Beej’s Guide to C Programming

Library Reference

Brian “Beej Jorgensen” Hall

v0.9.9, Copyright © December 30, 2022

Contents

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

2 The C Language **4**
- 2.1 Background . . . 4
  - 2.1.1 Comments . . . 4
  - 2.1.2 Separators . . . 4
  - 2.1.3 Expressions . . . 4
  - 2.1.4 Statements . . . 4
  - 2.1.5 Booleans . . . 4
  - 2.1.6 Blocks . . . 4
  - 2.1.7 Code Examples . . . 4
- 2.2 Operators . . . 5
  - 2.2.1 Arithmetic Operators . . . 5
  - 2.2.2 Pre- and Post-Increment and -Decrement . . . 5
  - 2.2.3 Comparison Operators . . . 5
  - 2.2.4 Pointer Operators . . . 5
  - 2.2.5 Structure and Union Operators . . . 5
  - 2.2.6 Array Operators . . . 6
  - 2.2.7 Bitwise Operators . . . 6
  - 2.2.8 Assignment Operators . . . 6
  - 2.2.9 The `sizeof` Operator . . . 6
  - 2.2.10 Type Casts . . . 7
  - 2.2.11 `_Alignof` Operator . . . 7
  - 2.2.12 Comma Operator . . . 7
- 2.3 Type Specifiers . . . 7
- 2.4 Constant Types . . . 8
- 2.5 Composite Types . . . 8
  - 2.5.1 `struct` Types . . . 8
  - 2.5.2 `union` Types . . . 9
  - 2.5.3 `enum` Types . . . 9
- 2.6 Initializers . . . 10
- 2.7 Compound Literals . . . 11
- 2.8 Type Aliases . . . 11
- 2.9 Additional Type-Related Specifiers . . . 12
  - 2.9.1 Storage Class Specifiers . . . 12
  - 2.9.2 Type Qualifiers . . . 12
  - 2.9.3 Function Specifiers . . . 14
  - 2.9.4 Alignment Specifier . . . 14

2.10 `if` Statement . . . 14
2.11 `for` Statement . . . 14
2.12 `while` Statement . . . 15
2.13 `do-while` Statement . . . 15
2.14 `switch` Statement . . . 15
2.15 `break` Statement . . . 16
2.16 `continue` Statement . . . 16
2.17 `goto` Statement . . . 17
2.18 `return` Statement . . . 17
2.19 `_Static_assert` Statement . . . 17
2.20 Functions . . . 18
2.20.1 `main()` Function . . . 18
2.20.2 Variadic Functions . . . 18

3 `<assert.h>` Runtime and Compile-time Diagnostics 20
3.1 Macros . . . 20
3.2 `assert()` . . . 20
3.3 `static_assert()` . . . 22

4 `<complex.h>` Complex Number Functionality 24
4.1 `cacos()`, `cacosf()`, `cacosl()` . . . 26
4.2 `casin()`, `casinf()`, `casinl()` . . . 27
4.3 `catan()`, `catanf()`, `catanl()` . . . 28
4.4 `ccos()`, `ccosf()`, `ccosl()` . . . 28
4.5 `csin()`, `csinf()`, `csinl()` . . . 29
4.6 `ctan()`, `ctanf()`, `ctanl()` . . . 30
4.7 `cacosh()`, `cacoshf()`, `cacoshl()` . . . 31
4.8 `casinh()`, `casinhf()`, `casinhl()` . . . 32
4.9 `catanh()`, `catanhf()`, `catanhl()` . . . 33
4.10 `ccosh()`, `ccoshf()`, `ccoshl()` . . . 34
4.11 `csinh()`, `csinhf()`, `csinhl()` . . . 35
4.12 `ctanh()`, `ctanhf()`, `ctanhl()` . . . 36
4.13 `cexp()`, `cexpf()`, `cexpl()` . . . 36
4.14 `clog()`, `clogf()`, `clogl()` . . . 37
4.15 `cabs()`, `cabsf()`, `cabsl()` . . . 38
4.16 `cpow()`, `cpowf()`, `cpowl()` . . . 39
4.17 `csqrt()`, `csqrtf()`, `csqrtl()` . . . 40
4.18 `carg()`, `cargf()`, `cargl()` . . . 41
4.19 `cimag()`, `cimagf()`, `cimagl()` . . . 42
4.20 `CMPLX()`, `CMPLXF()`, `CMPLXL()` . . . 43
4.21 `conj()`, `conjf()`, `conjl()` . . . 44
4.22 `cproj()`, `cproj()`, `cproj()` . . . 45
4.23 `creal()`, `crealf()`, `creall()` . . . 46

5 `<ctype.h>` Character Classification and Conversion 47
5.1 `isalnum()` . . . 48
5.2 `isalpha()` . . . 48
5.3 `isblank()` . . . 49
5.4 `iscntrl()` . . . 50
5.5 `isdigit()` . . . 51
5.6 `isgraph()` . . . 52
5.7 `islower()` . . . 52
5.8 `isprint()` . . . 53
5.9 `ispunct()` . . . 54
5.10 `isspace()` . . . 55
5.11 `isupper()` . . . 56
5.12 `isxdigit()` . . . 57
5.13 `tolower()` . . . 57
5.14 `toupper()` . . . 58

6 `<errno.h>` Error Information 60
6.1 `errno` . . . 60

7 `<fenv.h>` Floating Point Exceptions and Environment 63
7.1 Types and Macros . . . 63
7.2 Pragmas . . . 64
7.3 `feclearexcept()` . . . 64
7.4 `fegetexceptflag()` `fesetexceptflag()` . . . 65
7.5 `feraiseexcept()` . . . 66
7.6 `fetestexcept()` . . . 67
7.7 `fegetround()` `fesetround()` . . . 68
7.8 `fegetenv()` `fesetenv()` . . . 69
7.9 `feholdexcept()` . . . 71
7.10 `feupdateenv()` . . . 72

8 `<float.h>` Floating Point Limits 74
8.1 Background . . . 75
8.2 `FLT_ROUNDS` Details . . . 76
8.3 `FLT_EVAL_METHOD` Details . . . 76
8.4 Subnormal Numbers . . . 76
8.5 How Many Decimal Places Can I Use? . . . 76
8.6 Comprehensive Example . . . 78

9 `<inttypes.h>` More Integer Conversions 81
9.1 Macros . . . 81
9.2 `imaxabs()` . . . 82
9.3 `imaxdiv()` . . . 83
9.4 `strtoimax()` `strtoumax()` . . . 84
9.5 `wcstoimax()` `wcstoumax()` . . . 85

10 `<iso646.h>` Alternative Operator Spellings 87

11 `<limits.h>` Numeric Limits 88
11.1 `CHAR_MIN` and `CHAR_MAX` . . . 88
11.2 Choosing the Correct Type . . . 89
11.3 Whither Two's Complement? . . . 89
11.4 Demo Program . . . 89

12 `<locale.h>` locale handling 91
12.1 `setlocale()` . . . 91
12.2 `localeconv()` . . . 93

13 `<math.h>` Mathematics 97
13.1 Math Function Idioms . . . 98
13.2 Math Types . . . 99
13.3 Math Macros . . . 99
13.4 Math Errors . . . 99
13.5 Math Pragmas . . . 100
13.6 `fpclassify()` . . . 100
13.7 `isfinite()`, `isinf()`, `isnan()`, `isnormal()` . . . 102
13.8 `signbit()` . . . 103
13.9 `acos()`, `acosf()`, `acosl()` . . . 103
13.10 `asin()`, `asinf()`, `asinl()` . . . 104
13.11 `atan()`, `atanf()`, `atanl()`, `atan2()`, `atan2f()`, `atan2l()` . . . 105
13.12 `cos()`, `cosf()`, `cosl()` . . . 107
13.13 `sin()`, `sinf()`, `sinl()` . . . 107
13.14 `tan()`, `tanf()`, `tanl()` . . . 108
13.15 `acosh()`, `acoshf()`, `acoshl()` . . . 109
13.16 `asinh()`, `asinhf()`, `asinhl()` . . . 110

13.17 atanh(), atanhf(), atanhl() . . . 111
13.18 cosh(), coshf(), coshl() . . . 112
13.19 sinh(), sinhf(), sinhl() . . . 112
13.20 tanh(), tanhf(), tanhl() . . . 113
13.21 exp(), expf(), expl() . . . 114
13.22 exp2(), exp2f(), exp2l() . . . 115
13.23 expm1(), expm1f(), expm1l() . . . 115
13.24 frexp(), frexpf(), frexpl() . . . 116
13.25 ilogb(), ilogbf(), ilogbl() . . . 117
13.26 ldexp(), ldexpf(), ldexpl() . . . 118
13.27 log(), logf(), logl() . . . 119
13.28 log10(), log10f(), log10l() . . . 120
13.29 log1p(), log1pf(), log1pl() . . . 121
13.30 log2(), log2f(), log2l() . . . 122
13.31 logb(), logbf(), logbl() . . . 123
13.32 modf(), modff(), modfl() . . . 124
13.33 scalbn(), scalbnf(), scalbnl() scalbln(), scalblnf(), scalblnl() . . . 125
13.34 cbrt(), cbrtf(), cbrtl() . . . 126
13.35 fabs(), fabsf(), fabsl() . . . 127
13.36 hypot(), hypotf(), hypotl() . . . 128
13.37 pow(), powf(), powl() . . . 129
13.38 sqrt() . . . 129
13.39 erf(), erff(), erfl() . . . 130
13.40 erfc(), erfcf(), erfcl() . . . 131
13.41 lgamma(), lgammaf(), lgammal() . . . 132
13.42 tgamma(), tgammaf(), tgammal() . . . 133
13.43 ceil(), ceilf(), ceill() . . . 134
13.44 floor(), floorf(), floorl() . . . 135
13.45 nearbyint(), nearbyintf(), nearbyintl() . . . 136
13.46 rint(), rintf(), rintl() . . . 137
13.47 lrint(), lrintf(), lrintl(), llrint(), llrintf(), llrintl() . . . 137
13.48 round(), roundf(), roundl() . . . 138
13.49 lround(), lroundf(), lroundl() llround(), llroundf(), llroundl() . . . 139
13.50 trunc(), truncf(), truncl() . . . 140
13.51 fmod(), fmodf(), fmodl() . . . 141
13.52 remainder(), remainderf(), remainderl() . . . 142
13.53 remquo(), remquof(), remquol() . . . 143
13.54 copysign(), copysignf(), copysignl() . . . 144
13.55 nan(), nanf(), nanl() . . . 145
13.56 nextafter(), nextafterf(), nextafterl() . . . 146
13.57 nexttoward(), nexttowardf(), nexttowardl() . . . 147
13.58 fdim(), fdimf(), fdiml() . . . 148
13.59 fmax(), fmaxf(), fmaxl(), fmin(), fminf(), fminl() . . . 149
13.60 fma(), fmaf(), fmal() . . . 150
13.61 isgreater(), isgreaterequal(), isless(), islessequal() . . . 150
13.62 islessgreater() . . . 151
13.63 isunordered() . . . 152

14 <setjmp.h> Non-local Goto 154
14.1 setjmp() . . . 154
14.2 longjmp() . . . 156

15 <signal.h> signal handling 159
15.1 signal() . . . 159
15.2 raise() . . . 162

16 <stdalign.h> Macros for Alignment 165
16.1 alignas() _Alignas() . . . 165
16.2 alignof() _Alignof() . . . 167

17 `<stdarg.h>` Variable Arguments **169**
17.1 `va_arg()` . . . 169
17.2 `va_copy()` . . . 170
17.3 `va_end()` . . . 172
17.4 `va_start()` . . . 173

18 `<stdatomic.h>` Atomic-Related Functions **175**
18.1 Atomic Types . . . 176
18.2 Lock-free Macros . . . 176
18.3 Atomic Flag . . . 177
18.4 Memory Order . . . 177
18.5 `ATOMIC_VAR_INIT()` . . . 178
18.6 `atomic_init()` . . . 178
18.7 `kill_dependency()` . . . 179
18.8 `atomic_thread_fence()` . . . 180
18.9 `atomic_signal_fence()` . . . 182
18.10 `atomic_is_lock_free()` . . . 183
18.11 `atomic_store()` . . . 184
18.12 `atomic_load()` . . . 185
18.13 `atomic_exchange()` . . . 186
18.14 `atomic_compare_exchange_*()` . . . 187
18.15 `atomic_fetch_*()` . . . 189
18.16 `atomic_flag_test_and_set()` . . . 190
18.17 `atomic_flag_clear()` . . . 192

19 `<stdbit.h>` Bit-Related Functions **195**
19.1 Existence Macro . . . 195
19.2 Endianess Macros . . . 195
19.3 General Structure of These Functions . . . 196
19.4 `stdc_leading_zeros()` . . . 197
19.5 `stdc_leading_ones()` . . . 198
19.6 `stdc_trailing_zeros()` . . . 199
19.7 `stdc_trailing_ones()` . . . 200
19.8 `stdc_first_leading_zero()` . . . 201
19.9 `stdc_first_leading_one()` . . . 202
19.10 `stdc_first_trailing_zero()` . . . 203
19.11 `stdc_first_trailing_one()` . . . 204
19.12 `stdc_count_zeros()` . . . 205
19.13 `stdc_count_ones()` . . . 205
19.14 `stdc_has_single_bit()` . . . 206
19.15 `stdc_bit_width()` . . . 207
19.16 `stdc_bit_floor()` . . . 208
19.17 `stdc_bit_ceil()` . . . 209

20 `<stdbool.h>` Boolean Types **211**
20.1 Example . . . 211
20.2 `_Bool`? . . . 212

21 `<stddef.h>` A Few Standard Definitions **213**
21.1 `ptrdiff_t` . . . 213
21.2 `size_t` . . . 214
21.3 `max_align_t` . . . 214
21.4 `wchar_t` . . . 215
21.5 `offsetof` . . . 215

22 `<stdint.h>` More Integer Types **216**
22.1 Specific-Width Integers . . . 216
22.2 Other Integer Types . . . 217
22.3 Macros . . . 217

22.4 Other Limits . . . 218
22.5 Macros for Declaring Constants . . . 218

23 `<stdio.h>` Standard I/O Library 219
23.1 `remove()` . . . 221
23.2 `rename()` . . . 222
23.3 `tmpfile()` . . . 222
23.4 `tmpnam()` . . . 223
23.5 `fclose()` . . . 225
23.6 `fflush()` . . . 226
23.7 `fopen()` . . . 227
23.8 `freopen()` . . . 229
23.9 `setbuf()`, `setvbuf()` . . . 230
23.10 `printf()`, `fprintf()`, `sprintf()`, `snprintf()` . . . 231
23.11 `scanf()`, `fscanf()`, `sscanf()` . . . 238
23.12 `vprintf()`, `vfprintf()`, `vsprintf()`, `vsnprintf()` . . . 243
23.13 `vscanf()`, `vfscanf()`, `vsscanf()` . . . 245
23.14 `getc()`, `fgetc()`, `getchar()` . . . 246
23.15 `gets()`, `fgets()` . . . 247
23.16 `putc()`, `fputc()`, `putchar()` . . . 249
23.17 `puts()`, `fputs()` . . . 250
23.18 `ungetc()` . . . 251
23.19 `fread()` . . . 252
23.20 `fwrite()` . . . 253
23.21 `fgetpos()`, `fsetpos()` . . . 254
23.22 `fseek()`, `rewind()` . . . 256
23.23 `ftell()` . . . 257
23.24 `feof()`, `ferror()`, `clearerr()` . . . 258
23.25 `perror()` . . . 259

24 `<stdlib.h>` Standard Library Functions 262
24.1 `<stdlib.h>` Types and Macros . . . 263
24.2 `atof()` . . . 263
24.3 `atoi()`, `atol()`, `atoll()` . . . 264
24.4 `strtod()`, `strtof()`, `strtold()` . . . 265
24.5 `strtol()`, `strtoll()`, `strtoul()`, `strtoull()` . . . 267
24.6 `rand()` . . . 269
24.7 `srand()` . . . 271
24.8 `aligned_alloc()` . . . 272
24.9 `calloc()`, `malloc()` . . . 273
24.10 `free()` . . . 274
24.11 `realloc()` . . . 275
24.12 `abort()` . . . 277
24.13 `atexit()`, `at_quick_exit()` . . . 278
24.14 `exit()`, `quick_exit()`, `_Exit()` . . . 279
24.15 `getenv()` . . . 281
24.16 `system()` . . . 281
24.17 `bsearch()` . . . 283
24.18 `qsort()` . . . 284
24.19 `abs()`, `labs()`, `llabs()` . . . 286
24.20 `div()`, `ldiv()`, `lldiv()` . . . 287
24.21 `mblen()` . . . 288
24.22 `mbtowc()` . . . 289
24.23 `wctomb()` . . . 291
24.24 `mbstowcs()` . . . 292
24.25 `wcstombs()` . . . 293
24.26 `memalignment()` . . . 295

25 `<stdnoreturn.h>` Macros for Non-Returning Functions 297

26 `<string.h>` String Manipulation 298
26.1 `memcpy()`, `memccpy()`, `memmove()` . . . 299
26.2 `strcpy()`, `strncpy()` . . . 300
26.3 `strdup()`, `strndup()` . . . 301
26.4 `strcat()`, `strncat()` . . . 302
26.5 `strcmp()`, `strncmp()`, `memcmp()` . . . 303
26.6 `strcoll()` . . . 305
26.7 `strxfrm()` . . . 306
26.8 `strchr()`, `strrchr()`, `memchr()` . . . 308
26.9 `strspn()`, `strcspn()` . . . 309
26.10 `strpbrk()` . . . 310
26.11 `strstr()` . . . 311
26.12 `strtok()` . . . 312
26.13 `memset()`, ‘memset_explicit . . . 313
26.14 `strerror()` . . . 314
26.15 `strlen()` . . . 315

27 `<tgmath.h>` Type-Generic Math Functions 317
27.1 Example . . . 318

28 `<threads.h>` Multithreading Functions 320
28.1 `call_once()` . . . 321
28.2 `cnd_broadcast()` . . . 322
28.3 `cnd_destroy()` . . . 325
28.4 `cnd_init()` . . . 326
28.5 `cnd_signal()` . . . 328
28.6 `cnd_timedwait()` . . . 329
28.7 `cnd_wait()` . . . 331
28.8 `mtx_destroy()` . . . 333
28.9 `mtx_init()` . . . 334
28.10 `mtx_lock()` . . . 336
28.11 `mtx_timedlock()` . . . 337
28.12 `mtx_trylock()` . . . 339
28.13 `mtx_unlock()` . . . 341
28.14 `thrd_create()` . . . 342
28.15 `thrd_current()` . . . 344
28.16 `thrd_detach()` . . . 345
28.17 `thrd_equal()` . . . 346
28.18 `thrd_exit()` . . . 347
28.19 `thrd_join()` . . . 349
28.20 `thrd_sleep()` . . . 350
28.21 `thrd_yield()` . . . 351
28.22 `tss_create()` . . . 353
28.23 `tss_delete()` . . . 355
28.24 `tss_get()` . . . 357
28.25 `tss_set()` . . . 359

29 `<time.h>` Date and Time Functions 362
29.1 Thread Safety Warning . . . 363
29.2 `clock()` . . . 363
29.3 `difftime()` . . . 364
29.4 `mktime()` . . . 365
29.5 `timegm()` . . . 366
29.6 `time()` . . . 368
29.7 `timespec_get()` . . . 369
29.8 `asctime()` . . . 371
29.9 `ctime()` . . . 372
29.10 `gmtime()` . . . 373
29.11 `localtime()` . . . 374

29.12 strftime() . . . 374

30 <uchar.h> Unicode utility functions 378
30.1 Types . . . 378
30.2 OS X issue . . . 378
30.3 mbrtoc16() mbrtoc32() . . . 379
30.4 c16rtomb() c32rtomb() . . . 381

31 <wchar.h> Wide Character Handling 385
31.1 Restartable Functions . . . 386
31.2 wprintf(), fwprintf(), swprintf() . . . 386
31.3 wscanf() fwscanf() swscanf() . . . 388
31.4 vwprintf() vfwprintf() vswprintf() . . . 389
31.5 vwscanf(), vfwscanf(), vswscanf() . . . 390
31.6 getwc() fgetwc() getwchar() . . . 391
31.7 fgetws() . . . 392
31.8 putwchar() putwc() fputwc() . . . 393
31.9 fputws() . . . 394
31.10 fwide() . . . 395
31.11 ungetwc() . . . 397
31.12 wcstod() wcstof() wcstold() . . . 398
31.13 wcstol() wcstoll() wcstoul() wcstoull() . . . 399
31.14 wcscpy() wcsncpy() . . . 401
31.15 wmemcpy() wmemmove() . . . 401
31.16 wcscat() wcsncat() . . . 402
31.17 wcscmp(), wcsncmp(), wmemcmp() . . . 403
31.18 wcscoll() . . . 405
31.19 wcsxfrm() . . . 406
31.20 wcschr() wcsrchr() . . . 407
31.21 wcsspn() wcscspn() . . . 408
31.22 wcspbrk() . . . 409
31.23 wcsstr() . . . 410
31.24 wcstok() . . . 411
31.25 wcslen() . . . 412
31.26 wcsftime() . . . 413
31.27 btowc() wctob() . . . 414
31.28 mbsinit() . . . 415
31.29 mbrlen() . . . 417
31.30 mbrtowc() . . . 418
31.31 wcrtomb() . . . 419
31.32 mbsrtowcs() . . . 421
31.33 wcsrtombs() . . . 422

32 <wctype.h> Wide Character Classification and Transformation 425
32.1 iswalnum() . . . 425
32.2 iswalpha() . . . 426
32.3 iswblank() . . . 427
32.4 iswcntrl() . . . 428
32.5 iswdigit() . . . 429
32.6 iswgraph() . . . 429
32.7 iswlower() . . . 430
32.8 iswprint() . . . 431
32.9 iswpunct() . . . 432
32.10 iswspace() . . . 433
32.11 iswupper() . . . 434
32.12 iswxdigit() . . . 434
32.13 iswctype() . . . 435
32.14 wctype() . . . 437
32.15 towlower() . . . 438

32.16 `towupper()` 439
32.17 `towctrans()` 440
32.18 `wctrans()` 442

Chapter 1

Foreword

The door slowly creaks open revealing a long hall with dusty stacks of books of lore…

I admit, maybe not that.

But you have found the Library Reference portion of Beej's Guide to C!

This isn't a tutorial, but rather is a comprehensive set of manual pages (or *man pages* as Unix hackers like to say) that define *every* function in the C Standard Library, complete with examples.

> *"This book, sir, contains every word in our beloved language."*
> *"Every single one, sir?"*
> *"Every single one, sir!"*
> *"Ah, well in that case, sir, I hope you will not object if I also offer the doctor my most enthusiastic contrafribularities."*
>
> —Blackadder toying with Dr. Samuel Johnson

There are, in fact, a number of functions left out of this guide, most notably all the optional "safe" functions (with a `_s` suffix).

But everything you're likely to want is definitely covered in here. With examples.

Probably.

1.1 Audience

This guide is for people who are at least modestly proficient in C.

If you are not one of those people and wish to become one of those people, I can wholeheartedly recommend with zero bias the book *Beej's Guide to C Programming*[1], freely available wherever the Internet is sold.

1.2 How to Read This Book

Use the contents or index to find the function or category you're after.

Then grab a bowl of your favorite cereal and devour the delicious, delicious verbiage.

1.3 Platform and Compiler

I'll try to stick to Plain Ol'-Fashioned ISO-standard C[2]. Well, for the most part. Here and there I might go crazy and start talking about POSIX[3] or something, but we'll see.

[1] https://beej.us/guide/bgc/
[2] https://en.wikipedia.org/wiki/ANSI_C
[3] https://en.wikipedia.org/wiki/POSIX

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[4]. Or, if you're looking for a more Unix-like experience (recommended!), install WSL[5] and `gcc`.

Mac users will want to install XCode[6], 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/bgclr/`[7]. There used to be a note here about migrating off Chico State's computers (my alma mater), but that's something that happened roughly a zillion years ago and the wording remained here only because it was copied over from the Network Guide, [*breath*] which I apparently haven't read in its entirety for quite some time.

The End.

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!

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 Programming–Library Reference is Copyright © 2021 Brian "Beej Jorgensen" Hall.

[4] https://visualstudio.microsoft.com/vs/community/
[5] https://docs.microsoft.com/en-us/windows/wsl/install-win10
[6] https://developer.apple.com/xcode/
[7] https://beej.us/guide/bgclr/

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[8] who provide the bridge that leads from the spec to the real world.
- The helpful and knowledgeable folks on comp.lang.c[9] and r/C_Programming[10] who got me through the tougher parts of the language.
- Everyone who submitted corrections and pull-requests on everything from misleading instructions to typos.

Thank you! ♥

[8] https://en.cppreference.com/
[9] https://groups.google.com/g/comp.lang.c
[10] https://www.reddit.com/r/C_Programming/

Chapter 2

The C Language

This is just a quick overview of the fashionable and fun highlights of the syntax, keywords, and other animals in the C menagerie.

2.1 Background

Some things you'll need to make sense of the examples, below.

2.1.1 Comments

Comments in C start with `//` and go to the end of a line.

Multiline comments begin with `/*` and continue until a closing `*/`.

2.1.2 Separators

Expressions in C are separated by semicolons (`;`). These tend to appear at the ends of lines.

2.1.3 Expressions

If it's not a keyword or other reserved punctuation, it tends to be an expression. Think "math including function calls".

2.1.4 Statements

Think `if`, `while`, etc. Executable keywords.

2.1.5 Booleans

Ignoring the `bool` type, zero is false and non-zero is true.

2.1.6 Blocks

Multiple expressions and flow control keywords can be wrapped up in a block, made up of `{` followed by one or more expressions or statements, followed by `}`.

2.1.7 Code Examples

They are meant to give an idea of how to use various statements, but not be comprehensive in terms of examples.

In the examples, below, if either an expression or statement can be used, the word `code` is inserted.

2.2 Operators

2.2.1 Arithmetic Operators

The arithmetic operators: `+`, `-`, `*`, `/`, `%` (remainder).

Division is integer if all arguments are integers. Otherwise it's a floating result.

You can also negate an expression by putting `-` in front of it. (You can also put a `+` in front of it—this doesn't do anything mathematically, but it causes the Usual Arithmetic Conversions to be performed on the expression.)

2.2.2 Pre- and Post-Increment and -Decrement

The post-increment (`++`) and post-decrement (`--`) operators (after the variable) do their work *after* the rest of the expression has been evaluated.

```
int x = 10;
int y = 20;
int z = 30;

int w = (x++) + (y--) + (z++);

print("%d %d %d %d\n", x, y, z, w);  // 11 19 31 60
```

The pre-increment (`++`) and pre-decrement (`--`) operators (before the variable) do their work *before* the rest of the expression has been evaluated.

```
int x = 10;
int y = 20;
int z = 30;

int w = (++x) + (--y) + (++z);

print("%d %d %d %d\n", x, y, z, w);  // 11 19 31 61
```

2.2.3 Comparison Operators

All of these return a Boolean true-y or false-y value.

Less than, greater than, and equal to are: `<`, `>`, `==`, respectively.

Less than or equal to and greater than or equal to are `<=` and `>=`.

Not equal to is `!=`.

2.2.4 Pointer Operators

`*` in front of a pointer variable dereferences that variable.

`&` in front of a variable gives the address of that variable.

`+` and `-` arithmetic operators work on pointers for pointer arithmetic.

2.2.5 Structure and Union Operators

The dot operator (`.`) can get a field value out of a `struct` or `union`.

The arrow operator (`->`) can get a field value out of a pointer to a `struct` or `union`. These two are equivalent, assuming `p` is just such a pointer:

```
(*p).bar;
p->bar;
```

2.2.6 Array Operators

The square bracket operators can reference a value in an array:

```
a[10] = 99;
```

This is syntactic sugar over pointer arithmetic and referencing. The above line is equivalent to:

```
*(a + 10) = 99;
```

2.2.7 Bitwise Operators

Bit shift right: `>>`, bit shift left: `<<`.

```
int i = x << 3;  // left shift 3 bits
```

Whether or not a right shift on a signed value is sign-extended is implementation-defined.

Bitwise AND, OR, NOT, and XOR are `&`, `|`, `~`, and `^`, respectively.

2.2.8 Assignment Operators

A standalone `=` is your basic assignment.

But there are also compound assignments that are like a shorthand version. For example, these two are basically equivalent:

```
x = x + 1;
x += 1;
```

There are compound assignment operators for many of the other operators.

Arithmetic: `+=`, `-=`, `*=`, `/=`, and `%=`.

Bitwise: `|=`, `&=`, `~=`, and `^=`.

2.2.9 The `sizeof` Operator

This is a compile-time operator that gives you the size in bytes of the type of the argument. The type of the expression is used; the expression is not evaluated. `sizeof` works with any type, even user-defined composite types.

The return type is the integer type `size_t`.

```
float f;
size_t x = sizeof f;

printf("f is %zu bytes\n", x);
```

You can also specify a raw type name in there by wrapping it in parentheses:

```
size_t x = sizeof(int);

printf("int is %zu bytes\n", x);
```

2.2.10 Type Casts

You can force an expression to be another type (within reason) by *casting* to that type.

You give the new type name in parentheses.

Here we are forcing the subexpression `x` to be type `float` just before the division[1]. This causes the division, which would otherwise be an integer division, to be a floating point division.

```
int x = 17;
int y = 2;

float f = (float)x / y;
```

2.2.11 `_Alignof` Operator

You can get the byte alignment of any type with the `_Alignof` compile-time operator. If you include `<stdalign.h>`, you can use `alignof` instead.

Any type can be the argument to the operator, which must be in parenthesis. Unlike `sizeof`, the argument cannot be an expression.

```
printf("Alignment of int is %zu\n", alignof(int));
```

2.2.12 Comma Operator

You can separate subexpressions with commas, and each will be evaluated from left to right, and the value of the entire expression will be the value of the subexpression after the last comma.

```
int x = (1, 2, 3);  // Silly way to assign `x = 3`
```

Usually this is used in the various clauses in loops. For example, we can do multiple assignments in a `for` loop, and have multiple post expressions like this:

```
for (i = 2, j = 10; i < 100; i++, j += 4) { ... }
```

2.3 Type Specifiers

Integer types from smallest to largest capacity: `char`, `short`, `int`, `long`, `long long`.

Any integer type may be prefaced with `signed` (the default except for `char`) or `unsigned`.

Whether or not `char` is signed is implementation defined.

Floating types from least accuracy to most: `float`, `double`, `long double`.

`void` is a type representing lack of type.

`_Bool` is a Boolean type. This becomes `bool` in C23. Earlier versions of C must include `<stdbool.h>` to get `bool`.

`_Complex` indicates a complex floating type type, when paired with such a type. Include `<complex.h>` to use `complex` instead.

```
complex float x = 1.2 + 2.3*I;
complex double y = 1.2 + 2.3*I;
```

[1] This doesn't change the type of `x` in other contexts—it's just in this one usage in this expression.

`_Imaginary` is an optional keyword used to specify an imaginary type (the imaginary part of a complex number) when paired with a floating type. Include `<complex.h>` to use `imaginary` instead. Neither GCC nor clang support this.

```
imaginary float f = 2.3*I;
```

`_Generic` is a type "switcher" that allows you to emit different code at compile time depending on the type of the data.

2.4 Constant Types

You can declare constants to be of specific types (though it might be a larger type). In the following example unqualified types, case doesn't matter, and the `U` can come before or after the `L` or `LL`.

```
123              int or larger
123L             long int or larger
123LL            long long int

123U             unsigned int or larger
123UL            unsigned long int or larger
123ULL           unsigned long long int

123.4F           float
123.4            double
123.4L           long double

'a'              char
"hello, world"   char* (string)
```

You can specify the constant in other bases as well:

```
123              decimal
0x123            hexadecimal
0123             octal
```

You can also specify floating constants in base-10 exponential notation:

```
1.2e3            1.2 x 10^3
```

And you can specify floats in hex! Except in this case the exponent is still in decimal, and the base is 2 instead of 10:

```
0x1.2p3          0x1.2 x 2^3
```

2.5 Composite Types

2.5.1 `struct` Types

You can build a composite type made out of other types with `struct` and then declare variables to be of that type.

```
struct animal {
    char *name;
    int leg_count;
};
```

```
struct animal a;
struct animal b = {"goat", 4};
struct animal c = {.name="goat", .leg_count=4};
```

Accessing is done with the dot operator (`.`) or, if the variable is a pointer to a `struct`, the arrow operator (`->`).

```
struct animal *p = b;

printf("%d\n", b.leg_count);
printf("%d\n", p->leg_count);
```

2.5.2 `union` Types

These are like `struct` types in usage, except that you can only use one field at a time. (The fields all use the same region of memory.)

```
union dt {
    float distance;
    int time;
};

union dt a;
union dt b = {6};            // Initializes "distance", the first field
union dt c = {.distance=6};  // Initializes "distance"
union dt d = {.time=6};      // Initializes "time"
```

Accessing is done with the dot operator (`.`) or, if the variable is a pointer to a `union`, the arrow operator (`->`).

```
union dt *p = b;

printf("%d\n", b.time);
printf("%d\n", p->time);
```

2.5.3 `enum` Types

Gives you a typed way to have named constant integer values. These can be used with `switch()`, or as an array size, or any other place constant values are needed.

Names are conventionally capitalized.

```
enum animal {
    ANTELOPE,
    BADGER,
    CAT,
    DOG,
    ELEPHANT,
    FISH
};

enum animal a = CAT;

if (a == CAT)
    printf("The animal is a cat.\n");
```

The names have numeric values starting with zero and counting up. (In the example above, `DOG` would be `3`.)

The numeric value can be overridden by specifying an integer exactly. Subsequent values increment from the specified one.

```
enum animal {
    ANTELOPE = 4,
    BADGER,          // Will be 5
    CAT,             // Will be 6
    DOG = 3,
    ELEPHANT,        // Will be 4
    FISH             // Will be 5
};
```

As above, duplicate values are not illegal, but might be of marginal usefulness.

2.6 Initializers

You can do this when the variable is defined, but not elsewhere.

Initializing basic types:

```
int x = 12;
float y = 1.2;
char c = 'a';
char *s = "Hello, world!";
```

Initializing array types:

```
int a[3] = {1,2,3};
int a[] = {1,2,3};   // Same as a[3]

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

Initializing pointer types:

```
int q;
int *p = &q;
```

Initializing `struct`s:

```
struct s {
    int a;
    float b;
};

struct s x0 = {1, 2.2}; // Initialize fields in order

struct s x0 = {.a=1, .b=2.2}; // Initialize fields by name
struct s x0 = {.b=2.2, .a=1}; // Same thing

struct s x0 = {.b=2.2}; // All other fields initialized to 0
struct s x0 = {.b=2.2, .a-=0};  // Same thing
```

Initializing `union`s:

```
union u {
    int a;
    float b;
};

union u x0 = {1};  // Initialize the first field (a)

union u x0 = {.a=1};  // Initialize fields by name
union u x0 = {.b=2.2};

//union u x0 = {1, 2};       // ILLEGAL
//union u x0 = {.a1, ,b=2};  // ILLEGAL
```

2.7 Compound Literals

You can declare "unnamed" objects in C. This is often useful for passing a `struct` to a function that otherwise doesn't need a name.

You use the type name in parens followed by an initializer to make the object.

Here's an example of passing a compound literal to a function. Note that there's no `struct s` variable in `main()`:

```
#include <stdio.h>

struct s {
    int a, b;
};

int add(struct s x)
{
    return x.a + x.b;
}

int main(void)
{
    int t = add((struct s){.a=2, .b=4});  // <-- Here

    printf("%d\n", t);
}
```

Compound literals have the lifetime of their scope.

You can also pass a pointer to a compound literal by taking its address:

```
foo(&(struct s){1, 2});
```

2.8 Type Aliases

You can set up a type alias for convenience or abstraction.

Here we'll make a new type called `time_counter` that is just an `int`. It can only be used exactly like an `int`. It's just an alias for an `int`.

```c
typedef int time_counter;

time_counter t = 3490;
```

Also works with `structs` or `unions`:

```c
struct foo {
    int bar;
    float baz;
};

typedef struct foo funtype;

funtype f = {1, 2}; // "funtype" is an alias for "struct foo";
```

It also works inline, and with named or unnamed `structs` or `unions`:

```c
typedef struct {
    int bar;
    float baz;
} funtype;

funtype f = {1, 2}; // "funtype" is an alias for the unnamed struct
```

2.9 Additional Type-Related Specifiers

You can give the compiler more hints about what qualities a type should have using these specifiers and qualifiers.

2.9.1 Storage Class Specifiers

These can be placed before a type to provide more guidance about how the type is used.

```c
auto int a
register int a
static int a
extern int a
thread_local int a
```

`auto` is the default, so it's basically never used. Indicates automatic storage duration (things like local variables get freed automatically when they fall out of scope). In C23 this keyword changes to indicate type inference like C++.

`register` indicates that accessing this variable should be as quick as possible. Restricts some usage of the variable giving the compiler a chance to optimize. Rare in daily use.

`static` at function scope indicates that this variable's value should persist from call to call. At file scope indicates that this variable should not be visible outside of this source file.

`extern` indicates that this variable refers to one declared in another source file.

`_Thread_local` means that every thread gets its own copy of this variable. You can use `thread_local` if you include `<threads.h>`.

2.9.2 Type Qualifiers

These can be placed before a type to provide more guidance about how the type is used.

```
const int a
const int *p
int * const p
const int * const p
int * restrict p
volatile int a
atomic int a
```

`const` means the value can't be modified. You can use it with pointers, as well:

```
const int a = 10;         // Can't modify "a"

const int *p = &b         // Can't modify the thing "p" points to ("b")
int *const p = &b         // Can't modify "p"
const int *const p = &b   // Can't modify "p" or the thing it points to
```

`restrict` on a pointer means that there will only be one pointer to the item in question, freeing the compiler to make some optimizations.

`volatile` indicates that the value in a variable might change at any time and should be loaded from memory instead of being kept in a register. Usually used with memory-mapped hardware.

`_Atomic` (or `atomic` if you include `<stdatomic.h>`) tells the compiler that reads or writes to this type should happen atomically. (This might be accomplished with a lock depending on the platform and type.)

2.9.2.1 C23 Qualified Pseudo-Types: `QVoid*`, `QChar*`, etc.

There are some generic functions in C23 that will return a `const`-qualified type if one of the parameters is `const`, but not otherwise.

The spec makes up a fake type for this, with a `Q` at the front (for "qualified"). This is not a real type and will not compile—it's just for documentation purposes.

These pseudotypes are:

- `QVoid *`
- `QChar *`
- `QWchar_t *`

For example, the `strchr()` function, which searches a string for a character, has this prototype in the spec:

```
QChar *strchr(QChar *s, int c);
```

What is it? It basically means that if `s` is type `const char *`, then the return type of the function will also be `const char *`.

If `s` is merely `char *`, the return type of the function will merely be `char *`.

In other words, the `const`-ness of `s` is preserved in the return value.

Another way to look at it is that this:

```
QChar *strchr(QChar *s, int c);
```

is the same as:

```
char *strchr(char *s, int c);
const char *strchr(const char *s, int c);
```

The TLDR is when you see this, drop the leading `Q` and change the next letter to lowercase and you're there.

2.9.3 Function Specifiers

These are used on functions to provide additional guidance for the compiler.

`_Noreturn` indicates that a function will never return. It can only run forever or exit the program entirely. If you include `<stdnoreturn.h>`, you can use `noreturn` instead.

`inline` indicates that calls to this function should be as fast as possible. The intention here is that the code of the function be moved *inline* to remove the overhead of the call and return. The compiler regards `inline` as a suggestion, not a requirement.

2.9.4 Alignment Specifier

You can force the alignment of a variable with memory with `_Alignas`. If you include `<stdalign.h>` you can use `alignas` instead.

`alignas(0)` has no effect.

```
alignas(16) int a = 12;     // 16-byte alignment
alignas(long) int b = 34;   // Same alignment as "long"
```

2.10 `if` Statement

```
if (boolean_expression) code;

if (boolean_expression) {
    code;
    code;
    code;
}

if (boolean_expression) {
    code;
    code;
} else
    code;

if (boolean_expression) {
    code;
    code;
} else if {
    code;
    code;
    code;
} else {
    code;
}
```

2.11 `for` Statement

Classic `for`-loop.

The bit in parens comes in three parts separated by semicolons:

- Initialization, executed once.
- Block entry condition, evaluated every time before entering the loop body.
- Post expression, evaluated every time after the loop body.

For example, initialize `i` to `0`, enter the loop body while `i < 10`, and then increment `i` after each loop iteration:

```
for (i = 0; i < 10; i++) {
    code;
    code;
    code;
}
```

You can declare loop-local variables by specifying their type:

```
for (int i = 0; i < 10; i++) {
    code;
    code;
}
```

You can separate parts of the expressions with the comma operator:

```
for (i = 0, j = 5; i < 10; i++, j *= 3) {
    code;
    code;
}
```

2.12 `while` Statement

This loop won't enter if the Boolean expression is false. The continuation test happens before the loop.

```
while (boolean_expression) code;

while (boolean_expression) {
    code;
    code;
}
```

2.13 `do-while` Statement

This loop will run at least once even if the Boolean expression is false. The continuation test doesn't happen until after the loop.

```
do code while (boolean_expression);

do {
    code;
    code;
} while (boolean_expression);
```

2.14 `switch` Statement

Performs actions based on the value of an expression. The cases that it is compared against must be constant values.

If the optional `default` is present, that code is executed if none of the cases match. Braces are not required around the cases.

```
switch (expression) {
    case constant:
        code;
        code;
        break;

    case constant:
        code;
        code;
        break;

    default:
        code;
        break;
}
```

The final `break` in the `switch` is unnecessary if there are no cases after it.

If the `break` isn't present, the `case` falls through to the next one. It's nice to put a comment to that effect so other devs don't hate you.

```
switch (expression) {
    case constant:
        code;
        code;
        // fall through!

    case constant:
        code;
        break;
}
```

2.15 `break` Statement

This breaks out of a `switch` case, but it also can break out of any loop.

```
while (boolean_expression) {
    code;

    if (boolean_expression)
        break;

    code;
}
```

2.16 `continue` Statement

This can be used to short-circuit a loop and go to the next continuation condition test without completing the body of the loop.

```
while (boolean_expression) {
    code;
    code;
```

```
    if (boolean_expression_2)
        continue;

    // If boolean_expression_2, code down here will be skipped:

    code;
    code;
}
```

2.17 `goto` Statement

You can just jump anywhere within a function with `goto`. (You can't `goto` between functions, only within the same function as the `goto`.)

The destination of the `goto` is a *label*, which is an identifier followed by a colon (`:`). Labels are typically left-justified all the way to the margin to make them visually stand out.

```
{
    // Abusive demo code that should be a while loop

    int i = 0;

loop:

    printf("%d\n", i++);

    if (i < 10)
        goto loop;
}
```

2.18 `return` Statement

This is how you get back from a function. You can `return` multiple times or just once.

If a function with `void` return type falls off the end, the `return` is implicit.

If the return type is not `void`, the `return` statement must specify a return value of the same type.

Parentheses around the return value are not necessary (as it's a statement, not a function).

```
int increment(int a)
{
    return a + 1;
}
```

2.19 `_Static_assert` Statement

This is a way to prevent *compilation* of a program if a certain constant condition is not met.

```
_Static_assert(__STDC_VERSION__ >= 201112L, "You need at least C11!")
```

2.20 Functions

You need to specify the return type and parameter types for the function, and the body goes in a block afterward.

Variables in the function are local to that function.

```
// Function that adds two numbers

int add(int x, int y)
{
    int sum = x + y;

    return sum;
}
```

Functions that return nothing should be return type `void`. Functions that accept no parameters should have `void` as the parameter list.

```
// All side effects, all the time!

void foo(void)
{
    some_global = 12;
    printf("Here we go!\n");
}
```

2.20.1 `main()` Function

This is the function that runs when you first start the program. It will be one of these forms:

```
int main(void)
int main(int argc, char *argv[])
```

The first form ignores all command line parameters.

The second form stores the count of the command line parameters in `argc`, and stores the parameters themselves as an array of strings in `argv`. The first of these, `argv[0]`, is typically the name of the executable. The last `argv` pointer has the value `NULL`.

The return values usually show up as exit status codes in the OS. If there is no `return`, falling off the end of `main()` is an implied `return 0`[2].

2.20.2 Variadic Functions

Some functions can take a variable number of arguments. Every function must have at least one argument. The remaining arguments are specified by `...` and can be read with the `va_start()`, `va_arg()`, and `va_end()` macros.

Here's an example that adds up a variable number of integer values.

```
int add(int count, ...)
{
    int total = 0;
    va_list va;

    va_start(va, count);   // Start with arguments after "count"
```

[2]Note that this implication only for `main()`, and not for any other functions.

```
    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;
}
```

Chapter 3

`<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.

3.1 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()`.

3.2 `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, you can roll your own assert as a preprocessor macro:

```
#define ASSERT(c, m) \
do { \
    if (!(c)) { \
        fprintf(stderr, __FILE__ ":%d: assertion %s failed: %s\n", \
                        __LINE__, #c, m); \
        exit(1); \
    } \
} while(0)
```

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()`

3.3 `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()`

Chapter 4

`<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.

4.1 `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()`

4.2 `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()`

4.3 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()`

4.4 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()`

4.5 `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()`

4.6 `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()`

4.7 `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

```c
#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()`

4.8 `casinh()`, `casinhf()`, `casinhl()`

Compute the complex arc hyperbolic sine

Synopsis

```c
#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()`

4.9 `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()`

4.10 `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

```
#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()`

4.11 `csinh()`, `csinhf()`, `csinhl()`

Compute the complex hyperbolic sine

Synopsis

```
#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

```
#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()`

4.12 `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()`

4.13 `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()`

4.14 `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

```c
#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()`

4.15 `cabs()`, `cabsf()`, `cabsl()`

Compute the complex absolute value

Synopsis

```c
#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()`

4.16 `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()`

4.17 `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()`

4.18 `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
```

4.19 cimag(), cimagf(), cimagl()

Returns the imaginary part of a complex number

Synopsis

```c
#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:

```c
x == creal(x) + cimag(x) * I;
```

Return Value

Returns the imaginary part of `z`.

Example

```c
#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()`

4.20 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()`

4.21 `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
```

---

[1] https://en.wikipedia.org/wiki/Complex_conjugate

4.22 `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.

> `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:

[2] https://en.wikipedia.org/wiki/Riemann_sphere

```
Result: 1.000000 + 2.000000i
Result: inf + 0.000000i
```

4.23 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 5

`<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*!

5.1 `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()`

5.2 `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()`

5.3 `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()`

5.4 `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()`

5.5 `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()`

5.6 `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()`

5.7 `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()`

5.8 `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()`

5.9 `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()`

5.10 `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()`

5.11 `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()`

5.12 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()`

5.13 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()`

5.14 `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 6

`<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].

6.1 `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).

Value	Description
`0`	No error

[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

Value	Description
`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 7

`<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

7.1 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
`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.

7.2 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.

7.3 `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()`

7.4 `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!
}
```

7.5 `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()`

7.6 `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()`

7.7 `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

```
#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()`

7.8 `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.

```c
#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()`

7.9 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()`

7.10 `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.

```
#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);
}

```

```c
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()`

Chapter 8

`<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`
`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`

Macro	Minimum Magnitude	Description
`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!

8.1 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
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!

8.2 `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.

8.3 `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`

8.4 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`	Subnormals supported for this type

8.5 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:

[1] https://en.wikipedia.org/wiki/Subnormal_number

```c
#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. (And `DBL_DECIMAL_DIG` and `LDBL_DECIMAL_DIG` for those corresponding types.)

Here's some example output that shows how the value stored might have some extra decimal places at the end.

```c
#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?

8.6 Comprehensive Example

Here's a program that prints out the details for a particular machine:

```c
#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 9

`<inttypes.h>` More Integer Conversions

Function	Description
`imaxabs()`	Compute the absolute value of an `intmax_t`
`imaxdiv()`	Compute the quotient and remainder of `intmax_t`s
`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.

9.1 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:

```c
#include <stdio.h>     // for printf()
#include <inttypes.h>

int main(void)
{
    int16_t x = 32;

    printf("The value is %" PRId16 "!\n", x);
}
```

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.

9.2 `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()`

9.3 `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()`

9.4 `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`

9.5 `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 10

`<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:

```c
#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.

Chapter 11

`<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`

11.1 `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`.

Side benefit: you can tell at runtime if the system has signed or unsigned chars by checking to see if `CHAR_MIN` is `0`.

[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.

```c
#include <stdio.h>
#include <limits.h>

int main(void)
{
    printf("chars are %ssigned\n", CHAR_MIN == 0? "un": "");
}
```

On my system, `char`s are signed.

11.2 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 `int`s 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>`.

11.3 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.

11.4 Demo Program

Here's a program to print out the values of the macros:

```c
#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);
    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);
```

[2] https://en.wikipedia.org/wiki/Two%27s_complement

```
    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 `char`s are signed, and my `int`s are 32-bit.

Chapter 12

`<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.

12.1 `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>`
`LC_MONETARY`	Affects the monetary information returned from `localeconv()`

Category	Description
`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>`

12.2 `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. `"$"`.
`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.

[1] Remember that `char` is just a byte-sized integer.

Field	Description
`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.

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.

```c
#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 13

`<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.
`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.

Function	Description
`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.

13.1 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.

13.2 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.

13.3 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.

13.4 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.

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]:

[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.

`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.

13.5 Math Pragmas

In a nutshell, pragmas offer various ways to control the compiler's behavior. In this case, we're talking about controlling how C's math library works.

In specific, 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.

13.6 `fpclassify()`

Return the classification of a given floating point number.

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.

```c
#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)
{
    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!
}
```

[2] https://en.wikipedia.org/wiki/Denormal_number

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()`

13.7 `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()`.

Return Value

Returns non-zero for true, and zero for false.

Example

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

```

[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.

```
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()`,

13.8 `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()`

13.9 `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()`

13.10 `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()`

13.11 `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()`

13.12 `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()`

13.13 `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()`

13.14 `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()`

13.15 `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()`

13.16 `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()`

13.17 `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()`

13.18 `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()`

13.19 `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()`

13.20 `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()`

13.21 `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.

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
}
```

[4] https://en.wikipedia.org/wiki/E_(mathematical_constant)

See Also

`exp2()`, `expm1()`, `pow()`, `log()`

13.22 `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()`

13.23 `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()`

13.24 `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

```c
#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()`

13.25 `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()`

13.26 `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()`

13.27 `log()`, `logf()`, `logl()`

Compute the natural logarithm.

Synopsis

```
#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()`

13.28 `log10()`, `log10f()`, `log10l()`

Compute the log-base-10 of a number.

Synopsis

```c
#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()`

13.29 `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()`

13.30 `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()`

13.31 `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

```
#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
```

[5] https://en.wikipedia.org/wiki/Denormal_number

See Also

`ilogb()`

13.32 `modf()`, `modff()`, `modfl()`

Extract the integral and fractional parts of a number.

Synopsis

```
#include <math.h>

double modf(double value, double *iptr);

float modff(float value, float *iptr);

long double modfl(long double value, long double *iptr);
```

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

```
#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()`

13.33 `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()`

Suffix	Meaning
`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

```
#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()`

13.34 `cbrt()`, `cbrtf()`, `cbrtl()`

Compute the cube root.

Synopsis

```
#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

```
#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()`

13.35 `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()`

13.36 `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()`

---

[6] https://en.wikipedia.org/wiki/Pythagorean_theorem

13.37 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);
```

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()

13.38 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()`

13.39 `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.

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()`

13.40 `erfc()`, `erfcf()`, `erfcl()`

Compute the complementary error function of a value.

Synopsis

```
#include <math.h>

double erfc(double x);
```

[7] https://en.wikipedia.org/wiki/Error_function

```
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.

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()`

13.41 `lgamma()`, `lgammaf()`, `lgammal()`

Compute the natural logarithm of the absolute value of $\Gamma(x)$.

Synopsis

```
#include <math.h>

double lgamma(double x);
```

[8] https://en.wikipedia.org/wiki/Error_function

```
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.

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()`

13.42 `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);
```

[9] https://en.wikipedia.org/wiki/Gamma_function

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`, $\Gamma(x)$.

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()`

13.43 `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.)

[10] https://en.wikipedia.org/wiki/Gamma_function

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()`

13.44 `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()`

13.45 `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()`

13.46 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()`

13.47 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()`

13.48 `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()`

13.49 `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 fit in the return type, 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()`

13.50 `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()`

13.51 `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()`

13.52 `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()`

13.53 `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
}
```

See Also

`remainder()`, `imaxdiv()`

13.54 `copysign()`, `copysignf()`, `copysignl()`

Copy the sign of one value into another.

Synopsis

```c
#include <math.h>

double copysign(double x, double y);

float copysignf(float x, float y);

long double copysignl(long double x, long double y);
```

[11] https://en.cppreference.com/w/c/numeric/math/remquo
[12] https://en.cppreference.com/w/c/numeric/math/remquo

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()`

13.55 `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?

[13] A *quiet NaN* is one that doesn't raise any exceptions.

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>

int main(void)
{
    printf("%f\n", nan(""));        // nan
    printf("%f\n", nan("goats"));   // nan
    printf("%f\n", nan("!"));       // nan
}
```

See Also

`strtod()`

13.56 `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()`

13.57 `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()`

13.58 `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
}
```

13.59 `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
}
```

13.60 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
```

13.61 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()`

13.62 `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()`

13.63 `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

```c
#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 14

`<setjmp.h>` Non-local Goto

These functions enable you to rewind the call stack to an earlier point, with a bunch of gotchas. It is rarely used.

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()`.

14.1 `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()`.

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()`

14.2 `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.

```c
#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 15

`<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].

15.1 `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
- Perform the default action
- Have a specific function called

[1] https://man.archlinux.org/man/sigaction.2.en

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()`.

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()`

[2] As if might be sent from Unix's `kill` command.]

- `_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()`

15.2 `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()`

Chapter 16

`<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.

16.1 `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)
```

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`:

[1] https://en.wikipedia.org/wiki/Data_structure_alignment

```
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`, `memalignment()`

16.2 `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`, `memalignment()`

Chapter 17

`<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()`.

17.1 `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()`

17.2 `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()`

17.3 `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()`

17.4 `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 18

`<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.

18.1 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;
```

18.2 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.

18.3 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;
```

18.4 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.

18.5 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 `struct`s, 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()`

18.6 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()`

18.7 `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
}
```

18.8 `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)
`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

```c
#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()`

18.9 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()`

18.10 `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 `struct`s 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

18.11 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_*()`

18.12 `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()`

18.13 `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()`

18.14 `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
    }
```

[4]This effectively does the same thing, but it's clearly not atomic.

```
    *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.

(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);

```

[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."

```
    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_*()`

18.15 `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()`

18.16 `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;
}
```

[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.

```

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_clear()`

18.17 `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]:

```c
#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...
Thread 0: set s to {5, 6, 7}
Thread 0: releasing lock...
```

[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.

```
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 19

`<stdbit.h>` Bit-Related Functions

This header file and all the included functions are new in C23!

Caveat: none of my compilers support this, so all the code is untested!

Function	Description
`stdc_bit_ceil()`	Return the smallest power of 2 not less than a number
`stdc_bit_floor()`	Return the largest power of 2 not greater than a number
`stdc_bit_width()`	Return the number of bits needed to store a value
`stdc_count_ones()`	Count the ones in an unsigned number
`stdc_count_zeros()`	Count the zeros in an unsigned number
`stdc_first_leading_one()`	Find the first leading one in an unsigned number
`stdc_first_leading_zero()`	Find the first leading zero in an unsigned number
`stdc_first_trailing_one()`	Find the first trailing one in an unsigned number
`stdc_first_trailing_zero()`	Find the first trailing zero in an unsigned number
`stdc_has_single_bit()`	Test to see if an unsigned integer has only a single bit set
`stdc_leading_ones()`	Count the leading ones in an unsigned number
`stdc_leading_zeros()`	Count the leading zeros in an unsigned number
`stdc_trailing_ones()`	Count the trailing ones in an unsigned number
`stdc_trailing_zeros()`	Count the trailing zeros in an unsigned number

19.1 Existence Macro

Since this is a new feature, you can test for its existence by making sure the `__STDC_VERSION_STDBIT_H__` macro exists and is `202311L` or greater.

19.2 Endianess Macros

This header file defines a number of macros that can be used to determine the *endianess* of the system. (That is, in a multibyte value, do the first byte represent the most significant part of the value, or least? Or neither?)

There are two different values for the defined endianesses `__STDC_ENDIAN_BIG__` and `__STDC_ENDIAN_LITTLE__`.

Additionally, there's a macro `__STDC_ENDIAN_NATIVE__` that gives you the endianess of *this* system. You can compare that against the other two macros to see what you've got. If it doesn't compare equal to either of them, you must be on some other mixed-endian system.

Let's see what this system is:

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

int main(void)
{
    switch(__STDC_ENDIAN_NATIVE__) {
        case __STDC_ENDIAN_BIG__:
            puts("Big-endian!");
            break;

        case __STDC_ENDIAN_LITTLE__:
            puts("Little-endian!");
            break;

        default:
            puts("Other-endian!");
    }
}
```

19.3 General Structure of These Functions

All the functions in this header follow a standard pattern, so we might as well go over it once here up-front to save extraneous and excessive verbosity later.

Each function comes in six amazing forms, depending on the types involved.

First of all, they're just going to operate on unsigned integer types, so we'll get that out of the way.

And they're *mostly* going to return `unsigned int`.

And they come in two subforms: the type-specific and type-generic forms.

Let's look at the type specific forms first. We'll use the function that counts the leading zero bits in a value as an example. Here it is:

```
unsigned int stdc_leading_zeros_uc(unsigned char value);
unsigned int stdc_leading_zeros_us(unsigned short value);
unsigned int stdc_leading_zeros_ui(unsigned int value);
unsigned int stdc_leading_zeros_ul(unsigned long value);
unsigned int stdc_leading_zeros_ull(unsigned long long value);
```

Yikes. OK, what do we have?

Well, in order to prevent namespace pollution, they all start with `stdc_`.

And what about this `uc`, `us`, `ull`, etc. stuff? Those correspond to the type of the argument, you'll see with a little inspection.

Suffix	Parameter Type
`_uc`	`unsigned char`
`_us`	`unsigned short`
`_ui`	`unsigned int`
`_ul`	`unsigned long`
`_ull`	`unsigned long long`

But wait! There's more!

Each of these function families in the header also has a *generic* variant.

So, again with `count_leading_zeros` as an example, there's a version that doesn't need types to be specified, and it's just the function without any suffix.

```
generic_return_type stdc_leading_zeros(generic_value_type value);
```

In that example, the words `generic_return_type` and `generic_value_type` are **not** C keywords. They're just placeholders to let you know you need to put the right thing there.

```
unsigned short x = 3490;

unsigned int a = stdc_leading_zeros(x);
auto b = stdc_leading_zeros(1234);
```

The generic functions work with all unsigned types, not counting `bool`.

19.4 `stdc_leading_zeros()`

Count the leading zeros in an unsigned number

Synopsis

New in C23!

```
#include <stdbit.h>

unsigned int stdc_leading_zeros_uc(unsigned char value);
unsigned int stdc_leading_zeros_us(unsigned short value);
unsigned int stdc_leading_zeros_ui(unsigned int value);
unsigned int stdc_leading_zeros_ul(unsigned long value);
unsigned int stdc_leading_zeros_ull(unsigned long long value);

generic_return_type stdc_leading_zeros(generic_value_type value);
```

Description

This function returns the number of leading zero bits on a particular value, starting from the most-significant bit. The number will be influenced by the size of the argument.

Return Value

Returns the number of leading zero bits.

Example

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

int main(void)
{
    unsigned int value = 3490;
    unsigned int count = stdc_leading_zeros_ui(value);

    printf("%u\n", count);

```

```
    unsigned long long value2 = 3490;
    auto count2 = stdc_leading_zeros(value2);

    printf("%u\n", count2);
}
```

See Also

`stdc_leading_ones()`, `stdc_trailing_zeros()`

19.5 `stdc_leading_ones()`

Count the leading ones in an unsigned number

Synopsis

New in C23!

```
#include <stdbit.h>

unsigned int stdc_leading_ones_uc(unsigned char value);
unsigned int stdc_leading_ones_us(unsigned short value);
unsigned int stdc_leading_ones_ui(unsigned int value);
unsigned int stdc_leading_ones_ul(unsigned long value);
unsigned int stdc_leading_ones_ull(unsigned long long value);

generic_return_type stdc_leading_ones(generic_value_type value);
```

Description

This function returns the number of leading one bits on a particular value starting from the most-significant bit. The number will be influenced by the size of the argument.

Return Value

Returns the number of leading one bits.

Example

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

int main(void)
{
    unsigned int value = 3490;
    unsigned int count = stdc_leading_ones_ui(value);

    printf("%u\n", count);

    unsigned long long value2 = 3490;
    auto count2 = stdc_leading_ones(value2);

    printf("%u\n", count2);
```

```
}
```

See Also

`stdc_leading_zeros()`, `stdc_trailing_ones()`

19.6 `stdc_trailing_zeros()`

Count the trailing zeros in an unsigned number

Synopsis

New in C23!

```
#include <stdbit.h>

unsigned int stdc_trailing_zeros_uc(unsigned char value);
unsigned int stdc_trailing_zeros_us(unsigned short value);
unsigned int stdc_trailing_zeros_ui(unsigned int value);
unsigned int stdc_trailing_zeros_ul(unsigned long value);
unsigned int stdc_trailing_zeros_ull(unsigned long long value);

generic_return_type stdc_trailing_zeros(generic_value_type value);
```

Description

This function returns the number of trailing zero bits on a particular value, starting from the least-significant bit. The number will be influenced by the size of the argument.

Return Value

Returns the number of trailing zero bits.

Example

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

int main(void)
{
    unsigned int value = 3490;
    unsigned int count = stdc_trailing_zeros_ui(value);

    printf("%u\n", count);

    unsigned long long value2 = 3490;
    auto count2 = stdc_trailing_zeros(value2);

    printf("%u\n", count2);
}
```

See Also

`stdc_trailing_ones()`, `stdc_leading_zeros()`

19.7 `stdc_trailing_ones()`

Count the trailing ones in an unsigned number

Synopsis

New in C23!

```
#include <stdbit.h>

unsigned int stdc_trailing_ones_uc(unsigned char value);
unsigned int stdc_trailing_ones_us(unsigned short value);
unsigned int stdc_trailing_ones_ui(unsigned int value);
unsigned int stdc_trailing_ones_ul(unsigned long value);
unsigned int stdc_trailing_ones_ull(unsigned long long value);

generic_return_type stdc_trailing_ones(generic_value_type value);
```

Description

This function returns the number of trailing one bits on a particular value, starting from the least-significant bit. The number will be influenced by the size of the argument.

Return Value

Returns the number of trailing one bits.

Example

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

int main(void)
{
    unsigned int value = 3490;
    unsigned int count = stdc_trailing_ones_ui(value);

    printf("%u\n", count);

    unsigned long long value2 = 3490;
    auto count2 = stdc_trailing_ones(value2);

    printf("%u\n", count2);
}
```

See Also

`stdc_trailing_zeros()`, `stdc_leading_ones()`

19.8 stdc_first_leading_zero()

Find the first leading zero in an unsigned number

Synopsis

New in C23!

```
#include <stdbit.h>

unsigned int stdc_first_leading_zero_uc(unsigned char value);
unsigned int stdc_first_leading_zero_us(unsigned short value);
unsigned int stdc_first_leading_zero_ui(unsigned int value);
unsigned int stdc_first_leading_zero_ul(unsigned long value);
unsigned int stdc_first_leading_zero_ull(unsigned long long value);

generic_return_type stdc_first_leading_zero(generic_value_type value);
```

Description

This finds the index of the first leading zero in a number. Indexes are numbered starting from `1` being the most significant ("leftmost") bit position. (This might be in contrast to how you might be used to numbering bit indexes.)

It's one-based so that you can quickly use the return value as a Boolean expression for if it found a zero bit or not.

Return Value

Returns the 1-based index from the most significant bit of the first zero bit in the `value`, or `0` if there are no zero bits.

Example

```
#include <stdio.h>
#include <limits.h>
#include <stdbit.h>

int main(void)
{
    unsigned int value = UINT_MAX;
    unsigned int index = stdc_first_leading_zero_ui(value);

    printf("%u\n", index);

    unsigned long long value2 = UINT_MAX >> 2;
    auto index2 = stdc_first_leading_zero(value2);

    printf("%u\n", index2);
}
```

See Also

`stdc_first_leading_one()`, `stdc_first_trailing_zero()`

19.9 stdc_first_leading_one()

Find the first leading one in an unsigned number

Synopsis

New in C23!

```
#include <stdbit.h>

unsigned int stdc_first_leading_one_uc(unsigned char value);
unsigned int stdc_first_leading_one_us(unsigned short value);
unsigned int stdc_first_leading_one_ui(unsigned int value);
unsigned int stdc_first_leading_one_ul(unsigned long value);
unsigned int stdc_first_leading_one_ull(unsigned long long value);

generic_return_type stdc_first_leading_one(generic_value_type value);
```

Description

This finds the index of the first leading one in a number. Indexes are numbered starting from `1` being the most significant ("leftmost") bit position. (This might be in contrast to how you might be used to numbering bit indexes.)

It's one-based so that you can quickly use the return value as a Boolean expression for if it found a one bit or not.

Return Value

Returns the 1-based index from the most significant bit of the first one bit in the `value`, or `0` if there are no one bits.

Example

```
#include <stdio.h>
#include <limits.h>
#include <stdbit.h>

int main(void)
{
    unsigned int value = UINT_MAX;
    unsigned int index = stdc_first_leading_one_ui(value);

    printf("%u\n", index);

    unsigned long long value2 = UINT_MAX >> 2;
    auto index2 = stdc_first_leading_one(value2);

    printf("%u\n", index2);
}
```

See Also

`stdc_first_leading_zero()`, `stdc_first_trailing_one()`

19.10 `stdc_first_trailing_zero()`

Find the first trailing zero in an unsigned number

Synopsis

New in C23!

```
#include <stdbit.h>

unsigned int stdc_first_trailing_zero_uc(unsigned char value);
unsigned int stdc_first_trailing_zero_us(unsigned short value);
unsigned int stdc_first_trailing_zero_ui(unsigned int value);
unsigned int stdc_first_trailing_zero_ul(unsigned long value);
unsigned int stdc_first_trailing_zero_ull(unsigned long long value);

generic_return_type stdc_first_trailing_zero(generic_value_type value);
```

Description

This finds the index of the first trailing zero in a number. Indexes are numbered starting from `1` being the most significant ("leftmost") bit position. (This might be in contrast to how you might be used to numbering bit indexes.)

It's one-based so that you can quickly use the return value as a Boolean expression for if it found a zero bit or not.

Return Value

Returns the 1-based index from the most significant bit of the first zero bit in the `value`, or `0` if there are no zero bits.

Example

```
#include <stdio.h>
#include <limits.h>
#include <stdbit.h>

int main(void)
{
    unsigned int value = UINT_MAX;
    unsigned int index = stdc_first_trailing_zero_ui(value);

    printf("%u\n", index);

    unsigned long long value2 = UINT_MAX >> 2;
    auto index2 = stdc_first_trailing_zero(value2);

    printf("%u\n", index2);
}
```

See Also

`stdc_first_leading_zero()`, `stdc_first_trailing_one()`

19.11 stdc_first_trailing_one()

Find the first trailing one in an unsigned number

Synopsis

New in C23!

```
#include <stdbit.h>

unsigned int stdc_first_trailing_one_uc(unsigned char value);
unsigned int stdc_first_trailing_one_us(unsigned short value);
unsigned int stdc_first_trailing_one_ui(unsigned int value);
unsigned int stdc_first_trailing_one_ul(unsigned long value);
unsigned int stdc_first_trailing_one_ull(unsigned long long value);

generic_return_type stdc_first_trailing_one(generic_value_type value);
```

Description

This finds the index of the first trailing one in a number. Indexes are numbered starting from `1` being the most significant ("leftmost") bit position. (This might be in contrast to how you might be used to numbering bit indexes.)

It's one-based so that you can quickly use the return value as a Boolean expression for if it found a one bit or not.

Return Value

Returns the 1-based index from the most significant bit of the first one bit in the `value`, or `0` if there are no one bits.

Example

```
#include <stdio.h>
#include <limits.h>
#include <stdbit.h>

int main(void)
{
    unsigned int value = UINT_MAX;
    unsigned int index = stdc_first_trailing_one_ui(value);

    printf("%u\n", index);

    unsigned long long value2 = UINT_MAX >> 2;
    auto index2 = stdc_first_trailing_one(value2);

    printf("%u\n", index2);
}
```

See Also

`stdc_first_leading_one()`, `stdc_first_trailing_zero()`

19.12 stdc_count_zeros()

Count the zeros in an unsigned number

Synopsis

New in C23!

```
#include <stdbit.h>

unsigned int stdc_count_zeros_uc(unsigned char value);
unsigned int stdc_count_zeros_us(unsigned short value);
unsigned int stdc_count_zeros_ui(unsigned int value);
unsigned int stdc_count_zeros_ul(unsigned long value);
unsigned int stdc_count_zeros_ull(unsigned long long value);

generic_return_type stdc_count_zeros(generic_value_type value);
```

Description

This function returns the number of zero bits in a particular value. The number will be influenced by the size of the argument.

Return Value

Returns the number of zero bits.

Example

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

int main(void)
{
    unsigned int value = 3490;
    unsigned int count = stdc_count_zeros_ui(value);

    printf("%u\n", count);

    unsigned long long value2 = 123456;
    auto count2 = stdc_count_zeros(value2);

    printf("%u\n", count2);
}
```

See Also

`stdc_count_ones()()`, `stdc_leading_zeros()()`, `stdc_trailing_zeros()()`

19.13 stdc_count_ones()

Count the ones in an unsigned number

Synopsis

New in C23!

```
#include <stdbit.h>

unsigned int stdc_count_ones_uc(unsigned char value);
unsigned int stdc_count_ones_us(unsigned short value);
unsigned int stdc_count_ones_ui(unsigned int value);
unsigned int stdc_count_ones_ul(unsigned long value);
unsigned int stdc_count_ones_ull(unsigned long long value);

generic_return_type stdc_count_ones(generic_value_type value);
```

Description

This function returns the number of one bits in a particular value. The number will be influenced by the size of the argument.

Return Value

Returns the number of one bits.

Example

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

int main(void)
{
    unsigned int value = 3490;
    unsigned int count = stdc_count_ones_ui(value);

    printf("%u\n", count);

    unsigned long long value2 = 123456;
    auto count2 = stdc_count_ones(value2);

    printf("%u\n", count2);
}
```

See Also

`stdc_count_zeros()()`, `stdc_leading_ones()()`, `stdc_trailing_ones()()`

19.14 `stdc_has_single_bit()`

Test to see if an unsigned integer has only a single bit set

Synopsis

New in C23!

```
#include <stdbit.h>

bool stdc_has_single_bit_uc(unsigned char value);
bool stdc_has_single_bit_us(unsigned short value);
bool stdc_has_single_bit_ui(unsigned int value);
bool stdc_has_single_bit_ul(unsigned long value);
bool stdc_has_single_bit_ull(unsigned long long value);

bool stdc_has_single_bit(generic_value_type value);
```

Description

These functions return true if exactly one bit is `1` in an unsigned `value`.

If true, it also means the `value` is a power of 2.

Return Value

Return true if `value` has exactly a single bit set to `1`.

Example

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

int main(void)
{
    unsigned int value = 0b0001000;
    unsigned int result = stdc_has_single_bit_ui(value);

    printf("%d\n", result);    // 1

    unsigned long long value2 = 0b000011000
    auto result2 = stdc_has_single_bit(value2);

    printf("%d\n", result2);   // 0
}
```

19.15 `stdc_bit_width()`

Return the number of bits needed to store a value

Synopsis

New in C23!

```
#include <stdbit.h>

unsigned int stdc_bit_width_uc(unsigned char value);
unsigned int stdc_bit_width_us(unsigned short value);
unsigned int stdc_bit_width_ui(unsigned int value);
unsigned int stdc_bit_width_ul(unsigned long value);
unsigned int stdc_bit_width_ull(unsigned long long value);
```

```
generic_return_type stdc_bit_width(generic_value_type value);
```

Description

Given an unsigned integer value, what's the smallest number of bits needed to store it? That's the question this function answers.

Return Value

Returns the number of bits needed to store a positive `value`. Returns `0` if the `value` passed in is `0`.

Example

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

int main(void)
{
    unsigned int value = 0b0001000;
    unsigned int result = stdc_bit_width_ui(value);

    printf("%d\n", result);    // 4

    unsigned long long value2 = 0b000011010
    auto result2 = stdc_bit_width(value2);

    printf("%d\n", result2);   // 5
}
```

19.16 `stdc_bit_floor()`

Return the largest power of 2 not greater than a number

Synopsis

New in C23!

```
#include <stdbit.h>

unsigned char stdc_bit_floor_uc(unsigned char value);
unsigned short stdc_bit_floor_us(unsigned short value);
unsigned int stdc_bit_floor_ui(unsigned int value);
unsigned long stdc_bit_floor_ul(unsigned long value);
unsigned long long stdc_bit_floor_ull(unsigned long long value);

generic_value_type stdc_bit_floor(generic_value_type value);
```

Description

This function returns the largest power of 2 that is not greater than the `value`.

In other words "is less than or equal to" the `value`.

In other words, floor down to the nearest power of two.

In other words, return the value of the highest set bit in a number

For example: |`value`|Return| |-|-| |`0b101`|`0b100`| |`0b1000`|`0b1000`| |`0b100101`|`0b100000`|

Return Value

Returns the largest power of 2 not greater than `value`. Returns `0` if the `value` is `0`.

Example

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

int main(void)
{
    unsigned int value = 0b0001000;
    unsigned int result = stdc_bit_floor_ui(value);

    printf("%b\n", result);    // 1000

    unsigned long long value2 = 0b000011010
    auto result2 = stdc_bit_floor(value2);

    printf("%b\n", result2);   // 10000
}
```

See Also

`stdc_bit_ceil()`

19.17 `stdc_bit_ceil()`

Return the smallest power of 2 not less than a number

Synopsis

New in C23!

```
#include <stdbit.h>

unsigned char stdc_bit_ceil_uc(unsigned char value);
unsigned short stdc_bit_ceil_us(unsigned short value);
unsigned int stdc_bit_ceil_ui(unsigned int value);
unsigned long stdc_bit_ceil_ul(unsigned long value);
unsigned long long stdc_bit_ceil_ull(unsigned long long value);

generic_value_type stdc_bit_ceil(generic_value_type value);
```

Description

This function returns the smallest power of 2 that is not less than the `value`.

In other words “is greater than or equal to” the `value`.

In other words, ceil up to the nearest power of two.

For example: |`value`|Return| |-|-| |`0b101`|`0b1000`| |`0b1000`|`0b1000`| |`0b100101`|`0b1000000`|

If the result doesn't fit in the return type, the behavior is undefined.

Return Value

Returns the smallest power of 2 not less than `value`.

Example

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

int main(void)
{
    unsigned int value = 0b0001000;
    unsigned int result = stdc_bit_ceil_ui(value);

    printf("%b\n", result);    // 1000

    unsigned long long value2 = 0b000011010
    auto result2 = stdc_bit_ceil(value2);

    printf("%b\n", result2);   // 100000
}
```

See Also

`stdc_bit_floor()`

Chapter 20

`<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 one 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`.

20.1 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:

```
The universe still makes sense.
x is now 0
```

20.2 `_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 `typedef`s.

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 21

`<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

21.1 `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`.

```c
#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`:

```c
    printf("%td\n", d);  // Print decimal: 40
    printf("%tX\n", d);  // Print hex:     28
}
```

21.2 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);
}
```

21.3 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

21.4 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.

21.5 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 22

`<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.

22.1 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;

    // ...
```

22.2 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.

22.3 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 $2^{16} - 1$.

22.4 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).

22.5 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 23

`<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
`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

Function	Description
`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`.

23.1 `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()`

23.2 `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()`

23.3 `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)
}
```

See Also

`fopen()`, `fclose()`, `tmpnam()`

23.4 `tmpnam()`

Generate a unique name for a temporary file

[1] https://man.archlinux.org/man/unlinkat.2.en#DESCRIPTION

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.

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

```
#include <stdio.h>

int main(void)
{
    char filename[L_tmpnam];
    char *another_filename;
```

[2] https://man.archlinux.org/man/mkstemp.3.en

```

    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()`

23.5 `fclose()`

The opposite of `fopen()`—closes a file when you're done with it so that it frees system resources

Synopsis

```
#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()`

23.6 `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()`!

```
#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()`

23.7 `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

```c
#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()`

23.8 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()`

23.9 `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()`

23.10 `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*.

23.10.0.1 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?

23.10.0.2 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`.

The binary specifier is new in C23!

Conversion Specifier	Description
`d`	Print an `int` argument as a decimal number.
`i`	Identical to `d`.
`b`	Print an `unsigned int` in binary (base 2).
`B`	Identical to `b`, except for the alternate form, below.
`o`	Print an `unsigned int` in octal (base 8).
`u`	Print an `unsigned int` in decimal.

Conversion Specifier	Description
`x`	Print an `unsigned int` in hexadecimal with lowercase letters.
`X`	Print an `unsigned int` in hexadecimal with uppercase letters; also note the alternate form, below.
`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.

23.10.0.2.1 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.

23.10.0.2.2 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.

23.10.0.2.3 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
```

23.10.0.3 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`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Convert argument to `char` (signed or unsigned as appropriate) before printing.
`h`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Convert argument to `short int` (signed or unsigned as appropriate) before printing.
`l`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Argument is a `long int` (signed or unsigned as appropriate).
`ll`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Argument is a `long long int` (signed or unsigned as appropriate).
`j`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Argument is a `intmax_t` or `uintmax_t` (as appropriate).
`z`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Argument is a `size_t`.
`t`	`b`, `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.

23.10.0.4 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
b, 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
```

23.10.0.5 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);
```

23.10.0.6 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.
`b`	Prefix a non-zero number with `0b`.
`B`	Same as `b`, except capital `0B`.
`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.

23.10.0.7 `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

```
#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()`

23.11 `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`.
`%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.

23.11.0.1 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.

23.11.0.2 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.

23.11.0.3 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 *slooooooooowwwww*…

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.)

23.11.0.4 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

23.11.0.5 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`.

The binary conversion is new in C23!

Conversion Specifier	Description
`d`	Matches a decimal `int`. Can have a leading sign.
`b`	Matches a binary (base 2) `unsigned int`. Can have a leading sign.
`i`	Like `d`, except will handle it if you put a leading `0x` (hex) or `0` (octal) or `0b` (binary) 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`.

[3] https://nee.lv/2021/02/28/How-I-cut-GTA-Online-loading-times-by-70/

Conversion Specifier	Description
`f`	Match a floating point number (or scientific notation, or anything `strtod()` can handle).
`c`	Match a `char`, or mutiple `char`s if a field width is given.
`s`	Match a sequence of non-whitespace `char`s.
`[`	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`.

23.11.0.5.1 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 `char`s). Conversion stops when a character that is not in the set is matched.

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!

23.11.0.6 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`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Convert input to `char` (signed or unsigned as appropriate) before printing.
`h`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Convert input to `short int` (signed or unsigned as appropriate) before printing.
`l`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Convert input to `long int` (signed or unsigned as appropriate).
`ll`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Convert input to `long long int` (signed or unsigned as appropriate).
`j`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Convert input to `intmax_t` or `uintmax_t` (as appropriate).
`z`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Convert input to `size_t`.
`t`	`b`, `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.

Length Modifier	Conversion Specifier	Description
`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.

23.11.0.7 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.)

23.11.0.8 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()`

23.12 `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.

Function	Output Destination
`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()`

23.13 `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;

    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()`

23.14 `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);
```

[4] https://stackoverflow.com/questions/17017331/c99-vscanf-for-dummies/17018046#17018046

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..

```c
#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

23.15 `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

```c
#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()`

23.16 `putc()`, `fputc()`, `putchar()`

Write a single character to the console or to a file

Synopsis

```c
#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:

```
#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

23.17 `puts()`, `fputs()`

Write a string to the console or to a file

Synopsis

```
#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:

```
#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

23.18 `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()`

23.19 `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 `int`s in a file.

But wait—can't you use `fprintf()` with the "`%d`" format specifier to save the `int`s 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 `int`s 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()`

23.20 `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()`

23.21 `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()`

23.22 `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

```c
#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()`

23.23 `ftell()`

Tells you where a particular file is about to read from or write to

Synopsis

```c
#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

```
#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()`

23.24 `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:

```c
#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()`

23.25 `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 24

`<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 functionality 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.

24.1 `<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!

24.2 `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.

[1] http://man.cat-v.org/unix-1st/3/atof

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()`

24.3 `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);
```

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.

[2] http://man.cat-v.org/unix-1st/3/atoi

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()`

24.4 `strtod()`, `strtof()`, `strtold()`

Convert a string to a floating point number

Synopsis

```
#include <stdlib.h>

double strtod(const char * restrict nptr, char ** restrict endptr);

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

```
#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()`

24.5 `strtol()`, `strtoll()`, `strtoul()`, `strtoull()`

Convert a string to an integer

Synopsis

```
#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:

The binary conversion is new in C23!

- If the number has a leading `0b` or `0B`, it will be binary (base 2)
- 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

```c
#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
    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);
```

[3] https://en.wikipedia.org/wiki/Radix

```

    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()`

24.6 `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.

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.

[4] https://stackoverflow.com/questions/10984974/why-do-people-say-there-is-modulo-bias-when-using-a-random-number-generator

[5] https://mumble.net/~campbell/2014/04/28/uniform-random-float

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:

```c
#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());
}
```

[6] https://www.gnu.org/software/gsl/doc/html/rng.html

See Also

srand()

24.7 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?

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

```

[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.

[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.

```
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()`

24.8 `aligned_alloc()`

Allocate specifically-aligned memory

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.

[10] https://en.wikipedia.org/wiki/Data_structure_alignment

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);
}
```

See Also

`malloc()`, `calloc()`, `free()`

24.9 `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 `int`s:

```
#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()`

24.10 `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>

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()`

24.11 `realloc()`

Resize a previously allocated stretch of memory

[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

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

```
#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()`

24.12 `abort()`

Abruptly end program execution

Synopsis

```
#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.

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
```

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

See Also

signal()

24.13 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)
{
```

[13] quick_exit() differs from exit() in that open files might not be flushed and temporary files might not be removed.

```
    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)
{
    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()`

24.14 `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()`

24.15 `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.

Return Value

Returns a pointer to the environment variable value, or `NULL` if the variable doesn't exist.

Example

```
#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 [...]
```

24.16 `system()`

Run an external program

Synopsis

```
#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!
```

24.17 bsearch()

Binary Search (maybe) an array of objects

Synopsis

```
#include <stdlib.h>

// Pre-C23:

void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *, const void *));
// C23:

QVoid *bsearch(const void *key, QVoid *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.

New in C23: If `base` is `const`, the return type of the `bsearch()` function will also be `const`.

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()`

24.18 `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()`.

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]);

```

[14] "In-place" meaning that the original array will hold the results; no new array is allocated.

```
    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()`

24.19 `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()`

24.20 `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

```c
#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()`

24.21 `mblen()`

Return the number of bytes in a multibyte character

Synopsis

```c
#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()`

24.22 `mbtowc()`

Convert a multibyte character to a wide character

Synopsis

```
#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()`

24.23 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()`

24.24 `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.

```
#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()`

24.25 `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()`

24.26 memalignment()

Synopsis

New in C23!

```
#include <stdlib.h>

size_t memalignment(const void * p);
```

Description

This function will give you the memory alignment (in bytes) of the object pointed to.

The idea is that we're going to get the same information as we get with `alignof`, except we can do it with an untyped `void*`.

And you might want to do this because misaligned data could be slower or unusable on various platforms.

Return Value

Returns the integer alignment of `p`, which will be a power of 2. Returns `0` if `NULL` is passed in.

`0` means the pointer can't be used to access an object of any type.

Example

(Caveat: none of my compilers support this function yet, so the code is largely untested.)

The proposal for this feature suggests that a macro might be useful for determining if a pointer is well-aligned for a particular type, so we'll just steal that as an example:

```
#include <stdio.h>
#include <stdalign.h>
#include <stdlib.h>

#define isaligned(P, T) (memalignment (P) >= alignof(T))

void check(void *p)
{
    printf("%lu %lu\n", memalignment(p), alignof(int));
    if (isaligned(p, int)) {
        puts("The pointer p is well-aligned for an int! :)");

        // So I'm happy doing this:

        int *i = p;
        *i = 3490;

    } else
        puts("The pointer p is poorly-aligned for an int! :(");
}

int main(void)
{
    char *p = malloc(10);

    check(p);
    check(p + 1);
```

```
}
```

See Also

`alignof()`, `alignas()`, `max_align_t`

Chapter 25

`<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:

```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();
}
```

That's all there is to it.

Chapter 26

`<string.h>` String Manipulation

Function	Description
`memccpy()`	Copy a region of memory to another, stopping at a specified character.
`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.
`memset_explicit()`	Set a region of memory to a value in a manner that cannot be optimized away.
`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.
`strdup()`	Duplicate a string on the heap.
`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.
`strndup()`	Duplicate a string on the heap, 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.

26.1 `memcpy()`, `memccpy()`, `memmove()`

Copy bytes of memory from one location to another

Synopsis

`memccpy()` is new in C23!

```
#include <string.h>

void *memcpy(void * restrict s1, const void * restrict s2, size_t n);

void *memccpy(void * restrict s1, const void * restrict s2, int c,
              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 for as many bytes as you specify!

`memccpy()` gives you the additional option of stopping after a certain character is encountered in the source.

The main difference between the `memcpy()` variants and `memmove()` is that the former cannot safely copy overlapping memory regions, whereas the latter can.

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()`

26.2 `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()`

26.3 `strdup()`, `strndup()`

Duplicate a string on the heap

Synopsis

New in C23!

```
#include <string.h>

char *strdup(const char *s);

char *strndup(const char *s, size_t size);
```

Description

This is like `strcpy()`, except it allocates space for the new string, just as if it had been done with `malloc()`.

The pointer to the duplicate string it returns should be freed up with a call to `free()` once you're done with it.

`strndup()` works the same way, except you can limit the number of bytes duplicated. `strndup()` always produces a NUL-terminated string.

Return Value

Both functions return a pointer to the newly created string, or `NULL` if the memory for the duplicate cannot be allocated.

Example

Here's an example that duplicates a string and then capitalizes the first letter of the duplicate.

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

char *dup_cap(const char *s)
{
    char *d = strdup(s);

    if (d != NULL)
        d[0] = toupper(d[0]);

    return d;
}

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

    if (s != NULL)
        puts(s);  // Hello, world!
    else
        puts("Error duplicating string");

    free(s);
}
```

See Also

`strcpy()`, `strncpy()`, `malloc()`, `free()`

26.4 `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()`

26.5 `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

```c
#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()`

26.6 `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()`

26.7 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:

```
int my_strcoll(char *s1, char *s2)
{
    // Use n = 0 to just get the lengths of the transformed strings
    int len1 = strxfrm(NULL, s1, 0) + 1;
    int len2 = strxfrm(NULL, s2, 0) + 1;

    // Allocate enough room for each
    char *d1 = malloc(len1);
    char *d2 = malloc(len2);

    // Transform the strings for comparison
    strxfrm(d1, s1, len1);
    strxfrm(d2, s2, len2);

    // Compare the transformed strings
    int result = strcmp(d1, d2);

    // Free up the transformed strings
    free(d2);
    free(d1);

    return result;
}
```

[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.

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?

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()`

26.8 `strchr()`, `strrchr()`, `memchr()`

Find a character in a string

Synopsis

```
// Pre-C23:

#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);


// C23:

QChar *strchr(QChar *s, int c);

QChar *strrchr(QChar *s, int c);

QVoid *memchr(QVoid *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
```

26.9 `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()`

26.10 `strpbrk()`

Search a string for one of a set of characters

Synopsis

```
#include <string.h>

// Pre-C23:

char *strpbrk(const char *s1, const char *s2);

// C23:

QChar *strpbrk(QChar *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()`

26.11 `strstr()`

Find a string in another string

Synopsis

```
#include <string.h>

// Pre-C23:

char *strstr(const char *str, const char *substr);

// C23:

QChar *strstr(QChar *s1, const char *s2);
```

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()`

26.12 `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

```c
#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()`

26.13 `memset()`, ‘memset_explicit

Set a region of memory to a certain value

Synopsis

`memset_explicit()` is new in C23!

```c
#include <string.h>

void *memset(void *s, int c, size_t n);

void *memset_explicit(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`.

`memset_explicit()` only differs in that it will never be optimized away (like `memset()` might be). The idea is that you could use it to most-definitely remove sensitive information (like passwords) from memory before any nasty hackers get their hands on it.

Return Value

`memset()` and `memset_explicit()` return whatever you passed in as `s` for happy convenience.

Example

```c
#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()`

26.14 `strerror()`

Get a string version of an error number

Synopsis

```c
#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()`

26.15 `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 27

`<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.

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`
`copysign`	—	`copysign`
`fmax`	—	`fmax`
`erf`	—	`erf`
`fmin`	—	`fmin`
`erfc`	—	`erfc`
`fmod`	—	`fmod`

Real Function	Complex Function	Generic Macro
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

27.1 Example

Here's an example where we call the type-generic `sqrt()` function on a variety of types.

```c
#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 28

`<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!

Enjoy!

28.1 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!
```

28.2 `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.

Return Value

Returns `thrd_success` or `thrd_error` depending on how well things went.

[1] Which you should because of spurious wakeups.

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.

```
#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()`

28.3 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()`

28.4 `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()`

28.5 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);

    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()`

28.6 `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.

```c
#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()`

28.7 `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.

```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);       // <-- 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()`

28.8 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()`

28.9 `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()`

28.10 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:

```
#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()`

28.11 `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.

```c
#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()`

28.12 `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()`

28.13 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()`

28.14 `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()`.

[2] Well, as at least as many things as you have free cores. Your OS will schedule them as it can.

Return Value

In the case of goodness, returns `thrd_success`. If you're out of memory, will return `thrd_nomem`. Otherwise, `thrd_error`.

Example

```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()`

28.15 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()`

28.16 `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()`

28.17 `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()`

28.18 `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];

    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;
```

[3]For example, if a destructor caused more variables to be set.

```
        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()`

28.19 `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()`

28.20 `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
```

[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!

```
    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()`

28.21 `thrd_yield()`

Stop running that other threads might run

Synopsis

```
#include <threads.h>

void thrd_yield(void);
```

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()`

28.22 `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()`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

```c
#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()`

28.23 `tss_delete()`

Clean up a thread-specific storage variable

Synopsis

```c
#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()`.

```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); 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()`

28.24 `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()`

28.25 `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

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 29

`<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
};
```

You can convert between the two with `mktime()`, `gmtime()`, and `localtime()`.

You can print time information to strings with `ctime()`, `asctime()`, and `strftime()`.

[1] When you say GMT, unless you're talking specifically about the time zone and not the time, you probably mean "UTC".

29.1 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.

29.2 `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

```
#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);
```

[2]The spec doesn't actually say "clock ticks", but I… am.

```
}

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()`

29.3 `difftime()`

Compute the difference between two times

Synopsis

```
#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.

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,
    };

```

[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.

```
    // 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()`

29.4 `mktime()`

Convert a `struct tm` with local time 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 don't have a C23 compiler and you need input in UTC, see the non-standard functions `timegm()`[4] for Unix-likes and `_mkgmtime()`[5] for Windows.

[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

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.

```
#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

UTC        : Mon Apr 12 12:00:04 1982
```

See Also

`timegm()`, `localtime()`, `gmtime()`

29.5 `timegm()`

Convert a `struct tm` with UTC time into a `time_t`

Synopsis

New in C23!

```
#include <time.h>

time_t timegm(struct tm *timeptr);
```

Description

If you have a UTC 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.
- The spec doesn't say anything about the `tm_isdst` Daylight Saving flag, but since UTC is immune to DST, I'm assuming it is ignored or set to `0` for us.

If you don't have a C23 compiler and you need input in UTC, see the non-standard functions `timegm()`[6] for Unix-likes and `_mkgmtime()`[7] for Windows.

Return Value

Returns the UTC time in the `struct tm` as a `time_t` calendar time.

Returns `(time_t)(-1)` on error.

Example

```
#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 = timegm(&broken_down_time);

    char *days[] = {"Sunday", "Monday", "Tuesday",
        "Wednesday", "Furzeday", "Friday", "Saturday"};

    // This will print what was in broken_down_time
    printf("UTC        : %s", asctime(gmtime(&calendar_time)));
    printf("Day of week: %s\n\n", days[broken_down_time.tm_wday]);
```

[6] https://man.archlinux.org/man/timegm.3.en
[7] https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/mkgmtime-mkgmtime32-mkgmtime64?view=msvc-160

```

    // This will print UTC for the local time, above
    printf("Local time : %s", asctime(localtime(&calendar_time)));
}
```

See Also

`mktime()`, `timegm()`, `localtime()`, `gmtime()`

29.6 `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()`

29.7 `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[8], 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:

[8] https://en.wikipedia.org/wiki/Unix_time

```
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.
// 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()`

29.8 `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()`

29.9 `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()`

29.10 `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()`

29.11 `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()`

29.12 `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`]

Specifier	Description
`%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.
`%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[9].

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`.

[9] https://en.wikipedia.org/wiki/ISO_week_date

Specifier	Description
`%x`	Equivalent to `%m/%d/%y`.
`%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()`

Chapter 30

<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.

30.1 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.

30.2 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.

30.3 `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_t`s (or `char32_t`s 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.
`(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);

    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_t`s, requiring two calls to get them both.

[1] https://en.wikipedia.org/wiki/Symbols_for_Legacy_Computing

```c
#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()`

30.4 `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_t`s), 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 31

`<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
`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

Function	Description
`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.

31.1 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.

31.2 `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()`

31.3 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()`

31.4 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()`

31.5 `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.

[1] https://stackoverflow.com/questions/17017331/c99-vscanf-for-dummies/17018046#17018046

```c
#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()`

31.6 `getwc()` `fgetwc()` `getwchar()`

Get a wide character from an input stream

Synopsis

```c
#include <stdio.h>   // For getwc() and fgetwc()
#include <wchar.h>

wint_t getwchar(void);

wint_t getwc(FILE *stream);

wint_t fgetwc(FILE *stream);
```

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`

31.7 `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()`

31.8 `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`

31.9 `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()`

31.10 `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
```

31.11 ungetwc()

Pushes a wide character back into the input stream

Synopsis

```
#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.

```
#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()`

31.12 `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

```c
#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`

31.13 `wcstol()` `wcstoll()` `wcstoul()` `wcstoull()`

Convert a wide string to an integer value

Synopsis

```c
#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

```c
#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()`

31.14 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()`

31.15 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()`

31.16 `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()`

31.17 `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

```
#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()`

31.18 `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()`

31.19 `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

```
#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()`

31.20 `wcschr()` `wcsrchr()`

Find a wide character in a wide string

Synopsis

```
#include <wchar.h>

// Pre-C23:

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);

// C23:

QWchar_t *wcschr(QWchar_t *s, wchar_t c);

QWchar_t *wcsrchr(QWchar_t *s, wchar_t c);

QWchar_t *wmemchr(QWchar_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()`

31.21 `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()`

31.22 `wcspbrk()`

Search a wide string for one of a set of wide characters

Synopsis

```
#include <wchar.h>

// Pre-C23:

wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);

// C23:

QWchar_t *wcspbrk(QWchar_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()`

31.23 `wcsstr()`

Find a wide string in another wide string

Synopsis

```
#include <wchar.h>

// Pre-C23:
```

```
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);

// C23:

QWchar_t *wcsstr(QWchar_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()`

31.24 `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()`

31.25 `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()`

31.26 `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()`

31.27 `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()`

31.28 `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()`

31.29 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()`

31.30 `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.

```c
#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()`

31.31 `wcrtomb()`

Convert wide to multibyte characters restartably

Synopsis

```c
#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.

[2]This is a variable, not a macro, so if you use it to define an array, it'll be a variable-length array.

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.

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`

31.32 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`

31.33 `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.)

```c
#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 32

`<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").

32.1 `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()`

32.2 `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()`

32.3 `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()`

32.4 `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()`

32.5 `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()`

32.6 `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()`

32.7 `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()`

32.8 `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()`

32.9 `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()`

32.10 `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()`

32.11 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()`

32.12 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()`

32.13 `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:

```c
#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()`

32.14 `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()`

32.15 `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()`

32.16 `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()`

32.17 `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

```c
#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()`

32.18 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()`

Index

_Alignas() alignment specifier, *165*
_Alignof() operator, *167*
_Atomic type qualifier, *176*
_Atomic() type specifier, *176*
_Complex_I macro, *25*
_Exit() function, *279*
_Imaginary_I macro, *25*
__STDC_ENDIAN_BIG__ macro, 195–196
__STDC_ENDIAN_LITTLE__ macro, 195–196
__STDC_ENDIAN_NATIVE__ macro, 195–196
__STDC_NO_COMPLEX__, *24*
__STDC_VERSION_STDBIT_H__ macro, 195
__alignas_is_defined macro, *165*
__alignof_is_defined macro, *165*

abort() function, *277*
abs() function, *286*
acos() function, *103*, *317*
acosf() function, *103*
acosh() function, *109*, *317*
acoshf() function, *109*
acoshl() function, *109*
acosl() function, *103*
Addition operator, *see* + addition operator
alignas() alignment specifier, *165*
aligned_alloc() function, *272*
alignof() operator, *167*
and macro, *87*
and_eq macro, *87*
asctime() function, *371*
asin() function, *104*, *317*
asinf() function, *104*
asinh() function, *110*, *317*
asinhf() function, *110*
asinhl() function, *110*
asinl() function, *104*
assert() macro, *20*
assert.h header file, *20*
at_quick_exit() function, *278*
atan() function, *105*, *317*
atan2() function, *105*, *317*
atan2f() function, *105*
atan2l() function, *105*
atanf() function, *105*
atanh() function, *111*, *317*
atanhf() function, *111*
atanhl() function, *111*
atanl() function, *105*
atexit() function, *278*
atof() function, *263*
atoi() function, *264*
atol() function, *264*
atoll() function, *264*
atomic_bool type, *176*
ATOMIC_BOOL_LOCK_FREE macro, *176*
atomic_char type, *176*
atomic_char16_t type, *176*
ATOMIC_CHAR16_T_LOCK_FREE macro, *176*
atomic_char32_t type, *176*
ATOMIC_CHAR32_T_LOCK_FREE macro, *176*
ATOMIC_CHAR_LOCK_FREE macro, *176*
atomic_compare_exchange_*() function, *187*
atomic_exchange() function, *186*
atomic_fetch_*() function, *189*
atomic_flag type, *177*
atomic_flag_clear() function, *192*
atomic_flag_test_and_set() function, *190*
atomic_init() function, *178*
atomic_int type, *176*
atomic_int_fast16_t type, *176*
atomic_int_fast32_t type, *176*
atomic_int_fast64_t type, *176*
atomic_int_fast8_t type, *176*
atomic_int_least16_t type, *176*
atomic_int_least32_t type, *176*
atomic_int_least64_t type, *176*
atomic_int_least8_t type, *176*
ATOMIC_INT_LOCK_FREE macro, *176*
atomic_intmax_t type, *176*
atomic_intptr_t type, *176*
atomic_is_lock_free() function, *183*
atomic_llong type, *176*
ATOMIC_LLONG_LOCK_FREE macro, *176*
atomic_load() function, *185*
atomic_long type, *176*
ATOMIC_LONG_LOCK_FREE macro, *176*
ATOMIC_POINTER_LOCK_FREE macro, *176*
atomic_ptrdiff_t type, *176*
atomic_schar type, *176*
atomic_short type, *176*
ATOMIC_SHORT_LOCK_FREE macro, *176*
atomic_signal_fence() function, *182*
atomic_size_t type, *176*
atomic_store() function, *184*
atomic_thread_fence() function, *180*
atomic_uchar type, *176*
atomic_uint type, *176*
atomic_uint_fast16_t type, *176*

`atomic_uint_fast32_t` type, *176*
`atomic_uint_fast64_t` type, *176*
`atomic_uint_fast8_t` type, *176*
`atomic_uint_least16_t` type, *176*
`atomic_uint_least32_t` type, *176*
`atomic_uint_least64_t` type, *176*
`atomic_uint_least8_t` type, *176*
`atomic_uintmax_t` type, *176*
`atomic_uintptr_t` type, *176*
`atomic_ullong` type, *176*
`atomic_ulong` type, *176*
`atomic_ushort` type, *176*
`ATOMIC_VAR_INIT()` macro, *178*
`atomic_wchar_t` type, *176*
`ATOMIC_WCHAR_T_LOCK_FREE` macro, *176*

Bell, *see* `\a` operator
`bitand` macro, *87*
`bitor` macro, *87*
`bool` macro, *211*
Boolean AND, *see* `&&` operator
Boolean NOT, *see* `!` operator
Boolean OR, *see* `||` operator
`bsearch()` function, *283*
`btowc()` function, *414*

`c16rtomb()` function, *381*
`c32rtomb()` function, *381*
`cabs()` function, *38*
`cabsf()` function, *38*
`cabsl()` function, *38*
`cacos()` function, *26*
`cacosf()` function, *26*
`cacosh()` function, *31*
`cacoshf()` function, *31*
`cacoshl()` function, *31*
`cacosl()` function, *26*
`call_once()` function, *321*
`calloc()` function, *273*
`carg()` function, *41*, *317*
`cargf()` function, *41*
`cargl()` function, *41*
Carriage return, *see* `\r` operator
`casin()` function, *27*
`casinf()` function, *27*
`casinh()` function, *32*
`casinhf()` function, *32*
`casinhl()` function, *32*
`casinl()` function, *27*
`catan()` function, *28*
`catanf()` function, *28*
`catanh()` function, *33*
`catanhf()` function, *33*
`catanhl()` function, *33*
`catanl()` function, *28*
`cbrt()` function, *126*, *317*
`cbrtf()` function, *126*
`cbrtl()` function, *126*
`ccos()` function, *28*
`ccosf()` function, *28*
`ccosh()` function, *34*
`ccoshf()` function, *34*
`ccoshl()` function, *34*
`ccosl()` function, *28*
`ceil()` function, *134*, *317*
`ceilf()` function, *134*
`ceill()` function, *134*
`cexp()` function, *36*
`cexpf()` function, *36*
`cexpl()` function, *36*
`char16_t` type, *378*
`char32_t` type, *378*
`CHAR_BIT` macro, *88*
`CHAR_MAX` macro, *88*
`CHAR_MIN` macro, *88*
`cimag()` function, *42*, *317*
`cimagf()` function, *42*
`cimagl()` function, *42*
`clearerr()` function, *258*
`clock()` function, *363*
`clog()` function, *37*
`clogf()` function, *37*
`clogl()` function, *37*
`CMPLX()` macro, *43*
`CMPLXF()` macro, *43*
`CMPLXL()` macro, *43*
`cnd_broadcast()` function, *322*
`cnd_destroy()` function, *325*
`cnd_init()` function, *326*
`cnd_signal()` function, *328*
`cnd_timedwait()` function, *329*
`cnd_wait()` function, *331*
`compl` macro, *87*
`complex.h` header file, *24*
`conj()` function, *44*, *317*
`conjf()` function, *44*
`conjl()` function, *44*
`copysign()` function, *144*, *317*
`copysignf()` function, *144*
`copysignl()` function, *144*
`cos()` function, *107*, *317*
`cosf()` function, *107*
`cosh()` function, *112*, *317*
`coshf()` function, *112*
`coshl()` function, *112*
`cosl()` function, *107*
`cpow()` function, *39*
`cpowf()` function, *39*
`cpowl()` function, *39*
`cproj()` function, *45*, *317*
`cprojf()` function, *45*
`cprojl()` function, *45*
`creal()` function, *46*, *317*
`crealf()` function, *46*
`creall()` function, *46*
`csin()` function, *29*
`csinf()` function, *29*

`csinh()` function, *35*
`csinhf()` function, *35*
`csinhl()` function, *35*
`csinl()` function, *29*
`csqrt()` function, *40*
`csqrtf()` function, *40*
`csqrtl()` function, *40*
`ctan()` function, *30*
`ctanf()` function, *30*
`ctanh()` function, *36*
`ctanhf()` function, *36*
`ctanhl()` function, *36*
`ctanl()` function, *30*
`ctime()` function, *372*
`ctype.h` header file, *47*
`CX_LIMITED_RANGE` macro, *25*

`DBL_DECIMAL_DIG` macro, *77*
`DBL_DIG` macro, *76*
`DBL_EPSILON` macro, *75*
`DBL_HAS_SUBNORM` macro, *76*
`DBL_MANT_DIG` macro, *74*
`DBL_MAX` macro, *74*
`DBL_MAX_10_EXP` macro, *74*
`DBL_MAX_EXP` macro, *74*
`DBL_MIN` macro, *75*
`DBL_MIN_10_EXP` macro, *74*
`DBL_MIN_EXP` macro, *74*
`DBL_TRUE_MIN` macro, *75*
`DECIMAL_DIG` macro, *74*
`difftime()` function, *364*
`div()` function, *287*
`div_t` type, *263*
Division operator, *see* `/` division operator
`double complex` type, *25*
`double imaginary` type, *25*
`double_t` type, *99*

Endianess, 195–196
`erf()` function, *130*, *317*
`erfc()` function, *131*, *317*
`erfcf()` function, *131*
`erfcl()` function, *131*
`erff()` function, *130*
`erfl()` function, *130*
`errno` variable, *60*
`errno.h` header file, *60*
`exit()` function, *279*
`EXIT_FAILURE` macro, *263*
`EXIT_SUCCESS` macro, *263*
`exp()` function, *114*, *317*
`exp2()` function, *115*, *317*
`exp2f()` function, *115*
`exp2l()` function, *115*
`expf()` function, *114*
`expl()` function, *114*
`expm1()` function, *115*, *317*
`expm1f()` function, *115*
`expm1l()` function, *115*

`fabs()` function, *127*, *317*
`fabsf()` function, *127*
`fabsl()` function, *127*
`false` macro, *211*
`fclose()` function, *225*
`fdim()` function, *148*, *317*
`fdimf()` function, *148*
`fdiml()` function, *148*
`FE_ALL_EXCEPT` macro, *63*
`FE_DIVBYZERO` macro, *63*
`FE_INEXACT` macro, *63*
`FE_INVALID` macro, *63*
`FE_OVERFLOW` macro, *63*
`FE_UNDERFLOW` macro, *63*
`feclearexcept()` function, *64*
`fegetenv()` function, *69*
`fegetexceptflag()` function, *65*
`fegetround()` function, *68*
`feholdexcept()` function, *71*
`fenv.h` header file, *63*
`FENV_ACCESS` pragma, *64*
`fenv_t` type, *63*
`feof()` function, *258*
`feraiseexcept()` function, *66*
`ferror()` function, *258*
`fesetenv()` function, *69*
`fesetexceptflag()` function, *65*
`fesetround()` function, *68*
`fetestexcept()` function, *67*
`feupdateenv()` function, *72*
`fexcept_t` type, *63*
`fflush()` function, *226*
`fgetc()` function, *246*
`fgetpos()` function, *254*
`fgets()` function, *247*
`fgetwc()` function, *391*
`fgetws()` function, *392*
`FILE*` type, *220*
`float complex` type, *25*
`float imaginary` type, *25*
`float.h` header file, *74*
`float_t` type, *99*
`floor()` function, *135*, *317*
`floorf()` function, *135*
`floorl()` function, *135*
`FLT_DECIMAL_DIG` macro, *77*
`FLT_DIG` macro, *76*
`FLT_EPSILON` macro, *75*
`FLT_EVAL_METHOD` macro, *74*, *76*, *99*
`FLT_HAS_SUBNORM` macro, *76*
`FLT_MANT_DIG` macro, *74*
`FLT_MAX` macro, *74*
`FLT_MAX_10_EXP` macro, *74*
`FLT_MAX_EXP` macro, *74*
`FLT_MIN` macro, *75*
`FLT_MIN_10_EXP` macro, *74*
`FLT_MIN_EXP` macro, *74*
`FLT_RADIX` macro, *74*

FLT_ROUNDS macro, *76*
FLT_TRUE_MIN macro, *75*
fma() function, *150*, *317*
fmaf() function, *150*
fmal() function, *150*
fmax() function, *149*, *317*
fmaxf() function, *149*
fmaxl() function, *149*
fmin() function, *149*, *317*
fminf() function, *149*
fminl() function, *149*
fmod() function, *141*, *317*
fmodf() function, *141*
fmodl() function, *141*
fopen() function, *227*
FP_CONTRACT pragma, *100*
fpclassify() function, *100*
fprintf() function, *231*
fputc() function, *249*
fputwc() function, *393*
fputws() function, *394*
fread() function, *252*
free() function, *274*
freopen() function, *229*
frexp() function, *116*, *317*
frexpf() function, *116*
frexpl() function, *116*
fscanf() function, *238*
fseek() function, *256*
fsetpos() function, *254*
ftell() function, *257*
fwide() function, *395*
fwprintf() function, *386*
fwrite() function, *253*
fwscanf() function, *388*

getc() function, *246*
getchar() function, *246*
getenv() function, *281*
gets() function, *247*
getwc() function, *391*
getwchar() function, *391*
gmtime() function, *373*

Hexadecimal, *see* 0x hexadecimal
hypot() function, *128*, *317*
hypotf() function, *128*
hypotl() function, *128*

I macro, *25*
ilogb() function, *117*, *317*
ilogbf() function, *117*
ilogbl() function, *117*
imaxabs() function, *82*
imaxdiv() function, *83*
INT_FASTn_MAX macros, *217*
INT_FASTn_MIN macros, *217*
int_fastN_t types, *216*
INT_LEASTn_MAX macros, *217*
INT_LEASTn_MIN macros, *217*
int_leastN_t types, *216*
INT_MAX macro, *88*
INT_MIN macro, *88*
INTMAX_C() macro, *218*
INTMAX_MAX macros, *217*
INTMAX_MIN macros, *217*
intmax_t type, *217*
INTn_C() macros, *218*
INTn_MAX macros, *217*
INTn_MIN macros, *217*
intN_t types, *216*
INTPTR_MAX macros, *217*
INTPTR_MIN macros, *217*
intptr_t type, *217*
inttypes.h header file, *81*
isalnum() function, *48*
isalpha() function, *48*
isblank() function, *49*
iscntrl() function, *50*
isdigit() function, *51*
isfinite() function, *102*
isgraph() function, *52*
isgreater() function, *150*
isgreaterequal() function, *150*
isinf() function, *102*
isless() function, *150*
islessequal() function, *150*
islessgreater() function, *151*
islower() function, *52*
isnan() function, *102*
isnormal() function, *102*
iso646.h header file, *87*
isprint() function, *53*
ispunct() function, *54*
isspace() function, *55*
isunordered() function, *152*
isupper() function, *56*
iswalnum() function, *425*
iswalpha() function, *426*
iswblank() function, *427*
iswcntrl() function, *428*
iswctype() function, *435*
iswdigit() function, *429*
iswgraph() function, *429*
iswlower() function, *430*
iswprint() function, *431*
iswpunct() function, *432*
iswspace() function, *433*
iswupper() function, *434*
iswxdigit() function, *434*
isxdigit() function, *57*

kill_dependency() function, *179*

labs() function, *286*
LDBL_DECIMAL_DIG macro, *77*
LDBL_DIG macro, *76*
LDBL_EPSILON macro, *75*

LDBL_HAS_SUBNORM macro, *76*
LDBL_MANT_DIG macro, *74*
LDBL_MAX macro, *74*
LDBL_MAX_10_EXP macro, *74*
LDBL_MAX_EXP macro, *74*
LDBL_MIN macro, *75*
LDBL_MIN_10_EXP macro, *74*
LDBL_MIN_EXP macro, *74*
LDBL_TRUE_MIN macro, *75*
ldexp() function, *118*, *317*
ldexpf() function, *118*
ldexpl() function, *118*
ldiv() function, *287*
ldiv_t type, *263*
lgamma() function, *132*, *317*
lgammaf() function, *132*
lgammal() function, *132*
limits.h header file, *88*
llabs() function, *286*
lldiv() function, *287*
lldiv_t type, *263*
LLONG_MAX macro, *88*
LLONG_MIN macro, *88*
llrint() function, *137*, *317*
llrintf() function, *137*
llrintl() function, *137*
llround() function, *139*, *317*
llroundf() function, *139*
llroundl() function, *139*
locale.h. header file, *91*
localeconv() function, *93*
localtime() function, *374*
log() function, *119*, *317*
log10() function, *120*, *317*
log10f() function, *120*
log10l() function, *120*
log1p() function, *121*, *317*
log1pf() function, *121*
log1pl() function, *121*
log2() function, *122*, *317*
log2f() function, *122*
log2l() function, *122*
logb() function, *123*, *317*
logbf() function, *123*
logbl() function, *123*
logf() function, *119*
logl() function, *119*
long double complex type, *25*
long double imaginary type, *25*
LONG_MAX macro, *88*
LONG_MIN macro, *88*
longjmp() function, *156*
lrint() function, *137*, *317*
lrintf() function, *137*
lrintl() function, *137*
lround() function, *139*, *317*
lroundf() function, *139*
lroundl() function, *139*
malloc() function, *273*
math.h header file, *97*
MATH_ERREXCEPT macro, *99*
math_errhandling variable, *99*
MATH_ERRNO macro, *99*
max_align_t type, *214*
MB_CUR_MAX macro, *263*
MB_LEN_MAX macro, *88*
mblen() function, *288*
mbrlen() function, *417*
mbrtoc16() function, *379*
mbrtoc32() function, *379*
mbrtowc() function, *418*
mbsinit() function, *415*
mbsrtowcs() function, *421*
mbstate_t type, *378*, *386*
mbstowcs() function, *292*
mbtowc() function, *289*
memalignment() function, *295*
memccpy() function, *299*
memchr() function, *308*
memcmp() function, *303*
memcpy() function, *299*
memmove() function, *299*
memory_order_acq_rel enumerated type, *177*
memory_order_acquire enumerated type, *177*
memory_order_consume enumerated type, *177*
memory_order_relaxed enumerated type, *177*
memory_order_release enumerated type, *177*
memory_order_seq_cst enumerated type, *177*
memset() function, *313*
memset_explicit() function, *313*
mktime() function, *365*
modf() function, *124*
modff() function, *124*
modfl() function, *124*
Modulus operator, *see* % modulus operator
mtx_destroy() function, *333*
mtx_init() function, *334*
mtx_lock() function, *336*
mtx_timedlock() function, *337*
mtx_trylock() function, *339*
mtx_unlock() function, *341*
Multiplication operator, *see* * multiplication operator

NAN macro, *99*
nan() function, *145*
nanf() function, *145*
nanl() function, *145*
NDEBUG macro, *20*
nearbyint() function, *136*, *317*
nearbyintf() function, *136*
nearbyintl() function, *136*
New line, *see* \n newline
nextafter() function, *146*, *317*
nextafterf() function, *146*
nextafterl() function, *146*

nexttoward() function, *147*, *317*
nexttowardf() function, *147*
nexttowardl() function, *147*
noreturn macro, *297*
not macro, *87*
not_eq macro, *87*
NULL macro, *263*

offsetof operator, *215*
or macro, *87*
or_eq macro, *87*

perror() function, *259*
pow() function, *129*, *317*
powf() function, *129*
powl() function, *129*
PRIdFASTn macros, *82*
PRIdLEASTn macros, *82*
PRIdMAX macro, *82*
PRIdn macros, *82*
PRIdPTR macro, *82*
PRIiFASTn macros, *82*
PRIiLEASTn macros, *82*
PRIiMAX macro, *82*
PRIin macros, *82*
PRIiPTR macro, *82*
printf() function, *231*
PRIoFASTn macros, *82*
PRIoLEASTn macros, *82*
PRIoMAX macro, *82*
PRIon macros, *82*
PRIoPTR macro, *82*
PRIuFASTn macros, *82*
PRIuLEASTn macros, *82*
PRIuMAX macro, *82*
PRIun macros, *82*
PRIuPTR macro, *82*
PRIXFASTn macros, *82*
PRIxFASTn macros, *82*
PRIXLEASTn macros, *82*
PRIxLEASTn macros, *82*
PRIXMAX macro, *82*
PRIxMAX macro, *82*
PRIXn macros, *82*
PRIxn macros, *82*
PRIXPTR macro, *82*
PRIxPTR macro, *82*
PTRDIFF_MAX macro, *218*
PTRDIFF_MIN macro, *218*
ptrdiff_t type, *213*
putc() function, *249*
putchar() function, *249*
puts() function, *250*
putwc() function, *393*
putwchar() function, *393*

qsort() function, *284*
quick_exit() function, *279*

raise() function, *162*
rand() function, *269*
RAND_MAX macro, *263*
realloc() function, *275*
remainder() function, *142*, *317*
remainderf() function, *142*
remainderl() function, *142*
remove() function, *221*
remquo() function, *143*, *317*
remquof() function, *143*
remquol() function, *143*
rename() function, *222*
rewind() function, *256*
rint() function, *137*, *317*
rintf() function, *137*
rintl() function, *137*
round() function, *138*, *317*
roundf() function, *138*
roundl() function, *138*

scalbln() function, *125*, *317*
scalblnf() function, *125*
scalblnl() function, *125*
scalbn() function, *125*, *317*
scalbnf() function, *125*
scalbnl() function, *125*
scanf() function, *238*
SCHAR_MAX macro, *88*
SCHAR_MIN macro, *88*
SCNdFASTn macros, *82*
SCNdLEASTn macros, *82*
SCNdMAX macro, *82*
SCNdn macros, *82*
SCNdPTR macro, *82*
SCNiFASTn macros, *82*
SCNiLEASTn macros, *82*
SCNiMAX macro, *82*
SCNin macros, *82*
SCNiPTR macro, *82*
SCNoFASTn macros, *82*
SCNoLEASTn macros, *82*
SCNoMAX macro, *82*
SCNon macros, *82*
SCNoPTR macro, *82*
SCNuFASTn macros, *82*
SCNuLEASTn macros, *82*
SCNuMAX macro, *82*
SCNun macros, *82*
SCNuPTR macro, *82*
SCNxFASTn macros, *82*
SCNxLEASTn macros, *82*
SCNxMAX macro, *82*
SCNxn macros, *82*
SCNxPTR macro, *82*
setbuf() function, *230*
setjmp() function, *154*
setjmp.h header file, *154*
setlocale() function, *91*

`SHRT_MAX` macro, *88*
`SHRT_MIN` macro, *88*
`SIG_ATOMIC_MAX` macro, *218*
`SIG_ATOMIC_MIN` macro, *218*
`signal()` function, *159*
`signal.h` header file, *159*
`signbit()` function, *103*
`sin()` function, *107*, *317*
`sinf()` function, *107*
`sinh()` function, *112*, *317*
`sinhf()` function, *112*
`sinhl()` function, *112*
`sinl()` function, *107*
`SIZE_MAX` macro, *218*
`size_t` type, *214*, *263*, *378*
`snprintf()` function, *231*
`sprintf()` function, *231*
`sqrt()` function, *129*, *317*
`sqrtf()` function, *129*
`sqrtl()` function, *129*
`srand()` function, *271*
`sscanf()` function, *238*
`static_assert()` macro, *22*
`stdalign.h` header file, *165*
`stdarg.h` header file, *169*
`stdatomic.h` header file, *175*
`stdbit.h` header file, *195*
`stdbool.h` header file, *211*
`stdc_bit_ceil()` function, *209*
`stdc_bit_floor()` function, *208*
`stdc_bit_width()` function, *207*
`stdc_count_ones()` function, *205*
`stdc_count_zeros()` function, *205*
`stdc_first_leading_one()` function, *202*
`stdc_first_leading_zero()` function, *201*
`stdc_first_trailing_one()` function, *204*
`stdc_first_trailing_zero()` function, *203*
`stdc_has_single_bit()` function, *206*
`stdc_leading_ones()` function, *198*
`stdc_leading_zeros()` function, *197*
`stdc_trailing_ones()` function, *200*
`stdc_trailing_zeros()` function, *199*
`stddef.h` header file, *213*
`stderr` standard error, *220*
`stdin` standard input, *220*
`stdint.h` header file, *216*
`stdio.h` header file, *219*
`stdlib.h` header file, *262*
`stdnoreturn.h` header file, *297*
`stdout` standard output, *220*
`strcat()` function, *302*
`strchr()` function, *308*
`strcmp()` function, *303*
`strcoll()` function, *305*
`strcpy()` function, *300*
`strcspn()` function, *309*
`strdup()` function, *301*
`strerror()` function, *314*
`strftime()` function, *374*
String, *see* `char *`
`string.h` header file, *298*
`strlen()` function, *315*
`strncat()` function, *302*
`strncmp()` function, *303*
`strncpy()` function, *300*
`strndup()` function, *301*
`strpbrk()` function, *310*
`strrchr()` function, *308*
`strspn()` function, *309*
`strstr()` function, *311*
`strtod()` function, *265*
`strtof()` function, *265*
`strtoimax()` function, *84*
`strtok()` function, *312*
`strtol()` function, *267*
`strtold()` function, *265*
`strtoll()` function, *267*
`strtoul()` function, *267*
`strtoull()` function, *267*
`strtoumax()` function, *84*
`struct tm` type, *362*
`strxfrm()` function, *306*
Subtraction operator, *see* `-` subtraction operator
`swprintf()` function, *386*
`swscanf()` function, *388*
`system()` function, *281*

Tab (is better), *see* `\t` operator
`tan()` function, *108*, *317*
`tanf()` function, *108*
`tanh()` function, *113*, *317*
`tanhf()` function, *113*
`tanhl()` function, *113*
`tanl()` function, *108*
Ternary operator, *see* `?:` ternary operator
`tgamma()` function, *133*, *317*
`tgammaf()` function, *133*
`tgammal()` function, *133*
`tgmath.h` header file, *317*
`thrd_create()` function, *342*
`thrd_current()` function, *344*
`thrd_detach()` function, *345*
`thrd_equal()` function, *346*
`thrd_exit()` function, *347*
`thrd_join()` function, *349*
`thrd_yield()` function, *351*
`threads.h` header file, *320*
`time()` function, *368*
`time.h` header file, *362*
`time_t` type, *362*
`timegm()` function, *366*
`timespec_get()` function, *369*
`tmpfile()` function, *222*
`tmpnam()` function, *223*
`tolower()` function, *57*
`toupper()` function, *58*

towctrans() function, *440*
towlower() function, *438*
towupper() function, *439*
true macro, *211*
trunc() function, *140*, *317*
truncf() function, *140*
truncl() function, *140*
tss_create() function, *353*
tss_delete() function, *355*
tss_get() function, *357*
tss_set() function, *359*

UCHAR_MAX macro, *88*
UINT_FASTn_MAX macros, *217*
UINT_LEASTn_MAX macros, *217*
UINT_MAX macro, *88*
UINTMAX_C() macro, *218*
UINTMAX_MAX macros, *217*
uintmax_t type, *217*
UINTn_MAX macros, *217*
UINTPTR_MAX macros, *217*
uintptr_t type, *217*
ULLONG_MAX macro, *88*
ULONG_MAX macro, *88*
ungetc() function, *251*
ungetwc() function, *397*
USHRT_MAX macro, *88*

va_arg() macro, *169*
va_copy() macro, *170*
va_end() macro, *172*
va_list type, *169*
va_start() macro, *173*
vfprintf() function, *243*
vfscanf() function, *245*
vfwprintf() function, *389*
vfwscanf() function, *390*
vprintf() function, *243*
vscanf() function, *245*
vsnprintf() function, *243*
vsprintf() function, *243*
vsscanf() function, *245*
vswprintf() function, *389*
vswscanf() function, *390*
vwprintf() function, *389*
vwscanf() function, *390*

wchar.h header file, *385*
WCHAR_MAX macro, *218*
WCHAR_MIN macro, *218*
wchar_t type, *215*, *263*
wcrtomb() function, *419*
wcscat() function, *402*
wcschr() function, *407*
wcscmp() function, *403*
wcscoll() function, *405*
wcscpy() function, *401*
wcscspn() function, *408*
wcsftime() function, *413*
wcslen() function, *412*
wcsncat() function, *402*
wcsncmp() function, *403*
wcsncpy() function, *401*
wcspbrk() function, *409*
wcsrchr() function, *407*
wcsrtombs() function, *422*
wcsspn() function, *408*
wcsstr() function, *410*
wcstod() function, *398*
wcstof() function, *398*
wcstoimax() function, *85*
wcstok() function, *411*
wcstol() function, *399*
wcstold() function, *398*
wcstoll() function, *399*
wcstombs() function, *293*
wcstoul() function, *399*
wcstoull() function, *399*
wcstoumax() function, *85*
wcsxfrm() function, *406*
wctob() function, *414*
wctomb() function, *291*
wctrans() function, *442*
wctype() function, *437*
wctype.h header file, *425*
WINT_MAX macro, *218*
WINT_MIN macro, *218*
wmemcmp() function, *403*
wmemcpy() function, *401*
wmemmove() function, *401*
wprintf() function, *386*
wscanf() function, *388*

xor macro, *87*
xor_eq macro, *87*