by the percent code. These percent codes are called *format specifiers*.

Here are the most common format specifiers.

Specifier	Description
`%d`	Print the next argument as a signed decimal number, like `3490`. The argument printed this way should be an `int`, or something that gets promoted to `int`.
`%f`	Print the next argument as a signed floating point number, like `3.14159`. The argument printed this way should be a `double`, or something that gets promoted to a `double`.
`%c`	Print the next argument as a character, like `'B'`. The argument printed this way should be a `char` variant.
`%s`	Print the next argument as a string, like `"Did you remember your mittens?"`. The argument printed this way should be a `char*` or `char[]`.
`%%`	No arguments are converted, and a plain old run-of-the-mill percent sign is printed. This is how you print a '%' using `printf()`.

So those are the basics. I'll give you some more of the format specifiers in a bit, but let's get some more breadth before then. There's actually a lot more that you can specify in there after the percent sign.

For one thing, you can put a field width in there—this is a number that tells `printf()` how many spaces to put on one side or the other of the value you're printing. That helps you line things up in nice columns. If the number is negative, the result becomes left-justified instead of right-justified. Example:

```
printf("%10d", x);  /* prints X on the right side of the 10-space field */
printf("%-10d", x); /* prints X on the left side of the 10-space field */
```

If you don't know the field width in advance, you can use a little kung-foo to get it from the argument list just before the argument itself. Do this by placing your seat and tray tables in the fully upright position. The seatbelt is fastened by placing the—cough. I seem to have been doing way too much flying lately. Ignoring that useless fact completely, you can specify a dynamic field width by putting a `*` in for the width. If you are not willing or able to perform this task, please notify a flight attendant and we will reseat you.

```
int width = 12;
int value = 3490;

printf("%*d\n", width, value);
```

You can also put a "0" in front of the number if you want it to be padded with zeros:

```
int x = 17;
printf("%05d", x);  /* "00017" */
```

When it comes to floating point, you can also specify how many decimal places to print by making a field width of the form "`x.y`" where `x` is the field width (you can leave this off if you want it to be just wide enough) and `y` is the number of digits past the decimal point to print:

```
float f = 3.1415926535;

printf("%.2f", f);  /* "3.14" */
printf("%7.3f", f); /* "  3.141" <-- 7 spaces across */
```

Ok, those above are definitely the most common uses of `printf()`, but let's get *total coverage*.

61.10.0.1 Format Specifier Layout

Technically, the layout of the format specifier is these things in this order:

1. `%`, followed by…
2. Optional: zero or more flags, left justify, leading zeros, etc.
3. Optional: Field width, how wide the output field should be.
4. Optional: Precision, or how many decimal places to print.
5. Optional: Length modifier, for printing things bigger than `int` or `double`.
6. Conversion specifier, like `d`, `f`, etc.

In short, the whole format specifier is laid out like this:

`%[flags][fieldwidth][.precision][lengthmodifier]conversionspecifier`

What could be easier?

61.10.0.2 Conversion Specifiers

Let's talk conversion specifiers first. Each of the following specifies what type it can print, but it can also print anything that gets promoted to that type. For example, `%d` can print `int`, `short`, and `char`.

Conversion Specifier	Description
`d`	Print an `int` argument as a decimal number.
`i`	Identical to `d`.
`o`	Print an `unsigned int` in octal (base 8).
`u`	Print an `unsigned int` in decimal.
`x`	Print an `unsigned int` in hexadecimal with lowercase letters.
`X`	Print an `unsigned int` in hexadecimal with uppercase letters.
`f`	Print a `double` in decimal notation. Infinity is printed as `infinity` or `inf`, and NaN is printed as `nan`, any of which could have a leading minus sign.
`F`	Same as `f`, except it prints out `INFINITY`, `INF`, or `NAN` in all caps.
`e`	Print a number in scientific notation, e.g. `1.234e56`. Does infinity and NaN like `f`.
`E`	Just like `e`, except prints the exponent `E` (and infinity and NaN) in uppercase.
`g`	Print small numbers like `f` and large numbers like `e`. See note below.
`G`	Print small numbers like `F` and large numbers like `E`. See note below.
`a`	Print a `double` in hexadecimal form `0xh.hhhhpd` where `h` is a lowercase hex digit and `d` is a decimal exponent of 2. Infinity and NaN in the form of `f`. More below.
`A`	Like `a` except everything's uppercase.
`c`	Convert `int` argument to `unsigned char` and print as a character.
`s`	Print a string starting at the given `char*`.
`p`	Print a `void*` out as a number, probably the numeric address, possibly in hex.
`n`	Store the number of characters written so far in the given `int*`. Doesn't print anything. See below.
`%`	Print a literal percent sign.

61.10.0.2.1 Note on `%a` and `%A`

When printing floating point numbers in hex form, there is one number before the decimal point, and the rest of are out to the precision.

```
double pi = 3.14159265358979;

printf("%.3a\n", pi);  // 0x1.922p+1
```

C can choose the leading number in such a way to ensure subsequent digits align to 4-bit boundaries.

If the precision is left out and the macro `FLT_RADIX` is a power of 2, enough precision is used to represent the number exactly. If `FLT_RADIX` is not a power of two, enough precision is used to be able to tell any two floating values apart.

If the precision is `0` and the # flag isn't specified, the decimal point is omitted.

61.10.0.2.2 Note on `%g` and `%G` The gist of this is to use scientific notation when the number gets too "extreme", and regular decimal notation otherwise.

The exact behavior for whether these print as `%f` or `%e` depends on a number of factors:

If the number's exponent is greater than or equal to -4 **and** the precision is greater than the exponent, we use `%f`. In this case, the precision is converted according to $p = p - (x + 1)$, where p is the specified precision and x is the exponent.

Otherwise we use `%e`, and the precision becomes $p - 1$.

Trailing zeros in the decimal portion are removed. And if there are none left, the decimal point is removed, too. All this unless the # flag is specified.

61.10.0.2.3 Note on `%n` This specifier is cool and different, and rarely needed. It doesn't actually print anything, but stores the number of characters printed so far in the next pointer argument in the list.

```
int numChars;
float a = 3.14159;
int b = 3490;

printf("%f %d%n\n", a, b, &numChars);
printf("The above line contains %d characters.\n", numChars);
```

The above example will print out the values of `a` and `b`, and then store the number of characters printed so far into the variable `numChars`. The next call to `printf()` prints out that result.

```
3.141590 3490
The above line contains 13 characters
```

61.10.0.3 Length Modifiers

You can stick a *length* modifier in front of each of the conversion specifiers, if you want. most of those format specifiers work on `int` or `double` types, but what if you want larger or smaller types? That's what these are good for.

For example, you could print out a long long int with the `ll` modifier:

```
long long int x = 3490;

printf("%lld\n", x);  // 3490
```

Length Modifier	Conversion Specifier	Description
`hh`	`d`, `i`, `o`, `u`, `x`, `X`	Convert argument to `char` (signed or unsigned as appropriate) before printing.
`h`	`d`, `i`, `o`, `u`, `x`, `X`	Convert argument to `short int` (signed or unsigned as appropriate) before printing.
`l`	`d`, `i`, `o`, `u`, `x`, `X`	Argument is a `long int` (signed or unsigned as appropriate).
`ll`	`d`, `i`, `o`, `u`, `x`, `X`	Argument is a `long long int` (signed or unsigned as appropriate).
`j`	`d`, `i`, `o`, `u`, `x`, `X`	Argument is a `intmax_t` or `uintmax_t` (as appropriate).

www.dbooks.org

Length Modifier	Conversion Specifier	Description
z	d, i, o, u, x, X	Argument is a size_t.
t	d, i, o, u, x, X	Argument is a ptrdiff_t.
L	a, A, e, E, f, F, g, G	Argument is a long double.
l	c	Argument is in a wint_t, a wide character.
l	s	Argument is in a wchar_t*, a wide character string.
hh	n	Store result in signed char* argument.
h	n	Store result in short int* argument.
l	n	Store result in long int* argument.
ll	n	Store result in long long int* argument.
j	n	Store result in intmax_t* argument.
z	n	Store result in size_t* argument.
t	n	Store result in ptrdiff_t* argument.

61.10.0.4 Precision

In front of the length modifier, you can put a precision, which generally means how many decimal places you want on your floating point numbers.

To do this, you put a decimal point (.) and the decimal places afterward.

For example, we could print π rounded to two decimal places like this:

```
double pi = 3.14159265358979;

printf("%.2f\n", pi);  // 3.14
```

Conversion Specifier	Precision Value Meaning
d, i, o, u, x, X	For integer types, minimum number of digits (will pad with leading zeros)
a, e, f, A, E, F	For floating types, the precision is the number of digits past the decimal.
g, G	For floating types, the precision is the number of significant digits printed.
s	The maximum number of bytes (not multibyte characters!) to be written.

If no number is specified in the precision after the decimal point, the precision is zero.

If an * is specified after the decimal, something amazing happens! It means the int argument to printf() before the number to be printed holds the precision. You can use this if you don't know the precision at compile time.

```
int precision;
double pi = 3.14159265358979;

printf("Enter precision: "); fflush(stdout);
scanf("%d", &precision);

printf("%.*f\n", precision, pi);
```

Which gives:

```
Enter precision: 4
3.1416
```

61.10.0.5 Field Width

In front of the optional precision, you can indicate a field width. This is a decimal number that indicates how wide the region should be in which the argument is printed. The region is padding with leading (or trailing) spaces to make sure it's wide enough.

If the field width specified is too small to hold the output, it is ignored.

As a preview, you can give a negative field width to justify the item the other direction.

So let's print a number in a field of width 10. We'll put some angle brackets around it so we can see the padding spaces in the output.

```
printf("<<%10d>>\n", 3490);   // right justified
printf("<<%-10d>>\n", 3490);  // left justified
```

```
<<      3490>>
<<3490      >>
```

Like with the precision, you can use an asterisk (`*`) as the field width

```
int field_width;
int val = 3490;

printf("Enter field_width: "); fflush(stdout);
scanf("%d", &field_width);

printf("<<%*d>>\n", field_width, val);
```

61.10.0.6 Flags

Before the field width, you can put some optional flags that further control the output of the subsequent fields. We just saw that the `-` flag can be used to left- or right-justify fields. But there are plenty more!

Flag	Description
`-`	For a field width, left justify in the field (right is default).
`+`	If the number is signed, always prefix a `+` or `-` on the front.
[SPACE]	If the number is signed, prefix a space for positive, or a `-` for negative.
`0`	Pad the right-justified field with leading zeros instead of leading spaces.
`#`	Print using an alternate form. See below.

For example, we could pad a hexadecimal number with leading zeros to a field width of 8 with:

```
printf("%08x\n", 0x1234);  // 00001234
```

The `#` "alternate form" result depends on the conversion specifier.

Conversion Specifier	Alternate Form (`#`) Meaning
`o`	Increase precision of a non-zero number just enough to get one leading `0` on the octal number.
`x`	Prefix a non-zero number with `0x`.
`X`	Same as `x`, except capital `0X`.
`a`, `e`, `f`	Always print a decimal point, even if nothing follows it.
`A`, `E`, `F`	Identical to `a`, `e`, `f`.
`g`, `G`	Always print a decimal point, even if nothing follows it, and keep trailing zeros.

www.dbooks.org

61.10.0.7 sprintf() and snprintf() Details

Both sprintf() and snprintf() have the quality that if you pass in NULL as the buffer, nothing is written—but you can still check the return value to see how many characters *would* have been written.

snprintf() **always** terminates the string with a NUL character. So if you try to write out more than the maximum specified characters, the universe ends.

Just kidding. If you do, snprintf() will write $n - 1$ characters so that it has enough room to write the terminator at the end.

Return Value

Returns the number of characters outputted, or a negative number on error.

Example

```c
#include <stdio.h>

int main(void)
{
    int a = 100;
    float b = 2.717;
    char *c = "beej!";
    char d = 'X';
    int e = 5;

    printf("%d\n", a); /* "100"      */
    printf("%f\n", b); /* "2.717000" */
    printf("%s\n", c); /* "beej!"    */
    printf("%c\n", d); /* "X"        */
    printf("110%%\n"); /* "110%"     */

    printf("%10d\n", a);   /* "       100" */
    printf("%-10d\n", a);  /* "100       " */
    printf("%*d\n", e, a); /* "  100"      */
    printf("%.2f\n", b);   /* "2.72"       */

    printf("%hhd\n", d); /* "88" <-- ASCII code for 'X' */

    printf("%5d %5.2f %c\n", a, b, d); /* "  100  2.72 X" */
}
```

See Also

sprintf(), vprintf()

61.11 scanf(), fscanf(), sscanf()

Read formatted string, character, or numeric data from the console or from a file

Synopsis

```
#include <stdio.h>

int scanf(const char *format, ...);

int fscanf(FILE *stream, const char *format, ...);

int sscanf(const char * restrict s, const char * restrict format, ...);
```

Description

These functions read formatted output from a variety of sources.

Function	Input Source
`scanf()`	Read from the console (keyboard by default, typically).
`fscanf()`	Read from a file.
`sscanf()`	Read from a string.

The only differences between these is are the leading parameters that you pass to them before the `format` string.

Function	What you pass before `format`
`scanf()`	Nothing comes before `format`.
`fscanf()`	Pass a `FILE*`.
`sscanf()`	Pass a `char*` to a buffer to read from.

The `scanf()` family of functions reads data from the console or from a `FILE` stream, parses it, and stores the results away in variables you provide in the argument list.

The format string is very similar to that in `printf()` in that you can tell it to read a `"%d"`, for instance for an `int`. But it also has additional capabilities, most notably that it can eat up other characters in the input that you specify in the format string.

But let's start simple, and look at the most basic usage first before plunging into the depths of the function. We'll start by reading an `int` from the keyboard:

```
int a;

scanf("%d", &a);
```

`scanf()` obviously needs a pointer to the variable if it is going to change the variable itself, so we use the address-of operator to get the pointer.

In this case, `scanf()` walks down the format string, finds a "`%d`", and then knows it needs to read an integer and store it in the next variable in the argument list, `a`.

Here are some of the other format specifiers you can put in the format string:

Format Specifier	Description
`%d`	Reads an integer to be stored in an `int`. This integer can be signed.
`%u`	Reads an integer to be stored in an `unsigned int`.

Format Specifier	Description
`%f`	Reads a floating point number, to be stored in a `float`.
`%s`	Reads a string up to the first whitespace character.
`%c`	Reads a `char`.

And that's the end of the story!

Ha! Just kidding. If you've just arrived from the `printf()` page, you know there's a near-infinite amount of additional material.

61.11.0.1 Consuming Other Characters

`scanf()` will move along the format string matching any characters you include.

For example, you could read a hyphenated date like so:

```
scanf("%u-%u-%u", &yyyy, &mm, &dd);
```

In that case, `scanf()` will attempt to consume an unsigned decimal number, then a hyphen, then another unsigned number, then another hypen, then another unsigned number.

If it fails to match at any point (e.g. the user entered "foo"), `scanf()` will bail without consuming the offending characters.

And it will return the number of variables successfully converted. In the example above, if the user entered a valid string, `scanf()` would return `3`, one for each variable successfully read.

61.11.0.2 Problems with `scanf()`

I (and the C FAQ and a lot of people) recommend *against* using `scanf()` to read directly from the keyboard. It's too easy for it to stop consuming characters when the user enters some bad data.

If you have data in a file and you're confident it's in good shape, `fscanf()` can be really useful.

But in the case of the keyboard or file, you can always use `fgets()` to read a complete line into a buffer, and then use `sscanf()` to scan things out of the buffer. This gives you the best of both worlds.

61.11.0.3 Problems with `sscanf()`

A while back, a third-party programmer rose to fame for figuring out how to cut *GTA Online* load times by 70%[3].

What they'd discovered was that the implementation of `sscanf()` first effectively calls `strlen()`… so even if you're just using `sscanf()` to peel the first few characters off the string, it still runs all the way out to the end of the string first.

On small strings, no big deal, but on large strings with repeated calls (which is what was happening in *GTA*) it got *sloooooooooowwwww*…

So if you're just converting a string to a number, consider `atoi()`, `atof()`, or the `strtol()` and `strtod()` families of functions, instead.

(The programmer collected a $10,000 bug bounty for the effort.)

[3] https://nee.lv/2021/02/28/How-I-cut-GTA-Online-loading-times-by-70/

61.11.0.4 The Deep Details

Let's check out what a `scanf()`

And here are some more codes, except these don't tend to be used as often. You, of course, may use them as often as you wish!

First, the format string. Like we mentioned, it can hold ordinary characters as well as `%` format specifiers. And whitespace characters.

Whitespace characters have a special role: a whitespace character will cause `scanf()` to consume as many whitespace characters as it can up to the next non-whitespace character. You can use this to ignore all leading or trailing whitespace.

Also, all format specifiers except for `s`, `c`, and `[` automatically consume leading whitespace.

But I know what you're thinking: the meat of this function is in the format specifiers. What do those look like?

These consist of the following, in sequence:

1. A `%` sign
2. Optional: an `*` to suppress assignment—more later
3. Optional: a field width—max characters to read
4. Optional: length modifier, for specifying longer or shorter types
5. A conversion specifier, like `d` or `f` indicating the type to read

61.11.0.5 The Conversion Specifier

Let's start with the best and last: the *conversion specifier*.

This is the part of the format specifier that tells us what type of variable `scanf()` should be reading into, like `%d` or `%f`.

Conversion Specifier	Description
`d`	Matches a decimal `int`. Can have a leading sign.
`i`	Like `d`, except will handle it if you put a leading `0x` (hex) or `0` (octal) on the number.
`o`	Matches an octal (base 8) `unsigned int`. Leading zeros are ignored.
`u`	Matches a decimal `unsigned int`.
`x`	Matches a hex (base 16) `unsigned int`.
`f`	Match a floating point number (or scientific notation, or anything `strtod()` can handle).
`c`	Match a `char`, or mutiple `chars` if a field width is given.
`s`	Match a sequence of non-whitespace `chars`.
`[`	Match a sequence of characters from a set. The set ends with `]`. More below.
`p`	Match a pointer, the opposite of `%p` for `printf()`.
`n`	Store the number of characters written so far in the given `int*`. Doesn't consume anything.
`%`	Match a literal percent sign.

All of the following are equivalent to the `f` specifier: `a`, `e`, `g`, `A`, `E`, `F`, `G`.

And capital `X` is equivalent to lowercase `x`.

61.11.0.5.1 The Scanset `%[]` Conversion Specifier This is about the weirdest format specifier there is. It allows you to specify a set of characters (the *scanset*) to be stored away (likely in an array of `chars`). Conversion stops when a character that is not in the set is matched.

www.dbooks.org

For example, `%[0-9]` means "match all numbers zero through nine." And `%[AD-G34]` means "match A, D through G, 3, or 4".

Now, to convolute matters, you can tell `scanf()` to match characters that are *not* in the set by putting a caret (`^`) directly after the `%[` and following it with the set, like this: `%[^A-C]`, which means "match all characters that are *not* A through C."

To match a close square bracket, make it the first character in the set, like this: `%[]A-C]` or `%[^]A-C]`. (I added the "`A-C`" just so it was clear that the "`]`" was first in the set.)

To match a hyphen, make it the last character in the set, e.g. to match A-through-C or hyphen: `%[A-C-]`.

So if we wanted to match all letters *except* "%", "^", "]", "B", "C", "D", "E", and "-", we could use this format string: `%[^]%^B-E-]`.

Got it? Now we can go onto the next func—no wait! There's more! Yes, still more to know about `scanf()`. Does it never end? Try to imagine how I feel writing about it!

61.11.0.6 The Length Modifier

So you know that "`%d`" stores into an `int`. But how do you store into a `long`, `short`, or `double`?

Well, like in `printf()`, you can add a modifier before the type specifier to tell `scanf()` that you have a longer or shorter type. The following is a table of the possible modifiers:

Length Modifier	Conversion Specifier	Description
`hh`	`d`, `i`, `o`, `u`, `x`, `X`	Convert input to `char` (signed or unsigned as appropriate) before printing.
`h`	`d`, `i`, `o`, `u`, `x`, `X`	Convert input to `short int` (signed or unsigned as appropriate) before printing.
`l`	`d`, `i`, `o`, `u`, `x`, `X`	Convert input to `long int` (signed or unsigned as appropriate).
`ll`	`d`, `i`, `o`, `u`, `x`, `X`	Convert input to `long long int` (signed or unsigned as appropriate).
`j`	`d`, `i`, `o`, `u`, `x`, `X`	Convert input to `intmax_t` or `uintmax_t` (as appropriate).
`z`	`d`, `i`, `o`, `u`, `x`, `X`	Convert input to `size_t`.
`t`	`d`, `i`, `o`, `u`, `x`, `X`	Convert input to `ptrdiff_t`.
`L`	`a`, `A`, `e`, `E`, `f`, `F`, `g`, `G`	Convert input to `long double`.
`l`	`c`,`s`,`[`	Convert input to `wchar_t`, a wide character.
`l`	`s`	Argument is in a `wchar_t*`, a wide character string.
`hh`	`n`	Store result in `signed char*` argument.
`h`	`n`	Store result in `short int*` argument.
`l`	`n`	Store result in `long int*` argument.
`ll`	`n`	Store result in `long long int*` argument.
`j`	`n`	Store result in `intmax_t*` argument.
`z`	`n`	Store result in `size_t*` argument.
`t`	`n`	Store result in `ptrdiff_t*` argument.

61.11.0.7 Field Widths

The field width generally allows you to specify a maximum number of characters to consume. If the thing you're trying to match is shorter than the field width, that input will stop being processed before the field width is reached.

So a string will stop being consumed when whitespace is found, even if fewer than the field width characters are matched.

And a float will stop being consumed at the end of the number, even if fewer characters than the field width are matched.

But `%c` is an interesting one—it doesn't stop consuming characters on anything. So it'll go exactly to the field width. (Or 1 character if no field width is given.)

61.11.0.8 Skip Input with *

If you put an `*` in the format specifier, it tells `scanf()` do to the conversion specified, but not store it anywhere. It simply discards the data as it reads it. This is what you use if you want `scanf()` to eat some data but you don't want to store it anywhere; you don't give `scanf()` an argument for this conversion.

```
// Read 3 ints, but discard the middle one
scanf("%d %*d %d", &int1, &int3);
```

Return Value

`scanf()` returns the number of items assigned into variables. Since assignment into variables stops when given invalid input for a certain format specifier, this can tell you if you've input all your data correctly.

Also, `scanf()` returns `EOF` on end-of-file.

Example

```
#include <stdio.h>

int main(void)
{
    int a;
    long int b;
    unsigned int c;
    float d;
    double e;
    long double f;
    char s[100];

    scanf("%d", &a);  // store an int
    scanf(" %d", &a); // eat any whitespace, then store an int
    scanf("%s", s); // store a string
    scanf("%Lf", &f); // store a long double

    // store an unsigned, read all whitespace, then store a long int:
    scanf("%u %ld", &c, &b);

    // store an int, read whitespace, read "blendo", read whitespace,
    // and store a float:
    scanf("%d blendo %f", &a, &d);

    // read all whitespace, then store all characters up to a newline
    scanf(" %[^\n]", s);

    // store a float, read (and ignore) an int, then store a double:
    scanf("%f %*d %lf", &d, &e);

    // store 10 characters:
```

```
    scanf("%10c", s);
}
```

See Also

sscanf(), vscanf(), vsscanf(), vfscanf()

61.12 vprintf(), vfprintf(), vsprintf(), vsnprintf()

printf() variants using variable argument lists (va_list)

Synopsis

```
#include <stdio.h>
#include <stdarg.h>

int vprintf(const char * restrict format, va_list arg);

int vfprintf(FILE * restrict stream, const char * restrict format,
             va_list arg);

int vsprintf(char * restrict s, const char * restrict format, va_list arg);

int vsnprintf(char * restrict s, size_t n, const char * restrict format,
              va_list arg);
```

Description

These are just like the printf() variants except instead of taking an actual variable number of arguments, they take a fixed number—the last of which is a va_list that refers to the variable arguments.

Like with printf(), the different variants send output different places.

Function	Output Destination
vprintf()	Print to console (screen by default, typically).
vfprintf()	Print to a file.
vsprintf()	Print to a string.
vsnprintf()	Print to a string (safely).

Both vsprintf() and vsnprintf() have the quality that if you pass in NULL as the buffer, nothing is written—but you can still check the return value to see how many characters *would* have been written.

If you try to write out more than the maximum number of characters, vsnprintf() will graciously write only $n - 1$ characters so that it has enough room to write the terminator at the end.

As for why in the heck would you ever want to do this, the most common reason is to create your own specialized versions of printf()-type functions, piggybacking on all that printf() functionality goodness.

See the example for an example, predictably.

Return Value

`vprintf()` and `vfprintf()` return the number of characters printed, or a negative value on error.

`vsprintf()` returns the number of characters printed to the buffer, not counting the NUL terminator, or a negative value if an error occurred.

`vnsprintf()` returns the number of characters printed to the buffer. Or the number that *would* have been printed if the buffer had been large enough.

Example

In this example, we make our own version of `printf()` called `logger()` that timestamps output. Notice how the calls to `logger()` have all the bells and whistles of `printf()`.

```c
#include <stdio.h>
#include <stdarg.h>
#include <time.h>

int logger(char *format, ...)
{
    va_list va;
    time_t now_secs = time(NULL);
    struct tm *now = gmtime(&now_secs);

    // Output timestamp in format "YYYY-MM-DD hh:mm:ss : "
    printf("%04d-%02d-%02d %02d:%02d:%02d : ",
        now->tm_year + 1900, now->tm_mon + 1, now->tm_mday,
        now->tm_hour, now->tm_min, now->tm_sec);

    va_start(va, format);
    int result = vprintf(format, va);
    va_end(va);

    printf("\n");

    return result;
}

int main(void)
{
    int x = 12;
    float y = 3.2;

    logger("Hello!");
    logger("x = %d and y = %.2f", x, y);
}
```

Output:

```
2021-03-30 04:25:49 : Hello!
2021-03-30 04:25:49 : x = 12 and y = 3.20
```

See Also

`printf()`

61.13 vscanf(), vfscanf(), vsscanf()

scanf() variants using variable argument lists (va_list)

Synopsis

```
#include <stdio.h>
#include <stdarg.h>

int vscanf(const char * restrict format, va_list arg);

int vfscanf(FILE * restrict stream, const char * restrict format,
            va_list arg);

int vsscanf(const char * restrict s, const char * restrict format,
            va_list arg);
```

Description

These are just like the scanf() variants except instead of taking an actual variable number of arguments, they take a fixed number—the last of which is a va_list that refers to the variable arguments.

Function	Input Source
vscanf()	Read from the console (keyboard by default, typically).
vfscanf()	Read from a file.
vsscanf()	Read from a string.

Like with the vprintf() functions, this would be a good way to add additional functionality that took advantage of the power scanf() has to offer.

Return Value

Returns the number of items successfully scanned, or EOF on end-of-file or error.

Example

I have to admit I was wracking my brain to think of when you'd ever want to use this. The best example I could find was one on Stack Overflow[4] that error-checks the return value from scanf() against the expected. A variant of that is shown below.

```
#include <stdio.h>
#include <stdarg.h>
#include <assert.h>

int error_check_scanf(int expected_count, char *format, ...)
{
    va_list va;

```

[4] https://stackoverflow.com/questions/17017331/c99-vscanf-for-dummies/17018046#17018046

```
    va_start(va, format);
    int count = vscanf(format, va);
    va_end(va);

    // This line will crash the program if the condition is false:
    assert(count == expected_count);

    return count;
}

int main(void)
{
    int a, b;
    float c;

    error_check_scanf(3, "%d, %d/%f", &a, &b, &c);
    error_check_scanf(2, "%d", &a);
}
```

See Also

`scanf()`

61.14 `getc()`, `fgetc()`, `getchar()`

Get a single character from the console or from a file

Synopsis

```
#include <stdio.h>

int getc(FILE *stream);

int fgetc(FILE *stream);

int getchar(void);
```

Description

All of these functions in one way or another, read a single character from the console or from a `FILE`. The differences are fairly minor, and here are the descriptions:

`getc()` returns a character from the specified `FILE`. From a usage standpoint, it's equivalent to the same `fgetc()` call, and `fgetc()` is a little more common to see. Only the implementation of the two functions differs.

`fgetc()` returns a character from the specified `FILE`. From a usage standpoint, it's equivalent to the same `getc()` call, except that `fgetc()` is a little more common to see. Only the implementation of the two functions differs.

Yes, I cheated and used cut-n-paste to do that last paragraph.

`getchar()` returns a character from `stdin`. In fact, it's the same as calling `getc(stdin)`.

Return Value

All three functions return the `unsigned char` that they read, except it's cast to an `int`.

If end-of-file or an error is encountered, all three functions return `EOF`.

Example

This example reads all the characters from a file, outputting only the letter 'b's it finds..

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    int c;

    fp = fopen("spoon.txt", "r"); // error check this!

    // this while-statement assigns into c, and then checks against EOF:

    while((c = fgetc(fp)) != EOF) {
        if (c == 'b') {
            putchar(c);
        }
    }

    putchar('\n');

    fclose(fp);
}
```

See Also

61.15 `gets()`, `fgets()`

Read a string from console or file

Synopsis

```
#include <stdio.h>

char *fgets(char *s, int size, FILE *stream);
char *gets(char *s);
```

Description

These are functions that will retrieve a newline-terminated string from the console or a file. In other normal words, it reads a line of text. The behavior is slightly different, and, as such, so is the usage. For instance, here is the usage of `gets()`:

Don't use `gets()`. In fact, as of C11, it ceases to exist! This is one of the rare cases of a function being *removed* from the standard.

Admittedly, rationale would be useful, yes? For one thing, `gets()` doesn't allow you to specify the length of the buffer to store the string in. This would allow people to keep entering data past the end of your buffer, and believe me, this would be Bad News.

And that's what the `size` parameter in `fgets()` is for. `fgets()` will read at most `size-1` characters and then stick a `NUL` terminator on after that.

I was going to add another reason, but that's basically the primary and only reason not to use `gets()`. As you might suspect, `fgets()` allows you to specify a maximum string length.

One difference here between the two functions: `gets()` will devour and throw away the newline at the end of the line, while `fgets()` will store it at the end of your string (space permitting).

Here's an example of using `fgets()` from the console, making it behave more like `gets()` (with the exception of the newline inclusion):

```
char s[100];
gets(s);  // don't use this--read a line (from stdin)
fgets(s, sizeof(s), stdin); // read a line from stdin
```

In this case, the `sizeof()` operator gives us the total size of the array in bytes, and since a `char` is a byte, it conveniently gives us the total size of the array.

Of course, like I keep saying, the string returned from `fgets()` probably has a newline at the end that you might not want. You can write a short function to chop the newline off—in fact, let's just roll that into our own version of `gets()`

```
#include <stdio.h>
#include <string.h>

char *ngets(char *s, int size)
{
    char *rv = fgets(s, size, stdin);

    if (rv == NULL)
        return NULL;

    char *p = strchr(s, '\n');  // Find a newline

    if (p != NULL)  // if there's a newline
        *p = '\0';  // truncate the string there

    return s;
}
```

So, in summary, use `fgets()` to read a line of text from the keyboard or a file, and don't use `gets()`.

Return Value

Both `gets()` and `fgets()` return a pointer to the string passed.

On error or end-of-file, the functions return `NULL`.

Example

```
#include <stdio.h>

int main(void)
```

```
{
    FILE *fp;
    char s[100];

    gets(s); // read from standard input (don't use this--use fgets()!)

    fgets(s, sizeof s, stdin); // read 100 bytes from standard input

    fp = fopen("spoon.txt", "r"); // (you should error-check this)
    fgets(s, 100, fp); // read 100 bytes from the file datafile.dat
    fclose(fp);

    fgets(s, 20, stdin); // read a maximum of 20 bytes from stdin
}
```

See Also

`getc()`, `fgetc()`, `getchar()`, `puts()`, `fputs()`, `ungetc()`

61.16 `putc()`, `fputc()`, `putchar()`

Write a single character to the console or to a file

Synopsis

```
#include <stdio.h>

int putc(int c, FILE *stream);

int fputc(int c, FILE *stream);

int putchar(int c);
```

Description

All three functions output a single character, either to the console or to a `FILE`.

`putc()` takes a character argument, and outputs it to the specified `FILE`. `fputc()` does exactly the same thing, and differs from `putc()` in implementation only. Most people use `fputc()`.

`putchar()` writes the character to the console, and is the same as calling `putc(c, stdout)`.

Return Value

All three functions return the character written on success, or `EOF` on error.

Example

Print the alphabet:

```c
#include <stdio.h>

int main(void)
{
    char i;

    for(i = 'A'; i <= 'Z'; i++)
        putchar(i);

    putchar('\n'); // put a newline at the end to make it pretty
}
```

See Also

61.17 puts(), fputs()

Write a string to the console or to a file

Synopsis

```c
#include <stdio.h>

int puts(const char *s);

int fputs(const char *s, FILE *stream);
```

Description

Both these functions output a NUL-terminated string. `puts()` outputs to the console, while `fputs()` allows you to specify the file for output.

Return Value

Both functions return non-negative on success, or `EOF` on error.

Example

Read strings from the console and save them in a file:

```c
#include <stdio.h>

int main(void)
{
    FILE *fp;
    char s[100];

    fp = fopen("somefile.txt", "w"); // error check this!

    while(fgets(s, sizeof(s), stdin) != NULL) { // read a string
        fputs(s, fp);  // write it to the file we opened
```

```
    }

    fclose(fp);
}
```

See Also

61.18 ungetc()

Pushes a character back into the input stream

Synopsis

```
#include <stdio.h>

int ungetc(int c, FILE *stream);
```

Description

You know how `getc()` reads the next character from a file stream? Well, this is the opposite of that—it pushes a character back into the file stream so that it will show up again on the very next read from the stream, as if you'd never gotten it from `getc()` in the first place.

Why, in the name of all that is holy would you want to do that? Perhaps you have a stream of data that you're reading a character at a time, and you won't know to stop reading until you get a certain character, but you want to be able to read that character again later. You can read the character, see that it's what you're supposed to stop on, and then `ungetc()` it so it'll show up on the next read.

Yeah, that doesn't happen very often, but there we are.

Here's the catch: the standard only guarantees that you'll be able to push back *one character*. Some implementations might allow you to push back more, but there's really no way to tell and still be portable.

Return Value

On success, `ungetc()` returns the character you passed to it. On failure, it returns `EOF`.

Example

This example reads a piece of punctuation, then everything after it up to the next piece of punctuation. It returns the leading punctuation, and stores the rest in a string.

```
#include <stdio.h>
#include <ctype.h>

int read_punctstring(FILE *fp, char *s)
{
    int origpunct, c;

    origpunct = fgetc(fp);

    if (origpunct == EOF)  // return EOF on end-of-file
```

```
        return EOF;

    while (c = fgetc(fp), !ispunct(c) && c != EOF)
        *s++ = c;  // save it in the string

    *s = '\0'; // nul-terminate the string

    // if we read punctuation last, ungetc it so we can fgetc it next
    // time:
    if (ispunct(c))
        ungetc(c, fp);

    return origpunct;
}

int main(void)
{
    char s[128];
    char c;

    while((c = read_punctstring(stdin, s)) != EOF) {
        printf("%c: %s\n", c, s);
    }
}
```

Sample Input:

```
!foo#bar*baz
```

Sample output:

```
!: foo
#: bar
*: baz
```

See Also

`fgetc()`

61.19 `fread()`

Read binary data from a file

Synopsis

```
#include <stdio.h>

size_t fread(void *p, size_t size, size_t nmemb, FILE *stream);
```

Description

You might remember that you can call `fopen()` with the "b" flag in the open mode string to open the file in "binary" mode. Files open in not-binary (ASCII or text mode) can be read using standard character-oriented

calls like `fgetc()` or `fgets()`. Files open in binary mode are typically read using the `fread()` function.

All this function does is says, "Hey, read this many things where each thing is a certain number of bytes, and store the whole mess of them in memory starting at this pointer."

This can be very useful, believe me, when you want to do something like store 20 `ints` in a file.

But wait—can't you use `fprintf()` with the "%d" format specifier to save the `ints` to a text file and store them that way? Yes, sure. That has the advantage that a human can open the file and read the numbers. It has the disadvantage that it's slower to convert the numbers from `ints` to text and that the numbers are likely to take more space in the file. (Remember, an `int` is likely 4 bytes, but the string "12345678" is 8 bytes.)

So storing the binary data can certainly be more compact and faster to read.

Return Value

This function returns the number of items successfully read. If all requested items are read, the return value will be equal to that of the parameter `nmemb`. If EOF occurs, the return value will be zero.

To make you confused, it will also return zero if there's an error. You can use the functions `feof()` or `ferror()` to tell which one really happened.

Example

Read 10 numbers from a file and store them in an array:

```
#include <stdio.h>

int main(void)
{
    int i;
    int n[10]
    FILE *fp;

    fp = fopen("numbers.dat", "rb");
    fread(n, sizeof(int), 10, fp);  // read 10 ints
    fclose(fp);

    // print them out:
    for(i = 0; i < 10; i++)
        printf("n[%d] == %d\n", i, n[i]);
}
```

See Also

`fopen()`, `fwrite()`, `feof()`, `ferror()`

61.20 `fwrite()`

Write binary data to a file

Synopsis

```
#include <stdio.h>

size_t fwrite(const void *p, size_t size, size_t nmemb, FILE *stream);
```

Description

This is the counterpart to the `fread()` function. It writes blocks of binary data to disk. For a description of what this means, see the entry for `fread()`.

Return Value

`fwrite()` returns the number of items successfully written, which should hopefully be `nmemb` that you passed in. It'll return zero on error.

Example

Save 10 random numbers to a file:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int i;
    int n[10];
    FILE *fp;

    // populate the array with random numbers:
    for(i = 0; i < 10; i++) {
        n[i] = rand();
        printf("n[%d] = %d\n", i, n[i]);
    }

    // save the random numbers (10 ints) to the file
    fp = fopen("numbers.dat", "wb");
    fwrite(n, sizeof(int), 10, fp); // write 10 ints
    fclose(fp);
}
```

See Also

`fopen()`, `fread()`

61.21 `fgetpos()`, `fsetpos()`

Get the current position in a file, or set the current position in a file. Just like `ftell()` and `fseek()` for most systems

Synopsis

```
#include <stdio.h>

int fgetpos(FILE *stream, fpos_t *pos);

int fsetpos(FILE *stream, fpos_t *pos);
```

Description

These functions are just like `ftell()` and `fseek()`, except instead of counting in bytes, they use an *opaque* data structure to hold positional information about the file. (Opaque, in this case, means you're not supposed to know what the data type is made up of.)

On virtually every system (and certainly every system that I know of), people don't use these functions, using `ftell()` and `fseek()` instead. These functions exist just in case your system can't remember file positions as a simple byte offset.

Since the `pos` variable is opaque, you have to assign to it using the `fgetpos()` call itself. Then you save the value for later and use it to reset the position using `fsetpos()`.

Return Value

Both functions return zero on success, and `-1` on error.

Example

```
#include <stdio.h>

int main(void)
{
    char s[100];
    fpos_t pos;
    FILE *fp;

    fp = fopen("spoon.txt", "r");

    fgets(s, sizeof(s), fp); // read a line from the file
    printf("%s", s);

    fgetpos(fp, &pos);   // save the position after the read

    fgets(s, sizeof(s), fp); // read another line from the file
    printf("%s", s);

    fsetpos(fp, &pos);   // now restore the position to where we saved

    fgets(s, sizeof(s), fp); // read the earlier line again
    printf("%s", s);

    fclose(fp);
}
```

See Also

fseek(), ftell(), rewind()

61.22 fseek(), rewind()

Position the file pointer in anticipition of the next read or write

Synopsis

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);

void rewind(FILE *stream);
```

Description

When doing reads and writes to a file, the OS keeps track of where you are in the file using a counter generically known as the file pointer. You can reposition the file pointer to a different point in the file using the fseek() call. Think of it as a way to randomly access you file.

The first argument is the file in question, obviously. offset argument is the position that you want to seek to, and whence is what that offset is relative to.

Of course, you probably like to think of the offset as being from the beginning of the file. I mean, "Seek to position 3490, that should be 3490 bytes from the beginning of the file." Well, it *can* be, but it doesn't have to be. Imagine the power you're wielding here. Try to command your enthusiasm.

You can set the value of whence to one of three things:

whence	Description
SEEK_SET	offset is relative to the beginning of the file. This is probably what you had in mind anyway, and is the most commonly used value for whence.
SEEK_CUR	offset is relative to the current file pointer position. So, in effect, you can say, "Move to my current position plus 30 bytes," or, "move to my current position minus 20 bytes."
SEEK_END	offset is relative to the end of the file. Just like SEEK_SET except from the other end of the file. Be sure to use negative values for offset if you want to back up from the end of the file, instead of going past the end into oblivion.

Speaking of seeking off the end of the file, can you do it? Sure thing. In fact, you can seek way off the end and then write a character; the file will be expanded to a size big enough to hold a bunch of zeros way out to that character.

Now that the complicated function is out of the way, what's this rewind() that I briefly mentioned? It repositions the file pointer at the beginning of the file:

```
fseek(fp, 0, SEEK_SET); // same as rewind()
rewind(fp);             // same as fseek(fp, 0, SEEK_SET)
```

Return Value

For `fseek()`, on success zero is returned; `-1` is returned on failure.

The call to `rewind()` never fails.

Example

```
#include <stdio.h>

int main(void)
{
    FILE *fp;

    fp = fopen("spoon.txt", "r");

    fseek(fp, 100, SEEK_SET); // seek to the 100th byte of the file
    printf("100: %c\n", fgetc(fp));

    fseek(fp, -31, SEEK_CUR); // seek backward 30 bytes from the current pos
    printf("31 back: %c\n", fgetc(fp));

    fseek(fp, -12, SEEK_END); // seek to the 10th byte before the end of file
    printf("12 from end: %c\n", fgetc(fp));

    fseek(fp, 0, SEEK_SET);   // seek to the beginning of the file
    rewind(fp);               // seek to the beginning of the file, too
    printf("Beginning: %c\n", fgetc(fp));

    fclose(fp);
}
```

See Also

`ftell()`, `fgetpos()`, `fsetpos()`

61.23 `ftell()`

Tells you where a particular file is about to read from or write to

Synopsis

```
#include <stdio.h>

long ftell(FILE *stream);
```

Description

This function is the opposite of `fseek()`. It tells you where in the file the next file operation will occur relative to the beginning of the file.

It's useful if you want to remember where you are in the file, `fseek()` somewhere else, and then come back later. You can take the return value from `ftell()` and feed it back into `fseek()` (with `whence` parameter set to `SEEK_SET`) when you want to return to your previous position.

Return Value

Returns the current offset in the file, or `-1` on error.

Example

```c
#include <stdio.h>

int main(void)
{
    char c[6];
    FILE *fp;

    fp = fopen("spoon.txt", "r");

    long pos;

    // seek ahead 10 bytes:
    fseek(fp, 10, SEEK_SET);

    // store the current position in variable "pos":
    pos = ftell(fp);

    // Read some bytes
    fread(c, sizeof c  - 1, 1, fp);
    c[5] = '\0';
    printf("Read: \"%s\"\n", c);

    // and return to the starting position, stored in "pos":
    fseek(fp, pos, SEEK_SET);

    // Read the same bytes again
    fread(c, sizeof c  - 1, 1, fp);
    c[5] = '\0';
    printf("Read: \"%s\"\n", c);

    fclose(fp);
}
```

See Also

`fseek()`, `rewind()`, `fgetpos()`, `fsetpos()`

61.24 `feof()`, `ferror()`, `clearerr()`

Determine if a file has reached end-of-file or if an error has occurred

Synopsis

```
#include <stdio.h>

int feof(FILE *stream);

int ferror(FILE *stream);

void clearerr(FILE *stream);
```

Description

Each `FILE*` that you use to read and write data from and to a file contains flags that the system sets when certain events occur. If you get an error, it sets the error flag; if you reach the end of the file during a read, it sets the EOF flag. Pretty simple really.

The functions `feof()` and `ferror()` give you a simple way to test these flags: they'll return non-zero (true) if they're set.

Once the flags are set for a particular stream, they stay that way until you call `clearerr()` to clear them.

Return Value

`feof()` and `ferror()` return non-zero (true) if the file has reached EOF or there has been an error, respectively.

Example

Read binary data, checking for EOF or error:

```
#include <stdio.h>

int main(void)
{
    int a;
    FILE *fp;

    fp = fopen("numbers.dat", "r");

    // read single ints at a time, stopping on EOF or error:

    while(fread(&a, sizeof(int), 1, fp), !feof(fp) && !ferror(fp)) {
        printf("Read %d\n", a);
    }

    if (feof(fp))
        printf("End of file was reached.\n");

    if (ferror(fp))
        printf("An error occurred.\n");

    fclose(fp);
}
```

See Also

fopen(), fread()

61.25 perror()

Print the last error message to stderr

Synopsis

```
#include <stdio.h>
#include <errno.h>  // only if you want to directly use the "errno" var

void perror(const char *s);
```

Description

Many functions, when they encounter an error condition for whatever reason, will set a global variable called errno (in <errno.h>) for you. errno is just an interger representing a unique error.

But to you, the user, some number isn't generally very useful. For this reason, you can call perror() after an error occurs to print what error has actually happened in a nice human-readable string.

And to help you along, you can pass a parameter, s, that will be prepended to the error string for you.

One more clever trick you can do is check the value of the errno (you have to include errno.h to see it) for specific errors and have your code do different things. Perhaps you want to ignore certain errors but not others, for instance.

The standard only defines three values for errno, but your system undoubtedly defines more. The three that are defined are:

errno	Description
EDOM	Math operation outside domain.
EILSEQ	Invalid sequence in multibyte to wide character encoding.
ERANGE	Result of operation doesn't fit in specified type.

The catch is that different systems define different values for errno, so it's not very portable beyond the above 3. The good news is that at least the values are *largely* portable between Unix-like systems, at least.

Return Value

Returns nothing at all! Sorry!

Example

fseek() returns -1 on error, and sets errno, so let's use it. Seeking on stdin makes no sense, so it should generate an error:

```
#include <stdio.h>
#include <errno.h> // must include this to see "errno" in this example

int main(void)
```

```
{
    if (fseek(stdin, 10L, SEEK_SET) < 0)
        perror("fseek");

    fclose(stdin); // stop using this stream

    if (fseek(stdin, 20L, SEEK_CUR) < 0) {

        // specifically check errno to see what kind of
        // error happened...this works on Linux, but your
        // mileage may vary on other systems!

        if (errno == EBADF) {
            perror("fseek again, EBADF");
        } else {
            perror("fseek again");
        }
    }
}
```

And the output is:

```
fseek: Illegal seek
fseek again, EBADF: Bad file descriptor
```

See Also

`feof()`, `ferror()`, `strerror()`

Chapter 62

<stdlib.h> Standard Library Functions

Some of the following functions have variants that handle different types: `atoi()`, `strtod()`, `strtol()`, `abs()`, and `div()`. Only a single one is listed here for brevity.

Function	Description
`_Exit()`	Exit the currently-running program and don't look back
`abort()`	Abruptly end program execution
`abs()`	Compute the absolute value of an integer
`aligned_alloc()`	Allocate specifically-aligned memory
`at_quick_exit()`	Set up handlers to run when the program quickly exits
`atexit()`	Set up handlers to run when the program exits
`atof()`	Convert a string to a floating point value
`atoi()`	Convert an integer in a string into a integer type
`bsearch()`	Binary Search (maybe) an array of objects
`calloc()`	Allocate and clear memory for arbitrary use
`div()`	Compute the quotient and remainder of two numbers
`exit()`	Exit the currently-running program
`free()`	Free a memory region
`getenv()`	Get the value of an environment variable
`malloc()`	Allocate memory for arbitrary use
`mblen()`	Return the number of bytes in a multibyte character
`mbstowcs()`	Convert a multibyte string to a wide character string
`mbtowc()`	Convert a multibyte character to a wide character
`qsort()`	Quicksort (maybe) some data
`quick_exit()`	Exit the currently-running program quickly
`rand()`	Return a pseudorandom number
`realloc()`	Resize a previously allocated stretch of memory
`srand()`	Seed the built-in pseudorandom number generator
`strtod()`	Convert a string to a floating point number
`strtol()`	Convert a string to an integer
`system()`	Run an external program
`wcstombs()`	Convert a wide character string to a multibyte string
`wctomb()`	Convert a wide character to a multibyte character

The `<stdlib.h>` header has all kinds of—dare I say—miscellaneous functions bundled into it. This func-

www.dbooks.org

tionality includes:

- Conversions from numbers to strings
- Conversions from strings to numbers
- Pseudorandom number generation
- Dynamic memory allocation
- Various ways to exit the program
- Ability to run external programs
- Binary search (or some fast search)
- Quicksort (or some fast sort)
- Integer arithmetic functions
- Multibyte and wide character and string conversions

So, you know… a little of everything.

62.1 `<stdlib.h>` Types and Macros

A couple new types and macros are introduced, though some of these might also be defined elsewhere:

Type	Description
`size_t`	Returned from `sizeof` and used elsewhere
`wchar_t`	For wide character operations
`div_t`	For the `div()` function
`ldiv_t`	For the `ldiv()` function
`lldiv_t`	for the `lldiv()` function

And some macros:

Type	Description
`NULL`	Our good pointer friend
`EXIT_SUCCESS`	Good exit status when things go well
`EXIT_FAILURE`	Good exit status when things go poorly
`RAND_MAX`	The maximum value that can be returned by the `rand()` function
`MB_CUR_MAX`	Maximum number of bytes in a multibyte character in the current locale

And there you have it. Just a lot of fun, useful functions in here. Let's check 'em out!

62.2 `atof()`

Convert a string to a floating point value

Synopsis

```
#include <stdlib.h>

double atof(const char *nptr);
```

Description

This stood for "ASCII-To-Floating" back in the day[1], but no one would dare to use such coarse language now.

But the gist is the same: we're going to convert a string with numbers and (optionally) a decimal point into a floating point value. Leading whitespace is ignored, and translation stops at the first invalid character.

If the result doesn't fit in a `double`, behavior is undefined.

It generally works as if you'd called `strtod()`:

```
strtod(nptr, NULL)
```

So check out that reference page for more info.

In fact, `strtod()` is just better and you should probably use that.

Return Value

Returns the string converted to a `double`.

Example

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    double x = atof("3.141593");

    printf("%f\n", x);  // 3.141593
}
```

See Also

`atoi()`, `strtod()`

62.3 atoi(), atol(), atoll()

Convert an integer in a string into a integer type

Synopsis

```
#include <stdlib.h>

int atoi(const char *nptr);

long int atol(const char *nptr);

long long int atoll(const char *nptr);
```

[1] http://man.cat-v.org/unix-1st/3/atof

Description

Back in the day, `atoi()` stood for "ASCII-To_Integer"[2] but now the spec makes no mention of that.

These functions take a string with a number in them and convert it to an integer of the specified return type. Leading whitespace is ignored. Translation stops at the first invalid character.

If the result doesn't fit in the return type, behavior is undefined.

It generally works as if you'd called `strtol()` family of functions:

```
atoi(nptr)                  // is basically the same as...
(int)strtol(nptr, NULL, 10)

atol(nptr)                  // is basically the same as...
strtol(nptr, NULL, 10)

atoll(nptr)                 // is basically the same as...
strtoll(nptr, NULL, 10)
```

Again, the `strtol()` functions are generally better, so I recommend them instead of these.

Return Value

Returns an integer result corresponding to the return type.

Example

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int x = atoi("3490");

    printf("%d\n", x);  // 3490
}
```

See Also

`atof()`, `strtol()`

62.4 `strtod()`, `strtof()`, `strtold()`

Convert a string to a floating point number

Synopsis

```
#include <stdlib.h>

double strtod(const char * restrict nptr, char ** restrict endptr);
```

[2] http://man.cat-v.org/unix-1st/3/atoi

```
float strtof(const char * restrict nptr, char ** restrict endptr);

long double strtold(const char * restrict nptr, char ** restrict endptr);
```

Description

These are some neat functions that convert strings to floating point numbers (or even NaN or Infinity) and provide some error checking, besides.

Firstly, leading whitespace is skipped.

Then the functions attempt to convert characters into the floating point result. Finally, when an invalid character (or NUL character) is reached, they set `endptr` to point to the invalid character.

Set `endptr` to `NULL` if you don't care about where the first invalid character is.

If you didn't set `endptr` to `NULL`, it will point to a NUL character if the translation didn't find any bad characters. That is:

```
if (*endptr == '\0') {
    printf("What a perfectly-formed number!\n");
} else {
    printf("I found badness in your number: \"%s\"\n", endptr);
}
```

But guess what! You can also translate strings into special values, like NaN and Infinity!

If `nptr` points to a string containing `INF` or `INFINITY` (upper or lowercase), the value for Infinity will be returned.

If `nptr` points to a string containing `NAN`, then (a quiet, non-signalling) NaN will be returned. You can tag the `NAN` with a sequence of characters from the set `0-9`, `a-z`, `A-Z`, and _ by enclosing them in parens:

```
NAN(foobar_3490)
```

What your compiler does with this is implementation-defined, but it can be used to specify different kinds of NaN.

You can also specify a number in hexadecimal with a power-of-two exponent (2^x) if you lead with `0x` (or `0X`). For the exponent, use a `p` followed by a base 10 exponent. (You can't use `e` because that's a valid hex digit!)

Example:

```
0xabc.123p15
```

Which computes to $0xabc.123 \times 2^{15}$.

You can put in `FLT_DECIMAL_DIG`, `DBL_DECIMAL_DIG`, or `LDBL_DECIMAL_DIG` digits and get a correctly-rounded result for the type.

Return Value

Returns the converted number. If there was no number, returns `0`. `endptr` is set to point to the first invalid character, or the NUL terminator if all characters were consumed.

If there's an overflow, `HUGE_VAL`, `HUGE_VALF`, or `HUGE_VALL` is returned, signed like the input, and `errno` is set to `ERANGE`.

If there's an underflow, it returns the smallest number closest to zero with the input sign. `errno` may be set to `ERANGE`.

Example

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *inp = "   123.4567beej";
    char *badchar;

    double val = strtod(inp, &badchar);

    printf("Converted string to %f\n", val);
    printf("Encountered bad characters: %s\n", badchar);

    val = strtod("987.654321beej", NULL);
    printf("Ignoring bad chars for result: %f\n", val);

    val = strtod("11.2233", &badchar);

    if (*badchar == '\0')
        printf("No bad chars: %f\n", val);
    else
        printf("Found bad chars: %f, %s\n", val, badchar);
}
```

Output:

```
Converted string to 123.456700
Encountered bad characters: beej
Ignoring bad chars: 987.654321
No bad chars: 11.223300
```

See Also

`atof()`, `strtol()`

62.5 `strtol()`, `strtoll()`, `strtoul()`, `strtoull()`

Convert a string to an integer

Synopsis

```c
#include <stdlib.h>

long int strtol(const char * restrict nptr,
                char ** restrict endptr, int base);

long long int strtoll(const char * restrict nptr,
                      char ** restrict endptr, int base);

unsigned long int strtoul(const char * restrict nptr,
```

```
                                char ** restrict endptr, int base);

unsigned long long int strtoull(const char * restrict nptr,
                                char ** restrict endptr, int base);
```

Description

These convert a string to an integer like `atoi()`, but they have a few more bells and whistles.

Most notable, they can tell you where conversion started going wrong, i.e. where invalid characters, if any, appear. Leading spaces are ignored. A `+` or `-` sign may precede the number.

The basic idea is that if things go well, these functions will return the integer values contained in the strings. And if you pass in the `char**` typed `endptr`, it'll set it to point at the NUL at the end of the string.

If things don't go well, they'll set `endptr` to point at the first character where things have gone awry. That is, if you're converting a value `103z2!` in base 10, they'll send `endptr` to point at the `z` because that's the first non-numeric character.

You can pass in `NULL` for `endptr` if you don't care to do any of that kind of error checking.

Wait—did I just say we could set the number base for the conversion? Yes! Yes, I did. Now number bases[3] are out of scope for this document, but certainly some of the more well-known are binary (base 2), octal (base 8), decimal (base 10), and hexadecimal (base 16).

You can specify the number base for the conversion as the third parameter. Bases from 2 to 36 are supported, with case-insensitive digits running from `0` to `Z`.

If you specify a base of `0`, the function will make an effort to determine it. It'll default to base 10 except for a couple cases:

- If the number has a leading `0`, it will be octal (base 8)
- If the number has a leading `0x` or `0X`, it will be hex (base 16)

The locale might affect the behavior of these functions.

Return Value

Returns the converted value.

`endptr`, if not `NULL` is set to the first invalid character, or to the beginning of the string if no conversion was performed, or to the string terminal NUL if all characters were valid.

If there's overflow, one of these values will be returned: `LONG_MIN`, `LONG_MAX`, `LLONG_MIN`, `LLONG_MAX`, `ULONG_MAX`, `ULLONG_MAX`. And `errno` is set to `ERANGE`.

Example

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    // All output in decimal (base 10)

    printf("%ld\n", strtol("123", NULL, 0));      // 123
    printf("%ld\n", strtol("123", NULL, 10));     // 123
```

[3] https://en.wikipedia.org/wiki/Radix

```
    printf("%ld\n", strtol("101010", NULL, 2));   // binary, 42
    printf("%ld\n", strtol("123", NULL, 8));      // octal, 83
    printf("%ld\n", strtol("123", NULL, 16));     // hex, 291

    printf("%ld\n", strtol("0123", NULL, 0));     // octal, 83
    printf("%ld\n", strtol("0x123", NULL, 0));    // hex, 291

    char *badchar;
    long int x = strtol("   1234beej", &badchar, 0);

    printf("Value is %ld\n", x);               // Value is 1234
    printf("Bad chars at \"%s\"\n", badchar);  // Bad chars at "beej"
}
```

Output:

```
123
123
42
83
291
83
291
Value is 1234
Bad chars at "beej"
```

See Also

`atoi()`, `strtod()`, `setlocale()`, `strtoimax()`, `strtoumax()`

62.6 `rand()`

Return a pseudorandom number

Synopsis

```
#include <stdlib.h>

int rand(void);
```

Description

This gives us back a pseudorandom number in the range `0` to `RAND_MAX`, inclusive. (`RAND_MAX` will be at least 32767.)

If you want to force this to a certain range, the classic way to do this is to force it with the modulo operator `%`, although this introduces biases[4] if `RAND_MAX+1` is not a multiple of the number you're modding by. Dealing with this is out of scope for this guide.

[4] https://stackoverflow.com/questions/10984974/why-do-people-say-there-is-modulo-bias-when-using-a-random-number-generator

If you want to to make a floating point number between `0` and `1` inclusive, you can divide the result by `RAND_MAX`. Or `RAND_MAX+1` if you don't want to include `1`. But of course, there are out-of-scope problems with this, as well[5].

In short, `rand()` is a great way to get potentially poor random numbers with ease. Probably good enough for the game you're writing.

The spec elaborates:

> There are no guarantees as to the quality of the random sequence produced and some implementations are known to produce sequences with distressingly non-random low-order bits. Applications with particular requirements should use a generator that is known to be sufficient for their needs.

Your system probably has a good random number generator on it if you need a stronger source. Linux users have `getrandom()`, for example, and Windows has `CryptGenRandom()`.

For other more demanding random number work, you might find a library like the GNU Scientific Library[6] of use.

With most implementations, the numbers produced by `rand()` will be the same from run to run. To get around this, you need to start it off in a different place by passing a *seed* into the random number generator. You can do this with `srand()`.

Return Value

Returns a random number in the range `0` to `RAND_MAX`, inclusive.

Example

Note that all of these examples don't produce perfectly uniform distributions. But good enough for the untrained eye, and really common in general use when mediocre random number quality is acceptable.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    printf("RAND_MAX = %d\n", RAND_MAX);

    printf("0 to 9: %d\n", rand() % 10);

    printf("10 to 44: %d\n", rand() % 35 + 10);
    printf("0 to 0.99999: %f\n", rand() / ((float)RAND_MAX + 1));
    printf("10.5 to 15.7: %f\n", 10.5 + 5.2 * rand() / (float)RAND_MAX);
}
```

Output on my system:

```
RAND_MAX = 2147483647
0 to 9: 3
10 to 44: 21
0 to 0.99999: 0.783099
10.5 to 15.7: 14.651888
```

Example of seeding the RNG with the time:

[5] https://mumble.net/~campbell/2014/04/28/uniform-random-float
[6] https://www.gnu.org/software/gsl/doc/html/rng.html

www.dbooks.org

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
    // time(NULL) very likely returns the number of seconds since
    // January 1, 1970:

    srand(time(NULL));

    for (int i = 0; i < 5; i++)
        printf("%d\n", rand());
}
```

See Also

`srand()`

62.7 `srand()`

Seed the built-in pseudorandom number generator

Synopsis

```
#include <stdlib.h>

void srand(unsigned int seed);
```

Description

The dirty little secret of pseudorandom number generation is that they're completely deterministic. There's nothing random about them. They just look random.

If you use `rand()` and run your program several times, you might notice something *fishy*: they produce the same random numbers over and over again.

To mix it up, we need to give the pseudorandom number generator a new "starting point", if you will. We call that the *seed*. It's just a number, but it is used as the basic for subsequent number generation. Give a different seed, and you'll get a different sequence of random numbers. Give the same seed, and you'll get the same sequence of random numbers corresponding to it[7].

So if you call `srand(3490)` before you start generating numbers with `rand()`, you'll get the same sequence every time. `srand(37)` would also give you the same sequence every time, but it would be a different sequence than the one you got with `srand(3490)`.

But if you can't hardcode the seed (because that would give you the same sequence every time), how are you supposed to do this?

[7] Minecraft enthusiasts might recall that when generating a new world, they were given the option to enter a random number seed. That single value is used to generate that entire random world. And if your friend starts a world with the same seed you did, they'll get the same world you did.

It's really common to use the number of seconds since January 1, 1970 (this date is known as the *Unix epoch*[8]) to seed the generator. This sounds pretty arbitrary except for the fact that it's exactly the value most implementations return from the library call `time(NULL)`[9].

We'll do that in the example.

If you don't call `srand()`, it's as if you called `srand(1)`.

Return Value

Returns nothing!

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>    // for the time() call

int main(void)
{
    srand(time(NULL));

    for (int i = 0; i < 5; i++)
        printf("%d\n", rand() % 32);
}
```

Output:

```
4
20
22
14
9
```

Output from a subsequent run:

```
19
0
31
31
24
```

See Also

`rand()`, `time()`

62.8 `aligned_alloc()`

Allocate specifically-aligned memory

[8] https://en.wikipedia.org/wiki/Unix_time

[9] The C spec doesn't say exactly what `time(NULL)` will return, but the POSIX spec does! And virtually everyone returns exactly that: the number of seconds since epoch.

Synopsis

```
#include <stdlib.h>

void *aligned_alloc(size_t alignment, size_t size);
```

Description

Maybe you wanted `malloc()` or `calloc()` instead of this. But if you're sure you don't, read on!

Normally you don't have to think about this, since `malloc()` and `realloc()` both provide memory regions that are suitably aligned[10] for use with any data type.

But if you need a more specific alignment, you can specify it with this function.

When you're done using the memory region, be sure to free it with a call to `free()`.

Don't pass in `0` for the size. It probably won't do anything you want.

In case you're wondering, all dynamically-allocated memory is automatically freed by the system when the program ends. That said, it's considered to be *Good Form* to explicitly `free()` everything you allocate. This way other programmers don't think you were being sloppy.

Return Value

Returns a pointer to the newly-allocated memory, aligned as specified. Returns `NULL` if something goes wrong.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int main(void)
{
    int *p = aligned_alloc(256, 10 * sizeof(int));

    // Just for fun, let's convert to intptr_t and mod with 256
    // to make sure we're actually aligned on a 256-byte boundary.
    //
    // This is probably some kind of implementation-defined
    // behavior, but I'll bet it works.

    intptr_t ip = (intptr_t)p;

    printf("%ld\n", ip % 256);   // 0!

    // Free it up
    free(p);
}
```

[10] https://en.wikipedia.org/wiki/Data_structure_alignment

See Also

malloc(), calloc(), free()

62.9 calloc(), malloc()

Allocate memory for arbitrary use

Synopsis

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);

void *malloc(size_t size);
```

Description

Both of these functions allocate memory for general-purpose use. It will be aligned such that it's useable for storing any data type.

malloc() allocates exactly the specified number of bytes of memory in a contiguous block. The memory might be full of garbage data. (You can clear it with memset(), if you wish.)

calloc() is different in that it allocates space for nmemb objects of size bytes each. (You can do the same with malloc(), but you have to do the multiplication yourself.)

calloc() has an additional feature: it clears all the memory to 0.

So if you're planning to zero the memory anyway, calloc() is probably the way to go. If you're not, you can avoid that overhead by calling malloc().

When you're done using the memory region, free it with a call to free().

Don't pass in 0 for the size. It probably won't do anything you want.

In case you're wondering, all dynamically-allocated memory is automatically freed by the system when the program ends. That said, it's considered to be *Good Form* to explicitly free() everything you allocate. This way other programmers don't think you were being sloppy.

Return Value

Both functions return a pointer to the shiny, newly-allocated memory. Or NULL if something's gone awry.

Example

Comparison of malloc() and calloc() for allocating 5 ints:

```
#include <stdlib.h>

int main(void)
{
    // Allocate space for 5 ints
    int *p = malloc(5 * sizeof(int));

    p[0] = 12;
```

```
    p[1] = 30;

    // Allocate space for 5 ints
    // (Also clear that memory to 0)
    int *q = calloc(5, sizeof(int));

    q[0] = 12;
    q[1] = 30;

    // All done
    free(p);
    free(q);
}
```

See Also

`aligned_alloc()`, `free()`

62.10 `free()`

Free a memory region

Synopsis

```
#include <stdlib.h>

void free(void *ptr);
```

Description

You know that pointer you got back from `malloc()`, `calloc()`, or `aligned_alloc()`? You pass that pointer to `free()` to free the memory associated with it.

If you don't do this, the memory will stay allocated FOREVER AND EVER! (Well, until your program exits, anyway.)

Fun fact: `free(NULL)` does nothing. You can safely call that. Sometimes it's convenient.

Don't `free()` a pointer that's already been `free()`d. Don't `free()` a pointer that you didn't get back from one of the allocation functions. It would be *Bad*[11].

Return Value

Returns nothing!

Example

```
#include <stdlib.h>

```

[11] "Try to imagine all life as you know it stopping instantaneously and every molecule in your body exploding at the speed of light." —Egon Spengler

```
int main(void)
{
    // Allocate space for 5 ints
    int *p = malloc(5 * sizeof(int));

    p[0] = 12;
    p[1] = 30;

    // Free that space
    free(p);
}
```

See Also

malloc(), calloc(), aligned_alloc()

62.11 realloc()

Resize a previously allocated stretch of memory

Synopsis

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);
```

Description

This takes a pointer to some memory previously allocated with malloc() or calloc() and resizes it to the new size.

If the new size is smaller than the old size, any data larger than the new size is discarded.

If the new size is larger than the old size, the new larger part is uninitialized. (You can clear it with memset().)

Important note: the memory might move! If you resize, the system might need to relocate the memory to a larger continguous chunk. If this happens, realloc() will copy the old data to the new location for you.

Because of this, it's important to save the returned value to your pointer to update it to the new location if things move. (Also, be sure to error-check so that you don't overwrite your old pointer with NULL, leaking the memory.)

You can also relloc() memory allocated with aligned_alloc(), but it will potentially lose its alignment if the block is moved.

Return Value

Returns a pointer to the resized memory region. This might be equivalent to the ptr passed in, or it might be some other location.

Example

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    // Allocate space for 5 ints
    int *p = malloc(5 * sizeof(int));

    p[0] = 12;
    p[1] = 30;

    // Reallocate for 10 bytes
    int *new_p = realloc(p, 10 * sizeof(int));

    if (new_p == NULL) {
        printf("Error reallocing\n");
    } else {
        p = new_p;  // It's good; let's keep it
        p[7] = 99;
    }

    // All done
    free(p);
}
```

See Also

`malloc()`, `calloc()`

62.12 `abort()`

Abruptly end program execution

Synopsis

```c
#include <stdlib.h>

_Noreturn void abort(void);
```

Description

This ends program execution *abnormally* and immediately. Use this in rare, unexpected circumstances.

Open streams might not be flushed. Temporary files created might not be removed. Exit handlers are not called.

A non-zero exit status is returned to the environment.

On some systems, `abort()` might dump core[12], but this is outside the scope of the spec.

[12] https://en.wikipedia.org/wiki/Core_dump

You can cause the equivalent of an `abort()` by calling `raise(SIGABRT)`, but I don't know why you'd do that.

The only portable way to stop an `abort()` call midway is to use `signal()` to catch `SIGABRT` and then `exit()` in the signal handler.

Return Value

This function never returns.

Example

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int bad_thing = 1;

    if (bad_thing) {
        printf("This should never have happened!\n");
        fflush(stdout);  // Make sure the message goes out
        abort();
    }
}
```

On my system, this outputs:

```
This should never have happened!
zsh: abort (core dumped)  ./foo
```

See Also

`signal()`

62.13 `atexit()`, `at_quick_exit()`

Set up handlers to run when the program exits

Synopsis

```
#include <stdlib.h>

int atexit(void (*func)(void));

int at_quick_exit(void (*func)(void));
```

Description

When the program does a normal exit with `exit()` or returns from `main()`, it looks for previously-registered handlers to call on the way out. These handlers are registered with the `atexit()` call.

Think of it like, "Hey, when you're about to exit, do these extra things."

For the `quick_exit()` call, you can use the `at_quick_exit()` function to register handlers for that[13]. There's no crossover in handlers from `exit()` to `quick_exit()`, i.e. for a call to one, none of the other's handlers will fire.

You can register multiple handlers to fire—at least 32 handlers are supported by both `exit()` and `quick_exit()`.

The argument `func` to the functions looks a little weird—it's a pointer to a function to call. Basically just put the function name to call in there (without parentheses after). See the example, below.

If you call `atexit()` from inside your `atexit()` handler (or equivalent in your `at_quick_exit()` handler), it's unspecified if it will get called. So get them all registered before you exit.

When exiting, the functions will be called in the reverse order they were registered.

Return Value

These functions return `0` on success, or nonzero on failure.

Example

`atexit()`:

```
#include <stdio.h>
#include <stdlib.h>

void exit_handler_1(void)
{
    printf("Exit handler 1 called!\n");
}

void exit_handler_2(void)
{
    printf("Exit handler 2 called!\n");
}

int main(void)
{
    atexit(exit_handler_1);
    atexit(exit_handler_2);

    exit(0);
}
```

For the output:

```
Exit handler 2 called!
Exit handler 1 called!
```

And a similar example with `quick_exit()`:

```
#include <stdio.h>
#include <stdlib.h>

void exit_handler_1(void)
```

[13] `quick_exit()` differs from `exit()` in that open files might not be flushed and temporary files might not be removed.

```
{
    printf("Exit handler 1 called!\n");
}

void exit_handler_2(void)
{
    printf("Exit handler 2 called!\n");
}

int main(void)
{
    at_quick_exit(exit_handler_1);
    at_quick_exit(exit_handler_2);

    quick_exit(0);
}
```

See Also

`exit()`, `quick_exit()`

62.14 `exit()`, `quick_exit()`, `_Exit()`

Exit the currently-running program

Synopsis

```
#include <stdlib.h>

_Noreturn void exit(int status);

_Noreturn void quick_exit(int status);

_Noreturn void _Exit(int status);
```

Description

All these functions cause the program to exit, with various levels of cleanup performed.

`exit()` does the most cleanup and is the most normal exit.

`quick_exit()` is the second most.

`_Exit()` unceremoniously drops everything and ragequits on the spot.

Calling either of `exit()` or `quick_exit()` causes their respective `atexit()` or `at_quick_exit()` handlers to be called in the reverse order in which they were registered.

`exit()` will flush all streams and delete all temporary files.

`quick_exit()` or `_Exit()` might not perform that nicety.

`_Exit()` doesn't call any of the at-exit handlers, either.

For all functions, the exit `status` is returned to the environment.

Defined exit statuses are:

Status	Description
`EXIT_SUCCESS`	Typically returned when good things happen
`0`	Same as `EXIT_SUCCESS`
`EXIT_FAILURE`	Oh noes! Definitely failure!
Any positive value	Generally indicates another failure of some kind

OS X note: `quick_exit()` is not supported.

Return Value

None of these functions ever return.

Example

```
#include <stdlib.h>

int main(void)
{
    int contrived_exit_type = 1;

    switch(contrived_exit_type) {
        case 1:
            exit(EXIT_SUCCESS);

        case 2:
            // Not supported in OS X
            quick_exit(EXIT_SUCCESS);

        case 3:
            _Exit(2);
    }
}
```

See Also

`atexit()`, `at_quick_exit()`

62.15 `getenv()`

Get the value of an environment variable

Synopsis

```
#include <stdlib.h>

char *getenv(const char *name);
```

Description

The environment often provides variables that are set before the program run that you can access at runtime.

Of course the exact details are system dependent, but these variables are key/value pairs, and you can get the value by passing the key to `getenv()` as the `name` parameter.

You're not allowed to overwrite the string that's returned.

This is pretty limited in the standard, but your OS often provides better functionality. See the Environment Variables section for more details.

Return Value

Returns a pointer to the environment variable value, or `NULL` if the variable doesn't exist.

Example

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    printf("PATH is %s\n", getenv("PATH"));
}
```

Output (truncated in my case):

```
PATH is /usr/bin:/usr/local/bin:/usr/sbin:/home/beej/.cargo/bin [...]
```

62.16 system()

Run an external program

Synopsis

```c
#include <stdlib.h>

int system(const char *string);
```

Description

This will run an external program and then return to the caller.

The manner in which it runs the program is system-defined, but typically you can pass something to it just like you'd run on the command line, searching the `PATH`, etc.

Not all systems have this capability, but you can test for it by passing `NULL` to `system()` and seeing if it returns 0 (no command processor is available) or non-zero (a command processor is available! Yay!)

If you're getting user input and passing it to the `system()` call, be extremely careful to escape all special shell characters (everything that's not alphanumeric) with a backslash to keep a villain from running something you don't want them to.

Return Value

If NULL is passed, returns nonzero if a command processor is available (i.e. system() will work at all).

Otherwise returns an implementation-defined value.

Example

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    printf("Here's a directory listing:\n\n");

    system("ls -l");   // Run this command and return

    printf("\nAll done!\n");
}
```

Output:

```
Here's a directory listing:

total 92
drwxr-xr-x 3 beej beej  4096 Oct 14 21:38 bin
drwxr-xr-x 2 beej beej  4096 Dec 20 20:07 examples
-rwxr-xr-x 1 beej beej 16656 Feb 23 21:49 foo
-rw-rw-rw- 1 beej beej   155 Feb 23 21:49 foo.c
-rw-r--r-- 1 beej beej  1350 Jan 27 22:11 Makefile
-rw-r--r-- 1 beej beej  4644 Jan 18 09:12 README.md
drwxr-xr-x 3 beej beej  4096 Feb 23 20:21 src
drwxr-xr-x 6 beej beej  4096 Feb 21 20:24 stage
drwxr-xr-x 2 beej beej  4096 Sep 27 20:54 translations
drwxr-xr-x 2 beej beej  4096 Sep 27 20:54 website

All done!
```

62.17 bsearch()

Binary Search (maybe) an array of objects

Synopsis

```
#include <stdlib.h>

void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *, const void *));
```

Description

This crazy-looking function searches an array for a value.

It probably is a binary search or some fast, efficient search. But the spec doesn't really say.

However, the array must be sorted! So binary search seems likely.

- `key` is a pointer to the value to find.
- `base` is a pointer to the start of the array—the array must be sorted!
- `nmemb` is the number of elements in the array.
- `size` is the `sizeof` each element in the array.
- `compar` is a pointer to a function that will compare the key against other values.

The comparison function takes the key as the first argument and the value to compare against as the second. It should return a negative number if the key is less than the value, `0` if the key equals the value, and a positive number if the key is greater than the value.

This is commonly computed by taking the difference between the key and the value to be compared. If subtraction is supported.

The return value from the `strcmp()` function can be used for comparing strings.

Again, the array must be sorted according to the order of the comparison function before running `bsearch()`. Luckily for you, you can just call `qsort()` with the same comparison function to get this done.

It's a general-purpose function—it'll search any type of array for anything. The catch is you have to write the comparison function.

And that's not as scary as it looks. Jump down to the example

Return Value

The function returns a pointer to the found value, or `NULL` if it can't be found.

Example

```
#include <stdio.h>
#include <stdlib.h>

int compar(const void *key, const void *value)
{
    const int *k = key, *v = value;  // Need ints, not voids

    return *k - *v;
}

int main(void)
{
    int a[9] = {2, 6, 9, 12, 13, 18, 20, 32, 47};

    int *r, key;

    key = 12;  // 12 is in there
    r = bsearch(&key, a, 9, sizeof(int), compar);
    printf("Found %d\n", *r);

    key = 30;  // Won't find a 30
    r = bsearch(&key, a, 9, sizeof(int), compar);
    if (r == NULL)
        printf("Didn't find 30\n");
```

```

    // Searching with an unnamed key, pointer to 32
    r = bsearch(&(int){32}, a, 9, sizeof(int), compar);
    printf("Found %d\n", *r);  // Found it
}
```

Output:

```
Found 12
Didn't find 30
Found 32
```

See Also

`strcmp()`, `qsort()`

62.18 qsort()

Quicksort (maybe) some data

Synopsis

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Description

This function will quicksort (or some other sort, probably speedy) an array of data in-place[14].

Like `bsearch()`, it's data-agnostic. Any data for which you can define a relative ordering can be sorted, whether `int`s, `struct`s, or anything else.

Also like `bsearch()`, you have to give a comparison function to do the actual compare.

- `base` is a pointer to the start of the array to be sorted.
- `nmemb` is the number of elements in the array.
- `size` is the `sizeof` each element.
- `compar` is a pointer to the comparison function.

The comparison function takes pointers to two elements of the array as arguments and compares them. It should return a negative number if the first argument is less than the second, `0` if they are equal, and a positive number if the first argument is greater than the second.

This is commonly computed by taking the difference between the first argument and the second. If subtraction is supported.

The return value from the `strcmp()` function can provide sort order for strings.

If you have to sort a `struct`, just subtract the specific field you want to sort by.

This comparison function can be used by `bsearch()` to do searches after the list is sorted.

To reverse the sort, subtract the second argument from the first, i.e. negate the return value from `compar()`.

[14] "In-place" meaning that the original array will hold the results; no new array is allocated.

Return Value

Returns nothing!

Example

```c
#include <stdio.h>
#include <stdlib.h>

int compar(const void *elem0, const void *elem1)
{
    const int *x = elem0, *y = elem1;  // Need ints, not voids

    if (*x > *y) return 1;
    if (*x < *y) return -1;
    return 0;
}

int main(void)
{
    int a[9] = {14, 2, 3, 17, 10, 8, 6, 1, 13};

    // Sort the list

    qsort(a, 9, sizeof(int), compar);

    // Print sorted list

    for (int i = 0; i < 9; i++)
        printf("%d ", a[i]);

    putchar('\n');

    // Use the same compar() function to binary search
    // for 17 (passed in as an unnamed object)

    int *r = bsearch(&(int){17}, a, 9, sizeof(int), compar);
    printf("Found %d!\n", *r);
}
```

Output:

```
1 2 3 6 8 10 13 14 17
Found 17!
```

See Also

`strcmp()`, `bsearch()`

62.19 `abs()`, `labs()`, `llabs()`

Compute the absolute value of an integer

Synopsis

```
#include <stdlib.h>

int abs(int j);

long int labs(long int j);

long long int llabs(long long int j);
```

Description

Compute the absolute value of `j`. If you don't remember, that's how far from zero `j` is.

In other words, if `j` is negative, return it as a positive. If it's positive, return it as a positive. Always be positive. Enjoy life.

If the result cannot be represented, the behavior is undefined. Be especially aware of the upper half of unsigned numbers.

Return Value

Returns the absolute value of `j`, $|j|$.

Example

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    printf("|-2| = %d\n", abs(-2));
    printf("|4|  = %d\n", abs(4));
}
```

Output:

```
|-2| = 2
|4|  = 4
```

See Also

`fabs()`

62.20 `div()`, `ldiv()`, `lldiv()`

Compute the quotient and remainder of two numbers

Synopsis

```
#include <stdlib.h>
```

```
div_t div(int numer, int denom);

ldiv_t ldiv(long int numer, long int denom);

lldiv_t lldiv(long long int numer, long long int denom);
```

Description

These functions get you the quotient and remainder of a pair of numbers in one go.

They return a structure that has two fields, `quot`, and `rem`, the types of which match types of `numer` and `denom`. Note how each function returns a different variant of `div_t`.

These `div_t` variants are equivalent to the following:

```
typedef struct {
    int quot, rem;
} div_t;

typedef struct {
    long int quot, rem;
} ldiv_t;

typedef struct {
    long long int quot, rem;
} lldiv_t;
```

Why use these instead of the division operator?

The C99 Rationale says:

> Because C89 had implementation-defined semantics for division of signed integers when negative operands were involved, `div` and `ldiv`, and `lldiv` in C99, were invented to provide well-specified semantics for signed integer division and remainder operations. The semantics were adopted to be the same as in Fortran. Since these functions return both the quotient and the remainder, they also serve as a convenient way of efficiently modeling underlying hardware that computes both results as part of the same operation. Table 7.2 summarizes the semantics of these functions.

Indeed, K&R2 (C89) says:

> The direction of truncation for `/` and the sign of the result for `%` are machine-dependent for negative operands [...]

The Rationale then goes on to spell out what the signs of the quotient and remainder will be given the signs of a numerator and denominator when using the `div()` functions:

numer	denom	quot	rem
+	+	+	+
−	+	−	−
+	−	−	+
−	−	+	−

Return Value

A `div_t`, `ldiv_t`, or `lldiv_t` structure with the `quot` and `rem` fields loaded with the quotient and remainder of the operation of `numer/denom`.

Example

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    div_t d = div(64, -7);

    printf("64 / -7 = %d\n", d.quot);
    printf("64 %% -7 = %d\n", d.rem);
}
```

Output:

```
64 / -7 = -9
64 % -7 = 1
```

See Also

`fmod()`, `remainder()`

62.21 `mblen()`

Return the number of bytes in a multibyte character

Synopsis

```
#include <stdlib.h>

int mblen(const char *s, size_t n);
```

Description

If you have a multibyte character in a string, this will tell you how many bytes long it is.

`n` is the maximum number of bytes `mblen()` will scan before giving up.

If `s` is a `NULL` pointer, tests if this encoding has state dependency, as noted in the return value, below. It also resets the state, if there is one.

The behavior of this function is influenced by the locale.

Return Value

Returns the number of bytes used to encode this character, or `-1` if there is no valid multibyte character in the next `n` bytes.

Or, if `s` is NULL, returns true if this encoding has state dependency.

Example

For the example, I used my extended character set to put Unicode characters in the source. If this doesn't work for you, use the `\uXXXX` escape.

```c
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

int main(void)
{
    setlocale(LC_ALL, "");

    printf("State dependency: %d\n", mblen(NULL, 0));
    printf("Bytes for €: %d\n", mblen("€", 5));
    printf("Bytes for \u00e9: %d\n", mblen("\u00e9", 5));  // \u00e9 == é
    printf("Bytes for &: %d\n", mblen("&", 5));
}
```

Output (in my case, the encoding is UTF-8, but your mileage may vary):

```
State dependency: 0
Bytes for €: 3
Bytes for é: 2
Bytes for &: 1
```

See Also

`mbtowc()`, `mbstowcs()`), `setlocale()`

62.22 `mbtowc()`

Convert a multibyte character to a wide character

Synopsis

```c
#include <stdlib.h>

int mbtowc(wchar_t * restrict pwc, const char * restrict s, size_t n);
```

Description

If you have a multibyte character, this function will convert it to a wide character and stored at the address pointed to by `pwc`. Up to `n` bytes of the multibyte character will be analyzed.

If `pwc` is `NULL`, the resulting character will not be stored. (Useful for just getting the return value.)

If `s` is a `NULL` pointer, tests if this encoding has state dependency, as noted in the return value, below. It also resets the state, if there is one.

The behavior of this function is influenced by the locale.

Return Value

Returns the number of bytes used in the encoded wide character, or `-1` if there is no valid multibyte character in the next `n` bytes.

Returns `0` if `s` points to the NUL character.

Or, if s is NULL, returns true if this encoding has state dependency.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>

int main(void)
{
    setlocale(LC_ALL, "");

    printf("State dependency: %d\n", mbtowc(NULL, NULL, 0));

    wchar_t wc;
    int bytes;

    bytes = mbtowc(&wc, "€", 5);

    printf("L'%lc' takes %d bytes as multibyte char '€'\n", wc, bytes);
}
```

Output on my system:

```
State dependency: 0
L'€' takes 3 bytes as multibyte char '€'
```

See Also

mblen(), mbstowcs(), wcstombs(), setlocale()

62.23 wctomb()

Convert a wide character to a multibyte character

Synopsis

```
#include <stdlib.h>

int wctomb(char *s, wchar_t wc);
```

Description

If you have your hands on a wide character, you can use this to make it multibyte.

The wide character wc is stored as a multibyte character in the string pointed to by s. The buffer s points to should be at least MB_CUR_MAX characters long. Note that MB_CUR_MAX changes with locale.

If wc is a NUL wide character, a NUL is stored in s after the bytes needed to reset the shift state (if any).

If s is a NULL pointer, tests if this encoding has state dependency, as noted in the return value, below. It also resets the state, if there is one.

The behavior of this function is influenced by the locale.

Return Value

Returns the number of bytes used in the encoded multibyte character, or `-1` if `wc` does not correspond to any valid multibyte character.

Or, if `s` is NULL, returns true if this encoding has state dependency.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>

int main(void)
{
    setlocale(LC_ALL, "");

    printf("State dependency: %d\n", mbtowc(NULL, NULL, 0));

    int bytes;
    char mb[MB_CUR_MAX + 1];

    bytes = wctomb(mb, L'€');
    mb[bytes] = '\0';

    printf("L'€' takes %d bytes as multibyte char '%s'\n", bytes, mb);
}
```

Output on my system:

```
State dependency: 0
L'€' takes 3 bytes as multibyte char '€'
```

See Also

`mbtowc()`, `mbstowcs()`, `wcstombs()`, `setlocale()`

62.24 `mbstowcs()`

Convert a multibyte string to a wide character string

Synopsis

```
#include <stdlib.h>

size_t mbstowcs(wchar_t * restrict pwcs, const char * restrict s, size_t n);
```

Description

If you have a multibyte string (AKA a regular string), you can convert it wto a wide character string with this function.

At most `n` wide characters are written to the destination `pwcs` from the source `s`.

A NUL character is stored as a wide NUL character.

Non-portable POSIX extension: if you're using a POSIX-complaint library, this function allows `pwcs` to be `NULL` if you're only interested in the return value. Most notably, this will give you the number of characters in a multibyte string (as opposed to `strlen()` which counts the bytes.)

Return Value

Returns the number of wide characters written to the destination `pwcs`.

If an invalid multibyte character was found, returns `(size_t)(-1)`.

If the return value is `n`, it means the result was *not* NUL-terminated.

Example

This source uses an extended character set. If your compiler doesn't support it, you'll have to replace them with `\u` escapes.

```c
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <string.h>

int main(void)
{
    setlocale(LC_ALL, "");

    wchar_t wcs[128];
    char *s = "€200 for this spoon?";  // 20 characters

    size_t char_count, byte_count;

    char_count = mbstowcs(wcs, s, 128);
    byte_count = strlen(s);

    printf("Wide string: L\"%ls\"\n", wcs);
    printf("Char count : %zu\n", char_count);    // 20
    printf("Byte count : %zu\n\n", byte_count);  // 22 on my system

    // POSIX Extension that allows you to pass NULL for
    // the destination so you can just use the return
    // value (which is the character count of the string,
    // if no errors have occurred)

    s = "§¶°±π€•";  // 7 characters

    char_count = mbstowcs(NULL, s, 0);  // POSIX-only, nonportable
    byte_count = strlen(s);

```

```
    printf("Multibyte str: \"%s\"\n", s);
    printf("Char count   : %zu\n", char_count);  // 7
    printf("Byte count   : %zu\n", byte_count);  // 16 on my system
}
```

Output on my system (byte count will depend on your encoding):

```
Wide string: L"€200 for this spoon?"
Char count : 20
Byte count : 22

Multibyte str: "§¶°±π€•"
Char count   : 7
Byte count   : 16
```

See Also

`mblen()`, `mbtowc()`, `wcstombs()`, `setlocale()`

62.25 `wcstombs()`

Convert a wide character string to a multibyte string

Synopsis

```
#include <stdlib.h>

size_t wcstombs(char * restrict s, const wchar_t * restrict pwcs, size_t n);
```

Description

If you have a wide character string and you want it as multibyte string, this is the function for you!

It'll take the wide characters pointed to by `pwcs` and convert them to multibyte characters stored in `s`. No more than `n` bytes will be written to `s`.

Non-portable POSIX extension: if you're using a POSIX-complaint library, this function allows `s` to be `NULL` if you're only interested in the return value. Most notably, this will give you the number of bytes needed to encode the wide characters in a multibyte string.

Return Value

Returns the number of bytes written to `s`, or `(size_t)(-1)` if one of the characters can't be encoded into a multibyte string.

If the return value is `n`, it means the result was *not* NUL-terminated.

Example

This source uses an extended character set. If your compiler doesn't support it, you'll have to replace them with `\u` escapes.

```c
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <string.h>

int main(void)
{
    setlocale(LC_ALL, "");

    char mbs[128];
    wchar_t *wcs = L"€200 for this spoon?";  // 20 characters

    size_t byte_count;

    byte_count = wcstombs(mbs, wcs, 128);

    printf("Wide string: L\"%ls\"\n", wcs);
    printf("Multibyte  : \"%s\"\n", mbs);
    printf("Byte count : %zu\n\n", byte_count);  // 22 on my system

    // POSIX Extension that allows you to pass NULL for
    // the destination so you can just use the return
    // value (which is the character count of the string,
    // if no errors have occurred)

    wcs = L"§¶°±π€•";  // 7 characters

    byte_count = wcstombs(NULL, wcs, 0);  // POSIX-only, nonportable

    printf("Wide string: L\"%ls\"\n", wcs);
    printf("Byte count : %zu\n", byte_count);  // 16 on my system
}
```

Output on my system (byte count will depend on your encoding):

```
Wide string: L"€200 for this spoon?"
Multibyte  : "€200 for this spoon?"
Byte count : 22

Wide string: L"§¶°±π€•"
Byte count : 16
```

See Also

`mblen()`, `wctomb()`, `mbstowcs()`, `setlocale()`

Chapter 63

<stdnoreturn.h> Macros for Non-Returning Functions

This header provides a macro `noreturn` that is a handy alias for `_Noreturn`.

Use this macro to indicate to the compiler that a function will never return to the caller. It's undefined behavior if the so-marked function does return.

Here's a usage example:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void foo(void) // This function should never return!
{
    printf("Happy days\n");

    exit(1);            // And it doesn't return--it exits here!
}

int main(void)
{
    foo();
}
```

That's all there is to it.

www.dbooks.org

Chapter 64

<string.h> String Manipulation

Function	Description
memchr()	Find the first occurrence of a character in memory.
memcmp()	Compare two regions of memory.
memcpy()	Copy a region of memory to another.
memmove()	Move a (potentially overlapping) region of memory.
memset()	Set a region of memory to a value.
strcat()	Concatenate (join) two strings together.
strchr()	Find the first occurrence of a character in a string.
strcmp()	Compare two strings.
strcoll()	Compare two strings accounting for locale.
strcpy()	Copy a string.
strcspn()	Find length of a string not consisting of a set of characters.
strerror()	Return a human-readable error message for a given code.
strlen()	Return the length of a string.
strncat()	Concatenate (join) two strings, length-limited.
strncmp()	Compare two strings, length-limited.
strncpy()	Copy two strings, length-limited.
strpbrk()	Search a string for one of a set of character.
strrchr()	Find the last occurrence of a character in a string.
strspn()	Find length of a string consisting of a set of characters.
strstr()	Find a substring in a string.
strtok()	Tokenize a string.
strxfrm()	Prepare a string for comparison as if by strcoll().

As has been mentioned earlier in the guide, a string in C is a sequence of bytes in memory, terminated by a NUL character ('\0'). The NUL at the end is important, since it lets all these string functions (and printf() and puts() and everything else that deals with a string) know where the end of the string actually is.

Fortunately, when you operate on a string using one of these many functions available to you, they add the NUL terminator on for you, so you actually rarely have to keep track of it yourself. (Sometimes you do, especially if you're building a string from scratch a character at a time or something.)

In this section you'll find functions for pulling substrings out of strings, concatenating strings together, getting the length of a string, and so forth and so on.

64.1 memcpy(), memmove()

Copy bytes of memory from one location to another

Synopsis

```
#include <string.h>

void *memcpy(void * restrict s1, const void * restrict s2, size_t n);

void *memmove(void *s1, const void *s2, size_t n);
```

Description

These functions copy memory—as many bytes as you want! From source to destination!

The main difference between the two is that `memcpy()` cannot safely copy overlapping memory regions, whereas `memmove()` can.

On the one hand, I'm not sure why you'd want to ever use `memcpy()` instead of `memmove()`, but I'll bet it's possibly more performant.

The parameters are in a particular order: destination first, then source. I remember this order because it behaves like an "=" assignment: the destination is on the left.

Return Value

Both functions return whatever you passed in for parameter `s1` for your convenience.

Example

```
#include <string.h>

int main(void)
{
    char s[100] = "Goats";
    char t[100];

    memcpy(t, s, 6);       // Copy non-overlapping memory

    memmove(s + 2, s, 6);  // Copy overlapping memory
}
```

See Also

`strcpy()`, `strncpy()`

64.2 strcpy(), strncpy()

Copy a string

Synopsis

```
#include <string.h>

char *strcpy(char *dest, char *src);

char *strncpy(char *dest, char *src, size_t n);
```

Description

These functions copy a string from one address to another, stopping at the NUL terminator on the `src`string.

`strncpy()` is just like `strcpy()`, except only the first `n` characters are actually copied. Beware that if you hit the limit, `n` before you get a NUL terminator on the `src` string, your `dest` string won't be NUL-terminated. Beware! BEWARE!

(If the `src` string has fewer than `n` characters, it works just like `strcpy()`.)

You can terminate the string yourself by sticking the `'\0'` in there yourself:

```
char s[10];
char foo = "My hovercraft is full of eels."; // more than 10 chars

strncpy(s, foo, 9); // only copy 9 chars into positions 0-8
s[9] = '\0';        // position 9 gets the terminator
```

Return Value

Both functions return `dest` for your convenience, at no extra charge.

Example

```
#include <string.h>

int main(void)
{
    char *src = "hockey hockey hockey hockey hockey hockey hockey hockey";
    char dest[20];

    int len;

    strcpy(dest, "I like "); // dest is now "I like "

    len = strlen(dest);

    // tricky, but let's use some pointer arithmetic and math to append
    // as much of src as possible onto the end of dest, -1 on the length to
    // leave room for the terminator:
    strncpy(dest+len, src, sizeof(dest)-len-1);

    // remember that sizeof() returns the size of the array in bytes
    // and a char is a byte:
    dest[sizeof(dest)-1] = '\0'; // terminate

    // dest is now:       v null terminator
```

```
    // I like hockey hocke
    // 012345678901234567890123 45
}
```

See Also

memcpy(), strcat(), strncat()

64.3 strcat(), strncat()

Concatenate two strings into a single string

Synopsis

```
#include <string.h>

int strcat(const char *dest, const char *src);

int strncat(const char *dest, const char *src, size_t n);
```

Description

"Concatenate", for those not in the know, means to "stick together". These functions take two strings, and stick them together, storing the result in the first string.

These functions don't take the size of the first string into account when it does the concatenation. What this means in practical terms is that you can try to stick a 2 megabyte string into a 10 byte space. This will lead to unintended consequences, unless you intended to lead to unintended consequences, in which case it will lead to intended unintended consequences.

Technical banter aside, your boss and/or professor will be irate.

If you want to make sure you don't overrun the first string, be sure to check the lengths of the strings first and use some highly technical subtraction to make sure things fit.

You can actually only concatenate the first n characters of the second string by using strncat() and specifying the maximum number of characters to copy.

Return Value

Both functions return a pointer to the destination string, like most of the string-oriented functions.

Example

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char dest[30] = "Hello";
    char *src = ", World!";
    char numbers[] = "12345678";
```

```

    printf("dest before strcat: \"%s\"\n", dest); // "Hello"

    strcat(dest, src);
    printf("dest after strcat:  \"%s\"\n", dest); // "Hello, world!"

    strncat(dest, numbers, 3); // strcat first 3 chars of numbers
    printf("dest after strncat: \"%s\"\n", dest); // "Hello, world!123"
}
```

Notice I mixed and matched pointer and array notation there with `src` and `numbers`; this is just fine with string functions.

See Also

`strlen()`

64.4 `strcmp()`, `strncmp()`, `memcmp()`

Compare two strings or memory regions and return a difference

Synopsis

```
#include <string.h>

int strcmp(const char *s1, const char *s2);

int strncmp(const char *s1, const char *s2, size_t n);

int memcmp(const void *s1, const void *s2, size_t n);
```

Description

All these functions compare chunks of bytes in memory.

`strcmp()` and `strncmp()` operate on NUL-terminated strings, whereas `memcmp()` will compare the number of bytes you specify, brazenly ignoring any NUL characters it finds along the way.

`strcmp()` compares the entire string down to the end, while `strncmp()` only compares the first `n` characters of the strings.

It's a little funky what they return. Basically it's a difference of the strings, so if the strings are the same, it'll return zero (since the difference is zero). It'll return non-zero if the strings differ; basically it will find the first mismatched character and return less-than zero if that character in `s1` is less than the corresponding character in `s2`. It'll return greater-than zero if that character in `s1` is greater than that in `s2`.

So if they return `0`, the comparison was equal (i.e. the difference was `0`.)

These functions can be used as comparison functions for `qsort()` if you have an array of `char*`s you want to sort.

Return Value

Returns zero if the strings or memory are the same, less-than zero if the first different character in `s1` is less than that in `s2`, or greater-than zero if the first difference character in `s1` is greater than than in `s2`.

Example

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *s1 = "Muffin";
    char *s2 = "Muffin Sandwich";
    char *s3 = "Muffin";

    int r1 = strcmp("Biscuits", "Kittens");
    printf("%d\n", r1); // prints < 0 since 'B' < 'K'

    int r2 = strcmp("Kittens", "Biscuits");
    printf("%d\n", r2); // prints > 0 since 'K' > 'B'

    if (strcmp(s1, s2) == 0)
        printf("This won't get printed because the strings differ\n");

    if (strcmp(s1, s3) == 0)
        printf("This will print because s1 and s3 are the same\n");

    // this is a little weird...but if the strings are the same, it'll
    // return zero, which can also be thought of as "false". Not-false
    // is "true", so (!strcmp()) will be true if the strings are the
    // same. yes, it's odd, but you see this all the time in the wild
    // so you might as well get used to it:

    if (!strcmp(s1, s3))
        printf("The strings are the same!\n");

    if (!strncmp(s1, s2, 6))
        printf("The first 6 characters of s1 and s2 are the same\n");
}
```

See Also

`memcmp()`, `qsort()`

64.5 `strcoll()`

Compare two strings accounting for locale

Synopsis

```
#include <string.h>

int strcoll(const char *s1, const char *s2);
```

Description

This is basically `strcmp()`, except that it handles accented characters better depending on the locale.

For example, my `strcmp()` reports that the character "é" (with accent) is greater than "f". But that's hardly useful for alphabetizing.

By setting the `LC_COLLATE` locale value (either by name or via `LC_ALL`), you can have `strcoll()` sort in a way that's more meaningful by the current locale. For example, by having "é" appear sanely *before* "f".

It's also a lot slower than `strcmp()` so use it only if you have to. See `strxfrm()` for a potential speedup.

Return Value

Like the other string comparison functions, `strcoll()` returns a negative value if `s1` is less than `s2`, or a positive value if `s1` is greater than `s2`. Or `0` if they are equal.

Example

```
#include <stdio.h>
#include <string.h>
#include <locale.h>

int main(void)
{
    setlocale(LC_ALL, "");

    // If your source character set doesn't support "é" in a string
    // you can replace it with `\u00e9`, the Unicode code point
    // for "é".

    printf("%d\n", strcmp("é", "f"));   // Reports é > f, yuck.
    printf("%d\n", strcoll("é", "f"));  // Reports é < f, yay!
}
```

See Also

`strcmp()`

64.6 `strxfrm()`

Transform a string for comparing based on locale

Synopsis

```
#include <string.h>

size_t strxfrm(char * restrict s1, const char * restrict s2, size_t n);
```

Description

This is a strange little function, so bear with me.

Firstly, if you haven't done so, get familiar with `strcoll()` because this is closely related to that.

OK! Now that you're back, you can think of `strxfrm()` as the first part of the `strcoll()` internals. Basically, `strcoll()` has to transform a string into a form that can be compared with `strcmp()`. And it does this with `strxfrm()` for both strings every time you call it.

`strxform()` takes string `s2` and transforms it (readies it for `strcmp()`) storing the result in `s1`. It writes no more than `n` bytes, protecting us from terrible buffer overflows.

But hang on—there's another mode! If you pass `NULL` for `s1` and `0` for `n`, it will return the number of bytes that the transformed string *would have* used[1]. This is useful if you need to allocate some space to hold the transformed string before you `strcmp()` it against another.

What I'm getting at, not to be too blunt, is that `strcoll()` is slow compared to `strcmp()`. It does a lot of extra work running `strxfrm()` on all its strings.

In fact, we can see how it works by writing our own like this:

```
1  int my_strcoll(char *s1, char *s2)
2  {
3      // Use n = 0 to just get the lengths of the transformed strings
4      int len1 = strxfrm(NULL, s1, 0) + 1;
5      int len2 = strxfrm(NULL, s2, 0) + 1;
6
7      // Allocate enough room for each
8      char *d1 = malloc(len1);
9      char *d2 = malloc(len2);
10
11     // Transform the strings for comparison
12     strxfrm(d1, s1, len1);
13     strxfrm(d2, s2, len2);
14
15     // Compare the transformed strings
16     int result = strcmp(d1, d2);
17
18     // Free up the transformed strings
19     free(d2);
20     free(d1);
21
22     return result;
23 }
```

You see on lines 12, 13, and 16, above how we transform the two input strings and then call `strcmp()` on the result.

So why do we have this function? Can't we just call `strcoll()` and be done with it?

[1] It always returns the number of bytes the transformed string took, but in this case because `s1` was `NULL`, it doesn't actually write a transformed string.

The idea is that if you have one string that you're going to be comparing against a whole lot of other ones, maybe you just want to transform that string one time, then use the faster `strcmp()` saving yourself a bunch of the work we had to do in the function, above.

We'll do that in the example.

Return Value

Returns the number of bytes in the transformed sequence. If the value is greater than `n`, the results in `s1` are meaningless.

Example

```
#include <stdio.h>
#include <string.h>
#include <locale.h>
#include <stdlib.h>

// Transform a string for comparison, returning a malloc'd
// result
char *get_xfrm_str(char *s)
{
    int len = strxfrm(NULL, s, 0) + 1;
    char *d = malloc(len);

    strxfrm(d, s, len);

    return d;
}

// Does half the work of a regular strcoll() because the second
// string arrives already transformed.
int half_strcoll(char *s1, char *s2_transformed)
{
    char *s1_transformed = get_xfrm_str(s1);

    int result = strcmp(s1_transformed, s2_transformed);

    free(s1_transformed);

    return result;
}

int main(void)
{
    setlocale(LC_ALL, "");

    // Pre-transform the string to compare against
    char *s = get_xfrm_str("éfg");

    // Repeatedly compare against "éfg"
    printf("%d\n", half_strcoll("fgh", s));  // "fgh" > "éfg"
    printf("%d\n", half_strcoll("àbc", s));  // "àbc" < "éfg"
    printf("%d\n", half_strcoll("ĥij", s));  // "ĥij" > "éfg"
```

```

    free(s);
}
```

See Also

strcoll()

64.7 strchr(), strrchr(), memchr()

Find a character in a string

Synopsis

```
#include <string.h>

char *strchr(char *str, int c);

char *strrchr(char *str, int c);

void *memchr(const void *s, int c, size_t n);
```

Description

The functions strchr() and strrchr find the first or last occurrence of a letter in a string, respectively. (The extra "r" in strrchr() stands for "reverse"–it looks starting at the end of the string and working backward.) Each function returns a pointer to the char in question, or NULL if the letter isn't found in the string.

memchr() is similar, except that instead of stopping on the first NUL character, it continues searching for however many bytes you specify.

Quite straightforward.

One thing you can do if you want to find the next occurrence of the letter after finding the first, is call the function again with the previous return value plus one. (Remember pointer arithmetic?) Or minus one if you're looking in reverse. Don't accidentally go off the end of the string!

Return Value

Returns a pointer to the occurrence of the letter in the string, or NULL if the letter is not found.

Example

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    // "Hello, world!"
    //       ^  ^   ^
    //       A  B   C

```

```
    char *str = "Hello, world!";
    char *p;

    p = strchr(str, ',');       // p now points at position A
    p = strrchr(str, 'o');      // p now points at position B

    p = memchr(str, '!', 13);   // p now points at position C

    // repeatedly find all occurrences of the letter 'B'
    str = "A BIG BROWN BAT BIT BEEJ";

    for(p = strchr(str, 'B'); p != NULL; p = strchr(p + 1, 'B')) {
        printf("Found a 'B' here: %s\n", p);
    }
}
```

Output:

```
Found a 'B' here: BIG BROWN BAT BIT BEEJ
Found a 'B' here: BROWN BAT BIT BEEJ
Found a 'B' here: BAT BIT BEEJ
Found a 'B' here: BIT BEEJ
Found a 'B' here: BEEJ
```

64.8 strspn(), strcspn()

Return the length of a string consisting entirely of a set of characters, or of not a set of characters

Synopsis

```
#include <string.h>

size_t strspn(char *str, const char *accept);

size_t strcspn(char *str, const char *reject);
```

Description

`strspn()` will tell you the length of a string consisting entirely of the set of characters in `accept`. That is, it starts walking down `str` until it finds a character that is *not* in the set (that is, a character that is not to be accepted), and returns the length of the string so far.

`strcspn()` works much the same way, except that it walks down `str` until it finds a character in the `reject` set (that is, a character that is to be rejected.) It then returns the length of the string so far.

Return Value

The length of the string consisting of all characters in `accept` (for `strspn()`), or the length of the string consisting of all characters except `reject` (for `strcspn()`).

Example

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char str1[] = "a banana";
    char str2[] = "the bolivian navy on maenuvers in the south pacific";
    int n;

    // how many letters in str1 until we reach something that's not a vowel?
    n = strspn(str1, "aeiou");
    printf("%d\n", n);  // n == 1, just "a"

    // how many letters in str1 until we reach something that's not a, b,
    // or space?
    n = strspn(str1, "ab ");
    printf("%d\n", n);  // n == 4, "a ba"

    // how many letters in str2 before we get a "y"?
    n = strcspn(str2, "y");
    printf("%d\n", n);  // n = 16, "the bolivian nav"
}
```

See Also

`strchr()`, `strrchr()`

64.9 `strpbrk()`

Search a string for one of a set of characters

Synopsis

```
#include <string.h>

char *strpbrk(const char *s1, const char *s2);
```

Description

This function searches string `s1` for any of the characters that are found in string `s2`.

It's just like how `strchr()` searches for a specific character in a string, except it will match *any* of the characters found in `s2`.

Think of the power!

Return Value

Returns a pointer to the first character matched in `s1`, or NULL if the string isn't found.

Example

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    //  p points here after strpbrk
    //              v
    char *s1 = "Hello, world!";
    char *s2 = "dow!";  // Match any of these chars

    char *p = strpbrk(s1, s2);  // p points to the o

    printf("%s\n", p);  // "o, world!"
}
```

See Also

`strchr()`, `memchr()`

64.10 `strstr()`

Find a string in another string

Synopsis

```
#include <string.h>

char *strstr(const char *str, const char *substr);
```

Description

Let's say you have a big long string, and you want to find a word, or whatever substring strikes your fancy, inside the first string. Then `strstr()` is for you! It'll return a pointer to the `substr` within the `str`!

Return Value

You get back a pointer to the occurrence of the `substr` inside the `str`, or `NULL` if the substring can't be found.

Example

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *str = "The quick brown fox jumped over the lazy dogs.";
    char *p;

```

```
    p = strstr(str, "lazy");
    printf("%s\n", p == NULL? "null": p); // "lazy dogs."

    // p is NULL after this, since the string "wombat" isn't in str:
    p = strstr(str, "wombat");
    printf("%s\n", p == NULL? "null": p); // "null"
}
```

See Also

`strchr()`, `strrchr()`, `strspn()`, `strcspn()`

64.11 `strtok()`

Tokenize a string

Synopsis

```
#include <string.h>

char *strtok(char *str, const char *delim);
```

Description

If you have a string that has a bunch of separators in it, and you want to break that string up into individual pieces, this function can do it for you.

The usage is a little bit weird, but at least whenever you see the function in the wild, it's consistently weird.

Basically, the first time you call it, you pass the string, `str` that you want to break up in as the first argument. For each subsequent call to get more tokens out of the string, you pass `NULL`. This is a little weird, but `strtok()` remembers the string you originally passed in, and continues to strip tokens off for you.

Note that it does this by actually putting a NUL terminator after the token, and then returning a pointer to the start of the token. So the original string you pass in is destroyed, as it were. If you need to preserve the string, be sure to pass a copy of it to `strtok()` so the original isn't destroyed.

Return Value

A pointer to the next token. If you're out of tokens, `NULL` is returned.

Example

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    // break up the string into a series of space or
    // punctuation-separated words
    char str[] = "Where is my bacon, dude?";
```

```
    char *token;

    // Note that the following if-do-while construct is very very
    // very very very common to see when using strtok().

    // grab the first token (making sure there is a first token!)
    if ((token = strtok(str, ".,?! ")) != NULL) {
        do {
            printf("Word: \"%s\"\n", token);

            // now, the while continuation condition grabs the
            // next token (by passing NULL as the first param)
            // and continues if the token's not NULL:
        } while ((token = strtok(NULL, ".,?! ")) != NULL);
    }
}
```

Output:

```
Word: "Where"
Word: "is"
Word: "my"
Word: "bacon"
Word: "dude"
```

See Also

`strchr()`, `strrchr()`, `strspn()`, `strcspn()`

64.12 `memset()`

Set a region of memory to a certain value

Synopsis

```
#include <string.h>

void *memset(void *s, int c, size_t n);
```

Description

This function is what you use to set a region of memory to a particular value, namely `c` converted into `unsigned char`.

The most common usage is to zero out an array or `struct`.

Return Value

`memset()` returns whatever you passed in as `s` for happy convenience.

Example

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    struct banana {
        float ripeness;
        char *peel_color;
        int grams;
    };

    struct banana b;

    memset(&b, 0, sizeof b);

    printf("%d\n", b.ripeness == 0.0);     // True
    printf("%d\n", b.peel_color == NULL);  // True
    printf("%d\n", b.grams == 0);          // True
}
```

See Also

`memcpy()`, `memmove()`

64.13 `strerror()`

Get a string version of an error number

Synopsis

```
#include <string.h>

char *strerror(int errnum);
```

Description

This function ties closely into `perror()` (which prints a human-readable error message corresponding to `errno`). But instead of printing, `strerror()` returns a pointer to the locale-specific error message string.

So if you ever need that string back for some reason (e.g. you're going to `fprintf()` it to a file or something), this function will give it to you. All you need to do is pass in `errno` as an argument. (Recall that `errno` gets set as an error status by a variety of functions.)

You can actually pass in any integer for `errnum` you want. The function will return *some* message, even if the number doesn't correspond to any known value for `errno`.

The values of `errno` and the strings returned by `strerror()` are system-dependent.

Return Value

A string error message corresponding to the given error number.

You are not allowed to modify the returned string.

Example

```
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main(void)
{
    FILE *fp = fopen("NONEXISTENT_FILE.TXT", "r");

    if (fp == NULL) {
        char *errmsg = strerror(errno);
        printf("Error %d opening file: %s\n", errno, errmsg);
    }
}
```

Output:

```
Error 2 opening file: No such file or directory
```

See Also

`perror()`

64.14 strlen()

Returns the length of a string

Synopsis

```
#include <string.h>

size_t strlen(const char *s);
```

Description

This function returns the length of the passed null-terminated string (not counting the NUL character at the end). It does this by walking down the string and counting the bytes until the NUL character, so it's a little time consuming. If you have to get the length of the same string repeatedly, save it off in a variable somewhere.

Return Value

Returns the number of bytes in the string. Note that this might be different than the number of characters in a multibyte string.

Example

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *s = "Hello, world!"; // 13 characters

    // prints "The string is 13 characters long.":

    printf("The string is %zu characters long.\n", strlen(s));
}
```

See Also

Chapter 65

<tgmath.h> Type-Generic Math Functions

These are type-generic macros that are wrappers around the math functions in <math.h> and <complex.h>. This header includes both of those.

Deep down, these are implemented using type generics as described in the Types Part V chapter.

But on the surface, you can think of them as being able to use, say, the `sqrt()` function with any type without needed to think about if it's `double` or `long double` or even `complex`.

These are the defined macros—some of them don't have a counterpart in the real or complex space. Type suffixes are omitted in the table on the Real and Complex columns. None of the generic macros have type suffixes.

Real Function	Complex Function	Generic Macro
acos	cacos	acos
asin	casin	asin
atan	catan	atan
acosh	cacosh	acosh
asinh	casinh	asinh
atanh	catanh	atanh
cos	ccos	cos
sin	csin	sin
tan	ctan	tan
cosh	ccosh	cosh
sinh	csinh	sinh
tanh	ctanh	tanh
exp	cexp	exp
log	clog	log
pow	cpow	pow
sqrt	csqrt	sqrt
fabs	cabs	fabs
atan2	—	atan2
fdim	—	fdim
cbrt	—	cbrt
floor	—	floor
ceil	—	ceil
fma	—	fma

Real Function	Complex Function	Generic Macro
copysign	—	copysign
fmax	—	fmax
erf	—	erf
fmin	—	fmin
erfc	—	erfc
fmod	—	fmod
exp2	—	exp2
frexp	—	frexp
expm1	—	expm1
hypot	—	hypot
ilogb	—	ilogb
ldexp	—	ldexp
lgamma	—	lgamma
llrint	—	llrint
llround	—	llround
log10	—	log10
log1p	—	log1p
log2	—	log2
logb	—	logb
lrint	—	lrint
lround	—	lround
nearbyint	—	nearbyint
nextafter	—	nextafter
nexttoward	—	nexttoward
remainder	—	remainder
remquo	—	remquo
rint	—	rint
round	—	round
scalbn	—	scalbn
scalbln	—	scalbln
tgamma	—	tgamma
trunc	—	trunc
—	carg	carg
—	cimag	cimag
—	conj	conj
—	cproj	cproj
—	creal	creal

65.1 Example

Here's an example where we call the type-generic `sqrt()` function on a variety of types.

```
#include <stdio.h>
#include <tgmath.h>

int main(void)
{
    double x = 12.8;
    long double y = 34.9;
    double complex z = 1 + 2 * I;

    double x_result;
```

```
    long double y_result;
    double complex z_result;

    // We call the same sqrt() function--it's type-generic!
    x_result = sqrt(x);
    y_result = sqrt(y);
    z_result = sqrt(z);

    printf("x_result: %f\n", x_result);
    printf("y_result: %Lf\n", y_result);
    printf("z_result: %f + %fi\n", creal(z_result), cimag(z_result));
}
```

Output:

```
x_result: 3.577709
y_result: 5.907622
z_result: 1.272020 + 0.786151i
```

Chapter 66

<threads.h> Multithreading Functions

Function	Description
`call_once()`	Call a function one time no matter how many threads try
`cnd_broadcast()`	Wake up all threads waiting on a condition variable
`cnd_destroy()`	Free up resources from a condition variable
`cnd_init()`	Initialize a condition variable to make it ready for use
`cnd_signal()`	Wake up a thread waiting on a condition variable
`cnd_timedwait()`	Wait on a condition variable with a timeout
`cnd_wait()`	Wait for a signal on a condition variable
`mtx_destroy()`	Cleanup a mutex when done with it
`mtx_init()`	Initialize a mutex for use
`mtx_lock()`	Acquire a lock on a mutex
`mtx_timedlock()`	Lock a mutex allowing for timeout
`mtx_trylock()`	Try to lock a mutex, returning if not possible
`mtx_unlock()`	Free a mutex when you're done with the critical section
`thrd_create()`	Create a new thread of execution
`thrd_current()`	Get the ID of the calling thread
`thrd_detach()`	Automatically clean up threads when they exit
`thrd_equal()`	Compare two thread descriptors for equality
`thrd_exit()`	Stop and exit this thread
`thrd_join()`	Wait for a thread to exit
`thrd_yield()`	Stop running that other threads might run
`tss_create()`	Create new thread-specific storage
`tss_delete()`	Clean up a thread-specific storage variable
`tss_get()`	Get thread-specific data
`tss_set()`	Set thread-specific data

We have a bunch of good things at our disposal with this one:

- Threads
- Mutexes
- Condition Variables
- Thread-Specific Storage
- And, last but not least, the always-fun `call_once()` function!

www.dbooks.org

Enjoy!

66.1 call_once()

Call a function one time no matter how many threads try

Synopsis

```
#include <threads.h>

void call_once(once_flag *flag, void (*func)(void));
```

Description

If you have a bunch of threads running over the same piece of code that calls a function, but you only want that function to run one time, `call_once()` can help you out.

The catch is the function that is called doesn't return anything and takes no arguments.

If you need more than that, you'll have to set a threadsafe flag such as `atomic_flag`, or one that you protect with a mutex.

To use this, you need to pass it a pointer to a function to execute, `func`, and also a pointer to a flag of type `once_flag`.

`once_flag` is an opaque type, so all you need to know is that you initialize it to the value `ONCE_FLAG_INIT`.

Return Value

Returns nothing.

Example

```
#include <stdio.h>
#include <threads.h>

once_flag of = ONCE_FLAG_INIT;  // Initialize it like this

void run_once_function(void)
{
    printf("I'll only run once!\n");
}

int run(void *arg)
{
    (void)arg;

    printf("Thread running!\n");

    call_once(&of, run_once_function);

    return 0;
}
```

```

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_create(t + i, run, NULL);

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_join(t[i], NULL);
}
```

Output (might vary per run):

```
Thread running!
Thread running!
I'll only run once!
Thread running!
Thread running!
Thread running!
```

66.2 cnd_broadcast()

Wake up all threads waiting on a condition variable

Synopsis

```
#include <threads.h>

int cnd_broadcast(cnd_t *cond);
```

Description

This is just like `cnd_signal()` in that it wakes up threads that are waiting on a condition variable…. except instead of just rousing one thread, it wakes them all.

Of course, only one will get the mutex, and the rest will have to wait their turn. But instead of being asleep waiting for a signal, they'll be asleep waiting to reacquire the mutex. They're rearin' to go, in other words.

This can make a difference in a specific set of circumstances where `cnd_signal()` might leave you hanging.

If you're relying on subsequent threads to issue the next `cnd_signal()`, but you have the `cnd_wait()` in a `while` loop[1] that doesn't allow any threads to escape, you'll be stuck. No more threads will be woken up from the wait.

But if you `cnd_broadcast()`, all the threads will be woken, and presumably at least one of them will be allowed to escape the `while` loop, freeing it up to broadcast the next wakeup when its work is done.

[1] Which you should because of spurious wakeups.

Return Value

Returns `thrd_success` or `thrd_error` depending on how well things went.

Example

In the example below, we launch a bunch of threads, but they're only allowed to run if their ID matches the current ID. If it doesn't, they go back to waiting.

If you `cnd_signal()` to wake the next thread, it might not be the one with the proper ID to run. If it's not, it goes back to sleep and we hang (because no thread is awake to hit `cnd_signal()` again).

But if you `cnd_broadcast()` to wake them all, then they'll all try (one after another) to get out of the `while` loop. And one of them will make it.

Try switching the `cnd_broadcast()` to `cnd_signal()` to see likely deadlocks. It doesn't happen every time, but usually does.

```c
#include <stdio.h>
#include <threads.h>

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    int id = *(int*)arg;

    static int current_id = 0;

    mtx_lock(&mutex);

    while (id != current_id) {
        printf("THREAD %d: waiting\n", id);
        cnd_wait(&condvar, &mutex);

        if (id != current_id)
            printf("THREAD %d: woke up, but it's not my turn!\n", id);
        else
            printf("THREAD %d: woke up, my turn! Let's go!\n", id);
    }

    current_id++;

    printf("THREAD %d: signaling thread %d to run\n", id, current_id);

    //cnd_signal(&condvar);
    cnd_broadcast(&condvar);
    mtx_unlock(&mutex);

    return 0;
}

#define THREAD_COUNT 5

int main(void)
```

```
{
    thrd_t t[THREAD_COUNT];
    int id[] = {4, 3, 2, 1, 0};

    mtx_init(&mutex, mtx_plain);
    cnd_init(&condvar);

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_create(t + i, run, id + i);

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_join(t[i], NULL);

    mtx_destroy(&mutex);
    cnd_destroy(&condvar);
}
```

Example run with `cnd_broadcast()`:

```
THREAD 4: waiting
THREAD 1: waiting
THREAD 3: waiting
THREAD 2: waiting
THREAD 0: signaling thread 1 to run
THREAD 2: woke up, but it's not my turn!
THREAD 2: waiting
THREAD 4: woke up, but it's not my turn!
THREAD 4: waiting
THREAD 3: woke up, but it's not my turn!
THREAD 3: waiting
THREAD 1: woke up, my turn! Let's go!
THREAD 1: signaling thread 2 to run
THREAD 4: woke up, but it's not my turn!
THREAD 4: waiting
THREAD 3: woke up, but it's not my turn!
THREAD 3: waiting
THREAD 2: woke up, my turn! Let's go!
THREAD 2: signaling thread 3 to run
THREAD 4: woke up, but it's not my turn!
THREAD 4: waiting
THREAD 3: woke up, my turn! Let's go!
THREAD 3: signaling thread 4 to run
THREAD 4: woke up, my turn! Let's go!
THREAD 4: signaling thread 5 to run
```

Example run with `cnd_signal()`:

```
THREAD 4: waiting
THREAD 1: waiting
THREAD 3: waiting
THREAD 2: waiting
THREAD 0: signaling thread 1 to run
THREAD 4: woke up, but it's not my turn!
THREAD 4: waiting
```

```
[deadlock at this point]
```

See how THREAD 0 signaled that it was THREAD 1's turn? But—bad news—it was THREAD 4 that got woken up. So no one continued the process. cnd_broadcast() would have woken them all, so eventually THREAD 1 would have run, gotten out of the while, and broadcast for the next thread to run.

See Also

cnd_signal(), mtx_lock(), mtx_unlock()

66.3 cnd_destroy()

Free up resources from a condition variable

Synopsis

```
#include <threads.h>

void cnd_destroy(cnd_t *cond);
```

Description

This is the opposite of cnd_init() and should be called when all threads are done using a condition variable.

Return Value

Returns nothing!

Example

General-purpose condition variable example here, but you can see the cnd_destroy() down at the end.

```
#include <stdio.h>
#include <threads.h>

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    mtx_lock(&mutex);

    printf("Thread: waiting...\n");
    cnd_wait(&condvar, &mutex);
    printf("Thread: running again!\n");

    mtx_unlock(&mutex);

    return 0;
```

```
}

int main(void)
{
    thrd_t t;

    mtx_init(&mutex, mtx_plain);
    cnd_init(&condvar);

    printf("Main creating thread\n");
    thrd_create(&t, run, NULL);

    // Sleep 0.1s to allow the other thread to wait
    thrd_sleep(&(struct timespec){.tv_nsec=100000000L}, NULL);

    mtx_lock(&mutex);
    printf("Main: signaling thread\n");
    cnd_signal(&condvar);
    mtx_unlock(&mutex);

    thrd_join(t, NULL);

    mtx_destroy(&mutex);
    cnd_destroy(&condvar);  // <-- DESTROY CONDITION VARIABLE
}
```

Output:

```
Main creating thread
Thread: waiting...
Main: signaling thread
Thread: running again!
```

See Also

`cnd_init()`

66.4 `cnd_init()`

Initialize a condition variable to make it ready for use

Synopsis

```
#include <threads.h>

int cnd_init(cnd_t *cond);
```

Description

This is the opposite of `cnd_destroy()`. This prepares a condition variable for use, doing behind-the-scenes work on it.

Don't use a condition variable without calling this first!

Return Value

If all goes well, returns `thrd_success`. It all doesn't go well, it could return `thrd_nomem` if the system is out of memory, or `thread_error` in the case of any other error.

Example

General-purpose condition variable example here, but you can see the `cnd_init()` down at the start of `main()`.

```c
#include <stdio.h>
#include <threads.h>

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    mtx_lock(&mutex);

    printf("Thread: waiting...\n");
    cnd_wait(&condvar, &mutex);
    printf("Thread: running again!\n");

    mtx_unlock(&mutex);

    return 0;
}

int main(void)
{
    thrd_t t;

    mtx_init(&mutex, mtx_plain);
    cnd_init(&condvar);      // <-- INITIALIZE CONDITION VARIABLE

    printf("Main creating thread\n");
    thrd_create(&t, run, NULL);

    // Sleep 0.1s to allow the other thread to wait
    thrd_sleep(&(struct timespec){.tv_nsec=100000000L}, NULL);

    mtx_lock(&mutex);
    printf("Main: signaling thread\n");
    cnd_signal(&condvar);
    mtx_unlock(&mutex);

    thrd_join(t, NULL);

    mtx_destroy(&mutex);
```

```
    cnd_destroy(&condvar);
}
```

Output:

```
Main creating thread
Thread: waiting...
Main: signaling thread
Thread: running again!
```

See Also

cnd_destroy()

66.5 cnd_signal()

Wake up a thread waiting on a condition variable

Synopsis

```
#include <threads.h>

int cnd_signal(cnd_t *cond);
```

Description

If you have a thread (or a bunch of threads) waiting on a condition variable, this function will wake one of them up to run.

Compare to cnd_broadcast() that wakes up all the threads. See the cnd_broadcast() page for more information on when you're want to use that versus this.

Return Value

Returns thrd_success or thrd_error depending on how happy your program is.

Example

General-purpose condition variable example here, but you can see the cnd_signal() in the middle of main().

```
#include <stdio.h>
#include <threads.h>

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    mtx_lock(&mutex);
```

```c

    printf("Thread: waiting...\n");
    cnd_wait(&condvar, &mutex);
    printf("Thread: running again!\n");

    mtx_unlock(&mutex);

    return 0;
}

int main(void)
{
    thrd_t t;

    mtx_init(&mutex, mtx_plain);
    cnd_init(&condvar);

    printf("Main creating thread\n");
    thrd_create(&t, run, NULL);

    // Sleep 0.1s to allow the other thread to wait
    thrd_sleep(&(struct timespec){.tv_nsec=100000000L}, NULL);

    mtx_lock(&mutex);
    printf("Main: signaling thread\n");
    cnd_signal(&condvar);    // <-- SIGNAL CHILD THREAD HERE!
    mtx_unlock(&mutex);

    thrd_join(t, NULL);

    mtx_destroy(&mutex);
    cnd_destroy(&condvar);
}
```

Output:

```
Main creating thread
Thread: waiting...
Main: signaling thread
Thread: running again!
```

See Also

`cnd_init()`, `cnd_destroy()`

66.6 `cnd_timedwait()`

Wait on a condition variable with a timeout

Synopsis

```
#include <threads.h>

int cnd_timedwait(cnd_t *restrict cond, mtx_t *restrict mtx,
                  const struct timespec *restrict ts);
```

Description

This is like `cnd_wait()` except we get to specify a timeout, as well.

Note that the thread still must reacquire the mutex to get more work done even after the timeout. The the main difference is that regular `cnd_wait()` will only try to get the mutex after a `cnd_signal()` or `cnd_broadcast()`, whereas `cnd_timedwait()` will do that, too, **and** try to get the mutex after the timeout.

The timeout is specified as an absolute UTC time since Epoch. You can get this with the `timespec_get()` function and then add values on to the result to timeout later than now, as shown in the example.

Beware that you can't have more than 999999999 nanoseconds in the `tv_nsec` field of the `struct timespec`. Mod those so they stay in range.

Return Value

If the thread wakes up for a non-timeout reason (e.g. signal or broadcast), returns `thrd_success`. If woken up due to timeout, returns `thrd_timedout`. Otherwise returns `thrd_error`.

Example

This example has a thread wait on a condition variable for a maximum of 1.75 seconds. And it always times out because no one ever sends a signal. Tragic.

```
#include <stdio.h>
#include <time.h>
#include <threads.h>

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    mtx_lock(&mutex);

    struct timespec ts;

    // Get the time now
    timespec_get(&ts, TIME_UTC);

    // Add on 1.75 seconds from now
    ts.tv_sec += 1;
    ts.tv_nsec += 750000000L;

    // Handle nsec overflow
    ts.tv_sec += ts.tv_nsec / 1000000000L;
    ts.tv_nsec = ts.tv_nsec % 1000000000L;
```

```

    printf("Thread: waiting...\n");
    int r = cnd_timedwait(&condvar, &mutex, &ts);

    switch (r) {
        case thrd_success:
            printf("Thread: signaled!\n");
            break;

        case thrd_timedout:
            printf("Thread: timed out!\n");
            return 1;

        case thrd_error:
            printf("Thread: Some kind of error\n");
            return 2;
    }

    mtx_unlock(&mutex);

    return 0;
}

int main(void)
{
    thrd_t t;

    mtx_init(&mutex, mtx_plain);
    cnd_init(&condvar);

    printf("Main creating thread\n");
    thrd_create(&t, run, NULL);

    // Sleep 3s to allow the other thread to timeout
    thrd_sleep(&(struct timespec){.tv_sec=3}, NULL);

    thrd_join(t, NULL);

    mtx_destroy(&mutex);
    cnd_destroy(&condvar);
}
```

Output:

```
Main creating thread
Thread: waiting...
Thread: timed out!
```

See Also

`cnd_wait()`, `timespec_get()`

66.7 cnd_wait()

Wait for a signal on a condition variable

Synopsis

```
#include <threads.h>

int cnd_wait(cnd_t *cond, mtx_t *mtx);
```

Description

This puts the calling thread to sleep until it is awakened by a call to `cnd_signal()` or `cnd_broadcast()`.

Return Value

If everything's fantastic, returns `thrd_success`. Otherwise it returns `thrd_error` to report that something has gone fantastically, horribly awry.

Example

General-purpose condition variable example here, but you can see the `cnd_wait()` in the `run()` function.

```
#include <stdio.h>
#include <threads.h>

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    mtx_lock(&mutex);

    printf("Thread: waiting...\n");
    cnd_wait(&condvar, &mutex);       // <-- WAIT HERE!
    printf("Thread: running again!\n");

    mtx_unlock(&mutex);

    return 0;
}

int main(void)
{
    thrd_t t;

    mtx_init(&mutex, mtx_plain);
    cnd_init(&condvar);

    printf("Main creating thread\n");
    thrd_create(&t, run, NULL);

```

```
    // Sleep 0.1s to allow the other thread to wait
    thrd_sleep(&(struct timespec){.tv_nsec=100000000L}, NULL);

    mtx_lock(&mutex);
    printf("Main: signaling thread\n");
    cnd_signal(&condvar);    // <-- SIGNAL CHILD THREAD HERE!
    mtx_unlock(&mutex);

    thrd_join(t, NULL);

    mtx_destroy(&mutex);
    cnd_destroy(&condvar);
}
```

Output:

```
Main creating thread
Thread: waiting...
Main: signaling thread
Thread: running again!
```

See Also

`cnd_timedwait()`

66.8 `mtx_destroy()`

Cleanup a mutex when done with it

Synopsis

```
#include <threads.h>

void mtx_destroy(mtx_t *mtx);
```

Description

The opposite of `mtx_init()`, this function frees up any resources associated with the given mutex.

You should call this when all threads are done using the mutex.

Return Value

Returns nothing, the selfish ingrate!

Example

General-purpose mutex example here, but you can see the `mtx_destroy()` down at the end.

```
#include <stdio.h>
#include <threads.h>

```

```
cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    static int count = 0;

    mtx_lock(&mutex);

    printf("Thread: I got %d!\n", count);
    count++;

    mtx_unlock(&mutex);

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    mtx_init(&mutex, mtx_plain);

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_create(t + i, run, NULL);

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_join(t[i], NULL);

    mtx_destroy(&mutex);   // <-- DESTROY THE MUTEX HERE
}
```

Output:

```
Thread: I got 0!
Thread: I got 1!
Thread: I got 2!
Thread: I got 3!
Thread: I got 4!
```

See Also

`mtx_init()`

66.9 `mtx_init()`

Initialize a mutex for use

Synopsis

```
#include <threads.h>

int mtx_init(mtx_t *mtx, int type);
```

Description

Before you can use a mutex variable, you have to initialize it with this call to get it all prepped and ready to go.

But wait! It's not quite that simple. You have to tell it what `type` of mutex you want to create.

Type	Description
`mtx_plain`	Regular ol' mutex
`mtx_timed`	Mutex that supports timeouts
`mtx_plain|mtx_recursive`	Recursive mutex
`mtx_timed|mtx_recursive`	Recursive mutex that supports timeouts

As you can see, you can make a plain or timed mutex *recursive* by bitwise-ORing the value with `mtx_recursive`.

"Recursive" means that the holder of a lock can call `mtx_lock()` multiple times on the same lock. (They have to unlock it an equal number of times before anyone else can take the mutex.) This might ease coding from time to time, especially if you call a function that needs to lock the mutex when you already hold the mutex.

And the timeout gives a thread a chance to *try* to get the lock for a while, but then bail out if it can't get it in that timeframe. You use the `mtx_timedlock()` function with `mtx_timed` mutexes.

Return Value

Returns `thrd_success` in a perfect world, and potentially `thrd_error` in an imperfect one.

Example

General-purpose mutex example here, but you can see the `mtx_init()` down at the top of `main()`:

```
#include <stdio.h>
#include <threads.h>

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    static int count = 0;

    mtx_lock(&mutex);

    printf("Thread: I got %d!\n", count);
    count++;
```

```
    mtx_unlock(&mutex);

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    mtx_init(&mutex, mtx_plain);  // <-- CREATE THE MUTEX HERE

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_create(t + i, run, NULL);

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_join(t[i], NULL);

    mtx_destroy(&mutex);   // <-- DESTROY THE MUTEX HERE
}
```

Output:

```
Thread: I got 0!
Thread: I got 1!
Thread: I got 2!
Thread: I got 3!
Thread: I got 4!
```

See Also

`mtx_destroy()`

66.10 `mtx_lock()`

Acquire a lock on a mutex

Synopsis

```
#include <threads.h>

int mtx_lock(mtx_t *mtx);
```

Description

If you're a thread and want to enter a critical section, do I have the function for you!

A thread that calls this function will wait until it can acquire the mutex, then it will grab it, wake up, and run!

If the mutex is recursive and is already locked by this thread, it will be locked again and the lock count will increase. If the mutex is not recursive and the thread already holds it, this call will error out.

Return Value

Returns `thrd_success` on goodness and `thrd_error` on badness.

Example

General-purpose mutex example here, but you can see the `mtx_lock()` in the `run()` function:

```c
#include <stdio.h>
#include <threads.h>

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    static int count = 0;

    mtx_lock(&mutex);  // <-- LOCK HERE

    printf("Thread: I got %d!\n", count);
    count++;

    mtx_unlock(&mutex);

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    mtx_init(&mutex, mtx_plain);  // <-- CREATE THE MUTEX HERE

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_create(t + i, run, NULL);

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_join(t[i], NULL);

    mtx_destroy(&mutex);   // <-- DESTROY THE MUTEX HERE
}
```

Output:

```
Thread: I got 0!
Thread: I got 1!
Thread: I got 2!
Thread: I got 3!
Thread: I got 4!
```

See Also

mtx_unlock(), mtx_trylock(), mtx_timedlock()

66.11 mtx_timedlock()

Lock a mutex allowing for timeout

Synopsis

```
#include <threads.h>

int mtx_timedlock(mtx_t *restrict mtx, const struct timespec *restrict ts);
```

Description

This is just like mtx_lock() except you can add a timeout if you don't want to wait forever.

The timeout is specified as an absolute UTC time since Epoch. You can get this with the timespec_get() function and then add values on to the result to timeout later than now, as shown in the example.

Beware that you can't have more than 999999999 nanoseconds in the tv_nsec field of the struct timespec. Mod those so they stay in range.

Return Value

If everything works and the mutex is obtained, returns thrd_success. If a timeout happens first, returns thrd_timedout.

Otherwise, returns thrd_error. Because if nothing is right, everything is wrong.

Example

This example has a thread wait on a mutex for a maximum of 1.75 seconds. And it always times out because no one ever sends a signal.

```
#include <stdio.h>
#include <time.h>
#include <threads.h>

mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    struct timespec ts;

    // Get the time now
    timespec_get(&ts, TIME_UTC);

    // Add on 1.75 seconds from now
    ts.tv_sec += 1;
    ts.tv_nsec += 750000000L;
```

```

    // Handle nsec overflow
    ts.tv_sec += ts.tv_nsec / 1000000000L;
    ts.tv_nsec = ts.tv_nsec % 1000000000L;

    printf("Thread: waiting for lock...\n");
    int r = mtx_timedlock(&mutex, &ts);

    switch (r) {
        case thrd_success:
            printf("Thread: grabbed lock!\n");
            break;

        case thrd_timedout:
            printf("Thread: timed out!\n");
            break;

        case thrd_error:
            printf("Thread: Some kind of error\n");
            break;
    }

    mtx_unlock(&mutex);

    return 0;
}

int main(void)
{
    thrd_t t;

    mtx_init(&mutex, mtx_plain);

    mtx_lock(&mutex);

    printf("Main creating thread\n");
    thrd_create(&t, run, NULL);

    // Sleep 3s to allow the other thread to timeout
    thrd_sleep(&(struct timespec){.tv_sec=3}, NULL);

    mtx_unlock(&mutex);

    thrd_join(t, NULL);

    mtx_destroy(&mutex);
}
```

Output:

```
Main creating thread
Thread: waiting for lock...
Thread: timed out!
```

See Also

mtx_lock(), mtx_trylock(), timespec_get()

66.12 mtx_trylock()

Try to lock a mutex, returning if not possible

Synopsis

```
#include <threads.h>

int mtx_trylock(mtx_t *mtx);
```

Description

This works just like mtx_lock except that it returns instantly if a lock can't be obtained.

The spec notes that there's a chance that mtx_trylock() might spuriously fail with thrd_busy even if there are no other threads holding the lock. I'm not sure why this is, but you should defensively code against it.

Return Value

Returns thrd_success if all's well. Or thrd_busy if some other thread holds the lock. Or thrd_error, which means something went right. I mean "wrong".

Example

```
#include <stdio.h>
#include <time.h>
#include <threads.h>

mtx_t mutex;

int run(void *arg)
{
    int id = *(int*)arg;

    int r = mtx_trylock(&mutex);   // <-- TRY TO GRAB THE LOCK

    switch (r) {
        case thrd_success:
            printf("Thread %d: grabbed lock!\n", id);
            break;

        case thrd_busy:
            printf("Thread %d: lock already taken :(\n", id);
            return 1;

        case thrd_error:
            printf("Thread %d: Some kind of error\n", id);
            return 2;
```

```
    }

    mtx_unlock(&mutex);

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];
    int id[THREAD_COUNT];

    mtx_init(&mutex, mtx_plain);

    for (int i = 0; i < THREAD_COUNT; i++) {
        id[i] = i;
        thrd_create(t + i, run, id + i);
    }

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_join(t[i], NULL);

    mtx_destroy(&mutex);
}
```

Output (varies by run):

```
Thread 0: grabbed lock!
Thread 1: lock already taken :(
Thread 4: lock already taken :(
Thread 3: grabbed lock!
Thread 2: lock already taken :(
```

See Also

`mtx_lock()`, `mtx_timedlock()`, `mtx_unlock()`

66.13 `mtx_unlock()`

Free a mutex when you're done with the critical section

Synopsis

```
#include <threads.h>

int mtx_unlock(mtx_t *mtx);
```

Description

After you've done all the dangerous stuff you have to do, wherein the involved threads should not be stepping on each other's toes… you can free up your stranglehold on the mutex by calling `mtx_unlock()`.

Return Value

Returns `thrd_success` on success. Or `thrd_error` on error. It's not very original in this regard.

Example

General-purpose mutex example here, but you can see the `mtx_unlock()` in the `run()` function:

```
#include <stdio.h>
#include <threads.h>

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    static int count = 0;

    mtx_lock(&mutex);

    printf("Thread: I got %d!\n", count);
    count++;

    mtx_unlock(&mutex);  // <-- UNLOCK HERE

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    mtx_init(&mutex, mtx_plain);

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_create(t + i, run, NULL);

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_join(t[i], NULL);

    mtx_destroy(&mutex);
}
```

Output:

```
Thread: I got 0!
Thread: I got 1!
```

```
Thread: I got 2!
Thread: I got 3!
Thread: I got 4!
```

See Also

mtx_lock(), mtx_timedlock(), mtx_trylock()

66.14 thrd_create()

Create a new thread of execution

Synopsis

```
#include <threads.h>

int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
```

Description

Now *you* have the POWER!

Right?

This is how you launch new threads to make your program do multiple things at once[2]!

In order to make this happen, you need to pass a pointer to a thrd_t that will be used to represent the thread you're spawning.

That thread will start running the function you pass a pointer to in func. This is a value of type thrd_start_t, which is a pointer to a function that returns an int and takes a single void* as a parameter, i.e.:

```
int thread_run_func(void *arg)
```

And, as you might have guessed, the pointer you pass to thrd_create() for the arg parameter is passed on to the func function. This is how you can give additional information to the thread when it starts up.

Of course, for arg, you have to be sure to pass a pointer to an object that is thread-safe or per-thread.

If the thread returns from the function, it exits just as if it had called thrd_exit().

Finally, the value that the func function returns can be picked up by the parent thread with thrd_join().

Return Value

In the case of goodness, returns thrd_success. If you're out of memory, will return thrd_nomem. Otherwise, thrd_error.

Example

[2]Well, as at least as many things as you have free cores. Your OS will schedule them as it can.

```c
#include <stdio.h>
#include <threads.h>

int run(void *arg)
{
    int id = *(int*)arg;

    printf("Thread %d: I'm alive!!\n", id);

    return id;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];
    int id[THREAD_COUNT];  // One of these per thread

    for (int i = 0; i < THREAD_COUNT; i++) {
        id[i] = i; // Let's pass in the thread number as the ID
        thrd_create(t + i, run, id + i);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        int res;

        thrd_join(t[i], &res);

        printf("Main: thread %d exited with code %d\n", i, res);
    }
}
```

Output (might vary from run to run):

```
Thread 1: I'm alive!!
Thread 0: I'm alive!!
Thread 3: I'm alive!!
Thread 2: I'm alive!!
Main: thread 0 exited with code 0
Main: thread 1 exited with code 1
Main: thread 2 exited with code 2
Main: thread 3 exited with code 3
Thread 4: I'm alive!!
Main: thread 4 exited with code 4
```

See Also

`thrd_exit()`, `thrd_join()`

66.15 thrd_current()

Get the ID of the calling thread

Synopsis

```
#include <threads.h>

thrd_t thrd_current(void);
```

Description

Each thread has an opaque ID of type `thrd_t`. This is the value we see get initialized when we call `thrd_create()`.

But what if you want to get the ID of the currently running thread?

No problem! Just call this function and it will be returned to you.

Why? Who knows!

Well, to be honest, I could see it being used a couple places.

1. You could use it to have a thread detach itself with `thrd_detach()`. I'm not sure why you'd want to do this, however.
2. You could use it to compare this thread's ID with another you have stored in a variable somewhere by using the `thrd_equal()` function. Seems like the most legit use.
3. …
4. Profit!

If anyone has another use, please let me know.

Return Value

Returns the calling thread's ID.

Example

Here's a general example that shows getting the current thread ID and comparing it to a previously-recorded thread ID and taking exciting action based on the result! Starring Arnold Schwarzenegger!

```
#include <stdio.h>
#include <threads.h>

thrd_t first_thread_id;

int run(void *arg)
{
    (void)arg;

    thrd_t my_id = thrd_current();   // <-- GET MY THREAD ID

    if (thrd_equal(my_id, first_thread_id))
        printf("I'm the first thread!\n");
    else
        printf("I'm not the first!\n");

```

```
    return 0;
}

int main(void)
{
    thrd_t t;

    thrd_create(&first_thread_id, run, NULL);
    thrd_create(&t, run, NULL);

    thrd_join(first_thread_id, NULL);
    thrd_join(t, NULL);
}
```

Output:

```
Come on, you got what you want, Cohaagen! Give deez people ay-ah!
```

No, wait, that's an Arnold Schwarzenegger quote from *Total Recall*, one of the best science fiction films of all time. Watch it now and then come back to finish this reference page.

Man–what an ending! And Johnny Cab? So excellent. Anyway!

Output:

```
I'm the first thread!
I'm not the first!
```

See Also

`thrd_equal()`, `thrd_detach()`

66.16 `thrd_detach()`

Automatically clean up threads when they exit

Synopsis

```
#include <threads.h>

int thrd_detach(thrd_t thr);
```

Description

Normally you have to `thrd_join()` to get resources associated with a deceased thread cleaned up. (Most notably, its exit status is still floating around waiting to get picked up.)

But if you call `thrd_detach()` on the thread first, manual cleanup isn't necessary. They just exit and are cleaned up by the OS.

(Note that when the main thread dies, all the threads die in any case.)

Return Value

`thrd_success` if the thread successfully detaches, `thrd_error` otherwise.

Example

```
#include <stdio.h>
#include <threads.h>

thrd_t first_thread_id;

int run(void *arg)
{
    (void)arg;

    printf("Thread running!\n");

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t;

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_create(&t, run, NULL);
        thrd_detach(t);
    }

    // No need to thrd_join()!

    // Sleep a quarter second to let them all finish
    thrd_sleep(&(struct timespec){.tv_nsec=250000000}, NULL);
}
```

See Also

`thrd_join()`, `thrd_exit()`

66.17 `thrd_equal()`

Compare two thread descriptors for equality

Synopsis

```
#include <threads.h>

int thrd_equal(thrd_t thr0, thrd_t thr1);
```

Description

If you have two thread descriptors in `thrd_t` variables, you can test them for equality with this function.

For example, maybe one of the threads has special powers the others don't, and the run function needs to be able to tell them apart, as in the example.

Return Value

Returns non-zero if the threads are equal. Returns `0` if they're not.

Example

Here's a general example that shows getting the current thread ID and comparing it to a previously-recorded thread ID and taking boring action based on the result.

```
#include <stdio.h>
#include <threads.h>

thrd_t first_thread_id;

int run(void *arg)
{
    (void)arg;

    thrd_t my_id = thrd_current();

    if (thrd_equal(my_id, first_thread_id))  // <-- COMPARE!
        printf("I'm the first thread!\n");
    else
        printf("I'm not the first!\n");

    return 0;
}

int main(void)
{
    thrd_t t;

    thrd_create(&first_thread_id, run, NULL);
    thrd_create(&t, run, NULL);

    thrd_join(first_thread_id, NULL);
    thrd_join(t, NULL);
}
```

Output:

```
I'm the first thread!
I'm not the first!
```

See Also

`thrd_current()`

66.18 thrd_exit()

Stop and exit this thread

Synopsis

```
#include <threads.h>

_Noreturn void thrd_exit(int res);
```

Description

A thread commonly exits by returning from its run function. But if it wants to exit early (perhaps from deeper in the call stack), this function will get that done.

The `res` code can be picked up by a thread calling `thrd_join()`, and is equivalent to returning a value from the run function.

Like with returning from the run function, this will also properly clean up all the thread-specific storage associated with this thread—all the destructors for the threads TSS variables will be called. If there are any remaining TSS variables with destructors after the first round of destruction[3], the remaining destructors will be called. This happens repeatedly until there are no more, or the number of rounds of carnage reaches `TSS_DTOR_ITERATIONS`.

If the main thread calls this, it's as if you called `exit(EXIT_SUCCESS)`.

Return Value

This function never returns because the thread calling it is killed in the process. Trippy!

Example

Threads in this example exit early with result `22` if they get a `NULL` value for `arg`.

```
#include <stdio.h>
#include <threads.h>

thrd_t first_thread_id;

int run(void *arg)
{
    (void)arg;

    if (arg == NULL)
        thrd_exit(22);

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];
```

[3]For example, if a destructor caused more variables to be set.

```

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_create(t + i, run, i == 2? NULL: "spatula");


    for (int i = 0; i < THREAD_COUNT; i++) {
        int res;
        thrd_join(t[i], &res);

        printf("Thread %d exited with code %d\n", i, res);
    }
}
```

Output:

```
Thread 0 exited with code 0
Thread 1 exited with code 0
Thread 2 exited with code 22
Thread 3 exited with code 0
Thread 4 exited with code 0
```

See Also

`thrd_join()`

66.19 `thrd_join()`

Wait for a thread to exit

Synopsis

```
#include <threads.h>

int thrd_join(thrd_t thr, int *res);
```

Description

When a parent thread fires off some child threads, it can wait for them to complete with this call

Return Value

Example

Threads in this example exit early with result `22` if they get a `NULL` value for `arg`. The parent thread picks up this result code with `thrd_join()`.

```
#include <stdio.h>
#include <threads.h>

thrd_t first_thread_id;

int run(void *arg)
```

```
{
    (void)arg;

    if (arg == NULL)
        thrd_exit(22);

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_create(t + i, run, i == 2? NULL: "spatula");


    for (int i = 0; i < THREAD_COUNT; i++) {
        int res;
        thrd_join(t[i], &res);

        printf("Thread %d exited with code %d\n", i, res);
    }
}
```

Output:

```
Thread 0 exited with code 0
Thread 1 exited with code 0
Thread 2 exited with code 22
Thread 3 exited with code 0
Thread 4 exited with code 0
```

See Also

`thrd_exit()`

66.20 `thrd_sleep()`

Sleep for a specific number of seconds and nanoseconds

Synopsis

```
#include <threads.h>

int thrd_sleep(const struct timespec *duration, struct timespec *remaining);
```

Description

This function puts the current thread to sleep for a while[4] allowing other threads to run.

The calling thread will wake up after the time has elapsed, or if it gets interrupted by a signal or something.

If it doesn't get interrupted, it'll sleep at least as long as you asked. Maybe a tad longer. You know how hard it can be to get out of bed.

The structure looks like this:

```
struct timespec {
    time_t tv_sec;   // Seconds
    long   tv_nsec;  // Nanoseconds (billionths of a second)
};
```

Don't set `tv_nsec` greater than 999,999,999. I can't see what officially happens if you do, but on my system `thrd_sleep()` returns `-2` and fails.

Return Value

Returns `0` on timeout, or `-1` if interrupted by a signal. Or any negative value on some other error. Weirdly, the spec allows this "other error negative value" to also be `-1`, so good luck with that.

Example

```
#include <stdio.h>
#include <threads.h>

int main(void)
{
    // Sleep for 3.25 seconds
    thrd_sleep(&(struct timespec){.tv_sec=3, .tv_nsec=250000000}, NULL);

    return 0;
}
```

See Also

`thrd_yield()`

66.21 `thrd_yield()`

Stop running that other threads might run

Synopsis

```
#include <threads.h>

void thrd_yield(void);
```

[4] Unix-like systems have a `sleep()` syscall that sleeps for an integer number of seconds. But `thrd_sleep()` is likely more portable and gives subsecond resolution, besides!

Description

If you have a thread that's hogging the CPU and you want to give your other threads time to run, you can call `thrd_yield()`. If the system sees fit, it will put the calling thread to sleep and one of the other threads will run instead.

It's a good way to be "polite" to the other threads in your program if you want the encourage them to run instead.

Return Value

Returns nothing!

Example

This example's kinda poor because the OS is probably going to reschedule threads on the output anyway, but it gets the point across.

The main thread is giving other threads a chance to run after every block of dumb work it does.

```c
#include <stdio.h>
#include <threads.h>

int run(void *arg)
{
    int main_thread = arg != NULL;

    if (main_thread) {
        long int total = 0;

        for (int i = 0; i < 10; i++) {
            for (long int j = 0; j < 1000L; j++)
                total++;

            printf("Main thread yielding\n");
            thrd_yield();                       // <-- YIELD HERE
        }
    } else
        printf("Other thread running!\n");

    return 0;
}

#define THREAD_COUNT 10

int main(void)
{
    thrd_t t[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_create(t + i, run, i == 0? "main": NULL);

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_join(t[i], NULL);

```

```
    return 0;
}
```

The output will vary from run to run. Notice that even after `thrd_yield()` other threads might not yet be ready to run and the main thread will continue.

```
Main thread yielding
Main thread yielding
Main thread yielding
Other thread running!
Other thread running!
Other thread running!
Other thread running!
Main thread yielding
Other thread running!
Other thread running!
Main thread yielding
Main thread yielding
Main thread yielding
Other thread running!
Main thread yielding
Main thread yielding
Main thread yielding
Other thread running!
Other thread running!
```

See Also

`thrd_sleep()`

66.22 `tss_create()`

Create new thread-specific storage

Synopsis

```
#include <threads.h>

int tss_create(tss_t *key, tss_dtor_t dtor);
```

Description

This helps when you need per-thread storage of different values.

A common place this comes up is if you have a file scope variable that is shared between a bunch of functions and often returned. That's not threadsafe. One way to refactor is to replace it with thread-specific storage so that each thread gets their own code and doesn't step on other thread's toes.

To make this work, you pass in a pointer to a `tss_t` key—this is the variable you will use in subsequent `tss_set()` and `tss_get()` calls to set and get the value associated with the key.

The interesting part of this is the `dtor` destructor pointer of type `tss_dtor_t`. This is actually a pointer to a function that takes a `void*` argument and returns `void`, i.e.

```
void dtor(void *p) { ... }
```

This function will be called per thread when the thread exits with `thrd_exit()` (or returns from the run function).

It's unspecified behavior to call this function while other threads' destructors are running.

Return Value

Returns nothing!

Example

This is a general-purpose TSS example. Note the TSS variable is created near the top of `main()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <threads.h>

tss_t str;

void some_function(void)
{
    // Retrieve the per-thread value of this string
    char *tss_string = tss_get(str);

    // And print it
    printf("TSS string: %s\n", tss_string);
}

int run(void *arg)
{
    int serial = *(int*)arg;  // Get this thread's serial number
    free(arg);

    // malloc() space to hold the data for this thread
    char *s = malloc(64);
    sprintf(s, "thread %d! :)", serial);  // Happy little string

    // Set this TSS variable to point at the string
    tss_set(str, s);

    // Call a function that will get the variable
    some_function();

    return 0; // Equivalent to thrd_exit(0); fires destructors
}

#define THREAD_COUNT 15

int main(void)
{
    thrd_t t[THREAD_COUNT];

    // Make a new TSS variable, the free() function is the destructor
```

```
    tss_create(&str, free);                  // <-- CREATE TSS VAR!

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Holds a thread serial number
        *n = i;
        thrd_create(t + i, run, n);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }

    // And all threads are done, so let's free this
    tss_delete(str);
}
```

Output:

```
TSS string: thread 0! :)
TSS string: thread 2! :)
TSS string: thread 1! :)
TSS string: thread 5! :)
TSS string: thread 3! :)
TSS string: thread 6! :)
TSS string: thread 4! :)
TSS string: thread 7! :)
TSS string: thread 8! :)
TSS string: thread 9! :)
TSS string: thread 10! :)
TSS string: thread 13! :)
TSS string: thread 12! :)
TSS string: thread 11! :)
TSS string: thread 14! :)
```

See Also

`tss_delete()`, `tss_set()`, `tss_get()`, `thrd_exit()`

66.23 `tss_delete()`

Clean up a thread-specific storage variable

Synopsis

```
#include <threads.h>

void tss_delete(tss_t key);
```

Description

This is the opposite of `tss_create()`. You create (initialize) the TSS variable before using it, then, when all the threads are done that need it, you delete (deinitialize/free) it with this.

This doesn't call any destructors! Those are all called by `thrd_exit()`!

Return Value

Returns nothing!

Example

This is a general-purpose TSS example. Note the TSS variable is deleted near the bottom of `main()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <threads.h>

tss_t str;

void some_function(void)
{
    // Retrieve the per-thread value of this string
    char *tss_string = tss_get(str);

    // And print it
    printf("TSS string: %s\n", tss_string);
}

int run(void *arg)
{
    int serial = *(int*)arg;  // Get this thread's serial number
    free(arg);

    // malloc() space to hold the data for this thread
    char *s = malloc(64);
    sprintf(s, "thread %d! :)", serial);  // Happy little string

    // Set this TSS variable to point at the string
    tss_set(str, s);

    // Call a function that will get the variable
    some_function();

    return 0; // Equivalent to thrd_exit(0); fires destructors
}

#define THREAD_COUNT 15

int main(void)
{
    thrd_t t[THREAD_COUNT];

    // Make a new TSS variable, the free() function is the destructor
```

```
    tss_create(&str, free);

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Holds a thread serial number
        *n = i;
        thrd_create(t + i, run, n);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }

    // And all threads are done, so let's free this
    tss_delete(str);    // <-- DELETE TSS VARIABLE!
}
```

Output:

```
TSS string: thread 0! :)
TSS string: thread 2! :)
TSS string: thread 1! :)
TSS string: thread 5! :)
TSS string: thread 3! :)
TSS string: thread 6! :)
TSS string: thread 4! :)
TSS string: thread 7! :)
TSS string: thread 8! :)
TSS string: thread 9! :)
TSS string: thread 10! :)
TSS string: thread 13! :)
TSS string: thread 12! :)
TSS string: thread 11! :)
TSS string: thread 14! :)
```

See Also

`tss_create()`, `tss_set()`, `tss_get()`, `thrd_exit()`

66.24 `tss_get()`

Get thread-specific data

Synopsis

```
#include <threads.h>

void *tss_get(tss_t key);
```

Description

Once you've set a variable with `tss_set()`, you can retrieve the value with `tss_get()`—just pass in the key and you'll get a pointer to the value back.

Don't call this from a destructor.

Return Value

Returns the value stored for the given `key`, or `NULL` if there's trouble.

Example

This is a general-purpose TSS example. Note the TSS variable is retrieved in `some_function()`, below.

```
#include <stdio.h>
#include <stdlib.h>
#include <threads.h>

tss_t str;

void some_function(void)
{
    // Retrieve the per-thread value of this string
    char *tss_string = tss_get(str);    // <-- GET THE VALUE

    // And print it
    printf("TSS string: %s\n", tss_string);
}

int run(void *arg)
{
    int serial = *(int*)arg;  // Get this thread's serial number
    free(arg);

    // malloc() space to hold the data for this thread
    char *s = malloc(64);
    sprintf(s, "thread %d! :)", serial);  // Happy little string

    // Set this TSS variable to point at the string
    tss_set(str, s);

    // Call a function that will get the variable
    some_function();

    return 0; // Equivalent to thrd_exit(0); fires destructors
}

#define THREAD_COUNT 15

int main(void)
{
    thrd_t t[THREAD_COUNT];

    // Make a new TSS variable, the free() function is the destructor
```

```
    tss_create(&str, free);

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Holds a thread serial number
        *n = i;
        thrd_create(t + i, run, n);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }

    // And all threads are done, so let's free this
    tss_delete(str);
}
```

Output:

```
TSS string: thread 0! :)
TSS string: thread 2! :)
TSS string: thread 1! :)
TSS string: thread 5! :)
TSS string: thread 3! :)
TSS string: thread 6! :)
TSS string: thread 4! :)
TSS string: thread 7! :)
TSS string: thread 8! :)
TSS string: thread 9! :)
TSS string: thread 10! :)
TSS string: thread 13! :)
TSS string: thread 12! :)
TSS string: thread 11! :)
TSS string: thread 14! :)
```

See Also

`tss_set()`

66.25 `tss_set()`

Set thread-specific data

Synopsis

```
#include <threads.h>

int tss_set(tss_t key, void *val);
```

Description

Once you've set up your TSS variable with `tss_create()`, you can set it on a per thread basis with `tss_set()`.

`key` is the identifier for this data, and `val` is a pointer to it.

The destructor specified in `tss_create()` will be called for the value set when the thread exits.

Also, if there's a destructor *and* there is already at value for this key in place, the destructor will not be called for the already-existing value. In fact, this function will never cause a destructor to be called. So you're on your own, there—best clean up the old value before overwriting it with the new one.

Return Value

Returns `thrd_success` when happy, and `thrd_error` when not.

Example

This is a general-purpose TSS example. Note the TSS variable is set in `run()`, below.

```
#include <stdio.h>
#include <stdlib.h>
#include <threads.h>

tss_t str;

void some_function(void)
{
    // Retrieve the per-thread value of this string
    char *tss_string = tss_get(str);

    // And print it
    printf("TSS string: %s\n", tss_string);
}

int run(void *arg)
{
    int serial = *(int*)arg;  // Get this thread's serial number
    free(arg);

    // malloc() space to hold the data for this thread
    char *s = malloc(64);
    sprintf(s, "thread %d! :)", serial);  // Happy little string

    // Set this TSS variable to point at the string
    tss_set(str, s);    // <-- SET THE TSS VARIABLE

    // Call a function that will get the variable
    some_function();

    return 0; // Equivalent to thrd_exit(0); fires destructors
}

#define THREAD_COUNT 15

```

```c
int main(void)
{
    thrd_t t[THREAD_COUNT];

    // Make a new TSS variable, the free() function is the destructor
    tss_create(&str, free);

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Holds a thread serial number
        *n = i;
        thrd_create(t + i, run, n);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }

    // And all threads are done, so let's free this
    tss_delete(str);
}
```

Output:

```
TSS string: thread 0! :)
TSS string: thread 2! :)
TSS string: thread 1! :)
TSS string: thread 5! :)
TSS string: thread 3! :)
TSS string: thread 6! :)
TSS string: thread 4! :)
TSS string: thread 7! :)
TSS string: thread 8! :)
TSS string: thread 9! :)
TSS string: thread 10! :)
TSS string: thread 13! :)
TSS string: thread 12! :)
TSS string: thread 11! :)
TSS string: thread 14! :)
```

See Also

`tss_get()`

Chapter 67

<time.h> Date and Time Functions

Function	Description
`clock()`	How much processor time has been used by this process
`difftime()`	Compute the difference between two times
`mktime()`	Convert a `struct tm` into a `time_t`
`time()`	Get the current calendar time
`timespec_get()`	Get a higher resolution time, probably now
`asctime()`	Return a human-readable version of a `struct tm`
`ctime()`	Return a human-readable version of a `time_t`
`gmtime()`	Convert a calendar time into a UTC broken-down time
`localtime()`	Convert a calendar time into a broken-down local time
`strftime()`	Formatted date and time output

When it comes to time and C, there are two main types to look for:

- **`time_t`** holds a *calendar time*. This is an potentially opaque numeric type that represents an absolute time that can be converted to UTC[1] or local time.
- **`struct tm`** holds a *broken-down time*. This has things like the day of the week, the day of the month, the hour, the minute, the second, etc.

On POSIX systems and Windows, `time_t` is an integer and represents the number of seconds that have elapsed since January 1, 1970 at 00:00 UTC.

A `struct tm` contains the following fields:

```
struct tm {
    int tm_sec;    // seconds after the minute -- [0, 60]
    int tm_min;    // minutes after the hour -- [0, 59]
    int tm_hour;   // hours since midnight -- [0, 23]
    int tm_mday;   // day of the month -- [1, 31]
    int tm_mon;    // months since January -- [0, 11]
    int tm_year;   // years since 1900
    int tm_wday;   // days since Sunday -- [0, 6]
    int tm_yday;   // days since January 1 -- [0, 365]
    int tm_isdst;  // Daylight Saving Time flag
};
```

[1] When you say GMT, unless you're talking specifically about the time zone and not the time, you probably mean "UTC".

You can convert between the two with `mktime()`, `gmtime()`, and `localtime()`.

You can print time information to strings with `ctime()`, `asctime()`, and `strftime()`.

67.1 Thread Safety Warning

`asctime()`, `ctime()`: These two functions return a pointer to a `static` memory region. They both might return the same pointer. If you need thread safety, you'll need a mutex across them. If you need both results at once, `strcpy()` one of them out.

All these problems with `asctime()` and `ctime()` can be avoided by using the more flexible and thread-safe `strftime()` function instead.

`localtime()`, `gmtime()`: These other two functions also return a pointer to a `static` memory region. They both might return the same pointer. If you need thread safety, you'll need a mutex across them. If you need both results at once, copy the `struct` to another.

67.2 `clock()`

How much processor time has been used by this process

Synopsis

```
#include <time.h>

clock_t clock(void);
```

Description

Your processor is juggling a lot of things right now. Just because a process has been alive for 20 minutes doesn't mean that it used 20 minutes of "CPU time".

Most of the time your average process spends asleep, and that doesn't count toward the CPU time spent.

This function returns an opaque type representing the number of "clock ticks"[2] the process has spent in operation.

You can get the number of seconds out of that by dividing by the macro `CLOCKS_PER_SEC`. This is an integer, so you will have to cast part of the expression to a floating type to get a fractional time.

Note that this is not the "wall clock time" of the program. If you want to get that loosely use `time()` and `difftime()` (which might only offer 1-second resolution) or `timespec_get()` (which might only also offer low resolution, but at least it *might* go to nanosecond level).

Return Value

Returns the amount of CPU time spent by this process. This comes back in a form that can be divided by `CLOCKS_PER_SEC` to determine the time in seconds.

Example

[2] The spec doesn't actually say "clock ticks", but I… am.

www.dbooks.org

```c
#include <stdio.h>
#include <time.h>

// Deliberately naive Fibonacci
long long int fib(long long int n) {
    if (n <= 1) return n;

    return fib(n-1) + fib(n-2);
}

int main(void)
{
    printf("The 42nd Fibonacci Number is %lld\n", fib(42));

    printf("CPU time: %f\n", clock() / (double)CLOCKS_PER_SEC);
}
```

Output on my system:

```
The 42nd Fibonacci Number is 267914296
CPU time: 1.863078
```

See Also

`time()`, `difftime()`, `timespec_get()`

67.3 `difftime()`

Compute the difference between two times

Synopsis

```c
#include <time.h>

double difftime(time_t time1, time_t time0);
```

Description

Since the `time_t` type is technically opaque, you can't just straight-up subtract to get the difference between two of them[3]. Use this function to do it.

There is no guarantee as to the resolution of this difference, but it's probably to the second.

Return Value

Returns the difference between two `time_t`s in seconds.

[3] Unless you're on a POSIX system where `time_t` is definitely an integer, in which case you can subtract. But you should still use `difftime()` for maximum portability.

Example

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    // April 12, 1982 and change
    struct tm time_a = { .tm_year=82, .tm_mon=3, .tm_mday=12,
        .tm_hour=4, .tm_min=00, .tm_sec=04, .tm_isdst=-1,
    };

    // November 15, 2020 and change
    struct tm time_b = { .tm_year=120, .tm_mon=10, .tm_mday=15,
        .tm_hour=16, .tm_min=27, .tm_sec=00, .tm_isdst=-1,
    };

    time_t cal_a = mktime(&time_a);
    time_t cal_b = mktime(&time_b);

    double diff = difftime(cal_b, cal_a);

    double years = diff / 60 / 60 / 24 / 365.2425;  // close enough

    printf("%f seconds (%f years) between events\n", diff, years);
}
```

Output:

```
1217996816.000000 seconds (38.596783 years) between events
```

See Also

`time()`, `mktime()`

67.4 `mktime()`

Convert a `struct tm` into a `time_t`

Synopsis

```
#include <time.h>

time_t mktime(struct tm *timeptr);
```

Description

If you have a local date and time and want it converted to a `time_t` (so that you can `difftime()` it or whatever), you can convert it with this function.

Basically you fill out the fields in your `struct tm` in local time and `mktime()` will convert those to the UTC `time_t` equivalent.

A couple notes:

- Don't bother filling out `tm_wday` or `tm_yday`. `mktime()` will fill these out for you.
- You can set `tm_isdst` to `0` to indicate your time isn't Daylight Saving Time (DST), `1` to indicate it is, and `-1` to have `mktime()` fill it in according to your locale's preference.

If you need input in UTC, see the non-standard functions `timegm()`[4] for Unix-likes and `_mkgmtime()`[5] for Windows.

Return Value

Returns the local time in the `struct tm` as a `time_t` calendar time.

Returns `(time_t)(-1)` on error.

Example

In the following example, we have `mktime()` tell us if that time was DST or not.

```c
#include <stdio.h>
#include <time.h>

int main(void)
{
    struct tm broken_down_time = {
        .tm_year=82,   // years since 1900
        .tm_mon=3,     // months since January -- [0, 11]
        .tm_mday=12,   // day of the month -- [1, 31]
        .tm_hour=4,    // hours since midnight -- [0, 23]
        .tm_min=00,    // minutes after the hour -- [0, 59]
        .tm_sec=04,    // seconds after the minute -- [0, 60]
        .tm_isdst=-1,  // Daylight Saving Time flag
    };

    time_t calendar_time = mktime(&broken_down_time);

    char *days[] = {"Sunday", "Monday", "Tuesday",
        "Wednesday", "Furzeday", "Friday", "Saturday"};

    // This will print what was in broken_down_time
    printf("Local time : %s", asctime(localtime(&calendar_time)));
    printf("Is DST     : %d\n", broken_down_time.tm_isdst);
    printf("Day of week: %s\n\n", days[broken_down_time.tm_wday]);

    // This will print UTC for the local time, above
    printf("UTC        : %s", asctime(gmtime(&calendar_time)));
}
```

Output (for me in Pacific Time—UTC is 8 hours ahead):

```
Local time : Mon Apr 12 04:00:04 1982
Is DST     : 0
Day of week: Monday
```

[4] https://man.archlinux.org/man/timegm.3.en
[5] https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/mkgmtime-mkgmtime32-mkgmtime64?view=msvc-160

```
UTC        : Mon Apr 12 12:00:04 1982
```

See Also

localtime(), gmtime()

67.5 time()

Get the current calendar time

Synopsis

```
#include <time.h>

time_t time(time_t *timer);
```

Description

Returns the current calendar time right now. I mean, now. No, now!

If timer is not NULL, it gets loaded with the current time, as well.

This can be converted into a struct tm with localtime() or gmtime(), or printed directly with ctime().

Return Value

Returns the current calendar time. Also loads timer with the current time if it's not NULL.

Or returns (time_t)(-1) if the time isn't available because you've fallen out of the space-time continuum and/or the system doesn't support times.

Example

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    time_t now = time(NULL);

    printf("The local time is %s", ctime(&now));
}
```

Example output:

```
The local time is Mon Mar  1 18:45:14 2021
```

See Also

localtime(), gmtime(), ctime()

67.6 timespec_get()

Get a higher resolution time, probably now

Synopsis

```
#include <time.h>

int timespec_get(struct timespec *ts, int base);
```

Description

This function loads the current time UTC (unless directed otherwise) into the given `struct timespec`, `ts`.

That structure has two fields:

```
struct timespec {
    time_t tv_sec;   // Whole seconds
    long   tv_nsec;  // Nanoseconds, 0-999999999
}
```

Nanoseconds are billionths of a second. You can divide by 1000000000.0 to convert to seconds.

The `base` parameter has only one defined value, by the spec: `TIME_UTC`. So portably make it that. This will load `ts` with the current time in seconds since a system-defined Epoch[6], often January 1, 1970 at 00:00 UTC.

Your implementation might define other values for `base`.

Return Value

When `base` is `TIME_UTC`, loads `ts` with the current UTC time.

On success, returns `base`, valid values for which will always be non-zero. On error, returns `0`.

Example

```
struct timespec ts;

timespec_get(&ts, TIME_UTC);

printf("%ld s, %ld ns\n", ts.tv_sec, ts.tv_nsec);

double float_time = ts.tv_sec + ts.tv_nsec/1000000000.0;
printf("%f seconds since epoch\n", float_time);
```

Example output:

```
1614654187 s, 825540756 ns
1614654187.825541 seconds since epoch
```

Here's a helper function to add values to a `struct timespec` that handles negative values and nanosecond overflow.

```
#include <stdlib.h>

// Add delta seconds and delta nanoseconds to ts.
```

[6] https://en.wikipedia.org/wiki/Unix_time

```
// Negative values are allowed. Each component is added individually.
//
// Subtract 1.5 seconds from the current value:
//
// timespec_add(&ts, -1, -500000000L);

struct timespec *timespec_add(struct timespec *ts, long dsec, long dnsec)
{
    long sec = (long)ts->tv_sec + dsec;
    long nsec = ts->tv_nsec + dnsec;

    ldiv_t qr = ldiv(nsec, 1000000000L);

    if (qr.rem < 0) {
        nsec = 1000000000L + qr.rem;
        sec += qr.quot - 1;
    } else {
        nsec = qr.rem;
        sec += qr.quot;
    }

    ts->tv_sec = sec;
    ts->tv_nsec = nsec;

    return ts;
}
```

And here are some functions to convert from `long double` to `struct timespec` and back, just in case you like thinking in decimals. This is more limited in significant figures than using the integer values.

```
#include <math.h>

// Convert a struct timespec into a long double
long double timespec_to_ld(struct timespec *ts)
{
    return ts->tv_sec + ts->tv_nsec / 1000000000.0;
}

// Convert a long double to a struct timespec
struct timespec ld_to_timespec(long double t)
{
    long double f;
    struct timespec ts;
    ts.tv_nsec = modfl(t, &f) * 1000000000L;
    ts.tv_sec = f;

    return ts;
}
```

See Also

`time()`, `mtx_timedlock()`, `cnd_timedwait()`

67.7 asctime()

Return a human-readable version of a `struct tm`

Synopsis

```
#include <time.h>

char *asctime(const struct tm *timeptr)
```

Description

This takes a time in a `struct tm` and returns a string with that date in the form:

```
Sun Sep 16 01:03:52 1973
```

with a newline included at the end, rather unhelpfully. (`strftime()` will give you more flexibility.)

It's just like `ctime()`, except it takes a `struct tm` instead of a `time_t`.

WARNING: This function returns a pointer to a `static char*` region that isn't thread-safe and might be shared with the `ctime()` function. If you need thread safety, use `strftime()` or use a mutex that covers `ctime()` and `asctime()`.

Behavior is undefined for:

- Years less than 1000
- Years greater than 9999
- Any members of `timeptr` are out of range

Return Value

Returns a pointer to the human-readable date string.

Example

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    time_t now = time(NULL);

    printf("Local: %s", asctime(localtime(&now)));
    printf("UTC  : %s", asctime(gmtime(&now)));
}
```

Sample output:

```
Local: Mon Mar  1 21:17:34 2021
UTC  : Tue Mar  2 05:17:34 2021
```

See Also

`ctime()`, `localtime()`, `gmtime()`

67.8 ctime()

Return a human-readable version of a time_t

Synopsis

```
#include <time.h>

char *ctime(const time_t *timer);
```

Description

This takes a time in a time_t and returns a string with the local time and date in the form:

```
Sun Sep 16 01:03:52 1973
```

with a newline included at the end, rather unhelpfully. (strftime() will give you more flexibility.)

It's just like asctime(), except it takes a time_t instead of a struct tm.

WARNING: This function returns a pointer to a static char* region that isn't thread-safe and might be shared with the asctime() function. If you need thread safety, use strftime() or use a mutex that covers ctime() and asctime().

Behavior is undefined for:

- Years less than 1000
- Years greater than 9999
- Any members of timeptr are out of range

Return Value

A pointer to the human-readable local time and data string.

Example

```
time_t now = time(NULL);

printf("Local: %s", ctime(&now));
```

Sample output:

```
Local: Mon Mar  1 21:32:23 2021
```

See Also

asctime()

67.9 gmtime()

Convert a calendar time into a UTC broken-down time

Synopsis

```
#include <time.h>

struct tm *gmtime(const time_t *timer);
```

Description

If you have a `time_t`, you can run it through this function to get a `struct tm` back full of the corresponding broken-down UTC time information.

This is just like `localtime()`, except it does UTC instead of local time.

Once you have that `struct tm`, you can feed it to `strftime()` to print it out.

WARNING: This function returns a pointer to a `static struct tm*` region that isn't thread-safe and might be shared with the `localtime()` function. If you need thread safety use a mutex that covers `gmtime()` and `localtime()`.

Return Value

Returns a pointer to the broken-down UTC time, or `NULL` if it can't be obtained.

Example

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    time_t now = time(NULL);

    printf("UTC  : %s", asctime(gmtime(&now)));
    printf("Local: %s", asctime(localtime(&now)));
}
```

Sample output:

```
UTC  : Tue Mar  2 05:40:05 2021
Local: Mon Mar  1 21:40:05 2021
```

See Also

`localtime()`, `asctime()`, `strftime()`

67.10 `localtime()`

Convert a calendar time into a broken-down local time

Synopsis

```
#include <time.h>

struct tm *localtime(const time_t *timer);
```

Description

If you have a `time_t`, you can run it through this function to get a `struct tm` back full of the corresponding broken-down local time information.

This is just like `gmtime()`, except it does local time instead of UTC.

Once you have that `struct tm`, you can feed it to `strftime()` to print it out.

WARNING: This function returns a pointer to a `static struct tm*` region that isn't thread-safe and might be shared with the `gmtime()` function. If you need thread safety use a mutex that covers `gmtime()` and `localtime()`.

Return Value

Returns a pointer to the broken-down local time, or `NULL` if it can't be obtained.

Example

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    time_t now = time(NULL);

    printf("Local: %s", asctime(localtime(&now)));
    printf("UTC  : %s", asctime(gmtime(&now)));
}
```

Sample output:

```
Local: Mon Mar  1 21:40:05 2021
UTC  : Tue Mar  2 05:40:05 2021
```

See Also

`gmtime()`, `asctime()`, `strftime()`

67.11 `strftime()`

Formatted date and time output

Synopsis

```
#include <time.h>

size_t strftime(char * restrict s, size_t maxsize,
```

```
                const char * restrict format,
                const struct tm * restrict timeptr);
```

Description

This is the `sprintf()` of date and time functions. It'll take a `struct tm` and produce a string in just about whatever form you desire, for example:

```
2021-03-01
Monday, March 1 at 9:54 PM
It's Monday!
```

It's a super flexible version of `asctime()`. And thread-safe, besides, since it doesn't rely on a static buffer to hold the results.

Basically what you do is give it a destination, `s`, and its max size in bytes in `maxsize`. Also, provide a `format` string that's analogous to `printf()`'s format string, but with different format specifiers. And lastly, a `struct tm` with the broken-down time information to use for printing.

The `format` string works like this, for example:

```
"It's %A, %B %d!"
```

Which produces:

```
It's Monday, March 1!
```

The `%A` is the full day-of-week name, the `%B` is the full month name, and the `%d` is the day of the month. `strftime()` substitutes the right thing to produce the result. Brilliant!

So what are all the format specifiers? Glad you asked!

I'm going to be lazy and just drop this table in right from the spec.

Specifier	Description
`%a`	Locale's abbreviated weekday name. [`tm_wday`]
`%A`	Locale's full weekday name. [`tm_wday`]
`%b`	Locale's abbreviated month name. [`tm_mon`]
`%B`	Locale's full month name. [`tm_mon`]
`%c`	Locale's appropriate date and time representation.
`%C`	Year divided by 100 and truncated to an integer, as a decimal number (00–99). [`tm_year`]
`%d`	Day of the month as a decimal number (01–31). [`tm_mday`]
`%D`	Equivalent to `"%m/%d/%y"`. [`tm_mon`, `tm_mday`, `tm_year`]
`%e`	Day of the month as a decimal number (1–31); a single digit is preceded by a space. [`tm_mday`]
`%F`	Equivalent to "%Y-%m-%d" (the ISO 8601 date format). [`tm_year`, `tm_mon`, `tm_mday`]
`%g`	Last 2 digits of the week-based year (see below) as a decimal number (00–99). [`tm_year`, `tm_wday`, `tm_yday`]
`%G`	Week-based year (see below) as a decimal number (e.g., 1997). [`tm_year`, `tm_wday`, `tm_yday`]
`%h`	Equivalent to "%b". [`tm_mon`]
`%H`	Hour (24-hour clock) as a decimal number (00–23). [`tm_hour`]
`%I`	Hour (12-hour clock) as a decimal number (01–12). [`tm_hour`]
`%j`	Day of the year as a decimal number (001–366). [`tm_yday`]
`%m`	Month as a decimal number (01–12).
`%M`	Minute as a decimal number (00–59). [`tm_min`]
`%n`	A new-line character.

Specifier	Description
%p	Locale's equivalent of the AM/PM designations associated with a 12-hour clock. [tm_hour]
%r	Locale's 12-hour clock time. [tm_hour, tm_min, tm_sec]
%R	Equivalent to "%H:%M". [tm_hour, tm_min]
%S	Second as a decimal number (00–60). [tm_sec]
%t	A horizontal-tab character.
%T	Equivalent to "%H:%M:%S" (the ISO 8601 time format). [tm_hour, tm_min, tm_sec]
%u	ISO 8601 weekday as a decimal number (1–7), where Monday is 1. [tm_wday]
%U	Week number of the year (the first Sunday as the first day of week 1) as a decimal number (00–53). [tm_year, tm_wday, tm_yday]
%V	ISO 8601 week number (see below) as a decimal number (01–53). [tm_year, tm_wday, tm_yday]
%w	Weekday as a decimal number (0–6), where Sunday is 0.
%W	Week number of the year (the first Monday as the first day of week 1) as a decimal number (00–53). [tm_year, tm_wday, tm_yday]
%x	Locale's appropriate date representation.
%X	Locale's appropriate time representation.
%y	Last 2 digits of the year as a decimal number (00–99). [tm_year]
%Y	Year as a decimal number (e.g., 1997). [tm_year]
%z	Offset from UTC in the ISO 8601 format "-0430" (meaning 4 hours 30 minutes behind UTC, west of Greenwich), or by no characters if no time zone is determinable. [tm_isdst]
%Z	Locale's time zone name or abbreviation, or by no characters if no time zone is determinable. [tm_isdst]
%%	A plain ol' %

Phew. That's love.

%G, %g, and %v are a little funky in that they use something called the ISO 8601 week-based year. I'd never heard of it. But, again stealing from the spec, these are the rules:

> %g, %G, and %V give values according to the ISO 8601 week-based year. In this system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th, which is also the week that includes the first Thursday of the year, and is also the first week that contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January 1999, %G is replaced by 1998 and %V is replaced by 53. If December 29th, 30th, or 31st is a Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday 30th December 1997, %G is replaced by 1998 and %V is replaced by 01.

Learn something new every day! If you want to know more, Wikipedia has a page on it[7].

If you're in the "C" locale, the specifiers produce the following (again, stolen from the spec):

Specifier	Description
%a	The first three characters of %A.
%A	One of Sunday, Monday, ... , Saturday.
%b	The first three characters of %B.
%B	One of January, February, ... , December.
%c	Equivalent to %a %b %e %T %Y.
%p	One of AM or PM.
%r	Equivalent to %I:%M:%S %p.
%x	Equivalent to %m/%d/%y.

[7] https://en.wikipedia.org/wiki/ISO_week_date

www.dbooks.org

Specifier	Description
`%X`	Equivalent to `%T`.
`%Z`	Implementation-defined.

There are additional variants of the format specifiers that indicate you want to use a locale's alternative format. These don't exist for all locales. It's one of the format specifies above, with either an `E` or `O` prefix:

```
%Ec %EC %Ex %EX %Ey %EY %Od %Oe %OH %OI
%Om %OM %OS %Ou %OU %OV %Ow %OW %Oy
```

The `E` and `O` prefixes are ignored in the "C" locale.

Return Value

Returns the total number of bytes put into the result string, not including the NUL terminator.

If the result doesn't fit in the string, zero is returned and the value in `s` is indeterminate.

Example

```c
#include <stdio.h>
#include <time.h>

int main(void)
{
    char s[128];
    time_t now = time(NULL);

    // %c: print date as per current locale
    strftime(s, sizeof s, "%c", localtime(&now));
    puts(s);   // Sun Feb 28 22:29:00 2021

    // %A: full weekday name
    // %B: full month name
    // %d: day of the month
    strftime(s, sizeof s, "%A, %B %d", localtime(&now));
    puts(s);   // Sunday, February 28

    // %I: hour (12 hour clock)
    // %M: minute
    // %S: second
    // %p: AM or PM
    strftime(s, sizeof s, "It's %I:%M:%S %p", localtime(&now));
    puts(s);   // It's 10:29:00 PM

    // %F: ISO 8601 yyyy-mm-dd
    // %T: ISO 8601 hh:mm:ss
    // %z: ISO 8601 time zone offset
    strftime(s, sizeof s, "ISO 8601: %FT%T%z", localtime(&now));
    puts(s);   // ISO 8601: 2021-02-28T22:29:00-0800
}
```

See Also

`ctime()`, `asctime()`

www.dbooks.org

Chapter 68

<uchar.h> Unicode utility functions

Function	Description
`c16rtomb()`	Convert a `char16_t` to a multibyte character
`c32rtomb()`	Convert a `char32_t` to a multibyte character
`mbrtoc16()`	Convert a multibyte character to a `char16_t`
`mbrtoc32()`	Convert a multibyte character to a `char32_t`

These functions are *restartable*, meaning multiple threads can safely call them at once. They handle this by having their own conversion state variable (of type `mbstate_t`) per call.

68.1 Types

This header file defines four types.

Type	Description
`char16_t`	Type to hold 16-bit characters
`char32_t`	Type to hold 32-bit characters
`mbstate_t`	Holds the conversion state for restartable funcitons (also defined in `<wchar.h>`)
`size_t`	To hold various counts (also defined in `<stddef.h>`)

String literals for the character types are `u` for `char16_t` and `U` for `char32_t`.

```
char16_t *str1 = u"Hello, world!";
char32_t *str2 = U"Hello, world!";

char16_t *chr1 = u'A';
char32_t *chr2 = U'B';
```

Note that `char16_t` and `char32_t` *might* contain Unicode. Or not. If `__STDC_UTF_16__` or `__STDC_UTF_32__` is defined as `1`, then `char16_t` and `char32_t` use Unicode, respectively. Otherwise they don't and the actual value stored depend on the locale. And if you're not using Unicode, you have my commiserations.

68.2 OS X issue

This header file doesn't exist on OS X—bummer. If you just want the types, you can:

```
#include <stdint.h>

typedef int_least16_t char16_t;
typedef int_least32_t char32_t;
```

But if you also want the functions, that's all on you.

68.3 mbrtoc16() mbrtoc32()

Convert a multibyte character to a `char16_t` or `char32_t` restartably

Synopsis

```
#include <uchar.h>

size_t mbrtoc16(char16_t * restrict pc16, const char * restrict s, size_t n,
                mbstate_t * restrict ps);

size_t mbrtoc32(char32_t * restrict pc32, const char * restrict s, size_t n,
                mbstate_t * restrict ps);
```

Description

Given a source string `s` and a destination buffer `pc16` (or `pc32` for `mbrtoc32()`), convert the first character of the source to `char16_ts` (or `char32_ts` for `mbrtoc32()`).

Basically you have a regular character and you want it as `char16_t` or `char32_t`. Use these functions to do it. Note that only one character is converted no matter how many characters in `s`.

As the functions scan `s`, you don't want them to overrun the end. So you pass in `n` as the maximum number of bytes to inspect. The functions will quit after that many bytes or when they have a complete multibyte character, whichever comes first.

Since they're restartable, pass in a conversion state variable for the functions to do their work.

And the result will be placed in `pc16` (or `pc32` for `mbrtoc32()`).

Return Value

When successful this function returns a number between `1` and `n` inclusive representing the number of bytes that made up the multibyte character.

Or, also in the success category, they can return `0` if the source character is the NUL character (value `0`).

When not entirely successful, they can return a variety of codes. These are all of type `size_t`, but negative values cast to that type.

Return Value	Description
`(size_t)(-1)`	Encoding error—this isn't a valid sequence of bytes. `errno` is set to `EILSEQ`.
`(size_t)(-2)`	`n` bytes were examined and were a *partial* valid character, but not a complete one.

Return Value	Description
`(size_t)(-3)`	A subsequent value of a character that can't be represented as a single value. See below.

Case `(size_t)(-3)` is an odd one. Basically there are some characters that can't be represented with 16 bits and so can't be stored in a `char16_t`. These characters are store in something called (in the Unicode world) *surrogate pairs*. That is, there are *two* 16-bit values back to back that represent a larger Unicode value.

For example, if you want to read the Unicode character `\U0001fbc5` (which is a stick figure[1]—I'm just not putting it in the text because my font doesn't render it) that's more than 16 bits. But each call to `mbrtoc16()` only returns a single `char16_t`!

So subsequent calls to `mbrtoc16()` resolves the *next* value in the surrogate pair and returns `(size_t)(-3)` to let you know this has happened.

You can also pass `NULL` for `pc16` or `pc32`. This will cause no result to be stored, but you can use it if you're only interested in the return value from the functions.

Finally, if you pass `NULL` for `s`, the call is equivalent to:

```
mbrtoc16(NULL, "", 1, ps)
```

Since the character is a NUL in that case, this has the effect of setting the state in `ps` to the initial conversion state.

Example

Normal use case example where we get the first two character values from the multibyte string `"€Zillion"`:

```
#include <uchar.h>
#include <stdio.h>   // for printf()
#include <locale.h>  // for setlocale()
#include <string.h>  // for memset()

int main(void)
{
    char *s = "\u20acZillion";  // 20ac is "€"
    char16_t pc16;
    size_t r;
    mbstate_t mbs;

    setlocale(LC_ALL, "");
    memset(&mbs, 0, sizeof mbs);

    // Examine the next 8 bytes to see if there's a character in there
    r = mbrtoc16(&pc16, s, 8, &mbs);

    printf("%zu\n", r);     // Prints a value >= 1 (3 in UTF-8 locale)
    printf("%#x\n", pc16);  // Prints 0x20ac for "€"

    s += r;  // Move to next character

    // Examine the next 8 bytes to see if there's a character in there
    r = mbrtoc16(&pc16, s, 8, &mbs);
```

[1] https://en.wikipedia.org/wiki/Symbols_for_Legacy_Computing

```
    printf("%zu\n", r);     // Prints 1
    printf("%#x\n", pc16);  // Prints 0x5a for "Z"
}
```

Example with a surrogate pair. In this case we read plenty to get the entire character, but the result must be stored in two `char16_ts`, requiring two calls to get them both.

```
#include <uchar.h>
#include <stdio.h>   // for printf()
#include <string.h>  // for memset()
#include <locale.h>  // for setlocale()

int main(void)
{
    char *s = "\U0001fbc5*";   // Stick figure glyph, more than 16 bits
    char16_t pc16;
    mbstate_t mbs;
    size_t r;

    setlocale(LC_ALL, "");
    memset(&mbs, 0, sizeof mbs);

    r = mbrtoc16(&pc16, s, 8, &mbs);

    printf("%zd\n", r);     // r is 4 bytes in UTF-8 locale
    printf("%#x\n", pc16);  // First value of surrogate pair

    s += r;  // Move to next character

    r = mbrtoc16(&pc16, s, 8, &mbs);

    printf("%zd\n", r);     // r is (size_t)(-3) here to indicate...
    printf("%#x\n", pc16);  // ...Second value of surrogate pair

    // Since r is -3, it means we're still processing the same
    // character, so DON'T move to the next character this time
    //s += r;  // Commented out

    r = mbrtoc16(&pc16, s, 8, &mbs);

    printf("%zd\n", r);     // 1 byte for "*"
    printf("%#x\n", pc16);  // 0x2a for "*"
}
```

Output on my system, indicating the first character is represented by the pair `(0xd83e, 0xdfc5)` and the second character is represented by `0x2a`:

```
4
0xd83e
-3
0xdfc5
1
0x2a
```

See Also

c16rtomb(), c32rtomb()

68.4 c16rtomb() c32rtomb()

Convert a char16_t or char32_t to a multibyte character restartably

Synopsis

```
#include <uchar.h>

size_t c16rtomb(char * restrict s, char16_t c16, mbstate_t * restrict ps);

size_t c32rtomb(char * restrict s, char32_t c32, mbstate_t * restrict ps);
```

Description

If you have a character in a char16_t or char32_t, use these functions to convert them into a multibyte character.

These functions figure out how many bytes are needed for the multibyte character in the current locale and stores them in the buffer pointed to by s.

But how big to make that buffer? Luckily there is a macro to help: it needs be no larger than MB_CUR_MAX.

As a special case, if s is NULL, it's the same as calling

```
c16rtomb(buf, L'\0', ps);  // or...
c32rtomb(buf, L'\0', ps);
```

where buf is a buffer maintained by the system that you don't have access to.

This has the effect of setting the ps state to the initial state.

Finally for surrogate pairs (where the character has been split into two char16_ts), you call this once with the first of the pair—at this point, the function will return 0. Then you call it again with the second of the pair, and the function will return the number of bytes and store the result in the array s.

Return Value

Returns the number of bytes stored in the array pointed to by s.

Returns 0 if processing is not yet complete for the current character, as in the case of surrogate pairs.

If there is an encoding error, the functions return (size_t)(-1) and errno is set to EILSEQ.

Example

```
#include <uchar.h>
#include <stdlib.h>  // for MB_CUR_MAX
#include <stdio.h>   // for printf()
#include <string.h>  // for memset()
#include <locale.h>  // for setlocale()

```

```
int main(void)
{
    char16_t c16 = 0x20ac;  // Unicode for Euro symbol
    char dest[MB_CUR_MAX];
    size_t r;
    mbstate_t mbs;

    setlocale(LC_ALL, "");
    memset(&mbs, 0, sizeof mbs);  // Reset conversion state

    // Convert
    r = c16rtomb(dest, c16, &mbs);

    printf("r == %zd\n", r);  // r == 3 on my system

    // And this should print a Euro symbol
    printf("dest == \"%s\"\n", dest);
}
```

Output on my system:

```
r == 3
dest == "€"
```

This is a more complex example that converts a large-valued character in a multibyte string into a surrogate pair (as in the `mbrtoc16()` example, above) and then converts it back again into a multibyte string to print.

```
#include <uchar.h>
#include <stdlib.h>  // for MB_CUR_MAX
#include <stdio.h>   // for printf()
#include <string.h>  // for memset()
#include <locale.h>  // for setlocale()

int main(void)
{
    char *src = "\U0001fbc5*";   // Stick figure glyph, more than 16 bits
    char dest[MB_CUR_MAX];
    char16_t surrogate0, surrogate1;
    mbstate_t mbs;
    size_t r;

    setlocale(LC_ALL, "");
    memset(&mbs, 0, sizeof mbs);  // Reset conversion state

    // Get first surrogate character
    r = mbrtoc16(&surrogate0, src, 8, &mbs);

    // Get next surrogate character
    src += r;  // Move to next character
    r = mbrtoc16(&surrogate1, src, 8, &mbs);

    printf("Surrogate pair: %#x, %#x\n", surrogate0, surrogate1);

    // Now reverse it
    memset(&mbs, 0, sizeof mbs);  // Reset conversion state
```

```

    // Process first surrogate character
    r = c16rtomb(dest, surrogate0, &mbs);

    // r should be 0 at this point, because the character hasn't been
    // processed yet. And dest won't have anything useful... yet!
    printf("r == %zd\n", r);   // r == 0

    // Process second surrogate character
    r = c16rtomb(dest, surrogate1, &mbs);

    // Now we should be in business. r should have the number of
    // bytes, and dest should hold the character.
    printf("r == %zd\n", r);  // r == 4 on my system

    // And this should print a stick figure, if your font supports it
    printf("dest == \"%s\"\n", dest);
}
```

See Also

`mbrtoc16()`, `mbrtoc32()`

Chapter 69

<wchar.h> Wide Character Handling

Function	Description
btowc()	Convert a single byte character to a wide character
fgetwc()	Get a wide character from a wide stream
fgetws()	Read a wide string from a wide stream
fputwc()	Write a wide character to a wide stream
fputws()	Write a wide string to a wide stream
fwide()	Get or set the orientation of the stream
fwprintf()	Formatted wide output to a wide stream
fwscanf()	Formatted wide input from a wide stream
getwchar()	Get a wide character from stdin
getwc()	Get a wide character from stdin
mbrlen()	Compute the number of bytes in a multibyte character restartably
mbrtowc()	Convert multibyte to wide characters restartably
mbsinit()	Test if an mbstate_t is in the initial conversion state
mbsrtowcs()	Convert a multibyte string to a wide character string restartably
putwchar()	Write a wide character to stdout
putwc()	Write a wide character to stdout
swprintf()	Formatted wide output to a wide string
swscanf()	Formatted wide input from a wide string
ungetwc()	Pushes a wide character back into the input stream
vfwprintf()	Variadic formatted wide output to a wide stream
vfwscanf()	Variadic formatted wide input from a wide stream
vswprintf()	Variadic formatted wide output to a wide string
vswscanf()	Variadic formatted wide input from a wide string
vwprintf()	Variadic formatted wide output
vwscanf()	Variadic formatted wide input
wcscat()	Concatenate wide strings dangerously
wcschr()	Find a wide character in a wide string
wcscmp()	Compare wide strings
wcscoll()	Compare two wide strings accounting for locale
wcscpy()	Copy a wide string dangerously
wcscspn()	Count characters not from a start at the front of a wide string
wcsftime()	Formatted date and time output
wcslen()	Returns the length of a wide string
wcsncat()	Concatenate wide strings more safely
wcsncmp()	Compare wide strings, length limited

www.dbooks.org

Function	Description
`wcsncpy()`	Copy a wide string more safely
`wcspbrk()`	Search a wide string for one of a set of wide characters
`wcsrchr()`	Find a wide character in a wide string from the end
`wcsrtombs()`	Convert a wide character string to a multibyte string restartably
`wcsspn()`	Count characters from a set at the front of a wide string
`wcsstr()`	Find a wide string in another wide string
`wcstod()`	Convert a wide string to a `double`
`wcstof()`	Convert a wide string to a `float`
`wcstok()`	Tokenize a wide string
`wcstold()`	Convert a wide string to a `long double`
`wcstoll()`	Convert a wide string to a `long long`
`wcstol()`	Convert a wide string to a `long`
`wcstoull()`	Convert a wide string to an `unsigned long long`
`wcstoul()`	Convert a wide string to an `unsigned long`
`wcsxfrm()`	Transform a wide string for comparing based on locale
`wctob()`	Convert a wide character to a single byte character
`wctombr()`	Convert wide to multibyte characters restartably
`wmemcmp()`	Compare wide characters in memory
`wmemcpy()`	Copy wide character memory
`wmemmove()`	Copy wide character memory, potentially overlapping
`wprintf()`	Formatted wide output
`wscanf()`	Formatted wide input

These are the wide character variants of the functions found in `<stdio.h>`.

Remember that you can't mix-and-match multibyte output functions (like `printf()`) with wide character output functions (like `wprintf()`). The output stream has an *orientation* to either multibyte or wide that gets set on the first I/O call to that stream. (Or it can be set with `fwide()`.)

So choose one or the other and stick with it.

And you can specify wide character constants and string literals by prefixing `L` to the front of it:

```
wchar_t *s = L"Hello, world!";
wchar_t c = L'B';
```

This header also introduces a type `wint_t` that is used by the character I/O functions. It's a type that can hold any single wide character, but *also* the macro `WEOF` to indicate wide end-of-file.

69.1 Restartable Functions

Finally, a note on the "restartable" functions that are included here. When conversion is happening, some encodings require C to keep track of some *state* about the progress of the conversion so far.

For a lot of the functions, C uses an internal variable for the state that is shared between function calls. The problem is if you're writing multithreaded code, this state might get trampled by other threads.

To avoid this, each thread needs to maintain its own state in a variable of the opaque type `mbstate_t`. And the "restartable" functions allow you to pass in this state so that each thread can use their own.

69.2 wprintf(), fwprintf(), swprintf()

Formatted output with a wide string

Synopsis

```
#include <stdio.h>   // For fwprintf()
#include <wchar.h>

int wprintf(const wchar_t * restrict format, ...);

int fwprintf(FILE * restrict stream, const wchar_t * restrict format, ...);

int swprintf(wchar_t * restrict s, size_t n,
             const wchar_t * restrict format, ...);
```

Description

These are the wide versions of `printf()`, `fprintf()](#man-printf), and [`sprintf()‘.

See those pages for exact substantial usage.

These are the same except the `format` string is a wide character string instead of a multibyte string.

And that `swprintf()` is analogous to `snprintf()` in that they both take the size of the destination array as an argument.

And one more thing: the precision specified for a `%s` specifier corresponds to the number of wide characters printed, not the number of bytes. If you know of other difference, let me know.

Return Value

Returns the number of wide characters outputted, or `-1` if there's an error.

Example

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    char *mbs = "multibyte";
    wchar_t *ws = L"wide";

    wprintf(L"We're all wide for %s and %ls!\n", mbs, ws);

    double pi = 3.14159265358979;
    wprintf(L"pi = %f\n", pi);
}
```

Output:

```
We're all wide for multibyte and wide!
pi = 3.141593
```

See Also

printf(), vwprintf()

69.3 wscanf() fwscanf() swscanf()

Scan a wide stream or wide string for formatted input

Synopsis

```
#include <stdio.h>  // for fwscanf()
#include <wchar.h>

int wscanf(const wchar_t * restrict format, ...);

int fwscanf(FILE * restrict stream, const wchar_t * restrict format, ...);

int swscanf(const wchar_t * restrict s, const wchar_t * restrict format, ...);
```

Description

These are the wide variants of scanf(), fscanf(), and sscanf().

See the scanf() page for all the details.

Return Value

Returns the number of items successfully scanned, or EOF on some kind of input failure.

Example

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    int quantity;
    wchar_t item[100];

    wprintf(L"Enter \"quantity: item\"\n");

    if (wscanf(L"%d:%99ls", &quantity, item) != 2)
        wprintf(L"Malformed input!\n");
    else
        wprintf(L"You entered: %d %ls\n", quantity, item);
}
```

Output (input of 12: apples):

```
Enter "quantity: item"
12: apples
You entered: 12 apples
```

See Also

scanf(), vwscanf()

69.4 vwprintf() vfwprintf() vswprintf()

wprintf() variants using variable argument lists (va_list)

Synopsis

```
#include <stdio.h>   // For vfwprintf()
#include <stdarg.h>
#include <wchar.h>

int vwprintf(const wchar_t * restrict format, va_list arg);

int vswprintf(wchar_t * restrict s, size_t n,
              const wchar_t * restrict format, va_list arg);

int vfwprintf(FILE * restrict stream, const wchar_t * restrict format,
              va_list arg);
```

Description

These functions are the wide character variants of the vprintf(), functions. You can refer to that reference page for more details.

Return Value

Returns the number of wide characters stored, or a negative value on error.

Example

In this example, we make our own version of wprintf() called wlogger() that timestamps output. Notice how the calls to wlogger() have all the bells and whistles of wprintf().

```
#include <stdarg.h>
#include <wchar.h>
#include <time.h>

int wlogger(wchar_t *format, ...)
{
    va_list va;
    time_t now_secs = time(NULL);
    struct tm *now = gmtime(&now_secs);

    // Output timestamp in format "YYYY-MM-DD hh:mm:ss : "
    wprintf(L"%04d-%02d-%02d %02d:%02d:%02d : ",
        now->tm_year + 1900, now->tm_mon + 1, now->tm_mday,
        now->tm_hour, now->tm_min, now->tm_sec);

    va_start(va, format);
```

```
    int result = vwprintf(format, va);
    va_end(va);

    wprintf(L"\n");

    return result;
}

int main(void)
{
    int x = 12;
    float y = 3.2;

    wlogger(L"Hello!");
    wlogger(L"x = %d and y = %.2f", x, y);
}
```

Output:

```
2021-03-30 04:25:49 : Hello!
2021-03-30 04:25:49 : x = 12 and y = 3.20
```

See Also

`printf()`, `vprintf()`

69.5 `vwscanf()`, `vfwscanf()`, `vswscanf()`

`wscanf()` variants using variable argument lists (`va_list`)

Synopsis

```
#include <stdio.h>   // For vfwscanf()
#include <stdarg.h>
#include <wchar.h>

int vwscanf(const wchar_t * restrict format, va_list arg);

int vfwscanf(FILE * restrict stream, const wchar_t * restrict format,
             va_list arg);

int vswscanf(const wchar_t * restrict s, const wchar_t * restrict format,
             va_list arg);
```

Description

These are the wide counterparts to the `vscanf()` collection of functions. See their reference page for details.

Return Value

Returns the number of items successfully scanned, or `EOF` on some kind of input failure.

Example

I have to admit I was wracking my brain to think of when you'd ever want to use this. The best example I could find was one on Stack Overflow[1] that error-checks the return value from `scanf()` against the expected. A variant of that is shown below.

```
#include <stdarg.h>
#include <wchar.h>
#include <assert.h>

int error_check_wscanf(int expected_count, wchar_t *format, ...)
{
    va_list va;

    va_start(va, format);
    int count = vwscanf(format, va);
    va_end(va);

    // This line will crash the program if the condition is false:
    assert(count == expected_count);

    return count;
}

int main(void)
{
    int a, b;
    float c;

    error_check_wscanf(3, L"%d, %d/%f", &a, &b, &c);
    error_check_wscanf(2, L"%d", &a);
}
```

See Also

`wscanf()`

69.6 `getwc()` `fgetwc()` `getwchar()`

Get a wide character from an input stream

Synopsis

```
#include <stdio.h>   // For getwc() and fgetwc()
#include <wchar.h>

wint_t getwchar(void);

wint_t getwc(FILE *stream);

wint_t fgetwc(FILE *stream);
```

[1] https://stackoverflow.com/questions/17017331/c99-vscanf-for-dummies/17018046#17018046

Description

These are the wide variants of `fgetc()`.

`fgetwc()` and `getwc()` are identical except that `getwc()` might be implemented as a macro and is allowed to evaluate `stream` multiple times.

`getwchar()` is identical to `getwc()` with `stream` set to `stdin`.

I don't know why you'd ever use `getwc()` instead of `fgetwc()`, but if anyone knows, drop me a line.

Return Value

Returns the next wide character in the input stream. Return `WEOF` on end-of-file or error.

If an I/O error occurs, the error flag is also set on the stream.

If an invalid byte sequence is encountered, `errno` is set to `ILSEQ`.

Example

Reads all the characters from a file, outputting only the letter 'b's it finds in the file:

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    FILE *fp;
    wint_t c;

    fp = fopen("datafile.txt", "r"); // error check this!

    // this while-statement assigns into c, and then checks against EOF:

    while((c = fgetc(fp)) != WEOF)
        if (c == L'b')
            fputwc(c, stdout);

    fclose(fp);
}
```

See Also

`fputwc`, `fgetws`, `errno`

69.7 `fgetws()`

Read a wide string from a file

Synopsis

```
#include <stdio.h>
#include <wchar.h>
```

```
wchar_t *fgetws(wchar_t * restrict s, int n, FILE * restrict stream);
```

Description

This is the wide version of `fgets()`. See its reference page for details.

A wide `NUL` character is used to terminate the string.

Return Value

Returns `s` on success, or a `NULL` pointer on end-of-file or error.

Example

The following example reads lines from a file and prepends them with numbers:

```
#include <stdio.h>
#include <wchar.h>

#define BUF_SIZE 1024

int main(void)
{
    FILE *fp;
    wchar_t buf[BUF_SIZE];

    fp = fopen("textfile.txt", "r"); // error check this!

    int line_count = 0;

    while ((fgetws(buf, BUF_SIZE, fp)) != NULL)
        wprintf(L"%04d: %ls", ++line_count, buf);

    fclose(fp);
}
```

Example output for a file with these lines in them (without the prepended numbers):

```
0001: line 1
0002: line 2
0003: something
0004: line 4
```

See Also

`fgetwc()`, `fgets()`

69.8 `putwchar()` `putwc()` `fputwc()`

Write a single wide character to the console or to a file

Synopsis

```
#include <stdio.h>   // For putwc() and fputwc()
#include <wchar.h>

wint_t putwchar(wchar_t c);

wint_t putwc(wchar_t c, FILE *stream);

wint_t fputwc(wchar_t c, FILE *stream);
```

Description

These are the wide character equivalents to the 'fputc()' group of functions. You can find more information 'in that reference section'.

`fputwc()` and `putwc()` are identical except that `putwc()` might be implemented as a macro and is allowed to evaluate `stream` multiple times.

`putwchar()` is identical to `putwc()` with `stream` set to `stdin`.

I don't know why you'd ever use `putwc()` instead of `fputwc()`, but if anyone knows, drop me a line.

Return Value

Returns the wide character written, or `WEOF` on error.

If it's an I/O error, the error flag will be set for the stream.

If it's an encoding error, `errno` will be set to `EILSEQ`.

Example

Read all characters from a file, outputting only the letter 'b's it finds in the file:

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    FILE *fp;
    wint_t c;

    fp = fopen("datafile.txt", "r"); // error check this!

    // this while-statement assigns into c, and then checks against EOF:

    while((c = fgetc(fp)) != WEOF)
        if (c == L'b')
            fputwc(c, stdout);

    fclose(fp);
}
```

See Also

`fgetwc()`, `fputc()`, `errno`

69.9 fputws()

Write a wide string to a file

Synopsis

```
#include <stdio.h>
#include <wchar.h>

int fputws(const wchar_t * restrict s, FILE * restrict stream);
```

Description

This is the wide version of `fputs()`.

Pass in a wide string and an output stream, and it will so be written.

Return Value

Returns a non-negative value on success, or `EOF` on error.

Example

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    fputws(L"Hello, world!\n", stdout);
}
```

See Also

`fputwc()` `fputs()`

69.10 fwide()

Get or set the orientation of the stream

Synopsis

```
#include <stdio.h>
#include <wchar.h>

int fwide(FILE *stream, int mode);
```

Description

Streams can be either wide-oriented (meaning the wide functions are in use) or byte-oriented (that the regular multibyte functions are in use). Or, before an orientation is chosen, unoriented.

There are two ways to set the orientation of an unoriented stream:

- Implicitly: just use a function like `printf()` (byte oriented) or `wprintf()` (wide oriented), and the orientation will be set.
- Explicitly: use this function to set it.

You can set the orientation for the stream by passing different numbers to `mode`:

`mode`	Description
`0`	Do not alter the orientation
`-1`	Set stream to byte-oriented
`1`	Set stream to wide-oriented

(I said `-1` and `1` there, but really it could be any positive or negative number.)

Most people choose the wide or byte functions (`printf()` or `wprintf()`) and just start using them and never use `fwide()` to set the orientation.

And once the orientation is set, you can't change it. So you can't use `fwide()` for that, either.

So what can you use it for?

You can *test* to see what orientation a stream is in by passing `0` as the `mode` and checking the return value.

Return Value

Returns greater than zero if the stream is wide-oriented.

Returns less than zero if the stream is byte-oriented.

Returns zero if the stream is unoriented.

Example

Example setting to byte-oriented:

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    printf("Hello world!\n");  // Implicitly set to byte

    int mode = fwide(stdout, 0);

    printf("Stream is %s-oriented\n", mode < 0? "byte": "wide");
}
```

Output:

```
Hello world!
Stream is byte-oriented
```

Example setting to wide-oriented:

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wprintf(L"Hello world!\n");  // Implicitly set to wide

    int mode = fwide(stdout, 0);

    wprintf(L"Stream is %ls-oriented\n", mode < 0? L"byte": L"wide");
}
```

Output:

```
Hello world!
Stream is wide-oriented
```

69.11 ungetwc()

Pushes a wide character back into the input stream

Synopsis

```c
#include <stdio.h>
#include <wchar.h>

wint_t ungetwc(wint_t c, FILE *stream);
```

Description

This is the wide character variant of `ungetc()`.

It performs the reverse operation of `fgetwc()`, pushing a character back on the input stream.

The spec guarantees you can do this one time in a row. You can probably do it more times, but it's up to the implementation. If you do too many calls without an intervening read, an error could be returned.

Setting the file position discards any characters pushed by `ungetwc()` without being subsequently read.

The end-of-file flag is cleared after a successful call.

Return Value

Returns the value of the pushed character on success, or `WEOF` on failure.

Example

This example reads a piece of punctuation, then everything after it up to the next piece of punctuation. It returns the leading punctuation, and stores the rest in a string.

```c
#include <stdio.h>
#include <wctype.h>
#include <wchar.h>

```

```
wint_t read_punctstring(FILE *fp, wchar_t *s)
{
    wint_t origpunct, c;

    origpunct = fgetwc(fp);

    if (origpunct == WEOF)  // return EOF on end-of-file
        return WEOF;

    while (c = fgetwc(fp), !iswpunct(c) && c != WEOF)
        *s++ = c;  // save it in the string

    *s = L'\0'; // nul-terminate the string

    // if we read punctuation last, ungetc it so we can fgetc it next
    // time:
    if (iswpunct(c))
        ungetwc(c, fp);

    return origpunct;
}

int main(void)
{
    wchar_t s[128];
    wint_t c;

    while ((c = read_punctstring(stdin, s)) != WEOF) {
        wprintf(L"%lc: %ls\n", c, s);
    }
}
```

Sample Input:

```
!foo#bar*baz
```

Sample output:

```
!: foo
#: bar
*: baz
```

See Also

`fgetwc()`, `ungetc()`

69.12 `wcstod()` `wcstof()` `wcstold()`

Convert a wide string to a floating point number

Synopsis

```
#include <wchar.h>

double wcstod(const wchar_t * restrict nptr, wchar_t ** restrict endptr);

float wcstof(const wchar_t * restrict nptr, wchar_t ** restrict endptr);

long double wcstold(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
```

Description

These are the wide counterparts to the `strtod()` family of functions. See their reference pages for details.

Return Value

Returns the string converted to a floating point value.

Returns `0` if there's no valid number in the string.

On overflow, returns an apporpriately-signed `HUGE_VAL`, `HUGE_VALF`. or `HUGE_VALL` depending on the return type, and `errno` is set to `ERANGE`.

On underflow, returns a number no greater than the smallest normalized positive number, appropriately signed. The implemention *might* set `errno` to `ERANGE`.

Example

```
#include <wchar.h>

int main(void)
{
    wchar_t *inp = L"   123.4567beej";
    wchar_t *badchar;

    double val = wcstod(inp, &badchar);

    wprintf(L"Converted string to %f\n", val);
    wprintf(L"Encountered bad characters: %ls\n", badchar);

    val = wcstod(L"987.654321beej", NULL);
    wprintf(L"Ignoring bad chars: %f\n", val);

    val = wcstod(L"11.2233", &badchar);

    if (*badchar == L'\0')
        wprintf(L"No bad chars: %f\n", val);
    else
        wprintf(L"Found bad chars: %f, %ls\n", val, badchar);
}
```

Output:

```
Converted string to 123.456700
Encountered bad characters: beej
```

```
Ignoring bad chars: 987.654321
No bad chars: 11.223300
```

See Also

wcstol(), strtod(), errno

69.13 wcstol() wcstoll() wcstoul() wcstoull()

Convert a wide string to an integer value

Synopsis

```
#include <wchar.h>

long int wcstol(const wchar_t * restrict nptr,
                wchar_t ** restrict endptr, int base);

long long int wcstoll(const wchar_t * restrict nptr,
                      wchar_t ** restrict endptr, int base);

unsigned long int wcstoul(const wchar_t * restrict nptr,
                          wchar_t ** restrict endptr, int base);

unsigned long long int wcstoull(const wchar_t * restrict nptr,
                                wchar_t ** restrict endptr, int base);
```

Description

These are the wide counterparts to the strtol() family of functions, so see their reference pages for the details.

Return Value

Returns the integer value of the string.

If nothing can be found, 0 is returned.

If the result is out of range, the value returned is one of LONG_MIN, LONG_MAX, LLONG_MIN, LLONG_MAX, ULONG_MAX or ULLONG_MAX, as appropriate. And errno is set to ERANGE.

Example

```
#include <wchar.h>

int main(void)
{
    // All output in decimal (base 10)

    wprintf(L"%ld\n", wcstol(L"123", NULL, 0));     // 123
    wprintf(L"%ld\n", wcstol(L"123", NULL, 10));    // 123
```

```
    wprintf(L"%ld\n", wcstol(L"101010", NULL, 2));  // binary, 42
    wprintf(L"%ld\n", wcstol(L"123", NULL, 8));     // octal, 83
    wprintf(L"%ld\n", wcstol(L"123", NULL, 16));    // hex, 291

    wprintf(L"%ld\n", wcstol(L"0123", NULL, 0));    // octal, 83
    wprintf(L"%ld\n", wcstol(L"0x123", NULL, 0));   // hex, 291

    wchar_t *badchar;
    long int x = wcstol(L"   1234beej", &badchar, 0);

    wprintf(L"Value is %ld\n", x);                  // Value is 1234
    wprintf(L"Bad chars at \"%ls\"\n", badchar);    // Bad chars at "beej"
}
```

Output:

```
123
123
42
83
291
83
291
Value is 1234
Bad chars at "beej"
```

See Also

`wcstod()`, `strtol()`, `errno`, `wcstoimax()`, `wcstoumax()`

69.14 `wcscpy()` `wcsncpy()`

Copy a wide string

Synopsis

```
#include <wchar.h>

wchar_t *wcscpy(wchar_t * restrict s1, const wchar_t * restrict s2);

wchar_t *wcsncpy(wchar_t * restrict s1,
                 const wchar_t * restrict s2, size_t n);
```

Description

These are the wide versions of `strcpy()` and `strncpy()`.

They'll copy a string up to a wide NUL. Or, in the case of the safer `wcsncpy()`, until then or until `n` wide characters are copied.

If the string in `s1` is shorter than `n`, `wcsncpy()` will pad `s2` with wide NUL characters until the `n`th wide character is reached.

Even though wcsncpy() is safer because it will never overrun the end of s2 (assuming you set n correctly), it's still unsafe a NUL is not found in s1 in the first n characters. In that case, s2 will not be NUL-terminated. Always make sure n is greater than the string length of s1!

Return Value

Returns s1.

Example

```
#include <wchar.h>

int main(void)
{
    wchar_t *s1 = L"Hello!";
    wchar_t s2[10];

    wcsncpy(s2, s1, 10);

    wprintf(L"\"%ls\"\n", s2);  // "Hello!"
}
```

See Also

wmemcpy(), wmemmove() strcpy(), strncpy()

69.15 wmemcpy() wmemmove()

Copy wide characters

Synopsis

```
#include <wchar.h>

wchar_t *wmemcpy(wchar_t * restrict s1,
                 const wchar_t * restrict s2, size_t n);

wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

These are the wide versions of memcpy() and memmove().

They copy n wide characters from s2 to s1.

They're the same except that wmemmove() is guaranteed to work with overlapping memory regions, and wmemcpy() is not.

Return Value

Both functions return the pointer s1.

Example

```
#include <wchar.h>

int main(void)
{
    wchar_t s[100] = L"Goats";
    wchar_t t[100];

    wmemcpy(t, s, 6);       // Copy non-overlapping memory

    wmemmove(s + 2, s, 6);  // Copy overlapping memory

    wprintf(L"s is \"%ls\"\n", s);
    wprintf(L"t is \"%ls\"\n", t);
}
```

Output:

```
s is "GoGoats"
t is "Goats"
```

See Also

`wcscpy()`, `wcsncpy()`, `memcpy()`, `memmove()`

69.16 `wcscat()` `wcsncat()`

Concatenate wide strings

Synopsis

```
#include <wchar.h>

wchar_t *wcscat(wchar_t * restrict s1, const wchar_t * restrict s2);

wchar_t *wcsncat(wchar_t * restrict s1,
                 const wchar_t * restrict s2, size_t n);
```

Description

These are the wide variants of `strcat()` and `strncat()`.

They concatenate `s2` onto the end of `s1`.

They're the same except `wcsncat()` gives you the option to limit the number of wide characters appended.

Note that `wcsncat()` always adds a NUL terminator to the end, even if `n` characters were appended. So be sure to leave room for that.

Return Value

Both functions return the pointer `s1`.

Example

```
#include <wchar.h>

int main(void)
{
    wchar_t dest[30] = L"Hello";
    wchar_t *src = L", World!";
    wchar_t numbers[] = L"12345678";

    wprintf(L"dest before strcat: \"%ls\"\n", dest); // "Hello"

    wcscat(dest, src);
    wprintf(L"dest after strcat:  \"%ls\"\n", dest); // "Hello, world!"

    wcsncat(dest, numbers, 3); // strcat first 3 chars of numbers
    wprintf(L"dest after strncat: \"%ls\"\n", dest); // "Hello, world!123"
}
```

See Also

`strcat()`, `strncat()`

69.17 `wcscmp()`, `wcsncmp()`, `wmemcmp()`

Compare wide strings or memory

Synopsis

```
#include <wchar.h>

int wcscmp(const wchar_t *s1, const wchar_t *s2);

int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);

int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

These are the wide variants of `memcmp()`, `strcmp()`, and `strncmp()`.

`wcscmp()` and `wcsncmp()` both compare strings until a NUL character.

`wcsncmp()` also has the additional restriction that it will only compare the first `n` characters.

`wmemcmp()` is like `wcsncmp()` except it won't stop at a NUL.

The comparison is done against the character value (which might (or might not) be its Unicode code point).

Return Value

Returns zero if both regions are equal.

Returns a negative number if the region pointed to by `s1` is less than `s2`.

Returns a positive number if the region pointed to by `s1` is greater than `s2`.

Example

```c
#include <wchar.h>

int main(void)
{
    wchar_t *s1 = L"Muffin";
    wchar_t *s2 = L"Muffin Sandwich";
    wchar_t *s3 = L"Muffin";

    wprintf(L"%d\n", wcscmp(L"Biscuits", L"Kittens")); // <0 since 'B' < 'K'
    wprintf(L"%d\n", wcscmp(L"Kittens", L"Biscuits")); // >0 since 'K' > 'B'

    if (wcscmp(s1, s2) == 0)
        wprintf(L"This won't get printed because the strings differ\n");

    if (wcscmp(s1, s3) == 0)
        wprintf(L"This will print because s1 and s3 are the same\n");

    // this is a little weird...but if the strings are the same, it'll
    // return zero, which can also be thought of as "false". Not-false
    // is "true", so (!wcscmp()) will be true if the strings are the
    // same. yes, it's odd, but you see this all the time in the wild
    // so you might as well get used to it:

    if (!wcscmp(s1, s3))
        wprintf(L"The strings are the same!\n");

    if (!wcsncmp(s1, s2, 6))
        wprintf(L"The first 6 characters of s1 and s2 are the same\n");
}
```

Output:

```
-1
1
This will print because s1 and s3 are the same
The strings are the same!
The first 6 characters of s1 and s2 are the same
```

See Also

`wcscoll()`, `memcmp()`, `strcmp()`, `strncmp()`

69.18 `wcscoll()`

Compare two wide strings accounting for locale

Synopsis

```
#include <wchar.h>

int wcscoll(const wchar_t *s1, const wchar_t *s2);
```

Description

This is the wide version of `strcoll()`. See `that reference page` for details.

This is slower than `wcscmp()`, so only use it if you need the locale-specific compare.

Return Value

Returns zero if both regions are equal in this locale.

Returns a negative number if the region pointed to by `s1` is less than `s2` in this locale.

Returns a positive number if the region pointed to by `s1` is greater than `s2` in this locale.

Example

```
#include <wchar.h>
#include <locale.h>

int main(void)
{
    setlocale(LC_ALL, "");

    // If your source character set doesn't support "é" in a string
    // you can replace it with `\u00e9`, the Unicode code point
    // for "é".

    wprintf(L"%d\n", wcscmp(L"é", L"f"));   // Reports é > f, yuck.
    wprintf(L"%d\n", wcscoll(L"é", L"f"));  // Reports é < f, yay!
}
```

See Also

`wcscmp()`, `wcsxfrm()`, `strcoll()`

69.19 `wcsxfrm()`

Transform a wide string for comparing based on locale

Synopsis

```
#include <wchar.h>

size_t wcsxfrm(wchar_t * restrict s1,
               const wchar_t * restrict s2, size_t n);
```

Description

This is the wide variant of `strxfrm()`. See that reference page for details.

Return Value

Returns the length of the transformed wide string in wide characters.

If the return value is greater than `n`, all bets are off for the result in `s1`.

Example

```c
#include <wchar.h>
#include <locale.h>
#include <stdlib.h>

// Transform a string for comparison, returning a malloc'd
// result
wchar_t *get_xfrm_str(wchar_t *s)
{
    int len = wcsxfrm(NULL, s, 0) + 1;
    wchar_t *d = malloc(len * sizeof(wchar_t));

    wcsxfrm(d, s, len);

    return d;
}

// Does half the work of a regular wcscoll() because the second
// string arrives already transformed.
int half_wcscoll(wchar_t *s1, wchar_t *s2_transformed)
{
    wchar_t *s1_transformed = get_xfrm_str(s1);

    int result = wcscmp(s1_transformed, s2_transformed);

    free(s1_transformed);

    return result;
}

int main(void)
{
    setlocale(LC_ALL, "");

    // Pre-transform the string to compare against
    wchar_t *s = get_xfrm_str(L"éfg");

    // Repeatedly compare against "éfg"
    wprintf(L"%d\n", half_wcscoll(L"fgh", s));  // "fgh" > "éfg"
    wprintf(L"%d\n", half_wcscoll(L"àbc", s));  // "àbc" < "éfg"
    wprintf(L"%d\n", half_wcscoll(L"ĥij", s));  // "ĥij" > "éfg"

    free(s);
```

```
}
```

Output:

```
1
-1
1
```

See Also

wcscmp(), wcscoll(), strxfrm()

69.20 wcschr() wcsrchr()

Find a wide character in a wide string

Synopsis

```
#include <wchar.h>

wchar_t *wcschr(const wchar_t *s, wchar_t c);

wchar_t *wcsrchr(const wchar_t *s, wchar_t c);

wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
```

Description

These are the wide equivalents to strchr(), strrchr(), and memchr().

They search for wide characters in a wide string from the front (wcschr()), the end (wcsrchr()) or for an arbitrary number of wide characters (wmemchr()).

Return Value

All three functions return a pointer to the wide character found, or NULL if the character, sadly, isn't found.

Example

```
#include <wchar.h>

int main(void)
{
    // "Hello, world!"
    //       ^  ^   ^
    //       A  B   C

    wchar_t *str = L"Hello, world!";
    wchar_t *p;

    p = wcschr(str, ',');       // p now points at position A
```

```
    p = wcsrchr(str, 'o');      // p now points at position B

    p = wmemchr(str, '!', 13);   // p now points at position C

    // repeatedly find all occurrences of the letter 'B'
    str = L"A BIG BROWN BAT BIT BEEJ";

    for(p = wcschr(str, 'B'); p != NULL; p = wcschr(p + 1, 'B')) {
        wprintf(L"Found a 'B' here: %ls\n", p);
    }
}
```

Output:

```
Found a 'B' here: BIG BROWN BAT BIT BEEJ
Found a 'B' here: BROWN BAT BIT BEEJ
Found a 'B' here: BAT BIT BEEJ
Found a 'B' here: BIT BEEJ
Found a 'B' here: BEEJ
```

See Also

`strchr()`, `strrchr()`, `memchr()`

69.21 `wcsspn()` `wcscspn()`

Return the length of a wide string consisting entirely of a set of wide characters, or of not a set of wide characters

Synopsis

```
#include <wchar.h>

size_t wcsspn(const wchar_t *s1, const wchar_t *s2);

size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

Description

The are the wide character counterparts to [`strspn()`] (#man-strspn)and `strcspn()`.

They compute the length of the string pointed to by `s1` consisting entirely of the characters found in `s2`. Or, in the case of `wcscspn()`, the characters *not* found in `s2`.

Return Value

The length of the string pointed to by `s1` consisting solely of the characters in `s2` (in the case of `wcsspn()`) or of the characters *not* in `s2` (in th ecase of `wcscspn()`).

Example

```
#include <wchar.h>

int main(void)
{
    wchar_t str1[] = L"a banana";
    wchar_t str2[] = L"the bolivian navy on maneuvers in the south pacific";
    int n;

    // how many letters in str1 until we reach something that's not a vowel?
    n = wcsspn(str1, L"aeiou");
    wprintf(L"%d\n", n);  // n == 1, just "a"

    // how many letters in str1 until we reach something that's not a, b,
    // or space?
    n = wcsspn(str1, L"ab ");
    wprintf(L"%d\n", n);  // n == 4, "a ba"

    // how many letters in str2 before we get a "y"?
    n = wcscspn(str2, L"y");
    wprintf(L"%d\n", n);  // n = 16, "the bolivian nav"
}
```

See Also

`wcschr()`, `wcsrchr()`, `strspn()`

69.22 `wcspbrk()`

Search a wide string for one of a set of wide characters

Synopsis

```
#include <wchar.h>

wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
```

Description

This is the wide character variant of `strpbrk()`.

It finds the first occurrance of any of a set of wide characters in a wide string.

Return Value

Returns a pointer to the first character in the string `s1` that exists in the string `s2`.

Or `NULL` if none of the characters in `s2` can be found in `s1`.

Example

```
#include <wchar.h>

int main(void)
{
    //  p points here after wcspbrk
    //                  v
    wchar_t *s1 = L"Hello, world!";
    wchar_t *s2 = L"dow!";  // Match any of these chars

    wchar_t *p = wcspbrk(s1, s2);  // p points to the o

    wprintf(L"%ls\n", p);  // "o, world!"
}
```

See Also

`wcschr()`, `wmemchr()`, `strpbrk()`

69.23 `wcsstr()`

Find a wide string in another wide string

Synopsis

```
#include <wchar.h>

wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
```

Description

This is the wide variant of `strstr()`.

It locates a substring in a string.

Return Value

Returns a pointer to the location in `s1` that contains `s2`.

Or `NULL` if `s2` cannot be found in `s1`.

Example

```
#include <wchar.h>

int main(void)
{
    wchar_t *str = L"The quick brown fox jumped over the lazy dogs.";
    wchar_t *p;

```

```
    p = wcsstr(str, L"lazy");
    wprintf(L"%ls\n", p == NULL? L"null": p); // "lazy dogs."

    // p is NULL after this, since the string "wombat" isn't in str:
    p = wcsstr(str, L"wombat");
    wprintf(L"%ls\n", p == NULL? L"null": p); // "null"
}
```

See Also

`wcschr()`, `wcsrchr()`, `wcsspn()`, `wcscspn()`, `strstr()`

69.24 `wcstok()`

Tokenize a wide string

Synopsis

```
#include <wchar.h>
wchar_t *wcstok(wchar_t * restrict s1, const wchar_t * restrict s2,
                wchar_t ** restrict ptr);
```

Description

This is the wide version of `strtok()`.

And, like that one, it modifies the string `s1`. So make a copy of it first if you want to preserve the original.

One key difference is that `wcstok()` can be threadsafe because you pass in the pointer `ptr` to the current state of the transformation. This gets initializers for you when `s1` is initially passed in as non-`NULL`. (Subsequent calls with a `NULL` `s1` cause the state to update.)

Return Value

Example

```
#include <wchar.h>

int main(void)
{
    // break up the string into a series of space or
    // punctuation-separated words
    wchar_t str[] = L"Where is my bacon, dude?";
    wchar_t *token;
    wchar_t *state;

    // Note that the following if-do-while construct is very very
    // very very very common to see when using strtok().

    // grab the first token (making sure there is a first token!)
    if ((token = wcstok(str, L".,?! ", &state)) != NULL) {
```

```
        do {
            wprintf(L"Word: \"%ls\"\n", token);

            // now, the while continuation condition grabs the
            // next token (by passing NULL as the first param)
            // and continues if the token's not NULL:
        } while ((token = wcstok(NULL, L".,?! ", &state)) != NULL);
    }
}
```

Output:

```
Word: "Where"
Word: "is"
Word: "my"
Word: "bacon"
Word: "dude"
```

See Also

`strtok()`

69.25 `wcslen()`

Returns the length of a wide string

Synopsis

```
#include <wchar.h>

size_t wcslen(const wchar_t *s);
```

Description

This is the wide counterpart to `strlen()`.

Return Value

Returns the number of wide characters before the wide NUL terminator.

Example

```
#include <wchar.h>

int main(void)
{
    wchar_t *s = L"Hello, world!"; // 13 characters

    // prints "The string is 13 characters long.":

```

```
    wprintf(L"The string is %zu characters long.\n", wcslen(s));
}
```

See Also

`strlen()`

69.26 wcsftime()

Formatted date and time output

Synopsis

```
#include <time.h>
#include <wchar.h>

size_t wcsftime(wchar_t * restrict s, size_t maxsize,
                const wchar_t * restrict format,
                const struct tm * restrict timeptr);
```

Description

This is the wide equivalent to `strftime()`. See that reference page for details.

`maxsize` here refers to the maximum number of wide characters that can be in the result string.

Return Value

If successful, returns the number of wide characters written.

If not successful because the result couldn't fit in the space alloted, `0` is returned and the contents of the string could be anything.

Example

```
#include <wchar.h>
#include <time.h>

#define BUFSIZE 128

int main(void)
{
    wchar_t s[BUFSIZE];
    time_t now = time(NULL);

    // %c: print date as per current locale
    wcsftime(s, BUFSIZE, L"%c", localtime(&now));
    wprintf(L"%ls\n", s);   // Sun Feb 28 22:29:00 2021

    // %A: full weekday name
    // %B: full month name
```

```
    // %d: day of the month
    wcsftime(s, BUFSIZE, L"%A, %B %d", localtime(&now));
    wprintf(L"%ls\n", s);   // Sunday, February 28

    // %I: hour (12 hour clock)
    // %M: minute
    // %S: second
    // %p: AM or PM
    wcsftime(s, BUFSIZE, L"It's %I:%M:%S %p", localtime(&now));
    wprintf(L"%ls\n", s);   // It's 10:29:00 PM

    // %F: ISO 8601 yyyy-mm-dd
    // %T: ISO 8601 hh:mm:ss
    // %z: ISO 8601 time zone offset
    wcsftime(s, BUFSIZE, L"ISO 8601: %FT%T%z", localtime(&now));
    wprintf(L"%ls\n", s);   // ISO 8601: 2021-02-28T22:29:00-0800
}
```

See Also

`strftime()`

69.27 `btowc()` `wctob()`

Convert a single byte character to a wide character

Synopsis

```
#include <wchar.h>

wint_t btowc(int c);

int wctob(wint_t c);
```

Description

These functions convert between single byte characters and wide characters, and vice-versa.

Even though `int`s are involved, don't let this mislead you; they're effectively converted to `unsigned char`s internally.

The characters in the basic character set are guaranteed to be a single byte.

Return Value

`btowc()` returns the single-byte character as a wide character. Returns `WEOF` if `EOF` is passed in, or if the byte doesn't correspond to a valid wide character.

`wctob()` returns the wide character as a single-byte character. Returns `EOF` if `WEOF` is passed in, or if the wide character doesn't correspond to a value single-byte character.

See `mbtowc()` and `wctomb()` for multibyte to wide character conversion.

Example

```
#include <wchar.h>

int main(void)
{
    wint_t wc = btowc('B');    // Convert single byte to wide char

    wprintf(L"Wide character: %lc\n", wc);

    unsigned char c = wctob(wc);  // Convert back to single byte

    wprintf(L"Single-byte character: %c\n", c);
}
```

Output:

```
Wide character: B
Single-byte character: B
```

See Also

`mbtowc()`, `wctomb()`

69.28 `mbsinit()`

Test if an `mbstate_t` is in the initial conversion state

Synopsis

```
#include <wchar.h>

int mbsinit(const mbstate_t *ps);
```

Description

For a given conversion state in a `mbstate_t` variable, this function determines if it's in the initial conversion state.

Return Value

Returns non-zero if the value pointed to by `ps` is in the initial conversion state, or if `ps` is `NULL`.

Returns `0` if the value pointed to by `ps` is **not** in the initial conversion state.

Example

For me, this example doesn't do anything exciting, saying that the `mbstate_t` variable is always in the initial state. Yay.

But if have a stateful encoding like 2022-JP, try messing around with this to see if you can get into an intermediate state.

This program has a bit of code at the top that reports if your locale's encoding requires any state.

```
#include <locale.h>   // For setlocale()
#include <string.h>   // For memset()
#include <stdlib.h>   // For mbtowc()
#include <wchar.h>

int main(void)
{
    mbstate_t state;
    wchar_t wc[128];

    setlocale(LC_ALL, "");

    int is_state_dependent = mbtowc(NULL, NULL, 0);

    wprintf(L"Is encoding state dependent? %d\n", is_state_dependent);

    memset(&state, 0, sizeof state);  // Set to initial state

    wprintf(L"In initial conversion state? %d\n", mbsinit(&state));

    mbrtowc(wc, "B", 5, &state);

    wprintf(L"In initial conversion state? %d\n", mbsinit(&state));
}
```

See Also

`mbtowc()`, `wctomb()`, `mbrtowc()`, `wcrtomb()`

69.29 `mbrlen()`

Compute the number of bytes in a multibyte character, restartably

Synopsis

```
#include <wchar.h>

size_t mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps);
```

Description

This is the restartable version of `mblen()`.

It inspects at most `n` bytes of the string `s` to see how many bytes in this character.

The conversion state is stored in `ps`.

This function doesn't have the functionality of `mblen()` that allowed you to query if this character encoding was stateful and to reset the internal state.

Return Value

Returns the number of bytes required for this multibyte character.

Returns `(size_t)(-1)` if the data in `s` is not a valid multibyte character.

Returns `(size_t)(-2)` if the data is `s` is a valid but not complete multibyte character.

Example

If your character set doesn't support the Euro symbol "€", substitute the Unicode escape sequence `\u20ac`, below.

```
#include <locale.h>   // For setlocale()
#include <string.h>   // For memset()
#include <wchar.h>

int main(void)
{
    mbstate_t state;
    int len;

    setlocale(LC_ALL, "");

    memset(&state, 0, sizeof state);  // Set to initial state

    len = mbrlen("B", 5, &state);

    wprintf(L"Length of 'B' is %d byte(s)\n", len);

    len = mbrlen("€", 5, &state);

    wprintf(L"Length of '€' is %d byte(s)\n", len);
}
```

Output:

```
Length of 'B' is 1 byte(s)
Length of '€' is 3 byte(s)
```

See Also

`mblen()`

69.30 `mbrtowc()`

Convert multibyte to wide characters restartably

Synopsis

```
#include <wchar.h>

size_t mbrtowc(wchar_t * restrict pwc, const char * restrict s,
               size_t n, mbstate_t * restrict ps);
```

Description

This is the restartable counterpart to `mbtowc()`.

It converts individual characters from multibyte to wide, tracking the conversion state in the variable pointed to by `ps`.

At most `n` bytes are inspected for conversion to a wide character.

These two variants are identical and cause the state pointed to by `ps` to be set to the initial conversion state:

```
mbrtowc(NULL, NULL, 0, &state);
mbrtowc(NULL, "", 1, &state);
```

Also, if you're just interested in the length in bytes of the multibyte character, you can pass `NULL` for `pwc` and nothing will be stored for the wide character:

```
int len = mbrtowc(NULL, "€", 5, &state);
```

This function doesn't have the functionality of `mbtowc()` that allowed you to query if this character encoding was stateful and to reset the internal state.

Return Value

On success, returns a positive number corresponding to the number of bytes in the multibyte character.

Returns `0` if the character encoded is a wide NUL character.

Returns `(size_t)(-1)` if the data in `s` is not a valid multibyte character.

Returns `(size_t)(-2)` if the data is `s` is a valid but not complete multibyte character.

Example

If your character set doesn't support the Euro symbol "€", substitute the Unicode escape sequence `\u20ac`, below.

```
#include <string.h>  // For memset()
#include <stdlib.h>  // For mbtowc()
#include <locale.h>  // For setlocale()
#include <wchar.h>

int main(void)
{
    mbstate_t state;

    memset(&state, 0, sizeof state);

    setlocale(LC_ALL, "");

    wprintf(L"State dependency: %d\n", mbtowc(NULL, NULL, 0));

    wchar_t wc;
    int bytes;

    bytes = mbrtowc(&wc, "€", 5, &state);

    wprintf(L"L'%lc' takes %d bytes as multibyte char '€'\n", wc, bytes);
}
```

Output on my system:

```
State dependency: 0
L'€' takes 3 bytes as multibyte char '€'
```

See Also

`mbtowc()`, `wcrtomb()`

69.31 wcrtomb()

Convert wide to multibyte characters restartably

Synopsis

```
#include <wchar.h>

size_t wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);
```

Description

This is the restartable counterpart to `wctomb()`.

It converts individual characters from wide to multibyte, tracking the conversion state in the variable pointed to by `ps`.

The destination array `s` should be at least `MB_CUR_MAX`[2] bytes in size—you won't get anything bigger back from this function.

Note that the values in this result array won't be NUL-terminated.

If you pass a wide NUL character in, the result will contain any bytes needed to restore the conversion state to its initial state followed by a NUL character, and the state pointed to by `ps` will be reset to its initial state:

```
// Reset state
wcrtomb(mb, L'\0', &state)
```

If you don't care about the results (i.e. you're just interested in resetting the state or getting the return value), you can do this by passing `NULL` for `s`:

```
wcrtomb(NULL, L'\0', &state);                // Reset state

int byte_count = wctomb(NULL, "X", &state);  // Count bytes in 'X'
```

This function doesn't have the functionality of `wctomb()` that allowed you to query if this character encoding was stateful and to reset the internal state.

Return Value

On success, returns the number of bytes needed to encode this wide character in the current locale.

If the input is an invalid wide character, `errno` will be set to `EILSEQ` and the function returns `(size_t)(-1)`. If this happens, all bets are off for the conversion state, so you might as well reset it.

[2] This is a variable, not a macro, so if you use it to define an array, it'll be a variable-length array.

Example

If your character set doesn't support the Euro symbol "€", substitute the Unicode escape sequence `\u20ac`, below.

```
#include <string.h>  // For memset()
#include <stdlib.h>  // For mbtowc()
#include <locale.h>  // For setlocale()
#include <wchar.h>

int main(void)
{
    mbstate_t state;

    memset(&state, 0, sizeof state);

    setlocale(LC_ALL, "");

    wprintf(L"State dependency: %d\n", mbtowc(NULL, NULL, 0));

    char mb[10] = {0};
    int bytes = wcrtomb(mb, L'€', &state);

    wprintf(L"L'€' takes %d bytes as multibyte char '%s'\n", bytes, mb);
}
```

See Also

`mbrtowc()`, `wctomb()`, `errno`

69.32 `mbsrtowcs()`

Convert a multibyte string to a wide character string restartably

Synopsis

```
#include <wchar.h>

size_t mbsrtowcs(wchar_t * restrict dst, const char ** restrict src,
                 size_t len, mbstate_t * restrict ps);
```

Description

This is the restartable version of `mbstowcs()`.

It converts a multibyte string to a wide character string.

The result is put in the buffer pointed to by `dst`, and the pointer `src` is updated to indicate how much of the string was consumed (unless `dst` is `NULL`).

At most `len` wide characters will be stored.

This also takes a pointer to its own `mbstate_t` variable in `ps` for holding the conversion state.

You can set `dst` to `NULL` if you only care about the return value. This could be useful for getting the number of characters in a multibyte string.

In the normal case, the `src` string will be consumed up to the NUL character, and the results will be stored in the `dst` buffer, including the wide NUL character. In this case, the pointer pointed to by `src` will be set to `NULL`. And the conversion state will be set to the initial conversion state.

If things go wrong because the source string isn't a valid sequence of characters, conversion will stop and the pointer pointed to by `src` will be set to the address just after the last successfully-translated multibyte character.

Return Value

If successful, returns the number of characters converted, not including any NUL terminator.

If the multibyte sequence is invalid, the function returns `(size_t)(-1)` and `errno` is set to `EILSEQ`.

Example

Here we'll convert the string "€5 ± π" into a wide character string:

```
#include <locale.h>  // For setlocale()
#include <string.h>  // For memset()
#include <wchar.h>

#define WIDE_STR_SIZE 10

int main(void)
{
    const char *mbs = "€5 ± π";  // That's the exact price range

    wchar_t wcs[WIDE_STR_SIZE];

    setlocale(LC_ALL, "");

    mbstate_t state;
    memset(&state, 0, sizeof state);

    size_t count = mbsrtowcs(wcs, &mbs, WIDE_STR_SIZE, &state);

    wprintf(L"Wide string L\"%ls\" is %d characters\n", wcs, count);
}
```

Output:

```
Wide string L"€5 ± π" is 6 characters
```

Here's another example of using `mbsrtowcs()` to get the length in characters of a multibyte string even if the string is full of multibyte characters. This is in contrast to `strlen()`, which returns the total number of bytes in the string.

```
#include <stdio.h>   // For printf()
#include <locale.h>  // For setlocale()

#include <string.h>  // For memset()
#include <stdint.h>  // For SIZE_MAX
#include <wchar.h>

```

```
size_t mbstrlen(const char *mbs)
{
    mbstate_t state;

    memset(&state, 0, sizeof state);

    return mbsrtowcs(NULL, &mbs, SIZE_MAX, &state);
}

int main(void)
{
    setlocale(LC_ALL, "");

    char *mbs = "€5 ± π";  // That's the exact price range

    printf("\"%s\" is %zu characters...\n", mbs, mbstrlen(mbs));
    printf("but it's %zu bytes!\n", strlen(mbs));
}
```

Output on my system:

```
"€5 ± π" is 6 characters...
but it's 10 bytes!
```

See Also

`mbrtowc()`, `mbstowcs()`, `wcsrtombs()`, `strlen()`, `errno`

69.33 `wcsrtombs()`

Convert a wide character string to a multibyte string restartably

Synopsis

```
#include <wchar.h>

size_t wcsrtombs(char * restrict dst, const wchar_t ** restrict src,
                 size_t len, mbstate_t * restrict ps);
```

Description

If you have a wide character string, you can convert it to a multibyte character string in the current locale using this function.

At most `len` bytes of data will be stored in the buffer pointed to by `dst`. Conversion will stop just after the NUL terminator is copied, or `len` bytes get copied, or some other error occurs.

If `dst` is a `NULL` pointer, no result is stored. You might do this if you're just interested in the return value (nominally the number of bytes this would use in a multibyte string, not including the NUL terminator).

If `dst` is not a `NULL` pointer, the pointer pointed to by `src` will get modified to indicate how much of the data was copied. If it contains `NULL` at the end, it means everything went well. In this case, the state `ps` will be set to the initial conversion state.

If `len` was reached or an error occurred, it'll point one address past `dst+len`.

Return Value

If everything goes well, returns the number of bytes needed for the multibyte string, not counting the NUL terminator.

If any character in the string doesn't correspond to a valid multibyte character in the currently locale, it returns `(size_t)(-1)` and `EILSEQ` is stored in `errno`.

Example

Here we'll convert the wide string "€5 ± π" into a multibyte character string:

```
#include <locale.h>  // For setlocale()
#include <string.h>  // For memset()
#include <wchar.h>

#define MB_STR_SIZE 20

int main(void)
{
    const wchar_t *wcs = L"€5 ± π";  // That's the exact price range

    char mbs[MB_STR_SIZE];

    setlocale(LC_ALL, "");

    mbstate_t state;
    memset(&state, 0, sizeof state);

    size_t count = wcsrtombs(mbs, &wcs, MB_STR_SIZE, &state);

    wprintf(L"Multibyte string \"%s\" is %d bytes\n", mbs, count);
}
```

Here's another example helper function that `malloc()`s just enough memory to hold the converted string, then returns the result. (Which must later be freed, of course, to prevent leaking memory.)

```
#include <stdlib.h>  // For malloc()
#include <locale.h>  // For setlocale()
#include <string.h>  // For memset()
#include <stdint.h>  // For SIZE_MAX
#include <wchar.h>

char *get_mb_string(const wchar_t *wcs)
{
    setlocale(LC_ALL, "");

    mbstate_t state;
    memset(&state, 0, sizeof state);

    // Need a copy of this because wcsrtombs changes it
    const wchar_t *p = wcs;

    // Compute the number of bytes needed to hold the result
```

```
    size_t bytes_needed = wcsrtombs(NULL, &p, SIZE_MAX, &state);

    // If we didn't get a good full conversion, forget it
    if (bytes_needed == (size_t)(-1))
        return NULL;

    // Allocate space for result
    char *mbs = malloc(bytes_needed + 1);  // +1 for NUL terminator

    // Set conversion state to initial state
    memset(&state, 0, sizeof state);

    // Convert and store result
    wcsrtombs(mbs, &wcs, bytes_needed + 1, &state);

    // Make sure things went well
    if (wcs != NULL) {
        free(mbs);
        return NULL;
    }

    // Success!
    return mbs;
}

int main(void)
{
    char *mbs = get_mb_string(L"€5 ± π");

    wprintf(L"Multibyte result: \"%s\"\n", mbs);

    free(mbs);
}
```

See Also

wcrtomb(), wcstombs(), mbsrtowcs(), errno

Chapter 70

<wctype.h> Wide Character Classification and Transformation

Function	Description
iswalnum()	Test if a wide character is alphanumeric.
iswalpha()	Tests if a wide character is alphabetic
iswblank()	Tests if this is a wide blank character
iswcntrl()	Tests if this is a wide control character.
iswctype()	Determine wide character classification
iswdigit()	Test if this wide character is a digit
iswgraph()	Test to see if a wide character is a printable non-space
iswlower()	Tests if a wide character is lowercase
iswprint()	Tests if a wide character is printable
iswpunct()	Test if a wide character is punctuation
iswspace()	Test if a wide character is whitespace
iswupper()	Tests if a wide character is uppercase
iswxdigit()	Tests if a wide character is a hexadecimal digit
towctrans()	Convert wide characters to upper or lowercase
towlower()	Convert an uppercase wide character to lowercase
towupper()	Convert a lowercase wide character to uppercase
wctrans()	Helper function for towctrans()
wctype()	Helper function for iswctype()

This is like <ctype.h> except for wide characters.

With it you can test for character classifications (like “is this character whitespace?”) or do basic character conversions (like “force this character to lowercase”).

70.1 iswalnum()

Test if a wide character is alphanumeric.

Synopsis

```
#include <wctype.h>

int iswalnum(wint_t wc);
```

Description

Basically tests if a character is alphabetic (A-Z or a-z) or a digit (0-9). But some other characters might also qualify based on the locale.

This is equivalent to testing if iswalpha() or iswdigit() is true.

Return Value

Returns true if the character is alphanumeric.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                  testing this char
    //                            v
    wprintf(L"%ls\n", iswalnum(L'a')? L"yes": L"no");  // yes
    wprintf(L"%ls\n", iswalnum(L'B')? L"yes": L"no");  // yes
    wprintf(L"%ls\n", iswalnum(L'5')? L"yes": L"no");  // yes
    wprintf(L"%ls\n", iswalnum(L'?')? L"yes": L"no");  // no
}
```

See Also

iswalpha(), iswdigit(), isalnum()

70.2 iswalpha()

Tests if a wide character is alphabetic

Synopsis

```
#include <wctype.h>

int iswalpha(wint_t wc);
```

Description

Basically tests if a character is alphabetic (A-Z or a-z). But some other characters might also qualify based on the locale. (If other characters qualify, they won't be control characters, digits, punctuation, or spaces.)

This is the same as testing for iswupper() or iswlower().

Return Value

Returns true if the character is alphabetic.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                  testing this char
    //                            v
    wprintf(L"%ls\n", iswalpha(L'a')? L"yes": L"no");  // yes
    wprintf(L"%ls\n", iswalpha(L'B')? L"yes": L"no");  // yes
    wprintf(L"%ls\n", iswalpha(L'5')? L"yes": L"no");  // no
    wprintf(L"%ls\n", iswalpha(L'?')? L"yes": L"no");  // no
}
```

See Also

`iswalnum()`, `isalpha()`

70.3 `iswblank()`

Tests if this is a wide blank character

Synopsis

```
#include <wctype.h>

int iswblank(wint_t wc);
```

Description

Blank characters are whitespace that are also used as word separators on the same line. In the “C” locale, the only blank characters are space and tab.

Other locales might define other blank characters.

Return Value

Returns true if this is a blank character.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
```

```
    //                    testing this char
    //                            v
    wprintf(L"%ls\n", iswblank(L' ')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswblank(L'\t')? L"yes": L"no");  // yes
    wprintf(L"%ls\n", iswblank(L'\n')? L"yes": L"no");  // no
    wprintf(L"%ls\n", iswblank(L'a')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswblank(L'?')? L"yes": L"no");   // no
}
```

See Also

`iswspace()`, `isblank()`

70.4 `iswcntrl()`

Tests if this is a wide control character.

Synopsis

```
#include <wctype.h>

int iswcntrl(wint_t wc);
```

Description

The spec is pretty barren, here. But I'm just going to assume that it works like the non-wide version. So let's look at that.

A *control character* is a locale-specific non-printing character.

For the "C" locale, this means control characters are in the range 0x00 to 0x1F (the character right before SPACE) and 0x7F (the DEL character).

Basically if it's not an ASCII (or Unicode less than 128) printable character, it's a control character in the "C" locale.

Probably.

Return Value

Returns true if this is a control character.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                    testing this char
    //                            v
    wprintf(L"%ls\n", iswcntrl(L'\t')? L"yes": L"no");  // yes (tab)
```

```
    wprintf(L"%ls\n", iswcntrl(L'\n')? L"yes": L"no");  // yes (newline)
    wprintf(L"%ls\n", iswcntrl(L'\r')? L"yes": L"no");  // yes (return)
    wprintf(L"%ls\n", iswcntrl(L'\a')? L"yes": L"no");  // yes (bell)
    wprintf(L"%ls\n", iswcntrl(L' ')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswcntrl(L'a')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswcntrl(L'?')? L"yes": L"no");   // no
}
```

See Also

`iscntrl()`

70.5 `iswdigit()`

Test if this wide character is a digit

Synopsis

```
#include <wctype.h>

int iswdigit(wint_t wc);
```

Description

Tests if the wide character is a digit (`0-9`).

Return Value

Returns true if the character is a digit.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                   testing this char
    //                            v
    wprintf(L"%ls\n", iswdigit(L'0')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswdigit(L'5')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswdigit(L'a')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswdigit(L'B')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswdigit(L'?')? L"yes": L"no");   // no
}
```

See Also

`iswalnum()`, `isdigit()`

70.6 iswgraph()

Test to see if a wide character is a printable non-space

Synopsis

```
#include <wctype.h>

int iswgraph(wint_t wc);
```

Description

Returns true if this is a printable (non-control) character and also not a whitespace character.

Basically if `iswprint()` is true and `iswspace()` is false.

Return Value

Returns true if this is a printable non-space character.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                   testing this char
    //                            v
    wprintf(L"%ls\n", iswgraph(L'0')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswgraph(L'a')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswgraph(L'B')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswgraph(L'?')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswgraph(L' ')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswgraph(L'\n')? L"yes": L"no");  // no
}
```

See Also

`iswprint()`, `iswspace()`, `isgraph()`

70.7 iswlower()

Tests if a wide character is lowercase

Synopsis

```
#include <wctype.h>

int iswlower(wint_t wc);
```

Description

Tests if a character is lowercase, in the range a-z.

In other locales, there could be other lowercase characters. In all cases, to be lowercase, the following must be true:

```
!iswcntrl(c) && !iswdigit(c) && !iswpunct(c) && !iswspace(c)
```

Return Value

Returns true if the wide character is lowercase.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                    testing this char
    //                            v
    wprintf(L"%ls\n", iswlower(L'c')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswlower(L'0')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswlower(L'B')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswlower(L'?')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswlower(L' ')? L"yes": L"no");   // no
}
```

See Also

islower(), iswupper(), iswalpha(), towupper(), towlower()

70.8 iswprint()

Tests if a wide character is printable

Synopsis

```
#include <wctype.h>

int iswprint(wint_t wc);
```

Description

Tests if a wide character is printable, including space (' '). So like isgraph(), except space isn't left out in the cold.

Return Value

Returns true if the wide character is printable, including space (' ').

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                    testing this char
    //                            v
    wprintf(L"%ls\n", iswprint(L'c')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswprint(L'0')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswprint(L' ')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswprint(L'\r')? L"yes": L"no");  // no
}
```

See Also

`isprint()`, `iswgraph()`, `iswcntrl()`

70.9 `iswpunct()`

Test if a wide character is punctuation

Synopsis

```
#include <wctype.h>

int iswpunct(wint_t wc);
```

Description

Tests if a wide character is punctuation.

This means for any given locale:

```
!isspace(c) && !isalnum(c)
```

Return Value

True if the wide character is punctuation.

Example

Results may vary based on locale.

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                    testing this char
    //                            v
```

```
    wprintf(L"%ls\n", iswpunct(L',')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswpunct(L'!')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswpunct(L'c')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswpunct(L'0')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswpunct(L' ')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswpunct(L'\n')? L"yes": L"no");  // no
}
```

See Also

`ispunct()`, `iswspace()`, `iswalnum()`

70.10 `iswspace()`

Test if a wide character is whitespace

Synopsis

```
#include <wctype.h>

int iswspace(wint_t wc);
```

Description

Tests if `c` is a whitespace character. These are probably:

- Space (`' '`)
- Formfeed (`'\f'`)
- Newline (`'\n'`)
- Carriage Return (`'\r'`)
- Horizontal Tab (`'\t'`)
- Vertical Tab (`'\v'`)

Other locales might specify other whitespace characters. `iswalnum()`, `iswgraph()`, and `iswpunct()` are all false for all whitespace characters.

Return Value

True if the character is whitespace.

Example

Results may vary based on locale.

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                    testing this char
    //                            v
    wprintf(L"%ls\n", iswspace(L' ')? L"yes": L"no");   // yes
```

```
    wprintf(L"%ls\n", iswspace(L'\n')? L"yes": L"no");  // yes
    wprintf(L"%ls\n", iswspace(L'\t')? L"yes": L"no");  // yes
    wprintf(L"%ls\n", iswspace(L',')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswspace(L'!')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswspace(L'c')? L"yes": L"no");   // no
}
```

See Also

`isspace()`, `iswblank()`

70.11 `iswupper()`

Tests if a wide character is uppercase

Synopsis

```
#include <wctype.h>

int iswupper(wint_t wc);
```

Description

Tests if a character is uppercase in the current locale.

To be uppercase, the following must be true:

```
!iscntrl(c) && !isdigit(c) && !ispunct(c) && !isspace(c)
```

Return Value

Returns true if the wide character is uppercase.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                    testing this char
    //                             v
    wprintf(L"%ls\n", iswupper(L'B')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswupper(L'c')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswupper(L'0')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswupper(L'?')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswupper(L' ')? L"yes": L"no");   // no
}
```

See Also

`isupper()`, `iswlower()`, `iswalpha()`, `towupper()`, `towlower()`

70.12 iswxdigit()

Tests if a wide character is a hexadecimal digit

Synopsis

```
#include <wctype.h>

int iswxdigit(wint_t wc);
```

Description

Returns true if the wide character is a hexadecimal digit. Namely if it's `0-9`, `a-f`, or `A-F`.

Return Value

True if the character is a hexadecimal digit.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                    testing this char
    //                            v
    wprintf(L"%ls\n", iswxdigit(L'B')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswxdigit(L'c')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswxdigit(L'2')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswxdigit(L'G')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswxdigit(L'?')? L"yes": L"no");   // no
}
```

See Also

`isxdigit()`, `iswdigit()`

70.13 iswctype()

Determine wide character classification

Synopsis

```
#include <wctype.h>

int iswctype(wint_t wc, wctype_t desc);
```

Description

This is the Swiss Army knife of classification functions; it's all the other ones rolled into one.

You call it with something like this:

```
if (iswctype(c, wctype("digit")))  // or "alpha" or "space" or...
```

and it behaves just like you'd called:

```
if (iswdigit(c))
```

The difference is that you can specify the type of matching you want to do as a string at runtime, which might be convenient.

`iswctype()` relies on the return value from the `wctype()` call to get its work done.

Stolen from the spec, here are the `iswctype()` calls and their equivalents:

`iswctype()` call	Hard-coded equivalent
`iswctype(c, wctype("alnum"))`	`iswalnum(c)`
`iswctype(c, wctype("alpha"))`	`iswalpha(c)`
`iswctype(c, wctype("blank"))`	`iswblank(c)`
`iswctype(c, wctype("cntrl"))`	`iswcntrl(c)`
`iswctype(c, wctype("digit"))`	`iswdigit(c)`
`iswctype(c, wctype("graph"))`	`iswgraph(c)`
`iswctype(c, wctype("lower"))`	`iswlower(c)`
`iswctype(c, wctype("print"))`	`iswprint(c)`
`iswctype(c, wctype("punct"))`	`iswpunct(c)`
`iswctype(c, wctype("space"))`	`iswspace(c)`
`iswctype(c, wctype("upper"))`	`iswupper(c)`
`iswctype(c, wctype("xdigit"))`	`iswxdigit(c)`

See the `wctype()` documentation for how that helper function works.

Return Value

Returns true if the wide character `wc` matches the character class in `desc`.

Example

Test for a given character classification at when the classification isn't known at compile time:

```
#include <stdio.h>  // for fflush(stdout)
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    wchar_t c;        // Holds a single wide character (to test)
```

```
    char desc[128];   // Holds the character class

    // Get the character and classification from the user
    wprintf(L"Enter a character and character class: ");
    fflush(stdout);
    wscanf(L"%lc %s", &c, desc);

    // Compute the type from the given class
    wctype_t t = wctype(desc);

    if (t == 0)
        // If the type is 0, it's an unknown class
        wprintf(L"Unknown character class: \"%s\"\n", desc);
    else {
        // Otherwise, let's test the character and see if its that
        // classification
        if (iswctype(c, t))
            wprintf(L"Yes! '%lc' is %s!\n", c, desc);
        else
            wprintf(L"Nope! '%lc' is not %s.\n", c, desc);
    }
}
```

Output:

```
Enter a character and character class: 5 digit
Yes! '5' is digit!

Enter a character and character class: b digit
Nope! 'b' is not digit.

Enter a character and character class: x alnum
Yes! 'x' is alnum!
```

See Also

`wctype()`

70.14 `wctype()`

Helper function for `iswctype()`

Synopsis

```
#include <wctype.h>

wctype_t wctype(const char *property);
```

Description

This function returns an opaque value for the given `property` that is meant to be passed as the second argument to `iswctype()`.

The returned value is of type `wctype_t`.

Valid properties in all locales are:

```
"alnum"   "alpha"   "blank"   "cntrl"
"digit"   "graph"   "lower"   "print"
"punct"   "space"   "upper"   "xdigit"
```

Other properties might be defined as determined by the `LC_CTYPE` category of the current locale.

See the `iswctype()` reference page for more usage details.

Return Value

Returns the `wctype_t` value associated with the given `property`.

If an invalid value is passed for `property`, returns `0`.

Example

Test for a given character classification at when the classification isn't known at compile time:

```
#include <stdio.h>  // for fflush(stdout)
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    wchar_t c;        // Holds a single wide character (to test)
    char desc[128];   // Holds the character class

    // Get the character and classification from the user
    wprintf(L"Enter a character and character class: ");
    fflush(stdout);
    wscanf(L"%lc %s", &c, desc);

    // Compute the type from the given class
    wctype_t t = wctype(desc);

    if (t == 0)
        // If the type is 0, it's an unknown class
        wprintf(L"Unknown character class: \"%s\"\n", desc);
    else {
        // Otherwise, let's test the character and see if its that
        // classification
        if (iswctype(c, t))
            wprintf(L"Yes! '%lc' is %s!\n", c, desc);
        else
            wprintf(L"Nope! '%lc' is not %s.\n", c, desc);
    }
}
```

Output:

```
Enter a character and character class: 5 digit
Yes! '5' is digit!

Enter a character and character class: b digit
```

```
Nope! 'b' is not digit.

Enter a character and character class: x alnum
Yes! 'x' is alnum!
```

See Also

iswctype()

70.15 towlower()

Convert an uppercase wide character to lowercase

Synopsis

```
#include <wctype.h>

wint_t towlower(wint_t wc);
```

Description

If the character is upper (i.e. iswupper(c) is true), this function returns the corresponding lowercase letter.

Different locales might have different upper and lowercase letters.

Return Value

If the letter wc is uppercase, a lowercase version of that letter will be returned according to the current locale.

If the letter is not uppercase, wc is returned unchanged.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                  changing this char
    //                            v
    wprintf(L"%lc\n", towlower(L'B'));  // b (made lowercase!)
    wprintf(L"%lc\n", towlower(L'e'));  // e (unchanged)
    wprintf(L"%lc\n", towlower(L'!'));  // ! (unchanged)
}
```

See Also

tolower(), towupper(), iswlower(), iswupper()

70.16 towupper()

Convert a lowercase wide character to uppercase

Synopsis

```
#include <wctype.h>

wint_t towupper(wint_t wc);
```

Description

If the character is lower (i.e. `iswlower(c)` is true), this function returns the corresponding uppercase letter.

Different locales might have different upper and lowercase letters.

Return Value

If the letter `wc` is lowercase, an uppercase version of that letter will be returned according to the current locale.

If the letter is not lowercase, `wc` is returned unchanged.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                  changing this char
    //                            v
    wprintf(L"%lc\n", towupper(L'B'));  // B (unchanged)
    wprintf(L"%lc\n", towupper(L'e'));  // E (made uppercase!)
    wprintf(L"%lc\n", towupper(L'!'));  // ! (unchanged)
}
```

See Also

`toupper()`, `towlower()`, `iswlower()`, `iswupper()`

70.17 towctrans()

Convert wide characters to upper or lowercase

Synopsis

```
#include <wctype.h>

wint_t towctrans(wint_t wc, wctrans_t desc);
```

Description

This is the Swiss Army knife of character conversion functions; it's all the other ones rolled into one. And by "all the other ones" I mean `towupper()` and `towlower()`, since those are the only ones there are.

You call it with something like this:

```
if (towctrans(c, wctrans("toupper")))  // or "tolower"
```

and it behaves just like you'd called:

```
towupper(c);
```

The difference is that you can specify the type of conversion you want to do as a string at runtime, which might be convenient.

`towctrans()` relies on the return value from the `wctrans()` call to get its work done.

`towctrans()` call	Hard-coded equivalent
`towctrans(c, wctrans("toupper"))`	`towupper(c)`
`towctrans(c, wctrans("tolower"))`	`towlower(c)`

See the `wctrans()` documentation for how that helper function works.

Return Value

Returns the character `wc` as if run through `towupper()` or `towlower()`, depending on the value of `desc`.

If the character already matches the classification, it is returned as-is.

Example

```
#include <stdio.h>  // for fflush(stdout)
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    wchar_t c;        // Holds a single wide character (to test)
    char desc[128];   // Holds the conversion type

    // Get the character and conversion type from the user
    wprintf(L"Enter a character and conversion type: ");
    fflush(stdout);
    wscanf(L"%lc %s", &c, desc);

    // Compute the type from the given conversion type
    wctrans_t t = wctrans(desc);

    if (t == 0)
        // If the type is 0, it's an unknown conversion type
        wprintf(L"Unknown conversion: \"%s\"\n", desc);
    else {
        // Otherwise, let's do the conversion
        wint_t result = towctrans(c, t);
        wprintf(L"'%lc' -> %s -> '%lc'\n", c, desc, result);
```

```
    }
}
```

Output on my system:

```
Enter a character and conversion type: b toupper
'b' -> toupper -> 'B'

Enter a character and conversion type: B toupper
'B' -> toupper -> 'B'

Enter a character and conversion type: B tolower
'B' -> tolower -> 'b'

Enter a character and conversion type: ! toupper
'!' -> toupper -> '!'
```

See Also

`wctrans()`, `towupper()`, `towlower()`

70.18 `wctrans()`

Helper function for `towctrans()`

Synopsis

```
#include <wctype.h>

wctrans_t wctrans(const char *property);
```

Description

This is a helper function for generating the second argument to `towctrans()`.

You can pass in one of two things for the `property`:

- `toupper` to make `towctrans()` behave like `towupper()`
- `tolower` to make `towctrans()` behave like `towlower()`

Return Value

On success, returns a value that can be used as the `desc` argument to `towctrans()`.

Otherwise, if the `property` isn't recognized, returns `0`.

Example

```
#include <stdio.h>  // for fflush(stdout)
#include <wchar.h>
#include <wctype.h>

```

```
int main(void)
{
    wchar_t c;        // Holds a single wide character (to test)
    char desc[128];   // Holds the conversion type

    // Get the character and conversion type from the user
    wprintf(L"Enter a character and conversion type: ");
    fflush(stdout);
    wscanf(L"%lc %s", &c, desc);

    // Compute the type from the given conversion type
    wctrans_t t = wctrans(desc);

    if (t == 0)
        // If the type is 0, it's an unknown conversion type
        wprintf(L"Unknown conversion: \"%s\"\n", desc);
    else {
        // Otherwise, let's do the conversion
        wint_t result = towctrans(c, t);
        wprintf(L"'%lc' -> %s -> '%lc'\n", c, desc, result);
    }
}
```

Output on my system:

```
Enter a character and conversion type: b toupper
'b' -> toupper -> 'B'

Enter a character and conversion type: B toupper
'B' -> toupper -> 'B'

Enter a character and conversion type: B tolower
'B' -> tolower -> 'b'

Enter a character and conversion type: ! toupper
'!' -> toupper -> '!'
```

See Also

`towctrans()`

Chapter 71

Exercises

These are the exercises for the various chapters.

Warning: if you look at the solution before you try to solve it yourself, you're only gaining 10%[1] of the potential learning. Work on it for a while before giving up and looking at the answer.

71.1 Intro

1. Modify the "Hello, world!" example to print your name instead. (Solution.[2])
2. Modify the "Hello, world!" example to print your name one one line, and your favorite fruit on the next. (Solution.[3])

71.2 Variables and Statements

1. An approximation of π is $208341/66317$. Write a program that computes this, stores the result in a `float` variable, and prints it out. Hint: put a `.0` after at least one of the numbers to force the computation to be `float` and not integer. (Solution.[4])
2. Choose a random integer in your head between 50 and 97. Call it x. Compute $(x + 10 \times x - 5)/2$ as an integer and print it. (Solution.[5])
3. Do the same thing as the previous problem, except divide it by `2.0` instead of `2` and store the result in a `float`. And print it. (Solution.[6])
4. Let's say you have a variable `i = 2`. Write a program that assigns `j = i + 7` (to get `9`) *and* then increments `i` on the same line of code. (Solution.[7])
5. Let's say you have a variable `i = 2`. Write a program that decrements `i` *and* then assigns `j = i + 7` (to get `8`) on the same line of code. (Solution.[8])
6. Choose a value for an integer variable `i`. Write a program that prints `"foo"` if `i` is greater than 17, otherwise it prints `"bar"`. (Solution.[9])

[1] In my non-scientific estimation. But for realsies, do the work before looking at the answer. It's the best way to learn.
[2] https://beej.us/guide/bgc/source/exercises/intro/hello_name.c
[3] https://beej.us/guide/bgc/source/exercises/intro/hello_fruit.c
[4] https://beej.us/guide/bgc/source/exercises/varstat/pi_approx.c
[5] https://beej.us/guide/bgc/source/exercises/varstat/formula1.c
[6] https://beej.us/guide/bgc/source/exercises/varstat/formula2.c
[7] https://beej.us/guide/bgc/source/exercises/varstat/postinc.c
[8] https://beej.us/guide/bgc/source/exercises/varstat/predec.c
[9] https://beej.us/guide/bgc/source/exercises/varstat/foo17.c

www.dbooks.org

7. If you used `if-else` for the previous problem, do the same thing using the ternary operator. Otherwise, do the reverse. (Solution.[10])

8. Use a while loop to determine how many times you can do an integer divide by 7 on the number 20754371 until it reaches zero. (Solution.[11])

9. Use a single `for` loop with two variables to print out:

```
0 0
1 1
2 3
3 6
4 10
5 15
```

   where the second number is its previous value plus the first number. Print up to 99 in the left column. The last value for `j` will be `4950`. (Solution.[12])

71.3 Functions

1. Write a function that has two integer parameters `x` and `y`, and computes

   $x \times 20 + x \times y$

   and returns the result. (Solution.[13])

2. Write a function to compute π through Gregory and Leibniz infinite series to a specified number of terms (where the first term is $1/1$ (AKA just 1), below):

   $\pi = 4(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots)$

   Print out the results of the series for terms 1 to 5000. The 5000th iteration should produce 3.141397. (Solution.[14])

71.4 Pointers

1. Write a function that takes a pointer to an `int` and a pointer to a `float` as arguments. It should add the `int` to the `float` and subtract the `float` from the `int`. It should appear as if both these happened simultaneously, for example input of `5` and `3.2` would give `1` and `8.2`.

   Since you're manipulating the value through the pointers, the result should be visible in the caller. (Solution.[15])

2. Write a function that takes two `int*` arguments and swaps them.

```c
int a = 10, b = 20;

swap(&a, &b);

printf("%d, %d\n", a, b); // Should print 20, 10
```

   (Solution.[16])

[10] https://beej.us/guide/bgc/source/exercises/varstat/foo17b.c
[11] https://beej.us/guide/bgc/source/exercises/varstat/whilediv7.c
[12] https://beej.us/guide/bgc/source/exercises/varstat/twofor.c
[13] https://beej.us/guide/bgc/source/exercises/functions/funcxy.c
[14] https://beej.us/guide/bgc/source/exercises/functions/pi.c
[15] https://beej.us/guide/bgc/source/exercises/pointers/pointers.c
[16] https://beej.us/guide/bgc/source/exercises/pointers/swap.c

71.5 Arrays

1. Declare and initialize an `int` array containing the following values:

   ```
   10, 5, 2, 30, 97, 64
   ```

   Write a function `sum()` that accepts the array and its length as arguments. Have the function return the sum of the elements in the array (which is `208`). (Solution.[17])

2. Declare an array of 2048 elements with an initializer that sets the element at index `312` to `3490`. All other elements should be initialized to zero. (Solution.[18])

3. Write a function that checks a tie-tac-toe[19] board for a winner. A `0` on the 3×3 board means no one has moved there yet. A `1` means "X" has moved there, and a `2` means "O" has. Return `0`, `1`, or `2` for no winners, "X" won, or "O" won, respectively. (Solution.[20])

71.6 Strings

1. Write a function that reverses a string in-place. Have it return the same pointer that was passed in. (Solution.[21])

2. Write a function that returns true if a string is entirely uppercase. Hint: see the `isupper()` function. (Solution.[22])

3. Write a function that returns a pointer to the first occurrence of a letter in a string. If the character doesn't appear in the string, return `NULL`. (Solution.[23])

71.7 Structs

1. Write a `struct point` that has two `double` members called `x` and `y`, useful for storing a 2D point. Write a function that adds two `struct points` together by adding their individual components. It should return a `struct point` as the result. (Solution.[24])

2. Do the same thing, except pass pointers to the `struct points` into the function. Have the function modify the first point so that it is the sum of the first and second points. (Solution.[25])

71.8 I/O

1. Write a program that reads in up to 50 lines of no more than 80 characters each from `stdin`, then prints them out in reverse order. Hint: 2D array of characters might help. (Solution.[26])

2. Write a program that reads in the 3D coordinates from `input.txt`[27] and computes the average 3D point, printing it out. The average point is the average of the Xs, then the average of the Ys, then Zs from the input. In the case of `input.txt`:

   ```
   Average point <46.296997,48.968998,49.845001>
   ```

[17] https://beej.us/guide/bgc/source/exercises/arrays/sum.c
[18] https://beej.us/guide/bgc/source/exercises/arrays/init312.c
[19] https://en.wikipedia.org/wiki/Tic-tac-toe
[20] https://beej.us/guide/bgc/source/exercises/arrays/tictactoe.c
[21] https://beej.us/guide/bgc/source/exercises/strings/reverse.c
[22] https://beej.us/guide/bgc/source/exercises/strings/str_isupper.c
[23] https://beej.us/guide/bgc/source/exercises/strings/my_strchr.c
[24] https://beej.us/guide/bgc/source/exercises/structs/point.c
[25] https://beej.us/guide/bgc/source/exercises/structs/point_ptr.c
[26] https://beej.us/guide/bgc/source/exercises/io/reverselines.c
[27] https://beej.us/guide/bgc/source/exercises/io/input.txt

www.dbooks.org

(Solution.[28])

71.9 Typedef

1. Write a `typedef` that creates a new type called `vector3d` for this `struct`:

```
struct {
   float x, y, z;
};
```

   Write a function called `dot3d` that computes and returns the vector dot product of two `vector3d`s. This function should accept two pointers to `vector3d`s as arguments, so it could be called with:

```
float result = dot3d(&v1, &v2);
```

   The vector dot product of two vectors is the sum of the products of the `x`, `y`, and `z` components:

   $v_1 = (x_1, y_1, z_1)$
   $v_2 = (x_2, y_2, z_2)$

   $v_1 \cdot v_2 = x_1 \times x_2 + y_1 \times y_2 + z_1 \times z_2$

   (Solution.[29])

71.10 Pointers II

1. Write `my_strchr()`, a function that returns a pointer to the first instance of a character in a string, like `strchr()`. Use pointer arithmetic to implement this. (Solution.[30])
2. Write `my_strrchr()`, a function that returns a pointer to the last instance of a character in a string, like `strrchr()`. Use pointer arithmetic to implement this. (Solution.[31])
3. Write a function `object_sum()` that computes the sum of the bytes of the object pointed to by a `void*` parameter.

```
int object_sum(void *p, size_t count)
```

```
int t;
float x = 3.14159;
int y = 3490;

t = object_sum(&x, sizeof x);
t = object_sum(&y, sizeof y);
```

   (Solution.[32])

4. Write a comparison function that sorts an `int` array with `qsort()` in ascending order. Write another comparison function that sorts it in descending order. (Solution.[33])

[28] https://beej.us/guide/bgc/source/exercises/io/point_avg.c
[29] https://beej.us/guide/bgc/source/exercises/typedef/vector3d.c
[30] https://beej.us/guide/bgc/source/exercises/pointers2/my_strchr.c
[31] https://beej.us/guide/bgc/source/exercises/pointers2/my_strrchr.c
[32] https://beej.us/guide/bgc/source/exercises/pointers2/object_sum.c
[33] https://beej.us/guide/bgc/source/exercises/pointers2/int_qsort.c

71.11 Manual Memory Allocation

1. Write a program that allocates space for an array of 10 `ints`, then fills that array for each index with the index number times 10. (Solution.[34])
2. Write a program that reads a file of arbitrary size into memory and returns a pointer to the data without knowing the file size in advance. The `fread()` function might help here. (Solution.[35])

71.12 Scope

1. What will this code print?

```c
#include <stdio.h>

int total = 0;

void add(int x)
{
    total += x;
    printf("Total is now %d\n", total);
}

int main(void)
{
    for (int i = 0; i < 3; i++)
        add(i);
}
```

Answer[36].

2. What will this code print?

```c
#include <stdio.h>

int main(void)
{
    int i = 0;

    if (1 < 2) {  // always true
        int i = 0;

        i += 5;
    }

    i += 5;

    printf("%d\n", i);
}
```

Answer[37].

3. What will this code print?

[34] https://beej.us/guide/bgc/source/exercises/manmem/tenints.c
[35] https://beej.us/guide/bgc/source/exercises/manmem/readfile.c
[36] `Total is now` with values `1`, `3`, `5`. `total` is at file scope so each call to `add()` affects it.
[37] `5`–the first increase in `i` is on a variable that hides the one at outer scope.

```c
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 3; i++) {
        for (int i = 0; i < 2; i++) {
            printf("Hi there!\n");
        }
    }
}
```

Answer[38].

71.13 Types II

1. Write a program that initializes an `unsigned char` to `0`. Then, in a loop, increment that value a total of 260 times, printing it out each time.

   What happens as you pass 255[39]?

   Answer[40]. (Solution.[41])

71.14 Types III

1. Write a function that prints out the whole number portion of a floating pointer number, followed by a `+` sign, followed by the fractional part of the number. For example, given the `float` value `3.14159`, it should print:

   ```
   3+0.14159
   ```

   Try to do this without any string manipulation, just numerically.

   (Solution.[42])

2. Write your own function that converts a string to an integer without using any of the built-in conversion functions (like `atoi()` or `strtol()`).

   ```c
   int my_atoi(char *s)
   {
       // TODO
   }

   printf("%d\n", my_atoi("3490"));  // prints 3490
   ```

   For bonus points:

   - Make it skip leading whitespace—see the `isspace()` function
   - Make it stop on the first non-numeric character—see the `isdigit()` function
   - Make it handle negative numbers

[38] Prints `Hi there` six times. The inner declaration of `i` hides the outer one.

[39] Assuming your machine has 8-bit bytes.

[40] Wraps around like an odometer from 255 to 0. This happens because 255 is binary `11111111`—all 8 bits are `1`, so adding one more gets us to `100000000`. But that's 9 bits, too big to fit in your byte. The highest bit is thrown out, leaving us with binary `00000000`. All unsigned types will do this when they are incremented from their highest value.

[41] https://beej.us/guide/bgc/source/exercises/types2/overflow.c

[42] https://beej.us/guide/bgc/source/exercises/types3/cutfrac.c

(Solution.[43])

71.15 Types IV

1. This is going to be a two-file project. One file contains functionality to add numbers to a running total. The other calls that functionality.

   The main file `runner.c` will contain the following:

```c
// runner.c

#include <stdio.h>

int main(void)
{
   for (int i = 0; i < 10; i++)
      add(i);

   printf("Total is %d over %d calls\n", total(), count);
}
```

   The file `total.c` will have three things that are callable from other source files:

   1. A function called `void add(int x)`. This will add on to the running total that starts at `0`.
   2. A function called `int total(void)` that returns the total so far.
   3. A shared global variable called `count` that returns the number of times `add()` has been called.

   Implement the functionality in `total.c` and then modify `runner.c` so it can call the functions `add()` and `total()` and can see the variable `count`.

   Make sure that `total()` is the only way `runner.c` can get the total.

   You'll have to make use of `static` and `extern` to solve this one.

   Normally this is a case where we'd use a header file, but we haven't talked about those yet.

   You can compile two files together by specifying them both on the command line:

```
gcc -Wall -Wextra  -o runner runner.c total.c
```

   (Solution.[44]) (Solution.[45])

71.16 Multifile Projects

1. Modify the exercise for Types IV, above, to use a header file `total.h`.

   Be sure to use the `#ifdef` wrapper on the header file.

   Bonus points for compiling the C files to object files before linking them all together.

   (Solution.[46]) (Solution.[47]) (Solution.[48])

[43] https://beej.us/guide/bgc/source/exercises/types3/my_atoi.c
[44] https://beej.us/guide/bgc/source/exercises/types4/runner.c
[45] https://beej.us/guide/bgc/source/exercises/types4/total.c
[46] https://beej.us/guide/bgc/source/exercises/multifile/runner.c
[47] https://beej.us/guide/bgc/source/exercises/multifile/total.c
[48] https://beej.us/guide/bgc/source/exercises/multifile/total.h

www.dbooks.org

71.17 The Outside Environment

1. Write a program that prints out the values of the environment variables that are specified on the command line.

   If the variable isn't set, it should print out the message `no such variable`.

   For example:

   ```
   $ ./printenv HOME LANG FOOBAR PATH
   ```

   ```
   HOME=/Users/beej
   LANG=en_US.UTF-8
   FOOBAR: no such variable
   PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
   ```

   (`HOME`, `LANG`, and `PATH` are variables commonly-set by the shell, so it's likely they're already there for you.)

   You'll want to use `getenv()` for this one.

   (Solution.[49])

71.18 The C Preprocessor

1. Write a macro that takes two arguments, multiplies them together, then adds the current source code line number to the product. The result of the macro should be integer type.

   (Solution.[50])

2. Write code that prints out `C89`, `C99`, or `C11 or higher` corresponding to your current C version.

   (Solution.[51])

3. Have a program `#define` a numeric value for `X`. Use conditional compilation to either print `X is greater than 30` if that is true. Otherwise, print out the values from `0` up to `X-1`.

   (Solution.[52])

[49] https://beej.us/guide/bgc/source/exercises/env/printenv.c
[50] https://beej.us/guide/bgc/source/exercises/cpp/macro2.c
[51] https://beej.us/guide/bgc/source/exercises/cpp/cversion.c
[52] https://beej.us/guide/bgc/source/exercises/cpp/condcomp.c

Index

! boolean NOT, 18
!= inequality operator, 17
' single quote, 91
* for VLA function prototypes, 220
* indirection operator, 33–34
* multiplication operator, 15
*= assignment operator, 15
+ addition operator, 15
++ increment operator, 16–17
+= assignment operator, 15
, comma operator, 17
- subtraction operator, 15
-- decrement operator, 16–17
-= assignment operator, 15
-> arrow operator, 54
... variadic arguments, 183–184
/ division operator, 15
/= assignment operator, 15
< less than operator, 17
<< shift left, 182
<<= assignment, 182
<= less or equal operator, 17
= assignment operator, 14
== equality operator, 17
> greater than operator, 17
>= less or equal operator, 17
>> shift right, 182
>>= assignment, 182
?: ternary operator, 15–16
null directive, 145
stringification, 140
concatenation, 140
#define directive, 7, 131, 137–140
    versus const, 131
#elif directive, 133–134
#else directive, 133
#endif directive, 132–133
#error directive, 143
#if 0 directive, 134
#if defined directive, 134–135
#if directive, 133–135
#ifdef directive, 132–133
#ifndef directive, 132–133
#include directive, 7, 130–131
    local files, 130–131
#line directive, 144–145
#pragma directive, 143–144
    nonstandard pragmas, 143
#undef directive, 135
#warning directive, 143
% modulus operator, 15
%= assignment operator, 15
& address-of operator, 31–32
& bitwise AND, 181
&= assignment, 181
&& boolean AND, 18
^ bitwise XOR, 181
^= assignment, 181
_Complex type, 251
_Complex_I macro, 252
_Exit() function, 209–210
_Generic keyword, 235–238
_Imaginary type, 252
_Imaginary_I macro, 252
_Pragma operator, 144
    in a macro, 144
_Thread_local storage class, 116, 274
__DATE__ macro, 135–136
__FILE__ macro, 135–136
__LINE__ macro, 135–136, 144–145
__STDC_ANALYZABLE__ macro, 137
__STDC_HOSTED__ macro, 135
__STDC_IEC_559_COMPLEX__ macro, 137, 251–252
__STDC_IEC_559__ macro, 137
__STDC_ISO_10646__ macro, 136, 198
__STDC_LIB_EXT1__ macro, 137
__STDC_MB_MIGHT_NEQ_WC__ macro, 136
__STDC_NO_ATOMICS__ macro, 137
__STDC_NO_COMPLEX__ macro, 137, 251
__STDC_NO_THREADS__ macro, 137, 266
__STDC_NO_VLA__ macro, 137, 217
__STDC_UTF_16__ macro, 137, 205
__STDC_UTF_32__ macro, 205
__STDC_UTF_32__ macro, 137
__STDC_VERSION__ macro, 135–136
__STDC__ macro, 135
__TIME__ macro, 135–136
__func__ identifier, 135–136
_mkgmtime() Windows function, 263

www.dbooks.org

`_putenv()` function, 128
`|` bitwise OR, 181
`|=` assignment, 181
`||` boolean OR, 18
`\` backslash escape, 161–164
`\'` single quote, 161–162
`\123` octal value, 164
`\?` question mark, 162–163
`\U` Unicode escape, 164, 195
`\a` alert, 162
`\b` backspace, 162–163
`\f` formfeed, 162
`\n` newline, 8, 162
`\r` carriage return, 162–163
`\t` tab, 162
`\u` Unicode escape, 164, 195
`\v` vertical tab, 162
`\x12` hexadecimal value, 164
`\\` backslash, 162
  bitwise NOT, 181
`0` octal, 97
`0b` binary, 97
`0x` hexadecimal, 97

`abort()` function, 210
Addition, *see* `+` addition operator
`aligned_alloc()` function, 83
`argc` parameter, 122–125
`argv` parameter, 122–125
Arithmetic Operators, 15, 15
Array initializers, 38–40
Arrays, 37–45
    as pointers, 42
    getting the length, 38
    indexing, 37–38
    modifying within functions, 43–44
    multidimensional, 41
    multidimensional initializers, 240–242
    out of bounds, 40
    passing to functions, 42–45
    `static` in parameter lists, 239–240
    type qualifiers in parameter lists, 239
    zero length, 150
`asctime()` function, 262
`atexit()` function, 208
`auto` storage class, 113
Automatic variables, 76

Bitwise operations, 181–182
`bool` type, 15
Boolean Operators, 18
Boolean types, 14–15
`break` statement, 23–25

C Preprocessor, 7
`c16rtomb()` function, 206
`c32rtomb()` function, 206
C3PO, 26
`cabs()` function, 255
`cacos()` function, 255
`cacosh()` function, 255
`call_once()` function, 284
`calloc()` function, 78–79
`carg()` function, 255
`case` statement, 23–24
`casin()` function, 255
`casinh()` function, 255
`catan()` function, 255
`catanh()` function, 255
`ccos()` function, 255
`ccosh()` function, 255
`cexp()` function, 255
`char *` type, 13
`char` type, 13, 18, 90–91
`char16_t` type, 205
`char32_t` type, 205
Character sets, 194–195
    basic, 194–195
    execution, 194, 195
    source, 194, 195
`cimag()` function, 253, 255
`cimagf()` function, 253
`cimagl()` function, 253
`clog()` function, 255
`CMPLX()` macro, 252–253, 255
`CMPLXF()` macro, 253
`CMPLXL()` macro, 253
`cnd_broadcast()` function, 284
`cnd_destroy()` function, 280–283
`cnd_init()` function, 280–283
`cnd_signal()` function, 280–283
`cnd_t` type, 280–283
`cnd_timedwait()` function, 283–284
`cnd_wait()` function, 280–283
Command line arguments, 121–125
Comments, 7
Comparison operators, 17
Compilation, 9
`complex double` type, 252
`complex float` type, 252
`complex long double` type, 252
Complex numbers, 251–255
    arithmetic, 253–254
    declaring, 252
`complex` type, 251
`complex.h` header file, 251
Compound literals, 232–235
    passing to functions, 233
    pointers to, 234

scope, 234–235
with `struct`, 233–234
Condition variables, 280–284
broadcasting, 284
spurious wakeup, 281
timeouts, 283–284
Conditional compilation, 132–135
Conditional Operators, 17–18
`conj()` function, 255
`const` type qualifier, 109–111
and pointers, 109–110
correctness, 110–111
`cpow()` function, 255
`cproj()` function, 255
`creal()` function, 253, 255
`crealf()` function, 253
`creall()` function, 253
`csin()` function, 255
`csinh()` function, 255
`csqrt()` function, 255
`ctan()` function, 255
`ctanh()` function, 255
`ctime()` function, 261–262
`CX_LIMITED_RANGE` pragma, 144

Data serialization, 62
Date and time, 260
differences, *265*, 264–265
`DBL_DECMIAL_DIG` macro, 96
`DBL_DIG` macro, 94–96
`DECMIAL_DIG` macro, 96
`default` label, 24
Dereferencing, 33–34
`difftime()` function, 264–265
Division, *see* `/` division operator
`do-while` statement, 21–22
in multiline macros, 140–141
`double` type, 94

Empty parameter lists, 29
Endianess, 61
`enum` enumerated types, 165–168
numbering order, 165–166
scope, 166
`enum` keyword, 25
`env` parameter, 129
`environ` variable, 128–129
Environment variables, 127–129
`EOF` end of file, 56–57
Escape sequences, 161–164
Exit status, 125–127
obtaining from shell, 126
`EXIT_FAILURE` macro, 125–126
`EXIT_SUCCESS` macro, 125–126
Exiting, 207–210
return from `main()`, 207
`extern` storage class, 114–115, 120

`F` float constant, 99
Fall through, 24–25
`false` value, 15
`fclose()` function, 56–57
`FENV_ACCESS` pragma, 144
`fgetc()` function, 56–57
`fgets()` function, 57–58
`fgetwc()` function, 200
`fgetws()` function, 200
File I/O, 55–62
binary files, 59–61
formatted input, 58–59
line by line, 57–58
text files, reading, 56
text files, writing, 59
with numeric values, 61–62
with `structs`, 61–62
`FILE*` type, 55–56
Fixed width integers, 256–259
`float` type, 13
Floating point constants, 99
Flow Control, 19–25
`FLT_DECMIAL_DIG` macro, 96
`FLT_DIG` macro, 94–96
`fopen()` function, 56–57
`for` statement, 22–23
`FP_CONTRACT` pragma, 144
`fprintf()` function, 59
`fputc()` function, 59
`fputs()` function, 59
`fputwc()` function, 200
`fputws()` function, 200
`fread()` function, 60–61
`free()` function, 77
`fscanf()` function, 58
Function arguments, 26
Function parameters, 26
Function prototypes, 28–29
Functions, 26–29
`fwide()` function, 200
`fwprintf()` function, 200
`fwrite()` function, 59–60
`fwscanf()` function, 200

`gcc` compiler, 8–11, 120
with threads, 266
Generic selections, 235–238
`getenv()` function, 127
`getwchar()` function, 200
`gmtime()` function, 262

www.dbooks.org

`goto` statement, 224–231
  as labeled `break`, 227
  as labeled `continue`, 225–226
  for bailing out, 226
  multilevel cleanup, 227–228
  restarting system calls, 229–230
  tail call optimzation, 228–229
  variable scope, 230–231
  with variable-length arrays, 231
Greenwich Mean time, 260

Hello, world, 7, 8
Hex floating point constants, 100

`I` macro, 252
I/O stream orientation, 200
`if` statement, 19–20
`if-else` statement, 20–21
`imaginary` type, 252
Implicit declaration, 29
Incomplete types, 247
  self-referential `structs`, 247–248
`int` type, 13
`INT_FASTn_MAX` macros, 258
`INT_FASTn_MIN` macros, 258
`int_fastN_t` types, 256–257
`INT_LEASTn_MAX` macros, 258
`INT_LEASTn_MIN` macros, 258
`int_leastN_t` types, 256–257
Integer constants, 97–99
Integer promotions, 181
Integrated Development Environment, 10
International Obfuscated C Code Contest, 2
`INTMAX_C()` macro, 257
`INTMAX_MAX` macro, 258
`INTMAX_MIN` macro, 258
`intmax_t` type, 257
`INTn_C()` macros, 257
`INTn_MAX` macros, 258
`INTn_MIN` macros, 258
`intN_t` types, 256–257
`isalpha()` function
  with UTF-8, 196
`iswalnum()` function, 202
`iswalpha()` function, 202
`iswblank()` function, 202
`iswcntrl()` function, 202
`iswdigit()` function, 202
`iswgraph()` function, 202
`iswlower()` function, 202
`iswprint()` function, 202
`iswpunct()` function, 202
`iswspace()` function, 202
`iswupper()` function, 202
`iswxdigit()` function, 202

`jmp_buf` type, 244

`L` long constant, 97–99
`L` long double constant, 99
`L` wide character prefix, 198
Labels, 224–225
Language versions, 10–136
`LDBL_DECMIAL_DIG` macro, 96
`LDBL_DIG` macro, 94–96
`LL` long long constant, 97–99
Local time, 260
Locale, 187–191
  money, 188–190
`locale.h` header file, 187
`localeconv()` function, 188–189
  `mon_grouping`, 189
  `sep_by_space`, 190
`localtime()` function, 262
`long double` type, 94
Long jumps, 243–246
`long long` type, 92–93
`long` type, 92–93
`longjmp()`, 244, 245
`longjmp()` function, 243–245
`lonjmp()`, 246

`main()` function, 8, 27
  command line options, 122–123
  returning from, 125
`malloc()` function, 76–77
  and arrays, 78
  error checking, 77–78
  with UTF-8, 196
Manual memory management, 76–84
`MB_LEN_MAX` macro, 196, 197
`mbrtoc16()` function, 206
`mbrtoc32()` function, 206
`mbstowcs()` function, 198–199
  with UTF-8, 196
`mbtowc()` function, 198
`memcpy()` function, 71–74
Memory address, 30
Memory alignment, 82–84
`mktime()` function, 262–263
`mtx_destroy()` function, 276–278, 281–283
`mtx_init()` function, 276–278, 281–283
`mtx_lock()` function, 276–278, 280–283
`mtx_plain` macro, 279
`mtx_recursive` macro, 279
`mtx_t` type, 277
`mtx_timed` macro, 279
`mtx_timedlock()` function, 279–280
`mtx_unlock` function, 276

mtx_unlock() function, 277–278, 280–283
Multibyte characters, 196–197
    parse state, 202–204
Multifile projects, 117–120
    extern storage class, 120
    function prototypes, 117–119
    includes, 117–120
    static storage class, 120
Multiplication, *see* * multiplication operator
Multithreading, 266–284
    and the standard library, 267
    one-time functions, 284
    race conditions, 270, 276–277
Mutexes, 276–280
    timeouts, 279–280
    types, 279–280

Newline, *see* \n newline
NULL pointer, **35**
    zero equivalence, 174

Object files, 120
Octothorpe, 7
offsetof() macro, 151–152
once_flag type, 284
ONCE_FLAG_INIT macro, 284

Pass by value, 27, 28
Pointer types, 32–33
Pointers, 30–36
    arithmetic, 67–75
    array equivalence, 70–71
    as arguments, 34–35
    as integers, 174–175
    casting, 175–177
    declarations, 36
    subtracting, 69, 177–178
    to functions, 178–180
    to multibyte values, 173–174
    to pointers, 169–171
    to pointers, const, 172
    with sizeof, 36
pow(), 15
Preprocessor, 7, 130–145
    macros, 131
    macros with arguments, 137–140
    macros with variable arguments, 139–140
    multiline macros, 140–141
    predefined macros, 135–137
PRIdFASTn macros, 258–259
PRIdLEASTn macros, 258–259
PRIdMAX macro, 258–259
PRIdn macros, 258–259
PRIiFASTn macros, 258–259
PRIiLEASTn macros, 258–259
PRIiMAX macro, 258–259
PRIin macros, 258–259
printf(), 8, 16
printf() function, 8, 14, 258
    with pointers, 31
    with UTF-8, 196
PRIoFASTn macros, 259
PRIoLEASTn macros, 259
PRIoMAX macros, 259
PRIon macros, 259
PRIuFASTn macros, 259
PRIuLEASTn macros, 259
PRIuMAX macros, 259
PRIun macros, 259
PRIXFASTn macros, 259
PRIxFASTn macros, 259
PRIXLEASTn macros, 259
PRIxLEASTn macros, 259
PRIXMAX macros, 259
PRIxMAX macros, 259
PRIXn macros, 259
PRIxn macros, 259
ptrdiff_t type, 177–178
    printing, 178
putenv() function, 128
putwchar() function, 200

qsort() function, 74–75
quick_exit() function, 208–209

realloc() function, 79–80
    with NULL argument, 82
register storage class, 115–116
restrict type qualifier, 111–112
return statement, 26

scanf() function, 258, 259
Scientific notation, 99–100
SCNdFASTn macros, 259
SCNdLEASTn macros, 259
SCNdMAX macros, 259
SCNdn macros, 259
SCNiFASTn macros, 259
SCNiLEASTn macros, 259
SCNiMAX macros, 259
SCNin macros, 259
SCNoFASTn macros, 259
SCNoLEASTn macros, 259
SCNoMAX macros, 259
SCNon macros, 259
SCNuFASTn macros, 259
SCNuLEASTn macros, 259
SCNuMAX macros, 259
SCNun macros, 259
SCNxFASTn macros, 259

www.dbooks.org

SCNxLEASTn macros, 259
SCNxMAX macros, 259
SCNxn macros, 259
Scope, 85–88
    block, 85–86
    file, 86–87
    for loop, 87
    function, 88
setenv(), 128
setjmp()
    in an expression, 245–246
setjmp() function, 243–245
setlocale() function, 187–188, 204–205
    LC_ALL macro, 191
    LC_COLLATE macro, 191
    LC_CTYPE macro, 191
    LC_MONETARY macro, 191
    LC_NUMERIC macro, 191
    LC_TIME macro, 191
short type, 92–93
sig_atomic_t type, 214–215
SIG_DFL macro, 212, 213, 215
SIG_ERR macro, 213
SIG_IGN macro, 212
SIG_INT signal, 213
SIGABRT signal, 210, 211
sigaction() function, 211, 214
SIGFPE signal, 211
SIGILL signal, 211
SIGINT signal, 211–212
Signal handling, 211–216
Signal handling-
    limitations, 214
signal() function, 211–214, 216
signed char type, 90–91
Significant digits, 94–96
SIGSEGV signal, 211
SIGTERM signal, 211
size_t type, 18
sizeof operator, 18–19
    with arrays, 38
    with malloc(), 77–78
static storage class, 113–114, 120
    in block scope, 113–114
    in file scope, 114
stdarg.h header file, 184
stdbool.h header file, 15
stderr standard error, 55
stdin standard input, 55
stdint.h header file, 256
stdio.h, 8
stdio.h header file, 7–8
stdout standard output, 55
Storage-Class Specifiers, 113–116
strchr() function
    with UTF-8, 196
strftime() function, 263–264
String, *see* char *
String literals, 46
String variables, 46
    as arrays, 47
Strings, 46–50
    copying, 49–50
    getting the length, 48
    initializers, 47–48
    termination, 48–49
strlen() function
    with UTF-8, 197
strstr() function
    with UTF-8, 196
strtok() function
    with UTF-8, 196
struct keyword, 51–54, 146–160
    anonymous, 148
    bit fields, 152–155
    comparing, 54
    compound literals, 233–234
    copying, 54
    declaring, 51–52
    flexible array members, 149–151
    initializers, 52, 146–148
    padding bytes, 151
    passing and returning, 52–53, 160
    self-referential, 149
struct timespec type, 264
struct tm type, 261
    conversion to time_t, 262–263
Subtraction, *see* - subtraction operator
switch statement, 23–25
swprintf() function, 200
swscanf() function, 200

Tail call optimzation
    with goto, 228–229
Ternary Operator, *see* ?: ternary operator
The heap, 76
The stack, 76
thrd_create() function, 267
thrd_create() function, 268–270
thrd_detach() function, 271–272
thrd_join() function, 267–270
thrd_start_t type, 268
thrd_t type, 267
thrd_timedout macro, 283
thrd_timedout() macro, 283–284
Thread local data, 272–274
Thread-specific storage, 274–276
thread_local storage class, 274

threads.h header file, 274
time() function, 261
time_t type, 261
    conversion to struct tm, 262
timegm() Unix function, 263
timespec_get() function, 264, 279–280, 283–284
tolower() function
    with UTF-8, 196
toupper() function
    with UTF-8, 196
towlower() function, 202
towupper() function, 202
Trigraphs, 163
true value, 15
tss_create() function, 275–276
tss_delete() function, 275–276
tss_dtor_t type, 275
tss_get() function, 275–276
tss_set() function, 275–276
tss_t type, 275–276
Type conversions, 101–108
    Boolean, 105
    casting, 107–108
    char, 104–105
    explicit, 107–108
    floating point, 106
    implicit, 106–107
    integer, 105–106
    numeric, 105–107
    strings, 101–104
Type qualifiers, 109–112
    arrays in parameter lists, 239
typedef keyword, 63–66
    scoping rules, 63–65
    with anonymous structs, 64–65
    with arrays, 66
    with pointers, 65
    with structs, 63–65
Types, 13
    character, 90–91
    signed and unsigned, 89–90

U Unicode prefix, 205
u Unicode prefix, 205
U unsigned constant, 97–99
u8 UTF-8 prefix, 204
UINT_FASTn_MAX macros, 258
uint_fastN_t types, 256–257
UINT_LEASTn_MAX macros, 258
uint_leastN_t types, 256–257
UINTMAX_C() macro, 257
UINTMAX_MAX macro, 258
uintmax_t type, 257
UINTn_C() macros, 257
UINTn_MAX macros, 258
uintN_t types, 256–257
UL unsigned long constant, 97–99
ULL unsigned long long constant, 97–99
ungetwc() function, 200
Unicode, 192–206
    code points, 192–193
    encoding, 193–194
    endianess, 194
    UTF-16, 193–194, 205
    UTF-32, 193–194, 205
    UTF-8, 193–194, 196, 204–205
union keyword, 155–160
    and unnamed structs, 159–160
    common initial sequences, 156–159
    passing and returning, 160
    pointers to, 156
    type punning, 155–156
Universal Coordinated Time, 260
unsetenv() function, 128
unsigned char type, 90–91
unsigned type, 89–90

va_arg() macro, 184–185
va_copy() macro, 185–186
va_end() macro, 184–185
va_list type, 184–186
    passing to functions, 186
va_start() macro, 184–185
Variable hiding, 86
Variable-length array, 217–223
    and sizeof(), 218–219
    controversy, 222–223
    defining, 217–218
    in function prototypes, 220
    multidimensional, 219
    passing to functions, 219–221
    with goto, 222
    with longjmp(), 222
    with regular arrays, 221–222
    with typedef, 222
Variables, 12–13
    uninitialized, 13
Variadic functions, 183–186
vfwprintf() function, 200
vfwscanf() function, 200
void type, 27, 29
    in function prototypes, 29
void* void pointer, 71–75
    caveats, 73
volatile type qualifier, 112
    with setjmp(), 244–245
vprintf() function, 186
vswprintf() function, 200

www.dbooks.org

`vswscanf()` function, 200
`vwprintf()` function, 200
`vwscanf()` function, 200

`wchar_t` type, 197–199
`wcscat()` function, 201
`wcschr()` function, 201
`wcscmp()` function, 201
`wcscoll()` function, 201
`wcscpy()` function, 201
`wcscspn()` function, 201
`wcsftime()` function, 202
`wcslen()` function, 199, 202
`wcsncat()` function, 201
`wcsncmp()` function, 201
`wcsncpy()` function, 201
`wcspbrk()` function, 201
`wcsrchr()` function, 201
`wcsspn()` function, 201
`wcsstr()` function, 201
`wcstod()` function, 200
`wcstof()` function, 200
`wcstok()` function, 201
`wcstol()` function, 200
`wcstold()` function, 200
`wcstoll()` function, 200
`wcstombs()` function, 198, 199
`wcstoul()` function, 200
`wcstoull()` function, 201
`wcsxfrm()` function, 201
`wctomb()` function, 198
`while` statement, 21
Wide characters, 197–204
`wint_t` type, 199
`wmemchr()` function, 201
`wmemcmp()` function, 201
`wmemcpy()` function, 201
`wmemmove()` function, 201
`wmemset()` function, 202
`wprintf()` function, 200
`wscanf()` function, 200