XII

CHAPTER

Standard Library Functions

Half the value of working with C comes from the standard library functions. Sure, it's nice to have that sexy `for` loop, and the similarity of arrays and pointers is convenient. When the rubber meets the road, though, what counts is how convenient it is to work with strings and files and such. Some programming languages do some parts of the task better; others do other parts better. When you have to do all of it, though, there's not much that does it better than C.

A lot is missing from the standard library. There are no functions for graphics, or even full-screen text manipulation. The `signal` mechanism (see FAQ XII.10) is pretty weak. There's absolutely no support for multitasking or for using anything but conventional memory. That's the point, though; the standard library provides functionality for all programs, whether they run in a multitasking, multiple-window environment, or on a dumb terminal, or in an expensive toaster. There are some de facto standards for the rest, and you can get some from your compiler vendor or a third-party library. What *is* in the standard library, though, is a very strong base to build on.

XII.1: Why should I use standard library functions instead of writing my own?

Answer:

The standard library functions have three advantages: they work, they're efficient, and they're portable.

They work: Your compiler vendor probably got them right. More important, the vendor is likely to have done a thorough test to prove they're right, more thorough than you probably have time for. (There are expensive test suites to make that job easier.)

They're efficient: Good C programmers use the standard library functions a lot, and good compiler vendors know that. There's a competitive advantage for the vendor to provide a good implementation. When competing compilers are compared for efficiency, a good compiler implementation can make all the difference. The vendor has more motivation than you do, and probably more time, to produce a fast implementation.

They're portable: In a world where software requirements change hourly, the standard library functions do the same thing, and mean the same thing, for every compiler, on every computer. They're one of the few things you, as a C programmer, can count on.

The funny thing is, one of the most standard pieces of information about the standard library is hard to find. For every function, there's one header file (or, rarely, two) that guarantees to give you that function's prototype. (You should always include the prototype for every function you call; see FAQ VIII.2.) What's funny? That header file might not be the file that actually contains the prototype. In some (sad!) cases, it's not even the header file recommended by the compiler manual. The same is true for macros, `typedef`s, and global variables.

To get the "right" header file, look up the function in a copy of the ANSI/ISO C standard. If you don't have a copy of the standard handy, use Table XII.2, shown in the next FAQ.

Cross Reference:

VIII.2: Why should I prototype a function?

XII.2: What header files do I need in order to define the standard library functions I use?

XII.2: What header files do I need in order to define the standard library functions I use?

Answer:

You need the ones that the ANSI/ISO standard says you should use. See Table XII.2.

The funny thing is, these are not necessarily the files that define what you're looking for. Your compiler guarantees that (for example) if you want the `EDOM` macro, you can get it by including `<errno.h>`. `EDOM` might be defined in `<errno.h>`, or `<errno.h>` might just include something that defines it. Worse, the next version of your compiler might define `EDOM` somewhere else.

Don't look in the files for the definition and use that file. Use the file that's supposed to define the symbol you want. It'll work.

A few names are defined in multiple files: `NULL`, `size_t`, and `wchar_t`. If you need a definition for one of these names, use a file you need to include anyway, or pick one arbitrarily. (`<stddef.h>` is a reasonable choice; it's small, and it defines common macros and types.)

Table XII.2. Standard library functions' header files.

Function/Macro	*Header File*
`abort`	stdlib.h
`abs`	stdlib.h
`acos`	math.h
`asctime`	time.h
`asin`	math.h
`assert`	assert.h
`atan`	math.h
`atan2`	math.h
`atexit`	stdlib.h
`atof`	stdlib.h
`atoi`	stdlib.h
`atol`	stdlib.h
`bsearch`	stdlib.h
`BUFSIZ`	stdio.h
`calloc`	stdlib.h
`ceil`	math.h
`clearerr`	stdio.h
`clock`	time.h
`CLOCKS_PER_SEC`	time.h
`clock_t`	time.h
`cos`	math.h
`cosh`	math.h
`ctime`	time.h
`difftime`	time.h
`div`	stdlib.h
`div_t`	stdlib.h
`EDOM`	errno.h

continues

Table XII.2. continued

Function/Macro	*Header File*
EOF	stdio.h
ERANGE	errno.h
errno	errno.h
exit	stdlib.h
EXIT_FAILURE	stdlib.h
EXIT_SUCCESS	stdlib.h
exp	math.h
fabs	math.h
fclose	stdio.h
feof	stdio.h
ferror	stdio.h
fflush	stdio.h
fgetc	stdio.h
fgetpos	stdio.h
fgets	stdio.h
FILE	stdio.h
FILENAME_MAX	stdio.h
floor	math.h
fmod	math.h
fopen	stdio.h
FOPEN_MAX	stdio.h
fpos_t	stdio.h
fprintf	stdio.h
fputc	stdio.h
fputs	stdio.h
fread	stdio.h
free	stdlib.h
freopen	stdio.h
frexp	math.h
fscanf	stdio.h
fseek	stdio.h
fsetpos	stdio.h
ftell	stdio.h
fwrite	stdio.h
getc	stdio.h

Function/Macro	*Header File*
getchar	stdio.h
getenv	stdlib.h
gets	stdio.h
gmtime	time.h
HUGE_VAL	math.h
_IOFBF	stdio.h
_IOLBF	stdio.h
_IONBF	stdio.h
isalnum	ctype.h
isalpha	ctype.h
iscntrl	ctype.h
isdigit	ctype.h
isgraph	ctype.h
islower	ctype.h
isprint	ctype.h
ispunct	ctype.h
isspace	ctype.h
isupper	ctype.h
isxdigit	ctype.h
jmp_buf	setjmp.h
labs	stdlib.h
LC_ALL	locale.h
LC_COLLATE	locale.h
LC_CTYPE	locale.h
LC_MONETARY	locale.h
LC_NUMERIC	locale.h
LC_TIME	locale.h
struct lconv	locale.h
ldexp	math.h
ldiv	stdlib.h
ldiv_t	stdlib.h
localeconv	locale.h
localtime	time.h
log	math.h
log10	math.h
longjmp	setjmp.h

continues

Table XII.2. continued

Function/Macro	*Header File*
L_tmpnam	stdio.h
malloc	stdlib.h
mblen	stdlib.h
mbstowcs	stdlib.h
mbtowc	stdlib.h
MB_CUR_MAX	stdlib.h
memchr	string.h
memcmp	string.h
memcpy	string.h
memmove	string.h
memset	string.h
mktime	time.h
modf	math.h
NDEBUG	assert.h
NULL	locale.h, stddef.h, stdio.h, stdlib.h, string.h, time.h
offsetof	stddef.h
perror	stdio.h
pow	math.h
printf	stdio.h
ptrdiff_t	stddef.h
putc	stdio.h
putchar	stdio.h
puts	stdio.h
qsort	stdlib.h
raise	signal.h
rand	stdlib.h
RAND_MAX	stdlib.h
realloc	stdlib.h
remove	stdio.h
rename	stdio.h
rewind	stdio.h
scanf	stdio.h
SEEK_CUR	stdio.h
SEEK_END	stdio.h
SEEK_SET	stdio.h

Function/Macro	*Header File*
`setbuf`	stdio.h
`setjmp`	setjmp.h
`setlocale`	locale.h
`setvbuf`	stdio.h
`SIGABRT`	signal.h
`SIGFPE`	signal.h
`SIGILL`	signal.h
`SIGINT`	signal.h
`signal`	signal.h
`SIGSEGV`	signal.h
`SIGTERM`	signal.h
`sig_atomic_t`	signal.h
`SIG_DFL`	signal.h
`SIG_ERR`	signal.h
`SIG_IGN`	signal.h
`sin`	math.h
`sinh`	math.h
`size_t`	stddef.h, stdlib.h, string.h, sprintf, stdio.h
`sqrt`	math.h
`srand`	stdlib.h
`sscanf`	stdio.h
`stderr`	stdio.h
`stdin`	stdio.h
`stdout`	stdio.h
`strcat`	string.h
`strchr`	string.h
`strcmp`	string.h
`strcoll`	string.h
`strcpy`	string.h
`strcspn`	string.h
`strerror`	string.h
`strftime`	time.h
`strlen`	string.h
`strncat`	string.h
`strncmp`	string.h
`strncpy`	string.h

continues

Table XII.2. continued

Function/Macro	*Header File*
`strpbrk`	string.h
`strrchr`	string.h
`strspn`	string.h
`strstr`	string.h
`strtod`	stdlib.h
`strtok`	string.h
`strtol`	stdlib.h
`strtoul`	stdlib.h
`strxfrm`	string.h
`system`	stdlib.h
`tan`	math.h
`tanh`	math.h
`time`	time.h
`time_t`	time.h
`struct tm`	time.h
`tmpfile`	stdio.h
`tmpnam`	stdio.h
`TMP_MAX`	stdio.h
`tolower`	ctype.h
`toupper`	ctype.h
`ungetc`	stdio.h
`va_arg`	stdarg.h
`va_end`	stdarg.h
`va_list`	stdarg.h
`va_start`	stdarg.h
`vfprintf`	stdio.h
`vprintf`	stdio.h
`vsprintf`	stdio.h
`wchar_t`	stddef.h, stdlib.h
`wcstombs`	stdlib.h
`wctomb`	stdlib.h

Cross Reference:

V.12: What is the difference between `#include <file>` and `#include "file"`?

XII.1: Why should I use standard library functions instead of writing my own?

XII.3: How can I write functions that take a variable number of arguments?

Answer:

Use `<stdarg.h>`. This defines some macros that let your program deal with variable numbers of arguments.

NOTE

The "variable arguments" functions used to be in a header file known as `<varargs.h>` or some such. Your compiler might or might not still have a file with that name; even if it does have the file now, it might not have it in the next release. Use `<stdarg.h>`.

There's no portable way for a C function, with no constraints on what it might be passed, to know how many arguments it might have gotten or what their types are. If a C function doesn't take a fixed number of arguments (of fixed types), it needs some convention for what the arguments are. For example, the first argument to `printf` is a string, which indicates what the remaining arguments are:

```
printf("Hello, world!\n");   /* no more arguments */
printf("%s\n", "Hello, world!");   /* one more string argument */
printf("%s, %s\n", "Hello", "world!");   /* two more string arguments */
printf("%s, %d\n", "Hello", 42);   /* one string, one int */
```

Listing XII.3 shows a simple `printf`-like function. The first argument is the format; from the format string, the number and types of the remaining arguments can be determined. As with the real `printf`, if the format doesn't match the rest of the arguments, the result is undefined. There's no telling what your program will do then (but probably something bad).

Listing XII.3. A simple `printf`-like function.

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

static char *
int2str(int n)
{
        int     minus = (n < 0);
```

continues

Listing XII.3. continued

```
        static char     buf[32];
        char    *p = &buf[31];

        if (minus)
                n = -n;
        *p = '\0';
        do {
                *--p = '0' + n % 10;
                n /= 10;
        } while (n > 0);
        if (minus)
                *--p = '-';
        return p;
}

/*
 * This is a simple printf-like function that handles only
 * the format specifiers %%, %s, and %d.
 */
void
simplePrintf(const char *format, ...)
{
        va_list ap; /* ap is our argument pointer. */
        int     i;
        char    *s;

        /*
         * Initialize ap to start with the argument
         * after "format"
         */
        va_start(ap, format);
        for ( ; *format; format++) {
                if (*format != '%') {
                        putchar(*format);
                        continue;
                }
                switch (*++format) {
                case 's':
                        /* Get next argument (a char*) */
                        s = va_arg(ap, char *);
                        fputs(s, stdout);
                        break;
                case 'd':
                        /* Get next argument (an int) */
                        i = va_arg(ap, int);
                        s = int2str(i);
                        fputs(s, stdout);
                        break;
                case '\0':
                        format--;
                        break;
                default:
                        putchar(*format);
                        break;
                }
        }
        /* Clean up varying arguments before returning */
        va_end(ap);
}

void
main()
{
        simplePrintf("The %s tax rate is %d%%.\n",
                "sales", 6);
}
```

Cross Reference:

XII.2: What header files do I need in order to define the standard library functions I use?

XII.4: What is the difference between a free-standing and a hosted environment?

Answer:

Not all C programmers write database management systems and word processors. Some write code for embedded systems, such as anti-lock braking systems and intelligent toasters. Embedded systems don't necessarily have any sort of file system, or much of an operating system at all. The ANSI/ISO standard calls these "free-standing" systems, and it doesn't require them to provide anything except the language itself. The alternative is a program running on a PC or a mainframe or something in-between; that's a "hosted" environment.

Even people developing for free-standing environments should pay attention to the standard library. For one thing, if a free-standing environment provides some functionality (such as a square root function), it's likely to provide it in a way that's compatible with the standard. (Reinventing the square root is like reinventing the square wheel; what's the point?) Beyond that, embedded programs are often tested on a PC before they're downloaded to a toaster (or whatever). Using the standard functions will increase the amount of code that can be identical in both the test and the real environments.

Cross Reference:

XII.1: Why should I use standard library functions instead of writing my own?

Chapter XV: Portability

XII.5: What standard functions are available to manipulate strings?

Answer:

Short answer: the functions in `<string.h>`.

C doesn't have a built-in string type. Instead, C programs use `char` arrays, terminated by the NUL (`'\0'`) character.

C programs (and C programmers) are responsible for ensuring that the arrays are big enough to hold all that will be put in them. There are three approaches:

- Set aside a lot of room, assume that it will be big enough, and don't worry what happens if it's not big enough (efficient, but this method can cause big problems if there's not enough room).
- Always allocate and reallocate the necessary amount of room (not too inefficient if done with `realloc`; this method can take lots of code and lots of runtime).
- Set aside what should be enough room, and stop before going beyond it (efficient and safe, but you might lose data).

NOTE

C++ is moving toward a fourth approach: leave it all behind and define a `string` type. For various reasons, that's a lot easier to do in C++ than in C. Even in C++, it's turning out to be rather involved. Luckily, after a standard C++ `string` type has been defined, even if it turns out to be hard to implement, it should be very easy for C++ programmers to use.

There are two sets of functions for C string programming. One set (`strcpy`, `strcat`, and so on) works with the first and second approaches. This set copies or uses as much as it's asked to—and there had better be room for it all, or the program might be buggy. Those are the functions most C programmers use. The other set (`strncpy`, `strncat`, and so on) takes the third approach. This set needs to know how much room there is, and it never goes beyond that, ignoring everything that doesn't fit.

The "n" (third) argument means different things to these two functions:

To `strncpy`, it means there is room for only "n" characters, including any NUL character at the end. `strncpy` copies exactly "n" characters. If the second argument doesn't have that many, `strncpy` copies extra NUL characters. If the second argument has more characters than that, `strncpy` stops before it copies any NUL character. That means, when using `strncpy`, you should always put a NUL character at the end of the string yourself; don't count on `strncpy` to do it for you.

To `strncat`, it means to copy up to "n" characters, plus a NUL character if necessary. Because what you really know is how many characters the destination can store, you usually need to use `strlen` to calculate how many characters you can copy.

The difference between `strncpy` and `strncat` is "historical." (That's a technical term meaning "It made sense to somebody, once, and it might be the right way to do things, but it's not obvious why right now.")

Listing XII.5a shows a short program that uses `strncpy` and `strncat`.

NOTE

Get to know the “string-n” functions. Using them is harder but leads to more robust, less buggy software.

If you’re feeling brave, try rewriting the program in Listing XII.5a with `strcpy` and `strcat`, and run it with big enough arguments that the buffer overflows. What happens? Does your computer hang? Do you get a General Protection Exception or a core dump? See FAQ VII.24 for a discussion.

Listing XII.5a. An example of the “string-n” functions.

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

/*
Normally, a constant like MAXBUF would be very large, to
help ensure that the buffer doesn’t overflow.  Here, it’s very
small, to show how the “string-n” functions prevent it from
ever overflowing.
*/

#define MAXBUF 16

int
main(int argc, char** argv)
{
        char buf[MAXBUF];
        int i;

        buf[MAXBUF - 1] = ‘\0’;

        strncpy(buf, argv[0], MAXBUF-1);
        for (i = 1; i < argc; ++i) {
                strncat(buf, “ “,
                  MAXBUF - 1 - strlen(buf));
                strncat(buf, argv[i],
                  MAXBUF - 1 - strlen(buf));
        }

        puts(buf);
        return 0;

}
```

NOTE

Many of the string functions take at least two string arguments. It’s convenient to refer to them as “the left argument” and “the right argument,” rather than “the first argument” and “the second argument,” for describing which one is which.

`strcpy` and `strncpy` copy a string from one array to another. The value on the right is copied to the value on the left; think of the order as being the same as that for assignment.

`strcat` and `strncat` “concatenate” one string onto the end of another. For example, if `a1` is an array that holds `“dog”` and `a2` is an array that holds `“wood”`, after calling `strcat(a1, a2)`, `a1` would hold `“dogwood”`.

`strcmp` and `strncmp` compare two strings. The return value is negative if the left argument is less than the right, zero if they’re the same, and positive if the left argument is greater than the right. There are two common idioms for equality and inequality:

```
if (strcmp(s1, s2)) {
    /* s1 != s2 */
}
```

and

```
if (! strcmp(s1, s2)) {
    /* s1 == s2 */
}
```

This code is not incredibly readable, perhaps, but it’s perfectly valid C code and quite common; learn to recognize it. If you need to take into account the current locale when comparing strings, use `strcoll`.

A number of functions search in a string. (In all cases, it’s the “left” or first argument being searched in.) `strchr` and `strrchr` look for (respectively) the first and last occurrence of a character in a string. (`memchr` and `memrchr` are the closest functions to the “n” equivalents `strchr` and `strrchr`.) `strspn`, `strcspn` (the “c” stands for “complement”), and `strpbrk` look for substrings consisting of certain characters or separated by certain characters:

```
n = strspn(“Iowa”, “AEIOUaeiou”);
/* n = 2; “Iowa” starts with 2 vowels */

n = strcspn(“Hello world”, “ \t”);
/* n = 5; white space after 5 characters */
p = strbrk(“Hello world”, “ \t”);
/* p points to blank */
```

`strstr` looks for one string in another:

```
p = strstr(“Hello world”, “or”);
/* p points to the second “o” */
```

`strtok` breaks a string into tokens, which are separated by characters given in the second argument. `strtok` is “destructive”; it sticks NUL characters in the original string. (If the original string should be changed, it should be copied, and the copy should be passed to `strtok`.) Also, `strtok` is not “reentrant”; it can’t be called from a signal-handling function, because it “remembers” some of its arguments between calls. `strtok` is an odd function, but very useful for pulling apart data separated by commas or white space. Listing XII.5b shows a simple program that uses `strtok` to break up the words in a sentence.

Listing XII.5b. An example of using `strtok`.

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

static char buf[] = “Now is the time for all good men ...”;
```

```
int
main()
{
        char* p;
        p = strtok(buf, " ");
        while (p) {
                printf("%s\n", p);
                p = strtok(NULL, " ");
        }
        return 0;
}
```

Cross Reference:

IV.18: How can I read and write comma-delimited text?

Chapter VI: Working with Strings

VII.23: What is the difference between `NULL` and `NUL`?

IX.9: What is the difference between a string and an array?

XII.8: What is a "locale"?

XII.10: What's a signal? What do I use signals for?

XII.6: What standard functions are available to manipulate memory?

Answer:

Several functions copy, compare, and fill arbitrary memory. These functions take `void*` (pointers to nothing in particular); they work with pointers to anything.

There are two functions (roughly like `strncpy`) for copying information. One, `memmove`, copies memory from one place to another, even if the two places overlap. Why is that important? Say you have a buffer with some data already in it, and you want to move it "to the right" to make room at the beginning of the buffer. Listing XII.6 shows a program that tries to perform this action but doesn't do it right.

Listing XII.6. A program that tries to move data but trashes it instead.

```
static char buf[] =
   {'R', 'I', 'G', 'H', 'T', '\0', '-', '-', '-'};
int
main()
{
    int i;
    for (i=0; i<6; ++i) {
        buf[i+3] = buf[i];
    }
}
```

The idea was to change `buf` from being `"RIGHT"` to being `"RIGRIGHT"` so that other data could be put in the first three bytes. Unfortunately, that's not what happened. If you unroll the `for` loop (or run the program with a debugger to see what it's doing), you'll see that the program really acted like this:

```
buf[3] = buf[0];
buf[4] = buf[1];
buf[5] = buf[2];
buf[6] = buf[3];
buf[7] = buf[4];
buf[8] = buf[5];
buf[9] = buf[6];
```

The effect on the data is shown in Figure XII.6a (the newly copied data is shown in bold). The program trashed some of the data it was supposed to move!

Figure XII.6a.

The wrong way to "move" overlapping data.

R	I	G	H	T	\0	–	–	–
R	I	G	**R**	T	\0	–	–	–
R	I	G	**R**	**I**	\0	–	–	–
R	I	G	**R**	**I**	**G**	–	–	–
R	I	G	**R**	**I**	**G**	**R**	–	–
R	I	G	**R**	**I**	**G**	**R**	**I**	–
R	I	G	**R**	**I**	**G**	**R**	**I**	**G**

For moving or copying data that overlaps, there's a simple rule. If the source and destination areas overlap, and the source is before the destination, start at the end of the source and work backward to the beginning. If the source is after the destination, start at the beginning of the source and work to the end. See Figure XII.6b.

Figure XII.6b.

The right ways to "move" overlapping data.

R	I	G	H	T	\0	–	–	–
R	I	G	H	T	\0	–	–	**\0**
R	I	G	H	T	\0	–	**T**	**\0**
R	I	G	H	T	\0	**H**	**T**	**\0**
R	I	G	H	T	**G**	**H**	**T**	**\0**
R	I	G	H	**I**	**G**	**H**	**T**	**\0**
R	I	G	**R**	**I**	**G**	**H**	**T**	**\0**
<	<	<	L	E	F	T	\0	
L	<	<	L	E	F	T	\0	
L	**E**	<	L	E	F	T	\0	
L	**E**	**F**	L	E	F	T	\0	
L	**E**	**F**	**T**	E	F	T	\0	
L	**E**	**F**	**T**	**\0**	F	T	\0	

The purpose for explaining all that is to tell you this: the `memmove` function knows that rule. It is guaranteed to copy data, even overlapping data, the right way. If you're copying or moving data and you're not sure whether the source and destination overlap, use `memmove`. If you're sure they don't overlap, `memcpy` might be marginally faster.

The `memcmp` function is like the `strncmp` function, except that it doesn't stop at bytes with NUL characters (`'\0'`). It shouldn't be used to compare `struct` values, though. Say that you have the following structure:

```
struct foo {
    short s;
    long l;
}
```

And say that on the system your program will run on, a `short` is two bytes (16 bits) long, and a `long` is four bytes (32 bits) long. On a 32-bit machine, many compilers put two bytes of "junk" between `s` and `l` so that `l` starts on a word boundary. If your program runs on a little-endian machine (the least significant byte is stored at the lowest address), the structure might be laid out like this:

`struct foo byte[0]`	least significant byte of `s`
`struct foo byte[1]`	most significant byte of `s`
`struct foo byte[2]`	junk (make `l` start on a `long` boundary)
`struct foo byte[3]`	junk (make `l` start on a `long` boundary)
`struct foo byte[4]`	least significant byte of `l`
`struct foo byte[5]`	second least significant byte of `lstruct`
`struct foo byte[6]`	second most significant byte of `l`
`struct foo byte[7]`	most significant byte of `l`

Two `struct foo`s with the same `s` and `l` values might not compare equal with `memcmp`, because the "junk" might be different.

`memchr` is like `strchr`, but it looks for a character anywhere in a specified part of memory; it won't stop at the first NUL byte.

`memset` is useful even for nonparanoid C programmers. It copies some byte into a specified part of memory. One common use is to initialize some structure to all zero bytes. If `p` is a pointer to a `struct`, then

```
memset(p, '\0', sizeof *p);
```

overwrites the thing `p` points to with zero (NUL or `'\0'`) bytes. (This also overwrites any "junk" used to get members on word boundaries, but that's OK; it's junk, nobody cares what you write there.)

Cross Reference:

VI.1: What is the difference between a string copy (`strcpy`) and a memory copy (`memcpy`)? When should each be used?

VI.3: How can I remove the leading spaces from a string?

IX.9: What is the difference between a string and an array?

XII.7: How do I determine whether a character is numeric, alphabetic, and so on?

Answer:

The header file ctype.h defines various functions for determining what class a character belongs to. These consist of the following functions:

Function	*Character Class*	*Returns Nonzero for Characters*
`isdigit()`	Decimal digits	0–9
`isxdigit()`	Hexadecimal digits	0–9, a–f, or A–F
`isalnum()`	Alphanumerics	0–9, a–z, or A–Z
`isalpha()`	Alphabetics	a–z or A–Z
`islower()`	Lowercase alphabetics	a–z
`isupper()`	Uppercase alphabetics	A–Z
`isspace()`	Whitespace	Space, tab, vertical tab, newline, form feed, or carriage return
`isgraph()`	Nonblank characters	Any character that appears nonblank when printed (ASCII 0x21 through 0x7E)
`isprint()`	Printable characters	All the `isgraph()` characters, plus space
`ispunct()`	Punctuation	Any character in `isgraph()` that is not in `isalnum()`
`iscntrl()`	Control characters	Any character not in `isprint()` (ASCII 0x00 through 0x1F plus 0x7F)

There are three very good reasons for calling these macros instead of writing your own tests for character classes. They are pretty much the same reasons for using standard library functions in the first place. First, these macros are fast. Because they are generally implemented as a table lookup with some bit-masking magic, even a relatively complicated test can be performed much faster than an actual comparison of the value of the character.

Second, these macros are correct. It's all too easy to make an error in logic or typing and include a wrong character (or exclude a right one) from a test.

Third, these macros are portable. Believe it or not, not everyone uses the same ASCII character set with PC extensions. You might not care today, but when you discover that your next computer uses Unicode rather than ASCII, you'll be glad you wrote code that didn't assume the values of characters in the character set.

The header file ctype.h also defines two functions to convert characters between upper- and lowercase alphabetics. These are `toupper()` and `tolower()`. The behavior of `toupper()` and `tolower()` is undefined if their arguments are not lower- and uppercase alphabetic characters, respectively, so you must remember to check using `islower()` and `isupper()` before calling `toupper()` and `tolower()`.

Cross Reference:

V.1: What is a macro, and how do you use it?

VI.2: How can I remove the trailing spaces from a string?

VI.3: How can I remove the leading spaces from a string?

XX.18: How do you tell whether a character is a letter of the alphabet?

XX.19: How do you tell whether a character is a number?

XII.8: What is a "locale"?

Answer:

A locale is a description of certain conventions your program might be expected to follow under certain circumstances. It's mostly helpful to internationalize your program.

If you were going to print an amount of money, would you always use a dollar sign? Not if your program was going to run in the United Kingdom; there, you'd use a pound sign. In some countries, the currency symbol goes before the number; in some, it goes after. Where does the sign go for a negative number? How about the decimal point? A number that would be printed `1,234.56` in the United States should appear as `1.234,56` in some other countries. Same value, different convention. How are times and dates displayed? The only short answer is, differently. These are some of the technical reasons why some programmers whose programs have to run all over the world have so many headaches.

Good news: Some of the differences have been standardized. C compilers support different "locales," different conventions for how a program acts in different places. For example, the `strcoll` (string collate) function is like the simpler `strcmp`, but it reflects how different countries and languages sort and order (collate) string values. The `setlocale` and `localeconv` functions provide this support.

Bad news: There's no standardized list of interesting locales. The only one your compiler is guaranteed to support is the "C" locale, which is a generic, American English convention that works best with ASCII characters between 32 and 127. Even so, if you need to get code that looks right, no matter where around the world it will run, thinking in terms of locales is a good first step. (Getting several locales your compiler supports, or getting your compiler to accept locales you define, is a good second step.)

Cross Reference:

None.

XII.9: Is there a way to jump out of a function or functions?

Answer:

The standard library functions `setjmp()` and `longjmp()` are used to provide a `goto` that can jump out of a function or functions, in the rare cases in which this action is useful. To correctly use `setjmp()` and `longjmp()`, you must apply several conditions.

You must `#include` the header file setjmp.h. This file provides the prototypes for `setjmp()` and `longjmp()`, and it defines the type `jmp_buf`. You need a variable of type `jmp_buf` to pass as an argument to both `setjmp()` and `longjmp()`. This variable will contain the information needed to make the jump occur.

You must call `setjmp()` to initialize the `jmp_buf` variable. If `setjmp()` returns 0, you have just initialized the `jmp_buf`. If `setjmp()` returns anything else, your program just jumped to that point via a call to `longjmp()`. In that case, the return value is whatever your program passed to `longjmp()`.

Conceptually, `longjmp()` works as if when it is called, the currently executing function returns. Then the function that called *it* returns, and so on, until the function containing the call to `setjmp()` is executing. Then execution jumps to where `setjmp()` was called from, and execution continues from the return of `setjmp()`, but with the return value of `setjmp()` set to whatever argument was passed to `longjmp()`.

In other words, if function `f()` calls `setjmp()` and later calls function `g()`, and function `g()` calls function `h()`, which calls `longjmp()`, the program behaves as if `h()` returned immediately, then `g()` returned immediately, then `f()` executed a `goto` back to the `setjmp()` call.

What this means is that for a call to `longjmp()` to work properly, the program must already have called `setjmp()` and must not have returned from the function that called `setjmp()`. If these conditions are not fulfilled, the operation of `longjmp()` is undefined (meaning your program will probably crash). The program in Listing XII.9 illustrates the use of `setjmp()` and `longjmp()`. It is obviously contrived, because it would be simpler to write this program without using `setjmp()` and `longjmp()`. In general, when you are tempted to use `setjmp()` and `longjmp()`, try to find a way to write the program without them, because they are easy to misuse and can make a program difficult to read and maintain.

Listing XII.9. An example of using `setjmp()` and `longjmp()`.

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

#define RETRY_PROCESS 1
#define QUIT_PROCESS  2

jmp_buf env;

int     nitems;

int
procItem()
{
        char    buf[256];
        if (gets(buf) && strcmp(buf, "done")) {
                if (strcmp(buf, "quit") == 0)
                        longjmp(env, QUIT_PROCESS);
                if (strcmp(buf, "restart") == 0)
                        longjmp(env, RETRY_PROCESS);
                nitems++;
                return 1;
        }
        return 0;
}

void
```

```
process()
{
        printf("Enter items, followed by 'done'.\n");
        printf("At any time, you can type 'quit' to exit\n");
        printf("or 'restart' to start over again\n");
        nitems = 0;
        while (procItem())
                ;
}

void
main()
{
        for ( ; ; ) {
                switch (setjmp(env)) {
                case 0:
                case RETRY_PROCESS:
                        process();
                        printf("You typed in %d items.\n",
                                nitems);
                        break;
                case QUIT_PROCESS:
                default:
                        exit(0);
                }
        }
}
```

Cross Reference:

I.8: What is the difference between `goto` and `longjmp()` and `setjmp()`?

VII.20: What is the stack?

XII.10: What's a signal? What do I use signals for?

Answer:

A signal is an exceptional condition that occurs during the execution of your program. It might be the result of an error in your program, such as a reference to an illegal address in memory; or an error in your program's data, such as a floating-point divided by 0; or an outside event, such as the user's pressing Ctrl-Break.

The standard library function `signal()` enables you to specify what action is to be taken on one of these exceptional conditions (a function that performs that action is called a "signal handler"). The prototype for `signal()` is

```
#include        <signal.h>
void (*signal(int num, void (*func)(int)))(int);
```

which is just about the most complicated declaration you'll see in the C standard library. It is easier to understand if you define a `typedef` first. The type `sigHandler_t`, shown next, is a pointer to a function that takes an `int` as its argument and returns a `void`:

```
typedef void (*sigHandler_t)(int);
sigHandler_t signal(int num, sigHandler_t func);
```

`signal()` is a function that takes an `int` and a `sigHandler_t` as its two arguments, and returns a `sigHandler_t` as its return value. The function passed in as the `func` argument will be the new signal handler for the exceptional condition numbered `num`. The return value is the previous signal handler for signal `num`. This value can be used to restore the previous behavior of a program, after temporarily setting a signal handler. The possible values for `num` are system dependent and are listed in signal.h. The possible values for `func` are any function in your program, or one of the two specially defined values `SIG_DFL` or `SIG_IGN`. The `SIG_DFL` value refers to the system's default action, which is usually to halt the program. `SIG_IGN` means that the signal is ignored.

The following line of code, when executed, causes the program containing it to ignore Ctrl-Break keystrokes unless the signal is changed again. Although the signal numbers are system dependent, the signal number `SIGINT` is normally used to refer to an attempt by the user to interrupt the program's execution (Ctrl-C or Ctrl-Break in DOS):

```
signal(SIGINT, SIG_IGN);
```

Cross Reference:

XX.16: How do you disable Ctrl-Break?

XII.11: Why shouldn't I start variable names with underscores?

Answer:

Identifier names beginning with two underscores or an underscore followed by a capital letter are reserved for use by the compiler or standard library functions wherever they appear. In addition, all identifier names beginning with an underscore followed by anything are reserved when they appear in file scope (when they are not local to a function).

If you use a reserved identifier for a variable name, the results are undefined (your program might not compile, or it might compile but crash). Even if you are lucky enough to pick an identifier that is not currently used by your compiler or library, remember that these identifiers are reserved for possible use later. Thus, it's best to avoid using an underscore at the beginning of variable and function names.

Cross Reference:

XIX.1: Should the underscore be used in variable names?

XII.12: Why does my compiler provide two versions of *malloc()*?

Answer:

By including stdlib.h, you can use `malloc()` and `free()` in your code. This function is put in your code by the compiler from the standard C library. Some compilers have a separate library that you can ask the compiler to use (by specifying a flag such as `-lmalloc` on the command line) to replace the standard library's versions of `malloc()` and `free()` with a different version.

These alternative versions of `malloc()` and `free()` do the same thing as the standard ones, but they are supposedly implemented to provide better performance at the cost of being less forgiving about memory allocation errors. I have never had a reason to use these alternative routines in 15 years of C programming. But in answering this FAQ, I wrote a simple test program to heavily exercise `malloc()` and `free()` and compiled it with a well-known commercial C compiler both with and without the `malloc` library. I couldn't detect any significant difference in performance, and because both versions of the routines were the same size, I suspect that this particular vendor used the same code for both implementations. For this reason, I will not name the compiler vendor.

The moral of the story is that you probably don't need to bother with the other version of `malloc()` and probably shouldn't count on it for performance improvements. If profiling shows that your program spends a large percentage of its time in `malloc()` and `free()`, and you can't fix the problem by changing the algorithm, you might be able to improve performance by writing your own "pool" allocator.

Programs that call `malloc()` and `free()` a lot are often allocating and freeing the same type of data, which has a fixed size. When you know the size of the data to be allocated and freed, a pool allocator can be much faster than `malloc()` and `free()`. A pool allocator works by calling `malloc()` to allocate many structures of the same size all at once, then hands them out one at a time. It typically never calls `free()`, and the memory stays reserved for use by the pool allocator until the program exits. Listing XII.12 shows a pool allocator for the hypothetical type `struct foo`.

Listing XII.12. An example of a pool allocator.

```
#include        <stdio.h>

/* declaration of hypothetical structure "foo" */
struct foo {
        int     dummy1;
        char    dummy2;
        long    dummy3;
};

/* start of code for foo pool allocator */

#include        <stdlib.h>

/* number of foos to malloc() at a time */
#define NFOOS   64

/*
 * A union is used to provide a linked list that
 * can be overlaid on unused foos.
 */
union foo_u {
        union foo_u     *next;
        struct foo      f;
};

static union foo_u      *free_list;
```

continues

Listing XII.12. continued

```
struct foo *
alloc_foo()
{
        struct foo      *ret = 0;
        if (!free_list) {
                int     i;
                free_list = (union foo_u *) malloc(NFOOS
                                * sizeof(union foo_u));
                if (free_list) {
                        for (i = 0; i < NFOOS - 1; i++)
                                free_list[i].next =
                                        &free_list[i + 1];
                        free_list[NFOOS - 1].next = NULL;
                }
        }
        if (free_list) {
                ret = &free_list->f;
                free_list = free_list->next;
        }
        return ret;
}

void
free_foo(struct foo *fp)
{
        union foo_u     *up = (union foo_u *) fp;
        up->next = free_list;
        free_list = up;
}

int
main(int argc, char **argv)
{
        int     i;
        int     n;
        struct foo      **a;

        if (argc < 2) {
                fprintf(stderr, “usage: %s f\n”, argv[0]);
                fprintf(stderr, “where f is the number of”);
                fprintf(stderr, “ ‘foo’s to allocate\n”);
                exit(1);
        }
        i = atoi(argv[1]);
        a = (struct foo **) malloc(sizeof(struct foo *) * i);
        for (n = 0; n < i; n++)
                a[n] = alloc_foo();
        for (n = 0; n < i; n++)
                free_foo(a[n]);
        return 0;
}
```

I compiled and ran this program with an argument of `300000` and compared the results to a similar program that replaced calls to `alloc_foo()` and `free_foo()` with calls to `malloc()` and `free()`. The CPU time used by the version of the program using the pool allocator was 0.46 seconds. The version of the program that used `malloc()` took 0.92 seconds.

Note that you should use a pool allocator only as a last resort. It might improve speed, but it can be very wasteful of memory. It also can lead to subtle memory allocation errors if you're not careful to return memory to the pool from which it came instead of calling `free()`.

Cross Reference:

VII.21: What is the heap?

VII.26: How does `free()` know how much memory to release?

XII.13: What math functions are available for integers? For floating point?

Answer:

The operations `+`, `-`, `*`, and `/` (addition, subtraction, multiplication, and division) are available for both integer and floating-point arithmetic. The operator `%` (remainder) is available for integers only.

For floating-point math, many other functions are declared in the header file math.h. Most of these functions operate in double-precision floating point, for increased accuracy. If these functions are passed an argument outside of their domain (the domain of a function is the set of legal values for which it is defined), the function will return some unspecified value and will set the variable `errno` to the value `EDOM`. If the return value of the function is too large or small to be represented by a `double` (causing overflow or underflow), the function will return `HUGE_VAL` (for overflow) or 0 (for underflow) and will set `errno` to `ERANGE`. The values `EDOM`, `ERANGE`, and `HUGE_VAL` are defined in math.h.

The following list describes the functions declared in math.h:

- `double cos(double)`, `double sin(double)`, `double tan(double)` take a value in radians and return the cosine, sine, and tangent of the value, respectively.
- `double acos(double)`, `double asin(double)`, `double atan(double)` take a value and return the arc cosine, arc sine, and arc tangent of the value, respectively. The value passed to `acos()` and `asin()` must be in the range –1 to 1, inclusive.
- `double atan2(double x, double y)` returns the arc tangent of the value represented by `x/y`, even if `x/y` is not representable as a `double` (if `y` is 0, for instance).
- `double cosh(double)`, `double sinh(double)`, `double tanh(double)` take a value in radians and return the hyperbolic cosine, hyperbolic sine, and hyperbolic tangent of the value, respectively.
- `double exp(double x)`, `double log(double x)`, `double log10(double x)` take a value and return `ex`, the natural logarithm of `x`, and the logarithm base 10 of `x`, respectively. The two logarithm functions will cause a range error (`ERANGE`) if `x` is 0 and a domain error (`EDOM`) if `x` is negative.
- `double sqrt(double)` returns the square root of its argument. It causes a domain error (`EDOM`) if the value passed to it is negative.

- `double ldexp(double n, int e)` returns `n * 2e`. This is somewhat analogous to the << operator for integers.
- `double pow(double b, double e)` returns `be`. It causes a domain error (`EDOM`) if `b` is 0 and `e` is less than or equal to 0, or if `b` is less than 0 and `e` is not an integral value.
- `double frexp(double n, int *i)` returns the *mantissa* of `n` and sets the `int` pointed to by `i` to the *exponent* of `n`. The mantissa is in the range 0.5 to 1 (excluding 1 itself), and the exponent is a number such that `n =` *mantissa* `* 2`*exponent*.
- `double modf(double n, int *i)` returns the fractional part of `n` and sets the `int` pointed to by `i` to the integer part of `n`.
- `double ceil(double)`, `double floor(double)` return the smallest integer greater than or equal to and the largest integer less than or equal to their arguments, respectively. For instance, `ceil(-1.1)` returns –1.0, and `floor(-1.1)` returns –2.0.
- `double fmod(double x, double y)` returns the remainder of `x/y`. This is similar to the `%` operator for integers, but it does not restrict its inputs or result to be `int`s. It causes a domain error (`EDOM`) if `y` is 0.
- `double fabs(double)` returns the absolute value of the value passed to it (a number with the same magnitude, but always positive). For instance, `fabs(-3.14)` returns 3.14.

Cross Reference:

II.11: Are there any problems with performing mathematical operations on different variable types?

XII.14: What are multibyte characters?

Answer:

Multibyte characters are another way to make internationalized programs easier to write. Specifically, they help support languages such as Chinese and Japanese that could never fit into eight-bit characters. If your programs will never need to deal with any language but English, you don't need to know about multibyte characters.

Inconsiderate as it might seem, in a world full of people who might want to use your software, not everybody reads English. The good news is that there are standards for fitting the various special characters of European languages into an eight-bit character set. (The bad news is that there are several such standards, and they don't agree.)

Go to Asia, and the problem gets more complicated. Some languages, such as Japanese and Chinese, have more than 256 characters. Those will never fit into any eight-bit character set. (An eight-bit character can store a number between 0 and 255, so it can have only 256 different values.)

The good news is that the standard library has the beginnings of a solution to this problem. `<stddef.h>` defines a type, `wchar_t`, that is guaranteed to be long enough to store any character in any language a C program can deal with. Based on all the agreements so far, 16 bits is enough. That's often a `short`, but it's better to trust that the compiler vendor got `wchar_t` right than to get in trouble if the size of a `short` changes.

The `mblen`, `mbtowc`, and `wctomb` functions transform byte strings into multibyte characters. See your compiler manuals for more information on these functions.

Cross Reference:

XII.15: How can I manipulate strings of multibyte characters?

XII.15: How can I manipulate strings of multibyte characters?

Answer:

Better than you might think.

Say your program sometimes deals with English text (which fits comfortably into 8-bit `char`s with a bit to spare) and sometimes Japanese text (which needs 16 bits to cover all the possibilities). If you use the same code to manipulate either country's text, will you need to set aside 16 bits for every character, even your English text? Maybe not. Some (but not all) ways of encoding multibyte characters can store information about whether more than one byte is necessary.

`mbstowcs` ("multibyte string to wide character string") and `wcstombs` ("wide character string to multibyte string") convert between arrays of `wchar_t` (in which every character takes 16 bits, or two bytes) and multibyte strings (in which individual characters are stored in one byte if possible).

There's no guarantee your compiler can store multibyte strings compactly. (There's no single agreed-upon way of doing this.) If your compiler can help you with multibyte strings, `mbstowcs` and `wcstombs` are the functions it provides for that.

Cross Reference:

XII.14: What are multibyte characters?