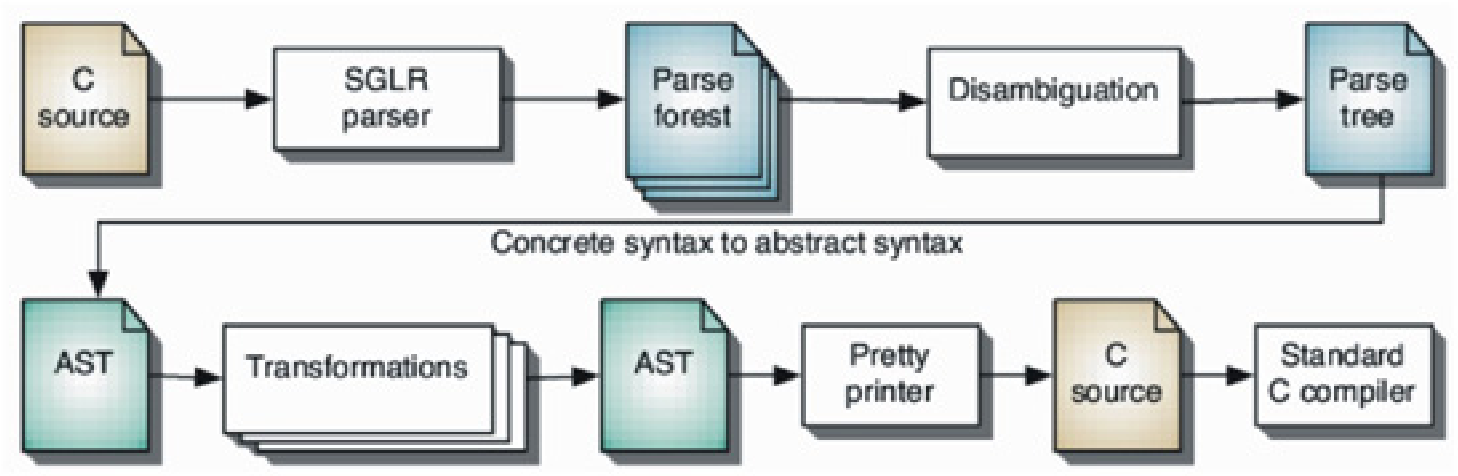
C Programming

Mark Lee



Training makes a difference...

Global Media

Education For Everyone

First Edition, 2007

ISBN 978 81 89940 36 2

Published by:

Global Media
1819, Bhagirath Palace,
Chandni Chowk, Delhi-110 006
Email: globalmedia@dkpd.com

Table of Contents

Introduction

Why Learn C?

The most popular Operating Systems right now are Microsoft Windows, Mac OS X, and /Linux. Each is written in C. Why? Because Operating Systems run directly on top of the hardware. There is no lower layer to mediate their requests. Originally, this OS software was written in the Assembly language, which results in very fast and efficient code. However, writing an OS in Assembly is a tedious process, and produces code that will only run on one CPU architecture, such as the Intel X86 or AMD64. Writing the OS in a higher level language, such as C, lets programmers re-target the OS to other architectures without re-writing the entire code.

But why 'C' and not Java or Basic or Perl? Mostly because of memory allocation. Unlike most computer languages, C allows the programmer to address memory the way he/she would using assembly language. Languages like Java and Perl shield the programmer from having to worry about memory allocation and pointers. This is usually a good thing. It's quite tedious to deal with memory allocation when building a high-level program like a quarterly income statement report. However, when dealing with low level code such as that part of the OS that moves the string of bytes that makes up that quarterly income report from the computer's memory to the network card's buffer so they can be shipped to the network printer, direct access to memory is critical -- something you just can't do with Java. C can be compiled into fast and efficient machine code.

So is it any wonder that C is such a popular language?

Like toppling dominoes, the next generation of programs follows the trend of its ancestors. Operating Systems designed in C always have system libraries designed in C. Those system libraries are in turn used to create higher-level libraries (like OpenGL, or GTK), and the designers of those libraries often decide to use the language the system libraries used. Application developers use the higher-level libraries to design word processors, games, media players, and the like. Many of them will choose to program in the language that higher-level library uses. And the pattern continues on and on and on... ← Why learn C? | What you need before you can learn →

History of the C Programming Language

In 1947, three scientists at Bell Telephone Laboratories, William Shockley, Walter Brattain, and John Bardeen created the transistor. Modern computing was beginning. In 1956 at MIT the first fully transistor based computer was completed, the TX-0. In 1958 at Texas Instruments, Jack Kilby created the first integrated circuit. But even before the first integrated circuit existed, the first high level language had already been written.

In 1954 Fortran, the Formula Translator, had been written. It began as Fortran I in 1956. Fortran begot Algol 58, the Algorithmic Language, in 1958. Algol 58 begot Algol 60 in 1960. Algol 60 begot CPL, the Combined Programming Language, in 1963. CPL begot BCPL, Basic CPL, in 1967. BCPL begot B in 1969. B begot C in 1971.

B was the first language in the C lineage directly, having been created at Bell Labs by Ken Thompson. B was an interpreted language, used in early, internal versions of the UNIX operating system. Thompson and Dennis Ritchie, also of Bell Labs, improved B, calling it NB; further extensions to NB created C, a compiled language. Most of UNIX was rewritten in NB and then C, which led to a more portable operating system.

B was of course named after BCPL, and C was its logical successor.

The portability of UNIX was the main reason for the initial popularity of both UNIX and C. So rather than creating a new operating system for each new machine, system programmers could simply write the few system-dependent parts required for the machine, and write a C compiler for the new system. Thereafter since most of the system utilities were written in C, it simply made sense to also write new utilities in that language. ← History | Using a Compiler →

Getting Started

This book is intended to be an introduction to C programming. Although some basic computer literacy is assumed, no special knowledge is needed.

The minimum software required to start programming in C is a **text editor** to create C source files, and **C compiler** to turn those source files into executable programs.

Many programmers, however, prefer to use a **IDE** (Integrated development environments). This is a program which combines editing, compiling and debugging into a convenient all-in-one interface. There are a variety of these available on almost every computer platform. Some can be downloaded free-of-charge while others are commercial products.

C Compilers:

		Platform	**License**	**Extra**
OpenWatcom	[1]	DOS, Windows, Netware, OS/2	Open source	
Borland C Compiler	[2]	Windows	Freeware	
C Compiler	[3]	DOS, Cygwin (w32), MinGW (w32)OS/2, Mac OS X, Unix	Open source	De facto standard. Ships with most Unix systems.
Tiny C Compiler (tcc)	[4]	/Linux, Windows	Open source	Small, fast compiler

IDEs:

		Platform	**License**	**Extra**
CDT	[5]	Windows, Mac OS X,	Open source	A C/C++ plug-in for Eclipse, a popular open source IDE.

		Unix		
Little C Compiler (LCC)	[6]	Windows	Free for non-commercial use.	
Anjuta	[7]	Unix	Open source	A GTK+2 IDE for the GNOME desktop environment
Xcode	[8]	Mac OS X	Freeware	Available on the "Developer Tools" disc with most recent-model Apple computers, or as download when registered (free) as ADC-member at http://developer.apple.com/ .
Pelles C	[9]	Windows, Pocket PC	"free"	

For Windows, Dev-C++ is recommended for its ease-of-use and simplicity of installation.

Installing the C Compiler on Linux can vary in method from Linux distribution to distribution.

- For Redhat, get a gcc RPM, e.g. using Rpmfind and then install (as root) using `rpm -ivh gcc-`*`version-release.arch`*`.rpm`
- For Fedora Core, install the GCC compiler (as root) by using `yum install gcc`.
- For Mandrake, install the GCC compiler (as root) by using `urpmi gcc`
- For Debian, install the GCC compiler (as root) by using `apt-get install gcc`.
- For Ubuntu, install the GCC compiler by using `sudo apt-get install gcc`, or by using Synaptic. You do not need Universe enabled.
- For Slackware, the package is available on their website - simply download, and type `installpkg gcc-xxxxx.tgz`
- For Gentoo, you should already have GCC already installed as it will have been used when you first installed. To update it run (as root) `emerge -uav gcc`
- For Arch /Linux, install the GCC compiler (as root) by using `pacman -Sy gcc`.
- For FreeBSD, NetBSD, OpenBSD, DragonFly BSD, Darwin the port of gcc is available in the base system, or it could be obtained using the ports collection or pkgsrc.
- If you cannot become root, get the GCC tarball from ftp://ftp..org/ and follow the instructions in it to compile and install in your home directory. Be warned though, you need a C compiler to do that - yes, gcc itself is written in C.
- You can use some commercial C compiler/IDE.

A text editor with syntax highlighting is recommended, as it can make code easier to read at a glance. Highlighting can also make it easy to spot syntax errors. Most programmers' text editors on Windows and Unix systems can do this. ← What you need before you can learn | A taste of C →

Dev-C++

Dev C++, as mentioned before, is an Integrated Development Enviroment (IDE) for the C++ programming language, available from Bloodshed Software.
C++ is a programming language which contains within itself most of the C language, plus a few

extensions - as such, most C++ compilers also compile C programs, sometimes with a few adjustments (like invoking it with a different name or commandline switch). Therefore you can use Dev C++ for C developement.

Dev C++ is not, however, the compiler: It is designed to use the MinGW or Cygwin versions of GCC - both of which can be downloaded as part of the Dev C++ package, although they are completely different projects.
Dev C++ simply provides an editor, syntax highlighting, some facilities for the visualisation of code (like class and package browsing) and a graphical interface to the chosen compiler. Because Dev C++ analyses the error messages produced by the compiler and attempts to distinguish the line numbers from the errors themselves, the use of other compiler software is discouraged since the format of their error messages is likely to be different.

The current version of Dev-C++ is a beta for version 5 - as such, it still has a significant number of bugs. However, all the features are there and it is quite usable - as such, it is still considered one of the best free software C IDEs available for Windows.

A version of Dev-C++ for Linux is in the pipeline; it is not quite usable yet, however Linux users already have a wealth of IDEs available to them (for example KDevelop and Anjuta.) Also, almost all the graphical text editors, and other common editors such as *emacs* and *vi(m)*, support syntax highlighting.

gcc

The GCC is a free set of compilers developed by the Free Software Foundation, with Richard Stallman as one of the main architects.

Detailed Steps for Compiling and Running Your First "Hello, world!" Program on Windows:

1. Open Notepad or another text editor (like the Crimson Editor listed above), and copy and paste this program into a new file:

```
#include <stdio.h>

int main()
{
   printf("Hello, world!\n");
   return (0);
}
```

2. Save this file as "hello.c" in the folder with your username, in the "home" folder in the Cygwin folder (i.e., somewhere like, "C:\cygwin\home\your-username-here").

3. Double-click the Cygwin icon on your desktop to start a Cygwin command prompt, and type "ls" to list the contents of your home folder; you should see your program "hello.c" listed if you have saved your program to the location listed in step #2, above.

4. Now type "gcc -o hello hello.c" and press enter to compile your program. If any error messages come up, make sure your "hello.c" file looks exactly like the code above, and make sure you are in the same folder as your "hello.c" file (you can enter "cd" at the prompt at any time to return to the "C:\cygwin\home\you-username-here" folder if you are unsure where you are.)

5. If all goes well and no error messages come up, type "ls" again at the prompt and you should now see "hello.c" as well as "hello.exe", your newly compiled program.

6. Type "hello.exe" and press enter to run your program; you should see "Hello, world!" printed out -- welcome to the miracle of computing! (On newer versions it may help to type "./hello.exe"

The current stable (usable) version is 4.0 published on 20 April 2005, which supports several platforms. In fact, GCC is not only a C compiler, but a family of compilers for several languages, such as C++, Ada, Java, and Fortran.

To get started using GCC, you can simply call **gcc** from the command line, followed by some of the modifiers:

-c: indicates that the compiler is supposed to generate an *object file*, which can be later linked to other files to form a final program.

-o: indicates that the next parameter is the name of the resulting program (or library). If this option is not specified, the compiled program will, for historic reasons, end up in a file called "a.out" or "a.exe" (for cygwin users).

-g3: indicates that *debugging information* should be added to the results of compilation

-O2 -ffast-math: indicates that the compilation should be optimized

-W -Wall -fno-common -Wcast-align -Wredundant-decls -Wbad-function-cast -Wwrite-strings -Waggregate-return -Wstrict-prototypes -Wmissing-prototypes: indicates that gcc should warn about many types of suspicious code that are likely to be incorrect

-E: indicates that gcc should only preprocess the code; this is useful when you are having trouble understanding what gcc is doing with #include and #define, among other things

For example, to compile the file *hello.c* into the program *hello*, use

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

← Using a Compiler | Intro exercise →

Like in every other programming language learning book we use the *Hello world* program to introduce you to C.

```
/*1*/  #include <stdio.h>
/*2*/
/*3*/  int main(void)
/*4*/  {
/*5*/     printf("Hello, world!\n");
/*6*/     return 0;
/*7*/  }
/*8*/
```

This program prints "Hello, world!" and then exits. The numbers are added for our benefit to refer to certain lines and would not be part of the real program.

Line 1 tells the C compiler to find a file called *stdio.h* and add the contents of that file to this program. In C, you often have to pull in extra optional components when you need them. *stdio.h* contains descriptions of standard input/output functions; in other words, stuff you can use to send messages to a user, or to read input from a user.

Line 3 is something you'll find in every C program. Every program has a *main* function. Generally, the main function is where a program begins. However, one C program can be scattered across multiple files, so you won't always find a main function in every file. The *int* at the beginning means that main will return an integer to whatever made it run when it is finished and *void* in the parenthesis means that main takes no parameters (parameters to main typically come from a shell when the program is invoked).

Line 5 is the statement that actually sends the message to the screen. *printf* is a function that is declared in the file *stdio.h* - which is why you had to *#include* that at the start of the program. *\n* is a so-called escape code which adds a new line at the end of the printed text.

Line 6 will return zero (which is the integer referred to on line 3) to the operating system. When a program runs successfully its return value is zero (GCC4 complains if it doesn't when compiling). A non-zero value is returned to indicate a warning or error.

Line 8 is there because it is (at least on UNIX) considered good practice to end a file with a new line. ← A taste of C | Preliminaries →

Introductory Exercises

If you are using a Unix(-like) system, such as /Linux, Mac OS X, or Solaris, it will probably have GCC installed. Type the hello world program into a file called first.c and then compile it with gcc. Just type:

```
gcc first.c
```

Then run the program by typing:

```
./a.out
```

or

```
a.exe
```

if you are using cygwin.

There are a lot of options you can use with the gcc compiler. For example, if you want the output to have a name other than a.out, you can use the -o option. Also, you can ask the compiler to print warnings while it handles your code. The following shows a few examples:

```
gcc -Wall -ansi -pedantic -o first first.c
```

All the options are well documented in the manual page for gcc and at even more length in the info material for gcc.

If you are using a commercial IDE you may have to select console project, and to compile you just select build from the menu or the toolbar. The executable will appear inside the project folder, but you should have a menu button so you can just run the executable from the IDE.

Simple Input and Output

When you take time to consider it, a computer would be pretty useless without some way to talk to the people who use it. Just like we need information in order to accomplish tasks, so do computers. And just as we supply information to others so that *they* can do tasks, so do computers.

These supplies and returns of information to a computer are called **input** and **output**. 'Input' is information supplied to a computer or program. 'Output' is information provided by a computer or program. Frequently, computer programmers will lump the discussion in the more general term *input/output* or simply, **I/O**.

In C, there are many different ways for a program to communicate with the user. Amazingly, the most simple methods usually taught to beginning programmers may also be the most powerful. In the "Hello, World" example at the beginning of this text, we were introduced to a Standard Library file stdio.h, and one of its functions, printf(). Here we discuss more of the functions that stdio.h gives us.

Output using printf()

Recall from the beginning of this text the demonstration program duplicated below:

```
#include <stdio.h>
int main(void)
{
   printf("Hello, world!\n");
   return 0;
}
```

If you compile and run this program, you will see the sentence below show up on your screen:

```
Hello, world!
```

This amazing accomplishment was achieved by using the *function* `printf()`. A function is like a "black box" that does something for you without exposing the internals inside. We can write functions ourselves in C, but we will cover that later.

You have seen that to use `printf()` one puts text, surrounded by quotes, in between the brackets. We call the text surrounded by quotes a *literal string* (or just a *string*), and we call that string an *argument* to printf.

As a note of explanation, it is sometimes convenient to include the open and closing parentheses after a function name to remind us that it is, indeed, a function. However usually when the name of the function we are talking about is understood, it is not necessary.

As you can see in the example above, using `printf()` can be as simple as typing in some text, surrounded by double quotes (note that these are double quotes and not two single quotes). So, for example, you can print any string by placing it as an argument to the `printf()` function:

```
printf("This sentence will print out exactly as you see it...");
```

And once it is contained in a proper `main()` function, it will show:

```
This sentence will print out exactly as you see it...
```

Printing numbers and escape sequences

Placeholder codes

The `printf` function is a powerful function, and is probably the most-used function in C programs.

For example, let us look at a problem. Say we don't know what 1905 + 31214 is. Let's use C to get the answer.

We start writing

```
#include <stdio.h> /* this is important, since printf
                      can't be used without this line */
```

(For more information about the line above, see The Preprocessor).

```
int main(void)
{
   printf("1905+31214 is");
   return 0;
}
```

but here we are stuck! `printf` only prints strings! Thankfully, printf has methods for printing numbers. What we do is put a *placeholder* format code in the string. We write:

```
  printf("1905+31214 is %d", 1905+31214);
```

The placeholder %d literally "holds the place" for the actual number that is the result of adding 1905 to 31214.

These placeholders are called **format specifiers**. Many other format specifiers work with `printf`. If we have a floating-point number, we can use `%f` to print out a floating-point number, decimal point and all. An incomplete list is:

- %i - int (same as %d)
- %f - float
- %lf - double
- %s - string
- %x - hexadecimal

A more complete list is in the File I/O section.

Tabs and newlines

What if, we want to achieve some output that will look like:

```
   1905
  31214 +
  -----
```

`printf` will not put line breaks in at the end of each statement: we must do this ourselves. But how?

What we can do is use the newline *escape character*. An escape character is a special character that we can write but will do something special onscreen, such as make a beep, write a tab, and so on. To write a newline we write `\n`. All escape characters start with a backslash.

So to achieve the output above, we write

```
printf(" 1905\n31214 +\n-----\n%d", 33119);
```

or to be a bit clearer, we can break this long printf statement over several lines. So our program will be:

```
#include <stdio.h>

int main(void)
{
   printf(" 1905\n");
   printf("31214 +\n");
   printf("-----\n");
   printf("%d", 33119);
   return 0;
}
```

There are other escape characters we can use. Another common one is to use `\t` to write a tab. You can use `\a` to ring the computer's bell, but you should not use this very much in your programs, as excessive use of sound is not very friendly to the user.

Other output methods

puts()

The puts() function is a very simple way to send a string to the screen when you have no placeholders to be concerned about. It works very much like the printf() function we saw the "Hello, World!" example:

```
puts("Print this string.");
```

will print to the screen:

```
Print this string.
```

followed by the newline character (as discussed above). (The `puts` function appends a newline character to its output.) The `fputs` function is similar:

```
fputs("Print this string via fputs", stdout);
```

will print to the stdout file (usually the screen):

```
Print this string via fputs
```

without a newline tacked on to the end.

Since puts() and fputs() do not allow the placeholders and the associated formatting that printf() allows, for most programmers learning printf() is sufficient for their needs.

Input using scanf()

The scanf() function is the input method equivalent to the printf() output function - simple yet powerful. In its simplest invocation, the scanf *format string* holds a single *placeholder* representing the type of value that will be entered by the user. These placeholders are exactly the same as the printf() function - %d for ints, %f for floats, and %lf for doubles.

There is, however, one variation to scanf() as compared to printf(). The scanf() function requires the memory address of the variable to which you want to save the input value. While *pointers* are possible here, this is a concept that won't be approached until later in the text. Instead, the simple technique is to use the *address-of* operator, **&**. For now it may be best to consider this "magic" before we discuss pointers.

A typical application might be like this:

```
#include <stdio.h>

int main(void)
{
   int a;

   printf("Please input an integer value: ");
   scanf("%d", &a);

}
```

If you are trying to input a string using *scanf*, you should **not** include the & operator.

If you were to describe the effect of the scanf() function call above, it might read as: "Read in an integer from the user and store it at the address of variable *a* ".

Note of caution on inputs: When data is typed at a keyboard, the information does not go straight to the program that is running. It is first stored in what is known as a **buffer** - a small amount of memory reserved for the input source. Sometimes there will be data left in the buffer when the program wants to read from the input source, and the scanf() function will read this data instead of waiting for the user to type something. The function fflush(stdin) may fix this issue on some computers and with some compilers, by clearing or "flushing" the input buffer. But this isn't generally considered good practice and may not be portable - if you take your code to a different computer with a different compiler, your code may not work properly.

Examples

← Simple Input and Output | Further math →

Operators and Assignments

In C, simple math is very easy to handle. The following operators exist: + (addition), - (subtraction), * (multiplication), / (division), and % (modulus); You likely know all of them from your math classes - except, perhaps, modulus. It returns the **remainder** of a division (e.g. 5 % 2 = 1).

Care must be taken with the modulus, because it's not the equivalent of the mathematical modulus: (-5) % 2 is not 1, but -1. Division of integers will return an integer, and the division of a negative integer by a positive integer will round towards zero instead of rounding down (e.g. (-5) / 3 = -1 instead of -2).

There is no inline operator to do the power (e.g. 5 ^ 2 is **not** 25, and 5 ** 2 is an error), but there is a power function.

The mathematical order of operations does apply. For example (2 + 3) * 2 = 10 while 2 + 3 * 2 = 8. The order of precedence in C is BFDMAS: Brackets, Functions, Division or Multiplication (from left to right, whichever comes first), Addition or Subtraction (also from left to right, whichever comes first).

Assignment in C is simple. You declare the type of variable, the name of the variable and what it's equal to. For example, int x = 0; double y = 0.0; char z = 'a';

```
#include <stdio.h>

int main()
{
 int i = 0, j = 0;

 while( (i < 5) && (j < 5) ) { /* while i is less than 5 AND j is less than 5, loop */
        ++j; /* prefix increment, increases by 1 immediately */
        printf("i equals: %d\tj equals: %d\n", i, j); /* will print current variable
values */
        i++; /* postfix increment, increases by 1 next time the variable is called,
therefore i will be equal to 0 in the beginning */
 }

 return 0;
}
```

will display the following:

```
i equals: 0     j equals: 1
i equals: 1     j equals: 2
i equals: 2     j equals: 3
i equals: 3     j equals: 4
i equals: 4     j equals: 5
```

← Simple math | Control →

The `<math.h>` header contains prototypes for several functions that deal with mathematics. In the 1990 version of the ISO standard, only the `double` versions of the functions were specified; the 1999 version added the `float` and `long double` versions.

The functions can be grouped into the following categories:

Trigonometric functions

The `acos` and `asin` functions

The `acos` functions return the arccosine of their arguments in radians, and the `asin` functions return the arcsine of their arguments in radians. All functions expect the argument in the range [-1,+1]. The arccosine returns a value in the range [0,π]; the arcsine returns a value in the range [-π/2,+π/2].

```
#include <math.h>
float asinf(float x); /* C99 */
float acosf(float x); /* C99 */
double asin(double x);
double acos(double x);
long double asinl(long double x); /* C99 */
long double acosl(long double x); /* C99 */
```

The `atan` and `atan2` functions

The `atan` functions return the arctangent of their arguments in radians, and the `atan2` function return the arctangent of `y/x` in radians. The `atan` functions return a value in the range [-π/2,+π/2] (the reason why ±π/2 are included in the range is because the floating-point value may represent infinity, and atan(±∞) = ±π/2); the `atan2` functions return a value in the range [-π,+π]. For `atan2`, a domain error may occur if both arguments are zero.

```
#include <math.h>
float atanf(float x); /* C99 */
float atan2f(float y, float x); /* C99 */
double atan(double x);
double atan2(double y, double x);

long double atanl(long double x); /* C99 */

long double atan2l(long double y, long double x); /* C99 */
```

The `cos`, `sin`, and `tan` functions

The `cos`, `sin`, and `tan` functions return the cosine, sine, and tangent of the argument, expressed in radians.

```
#include <math.h>
float cosf(float x); /* C99 */
float sinf(float x); /* C99 */
float tanf(float x); /* C99 */
double cos(double x);
double sin(double x);
double tan(double x);
long double cosl(long double x); /* C99 */
long double sinl(long double x); /* C99 */
long double tanl(long double x); /* C99 */
```

Hyperbolic functions

The `cosh`, `sinh` and `tanh` functions compute the hyperbolic cosine, the hyperbolic sine, and the hyperbolic tangent of the argument respectively. For the hyperbolic sine and cosine functions, a range error occurs if the magnitude of the argument is too large.

```
#include <math.h>
float coshf(float x); /* C99 */
float sinhf(float x); /* C99 */
float tanhf(float x); /* C99 */
double cosh(double x);
double sinh(double x);
double tanh(double x);
long double coshl(long double x); /* C99 */
long double sinhl(long double x); /* C99 */
long double tanhl(long double x); /* C99 */
```

Exponential and logarithmic functions

The `exp` functions

The exp functions compute the exponential function of x (e^x). A range error occurs if the magnitude of x is too large.

```
#include <math.h>
float expf(float x); /* C99 */
double exp(double x);
long double expl(long double x); /* C99 */
```

The frexp, ldexp, and modf functions

The frexp functions break a floating-point number into a normalized fraction and an integer power of 2. It stores the integer in the object pointed to by ex.

The frexp functions return the value x such that x has a magnitude of either [1/2, 1) or zero, and value equals x times 2 to the power *ex. If value is zero, both parts of the result are zero.

The ldexp functions multiply a floating-point number by a integral power of 2 and return the result. A range error may occur.

The modf function breaks the argument value into integer and fraction parts, each of which has the same sign as the argument. They store the integer part in the object pointed to by *iptr and return the fraction part.

```
#include <math.h>
float frexpf(float value, int *ex); /* C99 */
double frexp(double value, int *ex);
long double frexpl(long double value, int *ex); /* C99 */
float ldexpf(float x, int ex); /* C99 */
double ldexp(double x, int ex);
long double ldexpl(long double x, int ex); /* C99 */
float modff(float value, float *iptr); /* C99 */
double modf(double value, double *iptr);
long double modfl(long double value, long double *iptr); /* C99 */
```

The log and log10 functions

The log functions compute the natural logarithm of the argument and return the result. A domain error occurs if the argument is negative. A range error may occur if the argument is zero.

The log10 functions compute the common (base-10) logarithm of the argument and return the result. A domain error occurs if the argument is negative. A range error may occur if the argument is zero.

```
#include <math.h>
float logf(float x); /* C99 */
double log(double x);
long double logl(long double x); /* C99 */
float log10f(float x); /* C99 */
double log10(double x);
long double log10l(long double x); /* C99 */
```

Power functions

The `pow` functions

The `pow` functions compute `x` raised to the power `y` and return the result. A domain error occurs if `x` is negative and `y` is not an integral value. A domain error occurs if the result cannot be represented when `x` is zero and `y` is less than or equal to zero. A range error may occur.

```
#include <math.h>
float powf(float x, float y); /* C99 */
double pow(double x, double y);
long double powl(long double x, long double y); /* C99 */
```

The `sqrt` functions

The `sqrt` functions compute the nonnegative square root of `x` and return the result. A domain error occurs if the argument is negative.

```
#include <math.h>
float sqrtf(float x); /* C99 */
double sqrt(double x);
long double sqrtl(long double x); /* C99 */
```

Nearest integer, absolute value, and remainder functions

The `ceil` and `floor` functions

The `ceil` functions compute the smallest integral value not less than `x` and return the result; the `floor` functions compute the largest integral value not greater than `x` and return the result.

```
#include <math.h>
float ceilf(float x); /* C99 */
double ceil(double x);
long double ceill(long double x); /* C99 */
float floorf(float x); /* C99 */
double floor(double x);
long double floorl(long double x); /* C99 */
```

The `fabs` functions

The `fabs` functions compute the absolute value of a floating-point number `x` and return the result.

```
#include <math.h>
float fabsf(float x); /* C99 */
double fabs(double x);
long double fabsl(long double x); /* C99 */
```

The `fmod` functions

The `fmod` functions compute the floating-point remainder of `x/y` and return the value `x` - *i* * `y`, for some integer *i* such that, if `y` is nonzero, the result has the same sign as `x` and magnitude less than the magnitude of `y`. If `y` is zero, whether a domain error occurs or the `fmod` functions return zero is implementation-defined.

```
#include <math.h>
float fmodf(float x, float y); /* C99 */
double fmod(double x, double y);
long double fmodl(long double x, long double y); /* C99 */
```

← Further math | Procedures and functions →

Control

Very few C programs follow exactly one control path and have each instruction stated explicitly. In order to program effectively, it is necessary to understand how one can alter the steps taken by a program due to user input or other conditions, how some steps can be executed many times with few lines of code, and how programs can appear to demonstrate a rudimentary grasp of logic. C constructs known as conditionals and loops grant this power.

From this point forward, it is necessary to understand what is usually meant by the word *block*. A block is a group of code statements that are associated and intended to be executed as a unit. In C, the beginning of a block of code is denoted with { (left curly), and the end of a block is denoted with }. It is not necessary to place a semicolon after the end of a block. Blocks can be empty, as in {}. Blocks can also be nested; i.e. there can be blocks of code within larger blocks.

Conditionals

There is likely no meaningful program written in which a computer does not demonstrate basic decision-making skills. It can actually be argued that there is no meaningful human activity in which some sort of decision-making, instinctual or otherwise, does not take place. For example, when driving a car and approaching a traffic light, one does not think, "I will continue driving through the intersection." Rather, one thinks, "I will stop if the light is red, go if the light is green, and if yellow go only if I am traveling at a certain speed a certain distance from the intersection." These kinds of processes can be simulated in C using conditionals.

A conditional is a statement that instructs the computer to execute a certain block of code or alter certain data only if a specific condition has been met. The most common conditional is the If-Else statement, with conditional expressions and Switch-Case statements typically used as more shorthanded methods.

Before one can understand conditional statements, it is first necessary to understand how C expresses logical relations. C treats logic as being arithmetic. The value 0 (zero) represents false, and ***all other values*** represent true. If you chose some particular value to represent true and then compare values against it, sooner or later your code will fail when your assumed value (often 1) turns out to be incorrect. Code written by people uncomfortable with the C language can often be identified by the usage of #define to make a "TRUE" value.

Because logic is arithmetic in C, arithmetic operators and logical operators are one and the same. Nevertheless, there are a number of operators that are typically associated with logic:

Relational and Equivalence Expressions:

a < b	1 if a is less than b, 0 otherwise.
a > b	1 if a is greater than b, 0 otherwise.
a <= b	1 if a is less than or equal to b, 0 otherwise.
a >= b	1 if a is greater than or equal to b, 0 otherwise.
a == b	1 if a is equal to b, 0 otherwise.
a != b	1 if a is not equal to b, 0 otherwise

New programmers should take special note of the fact that the "equal to" operator is ==, not =. This is the cause of numerous coding mistakes and is often a difficult-to-find bug, as the statement (a = b) sets a equal to b and subsequently evaluates to b, while (a == b), which is usually intended, checks if a is equal to b. It needs to be pointed out that, if you confuse = with ==, your mistake will often not be brought to your attention by the compiler. A statement such as `if ( c = 20) {` is considered perfectly valid by the language, but will always assign 20 to c and evaluate as true.

A note regarding testing for equality against a truth constant: never do it.

```
#define TRUE 42
if (SomethingsAfoot() == TRUE)   // bad code  :^(
```

Instead it is much safer and more elegant to just write

```
if (SomethingsAfoot())    // good code  :^)
```

This is because someone could have defined TRUE erroneously such that an expression such as `(A < B) == TRUE` would actually evaluate to FALSE when A is indeed less than B. So let's repeat: Avoid testing for equality against TRUE.

One other thing to note is that the relational expressions do not evaluate as they would in mathematical texts. That is, an expression `myMin < value < myMax` does not evaluate as you probably think it might. Mathematically, this would test whether or not *value* is between *myMin* and *myMax*. But in C, what happens is that *value* is first compared with *myMin*. This produces either a 0 or a 1. It is this value that is compared against myMax. Example:

```
int value = 20;
...
if ( 0 < value < 10) {
   /* do some stuff */
}
```

Because value is greater than 0, the first comparison produces a value of 1. Now 1 is compared to be less than 10, which is true, so the statements in the if are executed. This probably is not what the programmer expected. The appropriate code would be

```
int value = 20;
...
if ( 0 < value && value < 20) {   // the && means "and"
 /* do some stuff */
}
```

If you're looking for a programming language that matches the mathematical notation, try Python.

Logical Expressions:

a || b	1 if either a or b is not zero, 0 otherwise.
a && b	1 if both a and b are not zero, 0 otherwise.
!a	1 if a is 0, 0 if a is not zero.

Here's an example of a logical expression. In the statement:

```
  e = ((a && b) || (c > d));
```

e is set equal to 1 if a and b are non-zero, or if c is greater than d. In all other cases, e is set to 0.

C uses short circuit evaluation of logical expressions. That is to say, once it is able to determine the truth of a logical expression, it does no further evaluation. This is often useful as in the following:

```
int myArray[12];
....
if ( i < 12 && myArray[i] > 3) {
....
```

In the snippit of code, the comparison of i with 12 is done first. If it evaluates to 0 (false), **i** would be out of bounds as an index to **myArray**. In this case, the program never attempts to access **myArray[i]** since the truth of the expression is known to be false. Hence we need not worry here about trying to access an out-of-bounds array element if it is already known that i is greater than or equal to zero. A similar thing happens with expressions involving the or || operator.

```
while( doThis() || doThat()) ...
```

DoThat() is never called if doThis() returns a non-zero (true) value.

Bitwise Boolean Expressions

The bitwise operators work bit by bit on the operands. The operands must be of integral type (one of the types used for integers). The six bitwise operators are & (AND), | (OR), ^ (exclusive OR, commonly called XOR), ~ (NOT, which changes 1 to 0 and 0 to 1), << (shift left), and >> (shift right). The negation operator is a unary operator which preceeds the operand. The others are binary operators which lie between the two operands. The precedence of these operators is lower than that of the relational and equivalence operators; it is often required to parenthesize expressions involving bitwise operators.

For this section, recall that a number starting with **0x** is hexadecimal, or hex for short. Unlike the normal decimal system using powers of 10 and digits 0123456789, hex uses powers of 16 and digits 0123456789abcdef. It is commonly used in C programs because a programmer can quickly convert it to or from binary (powers of 2 and digits 01). C does not directly support binary notation, which would be really verbose anyway.

a & b
: bitwise boolean and of a and b
: 0xc & 0xa produces the value 0x8 (in binary, 1100 & 1010 produces 1000)

a | b
: bitwise boolean or of a and b
: 0xc | 0xa produces the value 0xe (in binary, 1100 | 1010 produces 1110)

a ^ b
: bitwise xor of a and b
: 0xc ^ 0xa produces the value 0x6 (in binary, 1100 ^ 1010 produces 0110)

~a
: bitwise complement of a.
: ~0xc produces the value -1-0xc (in binary, ~1100 produces ...11110011 where "..." may be many more 1 bits)

a << b
: shift a left by b (multiply a by 2^b)
: 0xc << 1 produces the value 0x18 (in binary, 1100 << 1 produces the value 11000)

a >> b
: shift a right by b (divide a by 2^b)
: 0xc >> 1 produces the value 0x6 (in binary, 1100 >> 1 produces the value 110)

The If-Else statement

If-Else provides a way to instruct the computer to execute a block of code only if certain conditions have been met. The syntax of an If-Else construct is:

```
if (/* condition goes here */)
{
   /* if the condition is non-zero (true), this code will execute */
}
else
{
   /* if the condition is 0 (false), this code will execute */
}
```

The first block of code executes if the condition in parentheses directly after the *if* evaluates to non-zero (true); otherwise, the second block executes.

The *else* and following block of code are completely optional. If there is no need to execute code if a condition is not true, leave it out.

Also, keep in mind that an *if* can directly follow an *else* statement. While this can occasionally be useful, chaining more than two or three if-elses in this fashion is considered bad programming practice. We can get around this with the Switch-Case construct described later.

Two other general syntax notes need to be made that you will also see in other control constructs: First, note that there is no semicolon after *if* or *else*. There could be, but the block (code enclosed in { and }) takes the place of that. Second, if you only intend to execute one statement as a result of an *if* or *else*, curly braces are not needed. However, many programmers believe that inserting curly braces anyway in this case is good coding practice.

The following code sets a variable c equal to the greater of two variables a and b, or 0 if a and b are equal.

```
if(a > b)
{
   c = a;
}
else if(b > a)
     {
        c = b;
     }
     else
     {
        c = 0;
     }
```

Consider this question: why can't you just forget about *else* and write the code like:

```
if(a > b)
{
   c = a;
}

if(a < b)
{
  c = b;
}

if(a == b)
{
  c = 0;
}
```

There are several answers to this. Most importantly, if your conditionals are not mutually exclusive, *two* cases could execute instead of only one. If the code was different and the value of a or b changes somehow (e.g.: you reset the lesser of a and b to 0 after the comparison) during one of the blocks? You could end up with multiple *if* statements being invoked, which is not your intent. Also, evaluating *if* conditionals takes processor time. If you use *else* to handle these situations, in the case above assuming (a > b) is non-zero (true), the program is spared the expense of evaluating additional *if* statements. The bottom line is that it is usually best to insert an *else* clause for all cases in which a conditional will not evaluate to non-zero (true).

The conditional expression

A conditional expression is a way to set values conditionally in a more shorthand fashion than If-Else. The syntax is:

```
(/* logical expression goes here */) ? (/* if non-zero (true) */) : (/* if 0 (false) */)
```

The logical expression is evaluated. If it is non-zero (true), the overall conditional expression evaluates to the expression placed between the ? and :, otherwise, it evaluates to the expression after the :. Therefore, the above example (changing its function slightly such that c is set to b when a and b are equal) becomes:

```
c = (a > b) ? a : b;
```

Conditional expressions can sometimes clarify the intent of the code. Nesting the conditional operator should usually be avoided. It's best to use conditional expressions only when the expressions for a and b are simple. Also, contrary to a common beginner belief, conditional expressions do not make for faster code. As tempting as it is to assume that fewer lines of code result in faster execution times, there is no such correlation.

The Switch-Case statement

Say you write a program where the user inputs a number 1-5 (corresponding to student grades, A(represented as 1)-D(4) and F(5)), stores it in a variable **grade** and the program responds by printing to the screen the associated letter grade. If you implemented this using If-Else, your code would look something like this:

```
if(grade == 1)
{
   printf("A\n");
}
else if(grade == 2)
{
   printf("B\n");
}
else if /* etc. etc. */
```

Having a long chain of if-else-if-else-if-else can be a pain, both for the programmer and anyone reading the code. Fortunately, there's a solution: the Switch-Case construct, of which the basic syntax is:

```
switch(/* integer or enum goes here */)
{
  case /* potential value of the aforementioned int or enum */:
     /* code */
  case /* a different potential value */:
     /* different code */
  /* insert additional cases as needed */

  default:

     /* more code */
  }
```

The Switch-Case construct takes a variable, usually an int or an enum, placed after *switch*, and compares it to the value following the *case* keyword. If the variable is equal to the value specified after *case*, the construct "activates", or begins executing the code after the case statement. Once the construct has "activated", there will be no further evaluation of *case*s.

Switch-Case is syntactically "weird" in that no braces are required for code associated with a *case*.

Very important: Typically, the last statement for each case is a break statement. This causes program execution to jump to the statement following the closing bracket of the switch statement, which is what one would normally want to happen. However if the break statement is omitted, program execution continues with the first line of the next case, if any. This is called a *fall-through*. When a programmer desires this action, a comment should be placed at the end of the block of statements indicating the desire to fall through. Otherwise another programmer maintaining the code could consider the omission of the 'break' to be an error, and inadvertently 'correct' the problem. Here's an example:

```
switch ( someVariable ) {
case 1:
   printf("This code handles case 1\n");
   break;
case 2:
   printf("This prints when someVariable is 2, along with...\n");
   /* FALL THROUGH */
case 3:
   printf("This prints when someVariable is either 2 or 3.\n" );
   break;
}
```

If a *default* case is specified, the associated statements are executed if none of the other cases match. A *default* case is optional. Here's a switch statement that corresponds to the sequence of if - else if statements above.

Back to our example above. Here's what it would look like as Switch-Case:

```
switch (grade)
{
   case 1:
      printf("A\n");
      break;
   case 2:
      printf("B\n");
      break;
   case 3:
      printf("C\n");
      break;
   case 4:

      printf("D\n");

      break;
   default:
      printf("F\n");
      break;
}
```

A set of statements to execute can be grouped with more than one value of the variable as in the following example. (the fall-through comment is not necessary here because the intended behavior is obvious)

```
switch ( something)
{
  case 2:
  case 3:
  case 4:
      /* some statements to execute for 2, 3 or 4 */
      break;
  case 1:
  default:
     /* some statements to execute for 1 or other than 2,3,and 4 */
     break;
}
```

Switch-Case constructs are particularly useful when used in conjunction with user defined *enum* data types. Some compilers are capable of warning about an unhandled enum value, which may be helpful for avoiding bugs.

Loops

Often in computer programming, it is necessary to perform a certain action a certain number of times or until a certain condition is met. It is impractical and tedious to simply type a certain statement or group of statements a large number of times, not to mention that this approach is too inflexible and unintuitive to be counted on to stop when a certain event has happened. As a real-world analogy, someone asks a dishwasher at a restaurant what he did all night. He will respond, "I washed dishes all night long." He is not likely to respond, "I washed a dish, then washed a dish, then washed a dish, then...". The constructs that enable computers to perform certain repetitive tasks are called loops.

While loops

A while loop is the most basic type of loop. It will run as long as the condition is non-zero (true). For example, if you try the following, the program will appear to lock up and you will have to manually close the program down. A situation where the conditions for exiting the loop will never become true is called an infinite loop.

```
int a=1;
while(42) {
   a = a*2;
}
```

Here is another example of a while loop. It prints out all the exponents of two less than 100.

```
int a=1;
while(a<100) {
   printf("a is %d \n",a);
   a = a*2;
}
```

The flow of all loops can also be controlled by **break** and **continue** statements. A break statement will immediately exit the enclosing loop. A continue statement will skip the remainder of the block and start at the controlling conditional statement again. For example:

```
int a=1;
while (42) { // loops until the break statement in the loop is executed
   printf("a is %d ",a);
   a = a*2;
   if(a>100)
       break;
   else if(a==64)
       continue;  // Immediately restarts at while, skips next step
   printf("a is not 64\n");
}
```

In this example, the computer prints the value of a as usual, and prints a notice that a is not 64 (unless it was skipped by the continue statement).

Similar to If above, braces for the block of code associated with a While loop can be omitted if the code consists of only one statement, for example:

```
int a=1;
while(a < 100)
   a = a*2;
```

This will merely increase a until a is not less than 100.

It is very important to note, once the controlling condition of a While loop becomes 0 (false), the loop will not terminate until the block of code is finished and it is time to reevaluate the conditional. If you need to terminate a While loop immediately upon reaching a certain condition, consider using **break**.

A common idiom is to write:

```
int i = 5;
while(i--)
   printf("java and c# can't do this\n");
```

This executes the code in the while loop 5 times, with i having values that range from 4 down to 0. Conveniently, these are the values needed to access every item of an array containing 5 elements.

For loops

For loops generally look something like this:

```
for(initialization; test; increment)
{
   /* code */
}
```

The *initialization* statement is executed exactly once - before the first evaluation of the *test* condition. Typically, it is used to assign an initial value to some variable, although this is not strictly necessary. The *initialization* statement can also be used to declare and initialize variables used in the loop.

The *test* expression is evaluated each time before the code in the *for* loop executes. If this expression evaluates as 0 (false) when it is checked (i.e. if the expression is not true), the loop is not (re)entered and execution continues normally at the code immediately following the FOR-loop. If the expression is non-zero (true), the code within the braces of the loop is executed.

After each iteration of the loop, the *increment* statement is executed. This often is used to increment the loop index for the loop, the variable initialized in the initialization expression and tested in the test expression. Following this statement execution, control returns to the top of the loop, where the *test* action occurs. If a *continue* statement is executed within the *for* loop, the increment statement would be the next one executed.

Each of these parts of the for statement is optional and may be omitted. Because of the free-form nature of the for statement, some fairly fancy things can be done with it. Often a for loop is used to loop through items in an array, processing each item at a time.

```
int  myArray[12];
int ix;
for (ix = 0; ix<12; ix++)
```

```
   myArray[ix] = 5 * ix + 3;
```

The above for loop initializes each of the 12 elements of myArray. The loop index can start from any value. In the following case it starts from 1.

```
for(ix = 1; ix <= 10; ix++)
{
   printf("%d ", ix);
}
```

which will print

```
1 2 3 4 5 6 7 8 9 10
```

You will most often use loop indexes that start from 0, since arrays are indexed at zero, but you will sometimes use other values to initalize a loop index as well.

The *increment* action can do other things, such as *decrement*. So this kind of loop is common:

```
for (i = 5; i > 0; i--)
{
   printf("%d ",i);
}
```

which yields

```
5 4 3 2 1
```

Here's an example where the test condition is simply a variable. If the variable has a value of 0 or NULL, the loop exits, otherwise the statements in the body of the loop are executed.

```
for (t = list_head; t; t = NextItem(t) ) {
  /*body of loop */
}
```

A WHILE loop can be used to do the same thing as a FOR loop, however a FOR loop is a more condensed way to perform a set number of repetitions since all of the necessary information is in a one line statement.

A FOR loop can also be given no conditions, for example:

```
for(;;) {
  /* block of statements */
}
```

This is called a forever loop since it will loop forever unless there is a break statement within the statements of the for loop. The empty test condition effectively evaluates as true.

It is also common to use the comma operator in for loops to execute multiple statements.

```
int i, j, n = 10;
for(i = 0, j = 0; i <= n; i++,j+=2)
     printf("i = %d , j = %d \n",i,j);
```

Do-While loops

A DO-WHILE loop is a post-check while loop, which means that it checks the condition after each run. As a result, even if the condition is zero (false), it will run at least once. It follows the form of:

```
do
{
   /* do stuff */
} while (condition);
```

Note the terminating semicolon. This is required for correct syntax. Since this is also a type of while loop, **break** and **continue** statements within the loop function accordingly. A **continue** statement causes a jump to the test of the condition and a *break* statement exits the loop.

It is worth noting that Do-While and While are functionally almost identical, with one important difference: Do-While loops are always guaranteed to execute at least once, but While loops will not execute at all if their condition is 0 (false) on the first evaluation.

One last thing: goto

goto is a very simple and traditional control mechanism. It is a statement used to immediately and unconditionally jump to another line of code. To use goto, you must place a label at a point in your program. A label consists of a name followed by a colon (:) on a line by itself. Then, you can type "goto *label*;" at the desired point in your program. The code will then continue executing beginning with *label*. This looks like:

```
  MyLabel:
     /* some code */
  goto MyLabel;
```

The ability to transfer the flow of control enabled by gotos is so powerful that, in addition to the simple if, all other control constructs can be written using gotos instead. Here, we can let "S" and "T" be any arbitrary statements:

```
  if (cond) {
    S;
  }
  else {
    T;
  }
  /* ... */
```

The same statement could be accomplished using two gotos and two labels:

```
  if (cond) goto Label1:
  T;
  goto Label2;
Label1:
  S;
Label2:
  /* ... */
```

Here, the first goto is conditional on the value of "cond". The second goto is unconditional. We can perform the same translation on a loop:

```
  while (cond1) {
     S;
     if (cond2) break;
     T;
  }
  /* ... */
```

Which can be written as:

```
Start:
  if (!cond1) goto End;
  S;
  if (cond2) goto End;
  T;
  goto Start;
End:
  /* ... */
```

As these cases demonstrate, often the structure of what your program is doing can usually be expressed without using gotos. Undisciplined use of gotos can create unreadable, unmaintainable code when more idiomatic alternatives (such as if-elses, or for loops) can better express your structure. Theoretically, the goto construct does not ever *have* to be used, but there are cases when it can increase readability, avoid code duplication, or make control variables unnecessary. You should consider first mastering the idiomatic solutions first, and use goto only when necessary. Keep in mind that many, if not most, C style guidlines *strictly forbid* use of **goto**, with the the only common exceptions being the following examples.

One use of goto is to break out of a deeply nested loop. Since **break** will not work (it can only escape one loop), **goto** can be used to jump completely outside the loop. Breaking outside of deeply nested loops without the use of the goto is always possible, but often involves the creation and testing of extra variables that may make the resulting code far less readable than it would be with **goto**. The use of **goto** makes it easy to undo actions in an orderly fashion, typically to avoid failing to free memory that had been allocated.

Another accepted use is the creation of a state machine. This is a fairly advanced topic though, and not commonly needed.

Examples

← Control | Preprocessor →

Procedures and Functions

A **function** is a section of code that has some separate functionality or does some function that will be reused over and over again.

As a basic example, if you are writing code to print out the first 5 squares of numbers, then the first 5 cubes, then the next 5 squares again, instead of writing something like

```
for(i=1; i <= 5; i++)
{
   printf("%d ", i*i);
}
for(i=1; i <= 5; i++)
{
   printf("%d ", i*i*i);
}
for(i=1; i <= 5; i++)
{
   printf("%d ", i*i);
}
```

which duplicates the same loop twice. We may want to separate this code somehow and simply jump to this code when we want its functionality.

This is what precisely functions are for.

More on functions

A function is like a black box. It takes in an input, does something to that input, then spits out an answer.

Note that a function may not take any inputs at all, or it may not return anything at all. In the above example, if we were to make a function of that loop, we may not need any inputs, and we aren't returning anything at all (Text output doesn't count - when we speak of *returning* we mean to say meaningful data that the program that used the function can use).

We have some terminology to refer to functions:

- A function, call it *f*, that uses another function *g*, is said to *call g*. For example, *f* calls *g* to print the squares of ten numbers.
- A function's inputs are known as its *arguments*
- A function that wants to give *f* back some data that *g* calculated is said to *return* that data. For example, *g* returns the sum of its arguments.

Writing functions in C

It's always good to learn by example. Let's write a function that will return the square of a number.

```
int
square(int x)
{
   int square_of_x;
   square_of_x = x * x;
   return square_of_x;
}
```

To understand how to write such a function like this, it may help to look at what this function does as a whole. It takes in an `int`, x, and squares it, storing it in the variable square_of_x. Now this value is returned.

The first int at the beginning of the function declaration is the type of data that the function returns. In this case when we square an integer we get an integer, and we are returning this integer, and so we write `int` as the return type.

Next is the name of the function. It is good practice to use meaningful and descriptive names for functions you may write. It may help to name the function after what it is written to do. In this case we name the function "square", because that's what it does - it squares a number.

Next is the function's first and only argument, an `int`, which will be referred to in the function as x. This is the function's *input*.

Inbetween the braces is the actual guts of the function. It declares an integer variable called square_of_x that will be used to hold the value of the square of x. Note that the variable square_of_x can **only** be used within this function, and not outside. We'll learn more about this sort of thing later, and we will see that this property is very useful.

We then assign x multiplied by x, or x squared, to the variable square_of_x, which is what this function is all about. Following this is a `return` statement. We want to return the value of the square of x, so we must say that this function returns the contents of the variable square_of_x.

Our brace to close, and we have finished the declaration.

Note this should look familiar - you have been writing functions already, in fact - main is a function that is always written.

In general

In general, if we want to declare a function, we write

```
type
name(type1 arg1, type2 arg2, ...)
{
  /* code */
}
```

We've previously said that a function can take no arguments, or can return nothing, or both. What do we write for the type of nothing? We use C's `void` keyword. `void` basically means "nothing" - so if we want to write a function that returns nothing, for example, we write

```
void
sayhello(int number_of_times)
{
  int i;
  for(i=1; i <= number_of_times; i++)
     printf("Hello!\n");
}
```

Notice that there is no `return` statement in the function above. Since there's none, we write `void` as the return type.

What about a function that takes no arguments? If we want to do this, we can write for example

```
float
calculate_number(void)
{
  float to_return=1; int i;
  for(i=0; i < 100; i++)
  {
     to_return += 1;
     to_return = 1/to_return;
  }
  return to_return;
```

```
 }
```

Notice this function doesn't take any inputs, but merely returns a number calculated by this function.

Naturally, you can combine both void return and void in arguments together to get a valid function, also.

Recursion

Here's a simple function that does an infinite loop. It prints a line and calls itself, which again prints a line and calls itself again, and this continues until the stack overflows and the program crashes. A function calling itself is called recursion, and normally you will have a conditional that would stop the recursion after a small, finite number of steps.

```
void infinite_recursion()
     \\ don't run that!
 {
      printf("Infinite loop!\n");
      infinite_recursion();
 }
```

A simple check can be done like this. Note that ++depth is used so the increment will take place before the value is passed into the function. Alternatively you can increment on a separate line before the recursion call. If you say print_me(3,0); the function will print the line Recursion 3 times.

```
void print_me(int j, int depth)
{
   if(depth < j)
   {
       printf("Recursion! depth = %d j = %d\n",depth,j);//j always the same
       print_me(j, ++depth);
   }
}
```

Recursion is most often used for jobs such as directory tree scans, seeking for the end of a linked list, parsing a tree structure in a database and factorising numbers (and finding primes) among other things.

Static Functions

If a function is to be called only from within the file in which it is declared, it is appropriate to declare it as a static function. When a function is declared static, the compiler will now compile to an object file in a way that prevents the function from being called from code in other files. Example:

```
static short compare( short a, short b )
{
    return (a+4 < b)? a : b;
}
```

Using C functions

We can now *write* functions, but how do we use them? When we write main, we place the function outside the braces that encompass main.

When we want to use that function, say, using our `calculate_number` function above, we can write something like

```
float f;
f = calculate_number();
```

If a function takes in arguments, we can write something like

```
int square_of_10;
square_of_10 = square(10);
```

If a function doesn't return anything, we can just say

```
say_hello();
```

since we don't need a variable to catch its return value.

C's Built-in Functions

← Procedures and functions | Libraries →

Preprocessors are a way of making text processing with your C program before they are actually compiled. Before the actual compilation of every C program it is passed through a Preprocessor. The Preprocessor looks through the program trying to find out specific instructions called Preprocessor directives that it can understand. All Preprocessor directives begin with the # (hash) symbol.

The preprocessor is a part of the compiler which performs preliminary operations (conditionally compiling code, including files etc...) to your code before the compiler sees it. These transformations are lexical, meaning that the output of the preprocessor is still text.

NOTE: Technically the output of the preprocessing phase for C consists of a sequence of tokens, rather than source text, but it is simple to output source text which is equivalent to the given token sequence, and that is commonly supported by compilers via a `-E` or `/E` option -- although command line options to C compilers aren't completely standard, many follow similar rules.

Directives

Directives are special instructions directed to the preprocessor (preprocessor directive) or to the compiler (compiler directive) on how it should process part or all of your source code or set some flags on the final object and are used to make writing source code easier (more portable for instance) and to make the source code more understandable. Directives are handled by the preprocessor, which is either a separate program invoked by the compiler or part of the compiler itself.

#include

C has some features as part of the language and some others as part of a **standard library**, which is a repository of code that is available alongside every standard-conformant C compiler. When the C compiler compiles your program it usually also links it with the standard C library. For example, on encountering a `#include <stdio.h>` directive, it replaces the directive with the contents of the `stdio.h` header file.

When you use features from the library, C requires you to *declare* what you would be using. The first line in the program is a **preprocessing directive** which should look like this:

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

The above line causes the C declarations which are in the `stdio.h` header to be included for use in your program. Usually this is implemented by just inserting into your program the contents of a **header file** called `stdio.h` in a system-dependent location. The location of such files may be described in your compiler's documentation. A list of standard C header files is listed below in the Headers table.

The `stdio.h` header contains various declarations for input/output (I/O) using an abstraction of I/O mechanisms called **streams**. For example there is an output stream object called `stdout` which is used to output text to the standard output, which usually displays the text on the computer screen.

If using angle brackets like the example above, the preprocessor is instructed to search for the include file along the development environment path for the standard includes.

```
#include "other.h"
```

If you use quotation marks (`" "`), the preprocessor is expected to search in some additional, usually user-defined, locations for the header file, and to fall back to the standard include paths only if it is not found in those additional locations. It is common for this form to include searching in the same directory as the file containing the `#include` directive.

NOTE: You should check the documentation of the development environment you are using for any vendor specific implementations of the `#include` directive.

Headers

The C90 standard headers list:

- `assert.h`
- `ctype.h`
- `errno.h`
- `float.h`
- `limits.h`
- `locale.h`
- `math.h`
- `setjmp.h`
- `signal.h`
- `stdarg.h`
- `stddef.h`
- `stdio.h`
- `stdlib.h`
- `string.h`
- `time.h`

Headers added since C90:

- `complex.h`
- `iso646.h`
- `tgmath.h`

▪ `fenv.h`	▪ `stdbool.h`	▪ `wchar.h`
▪ `inttypes.h`	▪ `stdint.h`	▪ `wctype.h`

#pragma

The **pragma** (pragmatic information) directive is part of the standard, but the meaning of any pragma depends on the software implementation of the standard that is used.

Pragmas are used within the source program.

```
#pragma token(s)
```

You should check the software implementation of the C standard you intend on using for a list of the supported tokens.

For instance one of the most implemented preprocessor directives, `#pragma once` when placed at the beginning of a header file, indicates that the file where it resides will be skipped if included several times by the preprocessor.

NOTE: Other methods exist to do this action that is commonly refered as using **include guards**.

#define

The `#define` directive is used to define values or macros that are used by the preprocessor to manipulate the program source code before it is compiled. Because preprocessor definitions are substituted before the compiler acts on the source code, any errors that are introduced by `#define` are difficult to trace.

By convention, values defined using `#define` are named in uppercase. Although doing so is not a requirement, it is considered very bad practice to do otherwise. This allows the values to be easily identified when reading the source code.

Today, `#define` is primarily used to handle compiler and platform differences. E.g, a define might hold a constant which is the appropriate error code for a system call. The use of `#define` should thus be limited unless absolutely necessary; `typedef` statements and constant variables can often perform the same functions more safely.

Another feature of the `#define` command is that it can take arguments, making it rather useful as a pseudo-function creator. Consider the following code:

```
#define ABSOLUTE_VALUE( x ) ( ((x) < 0) ? -(x) : (x) )
...
int x = -1;
while( ABSOLUTE_VALUE( x ) ) {
...
}
```

It's generally a good idea to use extra parentheses when using complex macros. Notice that in the above example, the variable "x" is always within its own set of parentheses. This way, it will be evaluated in whole, before being compared to 0 or multiplied by -1. Also, the entire macro is surrounded by parentheses, to prevent it from being contaminated by other code. If you're not careful, you run the risk of having the compiler misinterpret your code.

Because of side-effects it is considered a very bad idea to use macro functions as described above.

```
int x = -10;
int y = ABSOLUTE_VALUE( x++ );
```

If ABSOLUTE_VALUE() was a real function 'x' would now have the value of '-9', but because it was an argument in a macro it was expanded 3 times (in this case) and thus has a value of -7.

NOTE: Try to use `const` and `inline` instead of `#define`.
It is common practice when using `#define` and macros to name them all upper and use "`_`" separators, this will make clear to a reader that the value is not alterable and in case of a macro, that the construct requires care, some subtle errors can be created if using enum and macros with the same name. A const is easier to debug, too, since the compiler and linker recognizes a constant variable name, unlike a macro.

Example:

To illustrate the dangers of macros, consider this naive macro

```
#define MAX(a,b) a>b?a:b
```

and the code

```
i = MAX(2,3)+5;
j = MAX(3,2)+5;
```

Take a look at this and consider what the the value after execution might be. The statements are turned into

```
int i = 2>3?2:3+5;
int j = 3>2?3:2+5;
```

Thus, after execution `i=8` and `j=3` instead of the expected result of `i=j=8`! This is why you were cautioned to use an extra set of parenthesis above, but even with these, the road is fraught with dangers. The alert reader might quickly realize that if `a,b` contains expressions, the definition must parenthesise every use of `a,b` in the macro defintion, like this:

```
#define MAX(a,b) ((a)>(b)?(a):(b))
```

This works, provided `a,b` have no side effects. Indeed,

```
i = 2;
j = 3;
k = MAX(i++, j++);
```

would result in `k=4`, `i=3` and `j=5`. This would be highly surprising to anyone expecting `MAX()` to behave like a function.

So what is the correct solution? The solution is not to use macro at all. A global, inline function, like this

```
inline max(int a, int b) { return a>b?a:b }
```

has none of the pitfalls above, but will not work with all types.

(#, ##)

The # and ## operators are used with the `#define` macro. Using # causes the first argument after the # to be returned as a string in quotes. For example, the command

```
#define as_string( s ) # s
```

will make the compiler turn this command

```
puts( as_string( Hello World! ) ) ;
```

into

```
puts( "Hello World!" );
```

Using ## concatenates what's before the ## with what's after it. For example, the command

```
#define concatenate( x, y ) x ## y
...
int xy = 10;
...
```

will make the compiler turn

```
printf( "%d", concatenate( x, y ));
```

into

```
printf( "%d", xy);
```

which will, of course, display `10` to standard output.

It is possible to concatenate a macro argument with a constant prefix or suffix to obtain a valid identifier as in

```
#define make_function( name ) int my_ ## name (int foo) {}
make_function( bar )
```

which will define a function called `my_bar()`. But it isn't possible to integrate a macro argument into a constant string using the concatenation operator. In order to obtain such an effect, one can use the ANSI C property that two or more consecutive string constants are considered equivalent to a single string constant when encountered. Using this property, one can write

```
#define eat( what ) puts( "I'm eating " #what " today." )
eat( fruit )
```

which the macro-processor will turn into

```
puts( "I'm eating " "fruit" " today." )
```

which in turn will be interpreted by the C parser as a single string constant.

macros

Macros aren't type-checked and so they do not evaluate arguments. Also, they do not obey scope properly, but simply take the string passed to them and replace each occurrence of the macro argument in the text of the macro with the actual string for that parameter (the code is literally copied into the location it was called from).

An example on how to use a macro:

```
#include <stdio.h>

#define SLICES 8
#define ADD(x) ( (x) / SLICES )

int main()
{
  int a = 0, b = 10, c = 6;

  a = ADD(b + c);
  printf("%d\n", a);
  return 0;
}
```

-- the result of "a" should be "2" (b + c = 16 -> passed to ADD -> 16 / SLICES -> result is "2")

NOTE:
It is usually bad practice to define macros in headers.

A macro should be defined only when it is not possible to achieve the same result with a function or some other mechanism. Some compilers are able to optimize code to where calls to small functions are replaced with inline code, negating any possible speed advantage. Using typedefs, enums, and `inline` (in C99) is often a better option.

#error

The **#error** directive halts compilation. When one is encountered the standard specifies that the compiler should emit a diagnostic containing the remaining tokens in the directive. This is mostly used for debugging purposes.

```
#error message
```

#undef

The **#undef** directive undefines a macro. The identifier need not have been previously defined.

if,else,elif,endif (conditionals)

The **#if** command checks whether a controlling conditional expression evaluates to zero or nonzero, and excludes or includes a block of code respectively. For example:

```
#if 1
/* This block will be included */
#endif
#if 0
/* This block will not be included */
#endif
```

The conditional expression could contain any C operator except for the assignment operators, increment, and decrement operators.

One unique operator used in preprocessing and nowhere else is the **defined** operator. It returns 1 if the macro name, optionally enclosed in parentheses, is currently defined; 0 if not.

The **#endif** command ends a block started by `#if`, `#ifdef`, or `#ifndef`.

The **#elif** command is similar to `#if`, except that it is used to extract one from a series of blocks of code. E.g.:

```
#if /* some expression */
  :
  :
  :
#elif /* another expression */
```

```
   :

/* imagine many more #elifs here ... */

   :
#else
/* The optional #else block is selected if none of the previous #if or
   #elif blocks are selected */
   :
   :
#endif /* The end of the #if block */
```

ifdef,ifndef

The **#ifdef** command is similar to `#if`, except that the code block following it is selected if a macro name is defined. In this respect,

```
#ifdef NAME
```

is equivalent to

```
#if defined NAME
```

.

The **#ifndef** command is similar to **#ifdef**, except that the test is reversed:

```
#ifndef NAME
```

is equivalent to

```
#if !defined NAME
```

.

← Preprocessor | Standard libraries →

A *library* in C is merely a group of functions and declarations. The library has an *interface* expressed in a file with a `.h` extension and an *implementation* expressed in a file with a `.c` extension (which may be precompiled or otherwise inaccessible).

Libraries may call functions in other libraries such as the Standard C or math libraries to do various tasks.

For example, suppose you want to write a function to parse arguments from the command line. Arguments on the command line could be by themselves:

```
    -i
```

have an optional argument that is concatenated to the letter:

```
    -ioptarg
```

or have the argument in a separate argv-element:

```
    -i optarg
```

Suppose you want the ability to bunch switches in one argv-element as well. Anyway, after much writing, you come up with this:

```
#include <stdio.h>              /* for fprintf() and EOF */
#include <string.h>             /* for strchr() */
/* variables */
int opterr = 1;                 /* getopt prints errors if this is on */
int optind = 1;                 /* token pointer */
int optopt;                     /* option character passed back to user */
char *optarg;                   /* flag argument (or value) */
/* function */
/* return option character, EOF if no more or ? if problem.
   The arguments to the function:
   argc, argv - the arguments to the main() function. An argument of "--"
   stops the processing.
   opts - a string containing the valid option characters.
   an option character followed by a colon (:) indicates that
   the option has a required argument.
 */
int
getopt (int argc, char **argv, char *opts)
{
  static int sp = 1;            /* character index into current token */
  register char *cp;            /* pointer into current token */
  if (sp == 1)
    {
      /* check for more flag-like tokens */
      if (optind >= argc || argv[optind][0] != '-' || argv[optind][1] == '\0')
        return EOF;
      else if (strcmp (argv[optind], "--") == 0)
        {
          optind++;
          return EOF;
        }
    }
  optopt = argv[optind][sp];
  if (optopt == ':' || (cp = strchr (opts, optopt)) == NULL)
    {
      if (opterr)
        fprintf (stderr, "%s: invalid option -- '%c'\n", argv[0], optopt);
      /* if no characters left in this token, move to next token */
      if (argv[optind][++sp] == '\0')

        {

          optind++;
          sp = 1;
        }
      return '?';
    }
  if (*++cp == ':')
    {
      /* if a value is expected, get it */
      if (argv[optind][sp + 1] != '\0')
        /* flag value is rest of current token */
        optarg = argv[optind++] + (sp + 1);
      else if (++optind >= argc)
        {
          if (opterr)
            fprintf (stderr, "%s: option requires an argument -- '%c'\n",
                     argv[0], optopt);
          sp = 1;
          return '?';
        }
      else
        /* flag value is next token */
        optarg = argv[optind++];
      sp = 1;
    }
  else
    {
      /* set up to look at next char in token, next time */
      if (argv[optind][++sp] == '\0')
        {
          /* no more in current token, so setup next token */
          sp = 1;
          optind++;
        }
      optarg = 0;
    }
  return optopt;
}
/* END OF FILE */
```

The implementation would be the code above. The interface would be the following:

```
#ifndef GETOPT_H
#define GETOPT_H
/* exported variables */
extern int opterr, optind, optopt;
extern char *optarg;
/* exported function */
int getopt(int, char **, char *);
#endif
/* END OF FILE */
```

All the programmer that is supposed to use this library sees (if he doesn't want to or can't look at the implementation) is the interface and the documentation that the library programmer wrote. The documentation should say that neither pointer can be null (or why would you be using the `getopt` function anyway?) and state what each parameter is for and the return value. The programmer that uses this library is not interested in the implementation of the library (unless the implementation has a bug, in which case he would want to complain somehow). ← Libraries | File IO →

Standard Libraries

Introduction

The `stdio.h` header declares a broad assortment of functions that perform input and output to files and devices such as the console. It was one of the earliest headers to appear in the C library. It declares more functions than any other standard header and also requires more explanation because of the complex machinery that underlies the functions.

The device-independent model of input and output has seen dramatic improvement over the years and has received little recognition for its success. FORTRAN II was touted as a machine-independent language in the 1960s, yet it was essentially impossible to move a FORTRAN program between architectures without some change. In FORTRAN II, you named the device you were talking to right in the FORTRAN statement in the middle of your FORTRAN code. So, you said `READ INPUT TAPE 5` on a tape-oriented IBM 7090 but `READ CARD` to read a card image on other machines. FORTRAN IV had more generic `READ` and `WRITE` statements, specifying a *logical unit number* (LUN) instead of the device name. The era of device-independent I/O had dawned.

Peripheral devices such as printers still had fairly strong notions about what they were asked to do. And then, *peripheral interchange* utilities were invented to handle bizarre devices. When cathode-ray tubes came onto the scene, each manufacturer of consoles solved problems such as console cursor movement in an independent manner, causing further headaches.

It was into this atmosphere that Unix was born. Ken Thompson and Dennis Ritchie, the developers of Unix, deserve credit for packing any number of bright ideas into the operating system. Their approach to device independence was one of the brightest.

The ANSI C `<stdio.h>` library is based on the original Unix file I/O primitives but casts a wider net to accommodate the least-common denominator across varied systems.

Streams

Input and output, whether to or from physical devices such as terminals and tape drives, or whether to or from files supported on structured storage devices, are mapped into logical data streams, whose properties are more uniform than their various inputs and outputs. Two forms of mapping are supported: text streams and binary streams.

A text stream is an ordered sequence of characters composed into lines, each line consisting of zero or more characters plus a terminating new-line character. Whether the last line requires a terminating new-line character is implementation-defined. Characters may have to be added, altered, or deleted on input and output to conform to differing conventions for representing text characters in a stream and those in the external representation. Data read in from a text stream will necessarily compare equal to the data that were earlier written out to that stream only if the data consist only of printable characters and the control characters horizontal tab and new-line, no new-line character is immediately preceded by space characters, and the last character is a new-line character. Whether space characters that are written out immediately before a new-line character appear when read in is implementation-defined.

Unix adopted a standard internal format for all text streams. Each line of text is terminated by a new-line character. That's what any program expects when it reads text, and that's what any program produces when it writes text. If such a convention doesn't meet the needs of a text-oriented peripheral attached to a Unix machine, then the fixup occurs out at the edges of the system. None of the code in the middle needs to change.

A binary stream is an ordered sequence of characters that can transparently record internal data. Data read in from a binary stream shall compare equal to the data that were earlier written out to that stream under the same implementation. Such a stream may, however, have an implementation-defined number of null characters appended to the end of the stream.

Nothing in Unix prevents the program from writing arbitrary 8-bit binary codes to any open file, or reading them back unchanged from an adequate repository. Thus, Unix obliterated the long-standing distinction between text streams and binary streams.

`FILE` pointers

The `<stdio.h>` header contains a definition for a type `FILE` (usually via a `typedef`) which is capable of recording all the information needed to control a stream, including its file position indicator, a pointer to the associated buffer (if any), an error indicator that records whether a read/write error has occurred, and an end-of-file indicator that records whether the end of the file has been reached.

It is considered bad manners to access the contents of `FILE` directly unless the programmer is writing an implementation of `<stdio.h>` and its contents. How, pray tell, is one going to know whether the file handle, for example, is spelt `handle` or `_Handle`? Access to the contents of `FILE` is better provided via the functions in `<stdio.h>`.

It can be said that the `FILE` type is an early example of object-oriented programming.

Opening and Closing Files

To open and close files, the `<stdio.h>` library has three functions: `fopen`, `freopen`, and `fclose`.

Opening Files

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

```
FILE *fopen(const char *filename, const char *mode);

FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

The `fopen` and `freopen` functions open files.

The `fopen` function opens the file whose name is in the string pointed to by `filename` and associates a stream with it.

The argument `mode` points to a string beginning with one of the following sequences:

```
r            open a text file for reading
w            truncate to zero length or create a text file for writing
a            append; open or create text file for writing at end-of-file
rb           open binary file for reading
wb           truncate to zero length or create a binary file for writing
ab           append; open or create binary file for writing at end-of-file
r+           open text file for update (reading and writing)
w+           truncate to zero length or create a text file for update
a+           append; open or create text file for update
r+b or rb+  open binary file for update (reading and writing)
w+b or wb+  truncate to zero length or create a binary file for update
a+b or ab+  append; open or create binary file for update
```

Opening a file with read mode ('`r`' as the first character in the `mode` argument) fails if the file does not exist or cannot be read.

Opening a file with append mode ('`a`' as the first character in the `mode` argument) causes all subsequent writes to the file to be forced to the then-current end-of-file, regardless of intervening calls to the `fseek` function. In some implementations, opening a binary file with append mode ('`b`' as the second or third character in the above list of `mode` arguments) may initially position the file position indicator for the stream beyond the last data written, because of null character padding.

When a file is opened with update mode ('`+`' as the second or third character in the above list of `mode` argument values), both input and output may be performed on the associated stream. However, output may not be directly followed by input without an intervening call to the `fflush` function or to a file positioning function (`fseek`, `fsetpos`, or `rewind`), and input may not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file. Opening (or creating) a text file with update mode may instead open (or create) a binary stream in some implementations.

When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators are cleared.

The `fopen` function returns a pointer to the object controlling the stream. If the open operation fails, `fopen` returns a null pointer.

The `freopen` function opens the file whose name is the string pointed to by `filename` and associates the stream pointed to by `stream` with it. The mode argument is used just as in the `fopen` function.

The `freopen` function first attempts to close any file that is associated with the specified stream. Failure to close the file successfully is ignored. The error and end-of-file indicators for the stream are cleared.

The `freopen` function returns a null pointer if the open operation fails, or the value `stream` if the open operation succeeds.

Closing Files

```
#include <stdio.h>
int fclose(FILE *stream);
```

The `fclose` function causes the stream pointed to by `stream` to be flushed and the associated file to be closed. Any unwritten buffered data for the stream are delivered to the host environment to be written to the file; any unread buffered data are discarded. The stream is disassociated from the file. If the associated buffer was automatically allocated, it is deallocated. The function returns zero if the stream was successfully closed or `EOF` if any errors were detected.

Other file access functions

The `fflush` function

```
#include <stdio.h>
int fflush(FILE *stream);
```

If `stream` points to an output stream or an update stream in which the most recent operation was not input, the `fflush` function causes any unwritten data for that stream to be deferred to the host environment to be written to the file; otherwise, the behavior is undefined.

If `stream` is a null pointer, the `fflush` function performs this flushing action on all streams for which the behavior is defined above.

The `fflush` functions returns `EOF` if a write error occurs, otherwise zero.

The reason for having a `fflush` function is because streams in C can have buffered input/output; that is, functions that write to a file actually write to a buffer inside the `FILE` structure. If the buffer is filled to capacity, the write functions will call `fflush` to actually "write" the data that is in the buffer to the file. Because `fflush` is only called every once in a while, calls to the operating system to do a raw write are minimized.

The `setbuf` function

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

Except that it returns no value, the `setbuf` function is equivalent to the `setvbuf` function invoked with the values `_IOFBF` for `mode` and `BUFSIZ` for `size`, or (if `buf` is a null pointer) with the value `_IONBF` for `mode`.

The `setvbuf` function

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

The `setvbuf` function may be used only after the stream pointed to by `stream` has been associated with an open file and before any other operation is performed on the stream. The argument `mode` determines how the stream will be buffered, as follows: `_IOFBF` causes input/output to be fully buffered; `_IOLBF` causes input/output to be line buffered; `_IONBF` causes input/output to be unbuffered. If `buf` is not a null pointer, the array it points to may be used instead of a buffer associated by the `setvbuf` function. (The buffer must have a lifetime at least as great as the open stream, so the stream should be closed before a buffer that has automatic storage duration is deallocated upon block exit.) The argument `size` specifies the size of the array. The contents of the array at any time are indeterminate.

The `setvbuf` function returns zero on success, or nonzero if an invalid value is given for `mode` or if the request cannot be honored.

Functions that Modify the File Position Indicator

The `stdio.h` library has five functions that affect the file position indicator besides those that do reading or writing: `fgetpos`, `fseek`, `fsetpos`, `ftell`, and `rewind`.

The `fseek` and `ftell` functions are older than `fgetpos` and `fsetpos`.

The `fgetpos` and `fsetpos` functions

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
```

The `fgetpos` function stores the current value of the file position indicator for the stream pointed to by `stream` in the object pointed to by `pos`. The value stored contains unspecified information usable by the `fsetpos` function for repositioning the stream to its position at the time of the call to the `fgetpos` function.

If successful, the `fgetpos` function returns zero; on failure, the `fgetpos` function returns nonzero and stores an implementation-defined positive value in `errno`.

The `fsetpos` function sets the file position indicator for the stream pointed to by `stream` according to the value of the object pointed to by `pos`, which shall be a value obtained from an earlier call to the `fgetpos` function on the same stream.

A successful call to the `fsetpos` function clears the end-of-file indicator for the stream and undoes any effects of the `ungetc` function on the same stream. After an `fsetpos` call, the next operation on an update stream may be either input or output.

If successful, the `fsetpos` function returns zero; on failure, the `fsetpos` function returns nonzero and stores an implementation-defined positive value in `errno`.

The `fseek` and `ftell` functions

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
long int ftell(FILE *stream);
```

The `fseek` function sets the file position indicator for the stream pointed to by `stream`.

For a binary stream, the new position, measured in characters from the beginning of the file, is obtained by adding `offset` to the position specified by `whence`. Three macros in `stdio.h` called `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` expand to unique values. If the position specified by `whence` is `SEEK_SET`, the specified position is the beginning of the file; if `whence` is `SEEK_END`, the specified position is the end of the file; and if `whence` is `SEEK_CUR`, the specified position is the current file position. A binary stream need not meaningfully support `fseek` calls with a `whence` value of `SEEK_END`.

For a text stream, either `offset` shall be zero, or `offset` shall be a value returned by an earlier call to the `ftell` function on the same stream and `whence` shall be `SEEK_SET`.

The `fseek` function returns nonzero only for a request that cannot be satisfied.

The `ftell` function obtains the current value of the file position indicator for the stream pointed to by `stream`. For a binary stream, the value is the number of characters from the beginning of the file; for a text stream, its file position indicator contains unspecified information, usable by the `fseek` function for returning the file position indicator for the stream to its position at the time of the `ftell` call; the difference between two such return values is not necessarily a meaningful measure of the number of characters written or read.

If successful, the `ftell` function returns the current value of the file position indicator for the stream. On failure, the `ftell` function returns `-1L` and stores an implementation-defined positive value in `errno`.

The `rewind` function

```
#include <stdio.h>
void rewind(FILE *stream);
```

The `rewind` function sets the file position indicator for the stream pointed to by `stream` to the beginning of the file. It is equivalent to

```
(void)fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared.

Error Handling Functions

The `clearerr` function

```
#include <stdio.h>
void clearerr(FILE *stream);
```

The `clearerr` function clears the end-of-file and error indicators for the stream pointed to by `stream`.

The `feof` function

```
#include <stdio.h>
int feof(FILE *stream);
```

The `feof` function tests the end-of-file indicator for the stream pointed to by `stream` and returns nonzero if and only if the end-of-file indicator is set for `stream`, otherwise it returns zero.

The `ferror` function

```
#include <stdio.h>
int ferror(FILE *stream);
```

The `ferror` function tests the error indicator for the stream pointed to by `stream` and returns nonzero if and only if the error indicator is set for `stream`, otherwise it returns zero.

The `perror` function

```
#include <stdio.h>
void perror(const char *s);
```

The `perror` function maps the error number in the integer expression `errno` to an error message. It writes a sequence of characters to the standard error stream thus: first, if `s` is not a null pointer and the character pointed to by `s` is not the null character, the string pointed to by `s` followed by a colon (`:`) and a space; then an appropriate error message string followed by a new-line character. The contents of the error message are the same as those returned by the `strerror` function with the argument `errno`, which are implementation-defined.

Other Operations on Files

The `stdio.h` library has a variety of functions that do some operation on files besides reading and writing.

The `remove` function

```
#include <stdio.h>
int remove(const char *filename);
```

The `remove` function causes the file whose name is the string pointed to by `filename` to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless it is created anew. If the file is open, the behavior of the `remove` function is implementation-defined.

The `remove` function returns zero if the operation succeeds, nonzero if it fails.

The `rename` function

```
#include <stdio.h>
int rename(const char *old_filename, const char *new_filename);
```

The `rename` function causes the file whose name is the string pointed to by `old_filename` to be henceforth known by the name given by the string pointed to by `new_filename`. The file named `old_filename` is no longer accessible by that name. If a file named by the string pointed to by `new_filename` exists prior to the call to the `rename` function, the behavior is implementation-defined.

The `rename` function returns zero if the operation succeeds, nonzero if it fails, in which case if the file existed previously it is still known by its original name.

The `tmpfile` function

```
#include <stdio.h>
FILE *tmpfile(void);
```

The `tmpfile` function creates a temporary binary file that will automatically be removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with `"wb+"` mode.

The `tmpfile` function returns a pointer to the stream of the file that it created. If the file cannot be created, the `tmpfile` function returns a null pointer.

The `tmpnam` function

```
#include <stdio.h>
char *tmpnam(char *s);
```

The `tmpnam` function generates a string that is a valid file name and that is not the name of an existing file.

The `tmpnam` function generates a different string each time it is called, up to `TMP_MAX` times. (`TMP_MAX` is a macro defined in `stdio.h`.) If it is called more than `TMP_MAX` times, the behavior is implementation-defined.

The implementation shall behave as if no library function calls the `tmpnam` function.

If the argument is a null pointer, the `tmpnam` function leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to the `tmpnam` function may modify the same object. If the argument is not a null pointer, it is assumed to point to an array of at least `L_tmpnam` characters (`L_tmpnam` is another macro in `stdio.h`); the `tmpnam` function writes its result in that array and returns the argument as its value.

The value of the macro `TMP_MAX` must be at least 25.

Reading from Files

Character Input Functions

The `fgetc` function

```
#include <stdio.h>
int fgetc(FILE *stream);
```

The `fgetc` function obtains the next character (if present) as an `unsigned char` converted to an `int`, from the input stream pointed to by `stream`, and advances the associated file position indicator for the stream (if defined).

The `fgetc` function returns the next character from the input stream pointed to by `stream`. If the stream is at end-of-file, the end-of-file indicator for the stream is set and `fgetc` returns `EOF` (`EOF` is a negative value defined in `<stdio.h>`, usually `(-1)`). If a read error occurs, the error indicator for the stream is set and `fgetc` returns `EOF`.

The `fgets` function

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

The `fgets` function reads at most one less than the number of characters specified by `n` from the stream pointed to by `stream` into the array pointed to by `s`. No additional characters are read after a new-line character (which is retained) or after end-of-file. A null character is written immediately after the last character read into the array.

The `fgets` function returns `s` if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointed is returned.

Warning: some filesystems use the terminator `\r\n` in text files; `fgets` will read those lines, removing the `\n` but keeping the `\r` as the last character of `s`. This expurious character should be removed in the string `s` before the string is used for anything.

The `getc` function

```
#include <stdio.h>
int getc(FILE *stream);
```

The `getc` function is equivalent to `fgetc`, except that it may be implemented as a macro. If it is implemented as a macro, the `stream` argument may be evaluated more than once, so the argument should never be an expression with side effects (i.e. have an assignment, increment, or decrement operators, or be a function call).

The `getc` function returns the next character from the input stream pointed to by `stream`. If the stream is at end-of-file, the end-of-file indicator for the stream is set and `getc` returns `EOF` (`EOF` is a negative value defined in `<stdio.h>`, usually `(-1)`). If a read error occurs, the error indicator for the stream is set and `getc` returns `EOF`.

The `getchar` function

```
#include <stdio.h>
int getchar(void);
```

The `getchar` function is equivalent to `getc` with the argument `stdin`.

The `getchar` function returns the next character from the input stream pointed to by `stdin`. If `stdin` is at end-of-file, the end-of-file indicator for `stdin` is set and `getchar` returns `EOF` (`EOF` is a negative value defined in `<stdio.h>`, usually `(-1)`). If a read error occurs, the error indicator for `stdin` is set and `getchar` returns `EOF`.

The `gets` function

```
#include <stdio.h>
char *gets(char *s);
```

The `gets` function reads characters from the input stream pointed to by `stdin` into the array pointed to by `s` until an end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

The `gets` function returns `s` if successful. If the end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

This function and description is only included here for completeness. Most C programmers nowadays shy away from using `gets`, as there is no way for the function to know how big the buffer is that the programmer wants to read into. Commandment #5 of Henry Spencer's *The Ten Commandments for C Programmers (Annotated Edition)* reads, "Thou shalt check the array bounds of all strings (indeed, all arrays), for surely where thou typest *foo* someone someday shall type *supercalifragilisticexpialidocious*." It mentions `gets` in the annotation: "As demonstrated by the deeds of the Great Worm, a consequence of this commandment is that robust production software should never make use of `gets()`, for it is truly a tool of the Devil. Thy interfaces should always inform thy servants of the bounds of thy arrays, and servants who spurn such advice or quietly fail to follow it should be dispatched forthwith to the Land Of Rm, where they can do no further harm to thee."

The `ungetc` function

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

The `ungetc` function pushes the character specified by `c` (converted to an `unsigned char`) back onto the input stream pointed to by stream. The pushed-back characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by `stream`) to a file-positioning function (`fseek`, `fsetpos`, or `rewind`) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

One character of pushback is guaranteed. If the `ungetc` function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

If the value of `c` equals that of the macro `EOF`, the operation fails and the input stream is unchanged.

A successful call to the `ungetc` function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back characters shall be the same as it was before the characters were pushed back. For a text stream, the value of its file-position indicator after a successful call to the `ungetc` function is unspecified until all pushed-back characters are read or discarded. For a binary stream, its file position indicator is decremented by each successful call to the `ungetc` function; if its value was zero before a call, it is indeterminate after the call.

The `ungetc` function returns the character pushed back after conversion, or `EOF` if the operation fails.

Direct input function: the `fread` function

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

The `fread` function reads, into the array pointed to by `ptr`, up to `nmemb` elements whose size is specified by `size`, from the stream pointed to by `stream`. The file position indicator for the stream (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. If a partial element is read, its value is indeterminate.

The `fread` function returns the number of elements successfully read, which may be less than `nmemb` if a read error or end-of-file is encountered. If `size` or `nmemb` is zero, `fread` returns zero and the contents of the array and the state of the stream remain unchanged.

Formatted input functions: the `scanf` family of functions

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
int scanf(const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

The `fscanf` function reads input from the stream pointed to by `stream`, under control of the string pointed to by `format` that specifies the admissible sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: one or more white-space characters; an ordinary multibyte character (neither `%` or a white-space character); or a conversion specification. Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

- An optional assignment-suppressing character `*`.
- An optional nonzero decimal integer that specifies the maximum field width.
- An optional `h`, `l` (ell) or `L` indicating the size of the receiving object. The conversion specifiers `d`, `i`, and `n` shall be preceded by `h` if the corresponding argument is a pointer to `short int` rather than a pointer to `int`, or by `l` if it is a pointer to `long int`. Similarly, the conversion specifiers `o`, `u`, and `x` shall be preceded by `h` if the corresponding argument is a pointer to `unsigned short int` rather than `unsigned int`, or by `l` if it is a pointer to `unsigned long int`. Finally, the conversion specifiers `e`, `f`, and `g` shall be preceded by `l` if the corresponding argument is a pointer to `double` rather than a pointer to `float`, or by `L` if it is a pointer to `long double`. If an `h`, `l`, or `L` appears with any other format specifier, the behavior is undefined.
- A character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The `fscanf` function executes each directive of the format in turn. If a directive fails, as detailed below, the `fscanf` function returns. Failures are described as input failures (due to the unavailability of input characters) or matching failures (due to inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread) or until no more characters remain unread.

A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space characters (as specified by the `isspace` function) are skipped, unless the specification includes a `[`, `c`, or `n` specifier. (The white-space characters are not counted against the specified field width.)

An input item is read from the stream, unless the specification includes an `n` specifier. An input item is defined as the longest matching sequences of input characters, unless that exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.

Except in the case of a `%` specifier, the input item (or, in the case of a `%n` directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails; this condition is a matching failure. Unless assignment suppression was indicated by a `*`, the result of the conversion is placed in the object pointed to by the first argument following the `format` argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion specifiers are valid:

`d`
: Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the `strtol` function with the value 10 for the `base` argument. The corresponding argument shall be a pointer to integer.

`i`
: Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the `strtol` function with the value 0 for the `base` argument. The corresponding argument shall be a pointer to integer.

`o`
: Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 8 for the `base` argument. The corresponding argument shall be a pointer to unsigned integer.

`u`
: Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 10 for the `base` argument. The corresponding argument shall be a pointer to unsigned integer.

`x`
: Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 16 for the `base` argument. The corresponding argument shall be a pointer to unsigned integer.

`e`, `f`, `g`
: Matches an optionally signed floating-point number, whose format is the same as expected for the subject string of the `strtod` function. The corresponding argument will be a pointer to floating.

`s`

Matches a sequence of non-white-space characters. (No special provisions are made for multibyte characters.) The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically.

`[`

Matches a nonempty sequence of characters (no special provisions are made for multibyte characters) from a set of expected characters (the *scanset*). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically. The conversion specifier includes all subsequent characters in the `format` string, up to and including the matching right bracket (`]`). The characters between the brackets (the *scanlist*) comprise the scanset, unless the character after the left bracket is a circumflex (`^`), in which case the scanset contains all the characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with `[]` or `[^]`, the right-bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification; otherwise, the first right bracket character is the one that ends the specification. If a `-` character is in the scanlist and is not the first, nor the second where the first character is a `^`, nor the last character, the behavior is implementation-defined.

`c`

Matches a sequence of characters (no special provisions are made for multibyte characters) of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence. No null character is added.

`p`

Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the `%p` conversion of the `fprintf` function. The corresponding argument shall be a pointer to `void`. The interpretation of the input then is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the `%p` conversion is undefined.

`n`

No input is consumed. The corresponding argument shall be a pointer to integer into which is to be written the number of characters read from the input stream so far by this call to the `fscanf` function. Execution of a `%n` directive does not increment the assignment count returned at the completion of execution of the `fscanf` function.

`%`

Matches a single `%`; no conversion or assignment occurs. The complete conversion specification shall be `%%`.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers `E`, `G`, and `X` are also valid and behave the same as, respectively, `e`, `g`, and `x`.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the `%n` directive.

The `fscanf` function returns the value of the macro `EOF` if an input failure occurs before any conversion. Otherwise, the `fscanf` funciton returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

The `scanf` function is equivalent to `fscanf` with the argument `stdin` interposed before the arguments to `scanf`. Its return value is similar to that of `fscanf`.

The `sscanf` function is equivalent to `fscanf`, except that the argument `s` specifies a string from which the input is to be obtained, rather than from a stream. Reaching the end of the string is equivalent to encountering the end-of-file for the `fscanf` function. If copying takes place between objects that overlap, the behavior is undefined.

Writing to Files

Character Output Functions

The `fputc` function

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

The `fputc` function writes the character specified by `c` (converted to an `unsigned char`) to the stream pointed to by `stream` at the position indicated by the associated file position indicator (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream is opened with append mode, the character is appended to the output stream. The function returns the character written, unless a write error occurs, in which case the error indicator for the stream is set and `fputc` returns `EOF`.

The `fputs` function

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

The `fputs` function writes the string pointed to by `s` to the stream pointed to by `stream`. The terminating null character is not written. The function returns `EOF` if a write error occurs, otherwise it returns a nonnegative value.

The `putc` function

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

The `putc` function is equivalent to `fputc`, except that if it is implemented as a macro, it may evaluate `stream` more than once, so the argument should never be an expression with side effects. The function returns the character written, unless a write error occurs, in which case the error indicator for the stream is set and the function returns `EOF`.

The `putchar` function

```
#include <stdio.h>
int putchar(int c);
```

The `putchar` function is equivalent to `putc` with the second argument `stdout`. It returns the character written, unless a write error occurs, in which case the error indicator for `stdout` is set and the function returns `EOF`.

The `puts` function

```
#include <stdio.h>
int puts(const char *s);
```

The `puts` function writes the string pointed to by `s` to the stream pointed to by `stdout`, and appends a new-line character to the output. The terminating null character is not written. The function returns `EOF` if a write error occurs; otherwise, it returns a nonnegative value.

Direct output function: the `fwrite` function

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

The `fwrite` function writes, from the array pointed to by `ptr`, up to `nmemb` elements whose size is specified by `size` to the stream pointed to by `stream`. The file position indicator for the stream (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. The function returns the number of elements successfully written, which will be less than `nmemb` only if a write error is encountered.

Formatted output functions: the `printf` family of functions

```
#include <stdarg.h>
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
int printf(const char *format, ...);
int sprintf(char *s, const char *format, ...);
int vfprintf(FILE *stream, const char *format, va_list arg);
int vprintf(const char *format, va_list arg);
int vsprintf(char *s, const char *format, va_list arg);
```

Note: Some length specifiers and format specifiers are new in C99. These may not be available in older compilers and versions of the stdio library, which adhere to the C89/C90 standard. Wherever possible, the new ones will be marked with (C99).

The `fprintf` function writes output to the stream pointed to by `stream` under control of the string pointed to by `format` that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The `fprintf` function returns when the end of the format string is encountered.

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not `%`), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

- Zero or more flags (in any order) that modify the meaning of the conversion specification.
- An optional minimum field width. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk `*` (described later) or a decimal integer. (Note that 0 is taken as a flag, not as the beginning of a field width.)
- An optional precision that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, and `X` conversions, the number of digits to appear after the decimal-point character for `a`, `A`, `e`, `E`, `f`, and `F` conversions, the maximum number of significant digits for the `g` and `G` conversions, or the maximum number of characters to be written from a string in `s` conversions. The precision takes the form of a period (`.`) followed either by an asterisk `*` (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- An optional length modifier that specifies the size of the argument.
- A conversion specifier character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an `int` argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a `-` flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag characters and their meanings are:

`-`
: The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)

`+`
: The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not specified. The results of all floating conversions of a negative zero, and of negative values that round to zero, include a minus sign.)

space
: If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the space and + flags both appear, the space flag is ignored.

`#`
: The result is converted to an "alternative form". For `o` conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For `x` (or `X`) conversion, a nonzero result has `0x` (or `0X`) prefixed to it. For `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, the result always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For `g` and `G` conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

`0`
: For `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the `0` and `-` flags both appear, the `0` flag is ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision is specified, the `0` flag is ignored. For other conversions, the behavior is undefined.

The length modifiers and their meanings are:

`hh`
: (C99) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `signed char` or `unsigned char` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `signed char` or `unsigned char` before printing); or that a following `n` conversion specifier applies to a pointer to a `signed char` argument.

`h`
: Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `short int` or `unsigned short int` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `short int` or `unsigned short int` before printing); or that a following `n` conversion specifier applies to a pointer to a `short int` argument.

`l` (ell)
: Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long int` or `unsigned long int` argument; that a following `n` conversion specifier applies to a pointer to a `long int` argument; (C99) that a following `c` conversion specifier applies to a `wint_t` argument; (C99) that a following `s` conversion specifier applies to a pointer to a `wchar_t` argument; or has no effect on a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier.

`ll` (ell-ell)
: (C99) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long long int` or `unsigned long long int` argument; or that a following `n` conversion specifier applies to a pointer to a `long long int` argument.

`j`
: (C99) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an `intmax_t` or `uintmax_t` argument; or that a following `n` conversion specifier applies to a pointer to an `intmax_t` argument.

`z`
: (C99) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `size_t` or the corresponding signed integer type argument; or that a following `n` conversion specifier applies to a pointer to a signed integer type corresponding to `size_t` argument.

`t`
: (C99) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `ptrdiff_t` or the corresponding unsigned integer type argument; or that a following `n` conversion specifier applies to a pointer to a `ptrdiff_t` argument.

`L`
: Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a `long double` argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

`d`, `i`
: The `int` argument is converted to signed decimal in the style *[−]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

`o`, `u`, `x`, `X`
: The `unsigned int` argument is converted to unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` or `X`) in the style *dddd*; the letters **`abcdef`** are used for `x` conversion and the letters **`ABCDEF`** for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

`f`, `F`
: A `double` argument representing a (finite) floating-point number is converted to decimal notation in the style *[−]ddd*`.`*ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as

6; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.
(C99) A `double` argument representing an infinity is converted in one of the styles *[-]*`inf` or *[-]*`infinity` — which style is implementation-defined. A double argument representing a NaN is converted in one of the styles *[-]*`nan` or *[-]*`nan(`*n-char-sequence*`)` — which style, and the meaning of any *n-char-sequence*, is implementation-defined. The `F` conversion specifier produces `INF`, `INFINITY`, or `NAN` instead of `inf`, `infinity`, or `nan`, respectively. (When applied to infinite and NaN values, the `-`, `+`, and *space* flags have their usual meaning; the `#` and `0` flags have no effect.)

`e`, `E`

A `double` argument representing a (finite) floating-point number is converted in the style *[−]d*`.`*ddd*`e`±*dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The `E` conversion specifier produces a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero.
(C99) A `double` argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`g`, `G`

A `double` argument representing a (finite) floating-point number is converted in style `f` or `e` (or in style `F` or `E` in the case of a `G` conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style `e` (or `E`) is used only if the exponent resulting from such a conversion is less than –4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result unless the `#` flag is specified; a decimal-point character appears only if it is followed by a digit.
(C99) A `double` argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`a`, `A`

(C99) A double argument representing a (finite) floating-point number is converted in the style *[−]*`0x`*h*`.`*hhhh*`p`±*d*, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character (Binary implementations can choose the hexadecimal digit to the left of the decimal-point character so that subsequent digits align to nibble [4-bit] boundaries.) and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and `FLT_RADIX` is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and `FLT_RADIX` is not a power of 2, then the precision is sufficient to distinguish (The precision *p* is sufficient to distinguish values of the source type if $16^{p-1} > b^n$ where *b* is `FLT_RADIX` and *n* is the number of base-*b* digits in the significand of the source type. A smaller *p* might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point character.) values of type `double`, except that trailing zeros may be omitted; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. The letters **`abcdef`** are used for `a` conversion and the letters **`ABCDEF`** for `A` conversion. The `A` conversion specifier produces a number with `X` and `P` instead of `x` and

`p`. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.
A `double` argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`c`

If no `l` length modifier is present, the `int` argument is converted to an `unsigned char`, and the resulting character is written.
(C99) If an `l` length modifier is present, the `wint_t` argument is converted as if by an `ls` conversion specification with no precision and an argument that points to the initial element of a two-element array of `wchar_t`, the first element containing the `wint_t` argument to the `lc` conversion specification and the second a null wide character.

`s`

If no `l` length modifier is present, the argument shall be a pointer to the initial element of an array of character type. (No special provisions are made for multibyte characters.) Characters from the array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.
(C99) If an `l` length modifier is present, the argument shall be a pointer to the initial element of an array of `wchar_t` type. Wide characters from the array are converted to multibyte characters (each as if by a call to the `wcrtomb` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array shall contain a null wide character. If a precision is specified, no more than that many characters (bytes) are written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the multibyte character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case is a partial multibyte character written. (Redundant shift sequences may result if multibyte characters have a state-dependent encoding.)

`p`

The argument shall be a pointer to `void`. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.

`n`

The argument shall be a pointer to signed integer into which is written the number of characters written to the output stream so far by this call to `fprintf`. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

`%`

A `%` character is written. No argument is converted. The complete conversion specification shall be `%%`.

If a conversion specification is invalid, the behavior is undefined. If any argument is not the correct type for the corresponding coversion specification, the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

For a and A conversions, if FLT_RADIX is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

It is recommended practice that if FLT_RADIX is not a power of 2, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

It is recommended practice that for e, E, f, F, g, and G conversions, if the number of significant decimal digits is at most DECIMAL_DIG, then the result should be correctly rounded. (For binary-to-decimal conversion, the result format's values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.) If the number of significant decimal digits is more than DECIMAL_DIG but the source value is exactly representable with DECIMAL_DIG digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having DECIMAL_DIG significant digits; the value of the resultant decimal string D should satisfy $L \leq D \leq U$, with the extra stipulation that the error should have a correct sign for the current rounding direction.

The fprintf function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

The printf function is equivalent to fprintf with the argument stdout interposed before the arguments to printf. It returns the number of characters transmitted, or a negative value if an output error occurred.

The sprintf function is equivalent to fprintf, except that the argument s specifies an array into which the generated input is to be written, rather than to a stream. A null character is written at the end of the characters written; it is not counted as part of the returned sum. If copying takes place between objects that overlap, the behavior is undefined. The function returns the number of characters written in the array, not counting the terminating null character.

The vfprintf function is equivalent to fprintf, with the variable argument list replaced by arg, which shall have been initialized by the va_start macro (and possibly subsequent va_arg calls). The vfprintf function does not invoke the va_end macro. The function returns the number of characters transmitted, or a negative value if an output error occurred.

The vprintf function is equivalent to printf, with the variable argument list replaced by arg, which shall have been initialized by the va_start macro (and possibly subsequent va_arg calls). The vprintf function does not invoke the va_end macro. The function returns the number of characters transmitted, or a negative value if an output error occurred.

The vsprintf function is equivalent to sprintf, with the variable argument list replaced by arg, which shall have been initialized by the va_start macro (and possibly subsequent va_arg calls). The vsprintf function does not invoke the va_end macro. If copying takes place between objects that overlap, the behavior is undefined. The function returns the number of characters written into the array, not counting the terminating null character. ← File IO | Arrays →

← Intro exercise | Compiling →

Basic Concepts

Before one jumps headlong into learning C syntax and programming constructs, it is beneficial to learn the meaning of a few key terms that are central to a thorough understanding of C. Good luck using this guide for your studies!

Compilation: How Does C Work?

Like any programming language, C by itself is completely incomprehensible to a microprocessor. Its purpose is to provide an intuitive way for humans to provide instructions that can be easily converted into machine code. The program you use to convert C code into executable machine code is called a ***compiler*** [10]. If you are working on a project in which several source code files are involved, a second program called the ***linker*** is invoked. The purpose of the linker is to "connect" the files and produce either an executable program or a ***library***. A library is a set of routines, not a stand-alone program, but can be used by other programs or programmers. Compilation and linking are so closely related that programmers usually treat them as one step. One thing to keep in mind is that compilation is a "one way street"; compiling a C source file into machine code is easy, but "decompiling" (turning machine code into the C source that creates it) is not. Decompilers for C do exist, but they rarely create useful code. Probably the most popular compiler available is the C Compiler, which comes with most UNIX and UNIX-like systems.

Integrated Development Environments (IDEs)

An IDE is a program that combines a set of programs that developers need into one convenient package, usually with a graphical user interface. These programs include a compiler, linker, and text editor. They also typically include a debugger, a tool that will preserve your C source code after compilation and enable you to do such things as step manually through it or alter data in an attempt to uncover errors. A very popular IDE is Microsoft Visual C++ (MS VC++); a popular free IDE is DevC++,If you are running Mac OS X the Xcode IDE is included on the Developer Tools CD that came with your computer, you can also download it free at the Apple Developer Connection. It is recommended you find a good IDE before you begin learning C (or any other language), but you do not *need* one. If you have a decent text editor (Microsoft Word and WordPerfect are not text editors, they are word processors. Notepad is a text editor though.) and a compiler, they will do as well. There are many text editors (see List of Text Editors on Wikipedia), the most popular being vi and its clones (such as vim) and Emacs.

Block Structure, Statements, Scope, and Whitespace

Now we **discuss the basic structure of a C program**. If you're familiar with PASCAL, you may have heard it referred to as a **block structured** language. C does not have complete block structure (and you'll find out why when you go over functions in detail) but it is still very important to understand what blocks are and how to use them. A **block** is a group of source code statements intended to control scope. In C, blocks begin with a "left curly" (`"{"`) and end with a "right curly" (`"}"`). Blocks can contain sub-blocks, which can contain their own sub-blocks, and so on.

So what's in a block? Generally, a block consists of executable **statements** and the **whitespace** that surrounds them. A statement is text the compiler will attempt to turn into executable instructions. Statements always end with a semicolon (;) character. Multiple statements can share a single line in the source file. There are several kinds of statements, including assignment, conditional and flow-control. A good portion of this book deals with statement construction.

Whitespace refers to the tab, space and newline/EOL (End Of Line) characters that separate the text characters that make up source code lines. Like many things in life, it's hard to appreciate whitespace until it's gone. To a C compiler, the source code

```
puts("Hello world"); return 0;
```

is the same as

```
puts("Hello world");
return 0;
```

is the same as

```
puts (
"Hello world") ;





return 0;
```

The compiler simply skips over whitespace. However, it is common practice to use spaces (and tabs) to organize source code for human readability.

In C, most of the time we do not want other functions or other programmer's routines accessing data that we are currently manipulating. This is why it is important to understand the concept of **scope**. **Scope** describes the level at which a piece of data or a function can be seen or manipulated. There are two kinds of scope in C, **local** and **global**. When we speak of something being **global**, we speak of something that can be seen or manipulated from anywhere in the program. When we speak of something being **local**, we speak of something that can be seen or manipulated only within the block it was declared, or its sub-blocks. (Sub-blocks can access data local to their encompassing block, but a block cannot access local data local to a sub-block.)

TIP: You can use blocks without an if, loop, et al statement to organize your code.

Basics of Using Functions

Functions are a big part of programming. A function is a special kind of block that performs a well-defined task. If a function is well-designed, it can enable a programmer to perform a task without knowing anything about how the function works. The act of requesting a function to perform its task is called a **function call**. Many functions require a caller to hand it certain pieces of data needed to perform its task; these are called **arguments**. Many functions also return a value to the caller when they're finished; this is called a **return value**.

- The things you need to know before calling a function are:
  - What the function does
  - The data type (discussed later) of the arguments and what they mean
  - The data type of the return value, and what it means

All code other than global data definitions and declarations needs to be a part of a function.

Every executable program needs to have one and only one **main** function, which is where the program begins executing.

The Standard Library

In 1983, when C was in the process of becoming standardized, the standardization committee decided that there needed to be a basic set of functions common to each implementation of C. This is called the **Standard Library**. The Standard Library provides functions for tasks such as input/output, string manipulation, mathematics, files, memory allocation, and more. The Standard Library does NOT provide functions for anything that might be dependent on hardware or operating system, like graphics, sound, or networking. In the "Hello, World", program, a Standard Library function is used - puts -- which outputs lines of text to the standard output stream.

Comments and Coding Style

Comments are text inserted into the source code of a program that serve no purpose other than documenting the code. In C, they begin with /* and end with */. Good commenting is considered essential to software development, not just because others may need to read your code, but **you** may need to come back to your code a long time after writing it and immediately understand how it works. In general, it is a good idea to comment anything that is not immediately obvious to a competent programmer. However, it is **not** a good idea to comment every line. This may actually make your code more difficult to read and it will waste space.

Good coding style habits are important to adopt for the simple reason that code should be intuitive and readable, which is, after all, the purpose of a high-level programming language like C. In general, provide ample white space, indent so that the opening brace of a block and the closing brace of a block are vertically aligned, and provide intuitive names for your functions and variables. Throughout this text we will be providing more style and coding style tips for you. Do try and follow these tips: they will make your code easier for you and others to read and understand.

The Preprocessor

Many times you will need to give special instructions to your compiler. This is done through inserting **preprocessor directives** into your code. When you begin compiling your code, a subprogram called the preprocessor first finds these directives and interprets them before the compilation process begins. One thing to remember is that these directives are NOT compiled as part of your source code. In C language, all preprocessor directives begin with the hash character (#). You can see one preprocessor directive in "Hello, World!", `#include`. `#include` opens a file and conceptually replaces the `#include` directive with the file's contents. There is another common preprocessor directive, `#define`, that will be discussed later.

Footnotes

1. ⇧ Actually, GCC's(C Compiler) **cc** (C Compiler) translates the input .c file to the target cpu's assembly, output is written to an .s file. Then **as** (assembler) generates a machine code file from the .s file. Pre-processing is done by another sub-program **cpp** (C PreProcessor).

← Preliminaries | Structure and style →

Now that we have covered the basic concepts of C programming, we can briefly discuss the process of *compilation*.

Compilation is basically translation -- a computer program called the *compiler* takes our C source code and translates it into the binary language used by computers. Of course, it is more complicated than that; but the basic idea applies.

To those new to programming, this seems fairly simple. A naive compiler might read in every source file, translate everything into machine code, and write out an executable. This could work but has two serious problems. First, for a large project, the computer may not have enough memory to read all of the source code at once. Second, if you make a change to a single source file, you would rather not have to recompile the *entire* application. To deal with these problems, compilers break their job down into steps; for each source file (each `.c` file), the compiler reads the file, reads the files it `#includes`, and translates it to machine code. The result of this is an "object file" (`.o`). Once every object file is made, a "linker" collects all of the object files and writes the actual program. This way, if you change one source file, only that file needs to be recompiled and then the application needs to be re-linked.

Without going into the painful details, it can be beneficial to have a superficial understanding of the compilation process, and here we will briefly discuss it:

Preprocessor

In this stage the "preprocessor directives" are dealt with. These include `#include`s, macros, and `#pragma` compiler settings. The result of the preprocessor is a text string.

Syntax Checking

This step ensures that the code is valid and will sequence into an executable program.

Object Code

The compiler produces a machine code equivalent of the source code that can then be linked into the final program. This step ensures that the code is valid and will sequence into an executable program.

Linking

Linking combines the separate object codes into one complete program by integrating libraries and the code into the final executable format. ← Compiling | Error handling →

C Structure and Style

This is a basic introduction to producing effective code structure in the C Programming Language. It is designed to provide information on how to effectively use Indentation, Comments, and other elements that will make your C code more readable. It is not a tutorial on actually programming in C.

New programmers will often not see the point in following structure in their programs, because they often think that code is designed purely for the reading by a compiler. This is usually not the case, as well-written code that follows a well-designed structure is usually much easier for programmers (who haven't worked on the code for months) to read, and edit.

In the following sections, we will attempt to explain good programming techniques that will in turn make your programs more effective.

Introduction

The following two blocks of code are essentially the same: Both of them contain exactly the same code, and will compile and execute with the same result; however there is one essential difference.

Which of the following programs do you think is easier to read?

```
#include <stdio.h>
int main()
{printf("Hello, World!\n");return(0);}
```

or

```
#include <stdio.h>

int main()
{
        printf("Hello, World!\n");
        return(0);
}
```

The simple use of indents and line breaks can greatly improve the readability of the code; without making any impact whatsoever on how the code performs. By having readable code, it is much easier to see where functions and procedures end, and which lines are part of which loops and procedures.

This book is going to focus on the above piece of code, and how to improve it. Please note that during the course of the tutorial, there will be many (apparently) redundant pieces of code added. These are only added to provide examples of techniques that we will be explaining, without breaking the overall flow of code that the program achieves.

Line Breaks and Indentation

The addition of white space inside your code is arguably the most important part of good code structure. Effective use of it can create a visual gauge of how your code flows, which can be very important when returning to your code when you want to maintain it.

Line Breaks

Line breaks should be used in three main parts of your code

- After precompiler declarations.
- After new variables are declared.
- Between new paths of code. (i.e. Before the declaration of the function or loop, and after the closing '}' bracket).

The following lines of code have line breaks between functions, but not any indention. *Note that we have added line numbers to the start of the lines. Using these in actual code will make your compiler fail, they are only there for reference in this book.*

```
10   #include <stdio.h>
20   int main()
30   {
40   int i=0;
50   printf("Hello, World!");
60   for (i=0;i<1;i++){
70   printf("\n");
80   break;
90   }
100  return(0);
110  }
120
```

Based on the rules we established earlier, there should now be four line breaks added.

- Between lines 10 and 20, as line 10 is a precompiler declaration
- Between lines 40 and 50, as the block above it contains variable declarations
- Between lines 50 and 60, as it is the beginning of a new path (the 'for' loop)
- Between lines 90 and 100, as it is the end of a path of code

This will make the code much more readable than it was before:

```
10   #include <stdio.h>
11
20   int main()
30   {
40   int i=0;
41
50   printf("Hello, World!");
51
60   for (i=0;i<1;i++){
70   printf("\n");
80   break;
90   }
91
100  return(0);
110  }
120
```

But this still isn't as readable as it can be.

Indentation

Although adding simple line breaks between key blocks of code can make code marginally easier to read, it provides no gauge of the flow of the program. The use of your tab key can be very useful now: indentation visually separates paths of code by moving their starting points to a new column in the line. This simple practice will make it much easier to read code. Indentation follows a fairly simple rule:

- All code inside a new path (i.e. Between the two '{' brackets '}') should be indented by one tab more than the code in the previous path.

So, based on our code from the previous section, there are two paths that require indentation:

- Lines 40 to 100
- Lines 70 and 80

```
10 #include <stdio.h>
11
20 int main()
30 {
40      int i=0;
41
50      printf("Hello, World!");
51
60      for (i=0;i<1;i++){
70              printf("\n");
80              break;
90      }
91
100     return(0);
110 }
120
```

It is now fairly obvious as to which parts of the program fit inside which paths of code. You can tell which parts of the program will loop, and which ones will not. Although it might not be immediately noticeable, once many nested loops and paths get added to the structure of the program, the use of indentation can be very important.

NOTE: Many text editors automatically indent appropriately when you hit the enter/return key.

Comments

Comments in code can be useful for a variety of purposes. They provide the easiest way to point out specific parts of code (and their purpose); as well as providing a visual "split" between various parts of your code. Having a good commentary throughout your code will make it much easier to remember what specific parts of your code do.

Comments in modern flavours of C (and many other languages) can come in two forms:

```
//Single Line Comments
```

and

```
/*Multi-Line
Comments*/
```

Single-line Comments

Single-line comments are most useful for simple 'side' notes that explain what certain parts of the code do. The best places to put these comments are next to variable declarations, and next to pieces of code that may need explanation.

Based on our previous program, there are two good places to place comments

- Line 40, to explain what 'int i' is going to do
- Line 80, to explain why there is a 'break' keyword.

This will make our program look something like

```
10   #include <stdio.h>
11
20   int main()
30   {
40      int i=0;                //Temporary variable used for 'for' loop.
41
50      printf("Hello, World!");
51
60      for (i=0;i<1;i++){
70         printf("\n");
80         break;               //Exits 'for' loop.
90      }
91
100     return(0);
110  }
```

Multi-line Comments

Multi-line comments are most useful for long explanations of code. They can be used as copyright/licensing notices, and they can also be used to explain the purpose of a path of code. This can be useful in two facets: They make your functions easier to understand, and they make it easier to spot errors in code (if you know what a path is *supposed* to do, then it is much easier to find the piece of code that is responsible).

As an example, suppose we had a program that was designed to print "Hello, World! " a certain number of times, on a certain number of lines. There would be many for loops in this program. For this example, we shall call the number of lines *i*, and the number of strings per line as *j*.

A good example of a multi-line comment that describes 'for' loop *i*'s purpose would be:

```
/* For Loop (int i)
   Loops the following procedure i times (for number of lines).  Performs 'for' loop j on
each loop,
   and prints a new line at end of each loop.
*/
```

This provides a good explanation of what 'i's purpose is, whilst not going into detail of what 'j' does. By going into detail over what the specific path does (and not ones inside it), it will be easier to troubleshoot the path.

Similarly, you should always include a multi-line comment as the first thing inside a function, to explain the name of the function; the input that it will take, how it takes that input; the output; and the overall procedure that the function is designed to perform. Always leave the technical details to the individual code paths inside your program - this makes it easier to troubleshoot.

A function descriptor should look something like:

```
/* Function : int hworld (int i,int j)
   Input    : int i (Number of lines), int j (Number of instances per line)
   Output   : 0 (on success)
   Procedure: Prints "Hello, World!" j times, and a new line to standard output over i
lines.
*/
```

This system allows for an at-a-glance explanation of what the function should do. You can then go into detail over how each aspect of the program is achieved later on in the program.

Finally, if you like to have aesthetically-pleasing source code, the multi-line comment system allows for the easy addition of starry borders to your comment. These make the comments stand out much more than they would without the border (especially buried deep in source code). They should take a format similar to:

```
/***************************************
 *  This is a multi line comment       *
 *  That is surrounded by a            *
 *  Cool, starry border!               *
 ***************************************/
```

Applied to our original program, we can now include a much more descriptive and readable source code:

```
#include <stdio.h>

int main()
{

/*************************************************************************************
  * Function: int main()
*
  * Input   : none
*
```

```
  * Output  : Returns 0 on success
*

  * Procedure: Prints "Hello, World!" and a new line to standard output then exits.
*

************************************************************************************/
    int i=0;                //Temporary variable used for 'for' loop.

    printf("Hello, World!");

    /* FOR LOOP (int i)
       Prints a new line to standard output, and exits */
    for (i=0;i<1;i++){
       printf("\n");
       break;               //Exits 'for' loop.
    }

    return(0);
 }
```

This will allow any outside users of the program an easy way to understand what the code does, and how it works. It also prevents confusion with other like-named functions.

Examples

Aladdin's C coding guidelines - A more definitive C coding guideline.

- C/C++ Programming Styles Coding styles & Linux Kernel Coding style

← Structure and style | Variables →

C has no native support for error handling (properly known as exception handling). The programmer must instead prevent errors from occurring in the first place, often testing return values from functions. -1 and NULL are used in several functions such as socket() (Unix socket programming) or malloc() respectively to indicate problems that the programmer should be aware about. In a worst case scenario where there is an unavoidable error and no way to recover from it, a C programmer usually tries to log the error and "gracefully" terminate the program.

There is an external variable called "errno", accessible by the programs after including <errno.h> - that file comes from the definition of the possible errors that can ocurr in some Operating Systems (e.g. Linux - in this case, the definition is in include/asm-generic/errno.h) when programs ask for resources. Such variable indexes error descriptions, that is accessible by the function 'strerror(errno)'.

The following code tests the return value from the library function malloc to see if dynamic memory allocation completed properly:

```
#include <stdio.h>
#include <errno.h>        // errno
#include <stdlib.h>       // malloc

#include <string.h>       // strerror


extern errno;

int main( void )
{
        /* pointer to characters (bytes) , requesting dynamic allocation of
5000000000000000000000 storage elements */
        char    *ptr = (char *) malloc( 5000000000000000000000  * sizeof( int ));

        if( ptr == NULL )
                fprintf( stdout, strerror( errno ) );
        else
        {
               /* the rest of the code hereafter can assume that 1000 bytes were
successfully allocated */
               /* ... */
               free( ptr );
        }

        exit(EXIT_SUCCESS); /* exiting program */
}
```

The code snippet above shows the use of the return value of the library function malloc to check for errors. Many library functions have return values that flag errors, and thus should be checked by the astute programmer. In the snippet above, a NULL pointer returned from malloc signals an error in allocation, so the program exits. In more complicated implementations, the program might try to handle the error and try to recover from the failed memory allocation.

Handling divide by zero errors

A common pitfall made by C programmers is not checking if a divisor is zero before a division command. The following code will produce a runtime error and in most cases, exit.

```
int dividend = 50;
int divisor = 0;
int quotient;

quotient = (dividend/divisor); /* This will produce a runtime error! */
```

For reasons beyond the scope of this document, you must check or make sure that a divisor is never zero. Alternatively for *nix processes, you can stop the OS from terminating your process by blocking the SIGFPE signal.

The code below fixes this by checking if the divisor is zero before dividing.

```
int dividend = 50;
int divisor = 0;
int quotient;

if(divisor == 0) {
    // Handle the error here...

}


quotient = (dividend/divisor);
```

← Error handling | Simple Input and Output →

Variables

Like most programming languages, C is able to use and process named variables and their contents. **Variables** are most simply described as names by which we refer to some location in memory - a location that holds a value with which we are working.

It often helps to think of variables as a "pigeonhole", or a placeholder for a value. You can think of a variable as being equivalent to its value. So, if you have a variable *i* that is initialized to 4, i+1 will equal 5.

Since C is a relatively low-level programming language, it is necessary for a C program to claim the memory needed to store the values for variables before using that memory. This is done by **declaring** variables, the way in which a C program shows the number of variables it needs, what they are going to be named, and how much memory they will need.

Within the C programming language, when we manage and work with variables, it is important for us to know the *type* of our variables and the *size* of these types. This is because C is a sufficiently low-level programming language that these aspects of its working can be hardware specific - that is, how the language is made to work on one type of machine can be different from how it is made to work on another.

All variables in C are typed. That is, you must give a type for every variable you declare.

Declaring, Initializing, and Assigning Variables

Here is an example of declaring an integer, which we've called `some_number`. (Note the semicolon at the end of the line - that is how your compiler separates one program *statement* from another.)

```
int some_number;
```

This statement means we're declaring some space for a variable called some_number, which will be used to store `integer` data. Note that we must specify the type of data that a variable will store. There are specific keywords to do this - we'll look at them in the next section.

You can also declare multiple variables with one statement:

```
int anumber, anothernumber, yetanothernumber;
```

We can also declare *and* assign some content to a variable at the same time. This is called *initialization* because it is the "initial" time a value has been assigned to the variable:

```
int some_number=3;
```

In C, all variable declarations (except for globals) must be done at the **beginning** of a block. You cannot declare your variables, insert some other statements, and then declare more variables. Variable declarations (if there are any) are always the **first** part of any block.

After declaring variables, you can assign a value to a variable later on using a statement like this:

```
some_number=3;
```

You can also assign a variable the value of another variable, like so:

```
anumber = anothernumber;
```

Or assign multiple variables the same value with one statement:

```
anumber = anothernumber = yetanothernumber = 3;
```

This is because the assignment (x = y) returns the value of the assignment. x = y = z is really shorthand for x = (y = z).

Naming Variables

Variable names in C are made up of letters (upper and lower case) and digits. The underscore character ("_") is also permitted. Names must not begin with a digit. Unlike some languages (such as Perl and some BASIC dialects), C does not use any special prefix characters on variable names.

Some examples of valid (but not very descriptive) C variable names:

```
foo
Bar
BAZ
foo_bar
_foo42
_

QuUx
```

Some examples of invalid C variable names:

```
2foo    (must not begin with a digit)
my foo  (spaces not allowed in names)
$foo    ($ not allowed -- only letters, digits, and _)
while   (language keywords cannot be used as names)
```

As the last example suggests, certain words are reserved as keywords in the language, and these cannot be used as variable names.

In addition there are certain sets of names that, while not language keywords, are reserved for one reason or another. For example, a C compiler might use certain names "behind the scenes", and this might cause problems for a program that attempts to use them. Also, some names are reserved for possible future use in the C standard library. The rules for determining exactly what names are reserved (and in what contexts they are reserved) are too complicated to describe here, and as a beginner you don't need to worry about them much anyway. For now, just avoid using names that begin with an underscore character.

The naming rules for C variables also apply to other language constructs such as function names, struct tags, and macros, all of which will be covered later.

Literals

Anytime within a program in which you specify a value explicitly instead of referring to a variable or some other form of data, that value is referred to as a **literal**. In the initialization example above, 3 is a literal. Literals can either take a form defined by their type (more on that soon), or one can use hexadecimal (hex) notation to directly insert data into a variable regardless of its type. Hex numbers are always preceded with *0x*. For now, though, you probably shouldn't be too concerned with hex.

The Four Basic Types

In Standard C there are four basic data types. They are **`int`**, **`char`**, **`float`**, and **`double`**.

The `int` type

The `int` type, which you've already seen, is meant to store integers, which you may also know as "whole numbers". An integer is typically the size of one machine word, which on most modern home PCs is 32 bits (4 octets). Examples of literals are whole numbers (integers) such as 1,2,3, 10, 100... When `int` is 32 bits (4 octets), it can store any whole number (integer) between -2147483648 and 2147483647. A 32 bit word (number) has the possibility of representing 4294967296 numbers (2 to the power of 32).

If you want to declare a new int variable, use the `int` keyword. For example:

```
int numberOfStudents, i, j=5;
```

In this declaration we declare 3 variables, numberOfStudents, i & j, j here is assigned the literal 5.

The `char` type

The `char` type is similar to the `int` type, yet it is only big enough to hold one ASCII character. It stores the same kind of data as an `int` (i.e. integers), but always has a size of one byte. It is most often used to store character data, hence its name.

Examples of character literals are 'a', 'b', '1', etc., as well as special characters such as '`\0`' (the null character) and '`\n`' (endline, recall "Hello, World").

The reason why one byte is seen as the ideal size for character data is that one byte is large enough to provide one slot for each member of the ASCII character set, which is a set of characters which maps one-to-one with a set of integers. At compile time, all character literals are converted into their corresponding integer. For example, 'A' will be converted to 65 (0x41). (Knowing about the ASCII character set is often useful.)

When we initialize a character variable, we can do it two ways. One is preferred, the other way is ***bad*** programming practice.

The first way is to write

```
 char letter1='a';
```

This is *good* programming practice in that it allows a person reading your code to understand that letter is being initialized with the letter "a" to start off with.

The second way, which should *not* be used when you are coding letter characters, is to write

```
char letter2=97; /* in ASCII, 97 = 'a' */
```

This is considered by some to be extremely ***bad*** practice, if we are using it to store a character, not a small number, in that if someone reads your code, most readers are forced to look up what character corresponds with the number 97 in the encoding scheme. In the end, letter1 and letter2 store both the same thing -- the letter "a", but the first method is clearer, easier to debug, and much more straightforward.

One important thing to mention is that characters for numerals are represented differently from their corresponding number, i.e. '1' is not equal to 1.

There is one more kind of literal that needs to be explained in connection with chars: the **string literal**. A string is a series of characters, usually intended to be output to the string. They are surrounded by double quotes (" ", not ' '). An example of a string literal is the "Hello, world!\n" in the "Hello, World" example.

The `float` type

`float` is short for Floating Point. It stores real numbers also, but is only one machine word in size. Therefore, it is used when less precision than a double provides is required. `float` literals must be suffixed with F or f, otherwise they will be interpreted as doubles. Examples are: 3.1415926f, 4.0f, 6.022e+23f. float variables can be declared using the `float` keyword.

The `double` type

The `double` and `float` types are very similar. The `float` type allows you to store single-precision floating point numbers, while the `double` keyword allows you to store double-precision floating point numbers - real numbers, in other words, both integer and non-integer values. Its size is typically two machine words, or 8 bytes on most machines. Examples of `double` literals are 3.1415926535897932, 4.0, 6.022e+23 (scientific notation). If you use 4 instead of 4.0, the 4 will be interpreted as an `int`.

The distinction between floats and doubles was made because of the differing sizes of the two types. When C was first used, space was at a minimum and so the judicious use of a float instead of a double saved some memory. Nowadays, with memory more freely available, you do not really need to conserve memory like this - it may be better to use doubles consistently. Indeed, some C implementations use doubles instead of floats when you declare a float variable.

If you want to use a double variable, use the `double` keyword.

`sizeof`

If you have any doubts as to the amount of memory actually used by any type (and this goes for types we'll discuss later, also), you can use the **`sizeof`** operator to find out for sure. (For completeness, it is important to mention that `sizeof` is an operator, not a function, even though it looks like a function. It does not have the overhead associated with a function, nor do you need to `#include` anything to use it.) Syntax is:

```
int i;

i = sizeof(int);
```

`i` will be set to 4, assuming a 32-bit system. The result of sizeof is the amount of data used for an object in multiples of `char`.

Data type modifiers

One can alter the data storage of any data type by preceding it with certain modifiers.

`long` and **`short`** are modifiers that make it possible for a data type to use either more or less memory. The `int` keyword need not follow the `short` and `long` keywords. This is most commonly the case. A `short` can be used where the values fall within a lesser range than that of an `int`, typically -32768 to 32767. A `long` can be used to contain an extended range of values. It is not guaranteed that a short uses less memory than an int, nor is it guaranteed that a long takes up more memory than an int. It is only guaranteed that sizeof(short) <= sizeof(int) <= sizeof(long). Typically a short is 2 bytes, an int is 4 bytes, and a long either 4 or 8 bytes.

In all of the types described above, one bit is used to store the sign (positive or negative) or a value. If you decide that a variable will never hold a negative value, you may use the **`unsigned`** modifier to use that one bit for storing other data, effectively doubling the range of values while mandating that those values be positive. The `unsigned` specifier may also be used without a trailing `int`, in which case the size defaults to that of an `int`. There is also a **`signed`** modifier which is the opposite, but it is not necessary and seldom used since all types are signed by default.

To use a modifier, just declare a variable with the data type and relevant modifiers attached:

```
 unsigned short int i;
 short  things;
 unsigned long apples;
```

`const` modifier

When **`const`** is added as a modifier, the declared variable must be initialized at declaration. It is then not allowed to be changed, unless a `cast` is done. While the idea of a variable that never changes may not seem useful, there are good reasons to use const. For one thing, many compilers can perform some small optimizations on data when it knows that data will never change. For example, if you need the value of π in your calculations, you can declare a const variable of `pi`, so a program or another function written by someone else cannot change the variable of `pi`.

Magic numbers

When you write C programs, you may be tempted to write code that will depend on certain numbers. For example, you may be writing a program for a grocery store. This complex program has thousands upon thousands of lines of code. The programmer decides to represent the cost of a can of corn, currently 99 cents, as a literal throughout the code. Now, assume the cost of a can of corn changes to 89 cents. The programmer must now go in and manually change each entry of 99 cents to 89. While this is not that big of a problem, considering the "global find-replace" function of many text editors, consider another problem: the cost of a can of green beans is also initially 99 cents. To reliably change the price, you have to look at every occurrence of the number 99.

C possesses certain functionality to avoid this. This functionality is approximately equivalent, though one method can be useful in one circumstance, over another.

Using the `const` keyword

The `const` keyword helps eradicate **magic numbers**. By declaring a variable `const corn` at the beginning of a block, a programmer can simply change that const and not have to worry about setting the value elsewhere.

There is also another method for avoiding magic numbers. It is much more flexible than `const`, and also much more problematic in many ways. It also involves the preprocessor, as opposed to the compiler. Behold...

`#define`

When you write programs, you can create what is known as a *macro*, so when the computer is reading your code, it will replace all instances of a word with the specified expression.

Here's an example. If you write

```
#define PRICE_OF_CORN 0.99
```

when you want to, for example, print the price of corn, you use the word `PRICE_OF_CORN` instead of the number 0.99 - the precompiler will replace all instances of `PRICE_OF_CORN` with the *text* "0.99", which the compiler will interpret as the literal `double` 0.99. Notice that, since this is a special directive (the compiler will never know that this line was there), there is no need for a semicolon.

It is important to note that `#define` has basically the same functionality as the "find-and-replace" function in a lot of text editors/word processors.

For some purposes, `#define` can be harmfully used, and it is usually preferable to use const if `#define` is unnecessary. It is possible, for instance, to `#define`, say, a macro `DOG` as the number 3, but if you try to print the macro, thinking that `DOG` represents a string that you can show on the screen, the program will have an error. `#define` also has no regard for type. It disregards the structure of your program, replacing the text *everywhere* (in effect, disregarding scope), which could be advantageous in some circumstances, but can be the source of problematic bugs.

You will see further instances of the `#define` directive later in the text. It is good convention to write `#defined` words in all capitals, so a programmer will know that this is not a variable that you have declared but a `#defined` macro.

Scope

In the Basic Concepts section, the concept of scope was introduced. It is important to revisit the distinction between local types and global types, and how to declare variables of each. To declare a local variable, you place the declaration at the beginning (i.e. before any non-declarative statements) of the block the variable is intended to be local to. To declare a global variable, declare the variable outside of any block. If a variable is global, it can be read, and written, from anywhere in your program.

Global variables are not considered good programming practice, and should be avoided whenever possible. They inhibit code readability, create naming conflicts, waste memory, and can create difficult-to-trace bugs. Excessive usage of globals is usually a sign of laziness and/or poor design. However, if there is a situation where local variables may create more obtuse and unreadable code, there's no shame in using globals. (Implementing malloc, which is a function discussed later, is one example of something that is simply too much more difficult to write without at least one global variable.)

Other Modifiers

Included here, for completeness, are more of the modifiers that standard C provides. For the beginning programmer, *static* and *extern* may be useful. *volatile* is more of interest to advanced programmers. *register* and *auto* are largely deprecated and are generally not of interest to either beginning or advanced programmers.

`static` is sometimes a useful keyword. It is a common misbelief that the only purpose is to make a variable stay in memory.
When you declare a function or global variable as *static* it will become internal. You cannot access the function or variable through the extern (see below) keyword from other files in your project.

When you declare a local variable as *static*, it is created just like any other variable. However, when the variable goes out of scope (i.e. the block it was local to is finished) the variable stays in memory, retaining its value. The variable stays in memory until the program ends. While this behaviour resembles that of global variables, static variables still obey scope rules and therefore cannot be accessed outside of their scope.
Variables declared static are initialized to zero (or for pointers, NULL) by default.

You can use static in (at least) two different ways. Consider this code, and imagine it is in a file called jfile.c:

```
static int j = 0;

void upj(void)
{
    static int k = 0;
    j++;
}

void downj(void)
{
    j--;
}
```

The j var is accessible by both upj and downj and retains its value. the k var also retains its value, but is only accessible to upj. static vars are a good way to implement encapsulation, a term from the object-oriented way of thinking that effectively means not allowing changes to be made to a variable except through function calls.

`extern` is used when a file needs to access a variable in another file that it may not have `#included` directly. Therefore, *extern* does not actually carve out space for a new variable, it just provides the compiler with sufficient information to access the remote variable.

`volatile` is a special type modifier which informs the compiler that the value of the variable may be changed by external entities other than the program itself. This is necessary for certain programs compiled with optimizations - if a variable were not defined `volatile` then the compiler may assume that certain operations involving the variable are safe to optimize away when in fact they aren't. *volatile* is particularly relevant when working with embedded systems (where a program may not have complete control of a variable) and multi-threaded applications.

`auto` is a modifier which specifies an "automatic" variable that is automatically created when in scope and destroyed when out of scope. If you think this sounds like pretty much what you've been doing all along when you declare a variable, you're right: all declared items within a block are implicitly "automatic". For this reason, the *auto* keyword is more like the answer to a trivia question than a useful modifier, and there are lots of very competent programmers that are unaware of its existence.

`register` is a hint to the compiler to attempt to optimize the storage of the given variable by storing it in a register of the computer's CPU when the program is run. Most optimizing compilers do this anyway, so use of this keyword is often unnecessary. In fact, ANSI C states that a compiler can ignore this keyword if it so desires -- and many do. Microsoft Visual C++ is an example of an implementation that completely ignores the *register* keyword.

In-depth C ideas

← Beginning exercises | Pointers and arrays →

Arrays & Strings

Arrays in C act to store related data under a single variable name with an index, also known as a *subscript*. It is easiest to think of an array as simply a list or ordered grouping of variables. As such, arrays often help a programmer organize collections of data efficiently and intuitively.

Later we will consider the concept of a *pointer*, fundamental to C, which extends the nature of the array. For now, we will consider just their declaration and their use.

Arrays

If we want an array of six integers , called "numbers", we write in C

```
int numbers[6];
```

For a character array called letters,

```
char letters[6];
```

and so on.

If we wish to initialize as we declare, we write

```
int vector[6]={0,0,1,0,0,0};
```

If we want to access a variable stored in an array, for example with the above declaration, the following code will store a 1 in the variable `x`

```
int x;
x = vector[2];
```

Arrays in C are indexed starting at 0, as opposed to starting at 1. The first element of the array above is `vector[0]`. The index to the last value in the array is the array size minus one. In the example above the subscripts run from 0 through 5. C does not do bounds checking on array accesses. The compiler will not complain about the following:

```
char y;
int z = 9;
char vector[6] = { 1, 2, 3, 4, 5, 6 };
//examples of accessing outside the array. A compile error is not raised
y = vector[15];
y = vector[-4];
y = vector[z];
```

During program execution, an out of bounds array access does not always cause a run time error. Your program may happily continue after retrieving a value from vector[-1]. To alleviate indexing problems, the sizeof() expression is commonly used when coding loops that process arrays.

```
int ix;
short anArray[]= { 3, 6, 9, 12, 15 };

for (ix=0; ix< (sizeof(anArray)/sizeof(short)); ++ix) {
   DoSomethingWith( anArray[ix] );
}
```

Notice in the above example, the size of the array was not explicitly specified. The compiler knows to size it at 5 because of the five values in the initializer list. Adding an additional value to the list will cause it to be sized to six, and because of the sizeof expression in the `for` loop, the code automatically adjusts to this change. This technique is often used by experienced C programmers.

C also supports multi dimensional arrays. The simplest type is a two dimensional array. This creates a rectangular array - each row has the same number of columns. To get a char array with 3 rows and 5 columns we write in C

```
char two_d[3][5];
```

To access/modify a value in this array we need two subscripts:

```
char ch;
ch = two_d[2][4];
```

or

```
two_d[0][0] = 'x';
```

Similarly, a multi-dimensional array can be initialized like this:

```
int two_d[2][3] = {{ 5, 2, 1 },
                   { 6, 7, 8 }};
```

There are also weird notations possible:

```
int a[100];
int i = 0;
if (a[i]==i[a])
{
    printf("Hello World!\n");
```

a[i] and i[a] point to the same location. (This is explained later in the next Chapter.)

Strings

C has no string handling facilities built in; consequently, strings are defined as arrays of characters.

```
char string[30];
```

However, there is a useful library of string handling routines which you can use by including another header file.

```
#include <stdio.h>
#include <string.h>  //new header file

int main(void) {
    ...
}
```

← Arrays | Memory management →

Pointers and Arrays

A **pointer** is a value that designates the address, or location in memory, of some value. There are four fundamental things you need to know about pointers:

- How to declare them
- How to assign to them
- How to reference the value associated with the pointer (dereferencing) and
- How they relate to arrays

We'll also discuss the relationship of pointers with text strings and the more advanced concept of function pointers.

Pointers are variables that hold a memory location -- the location of some other variable. One can access the value of the variable pointed to using the dereferencing operator '*'. Pointers can hold any data type, even functions.

The vast majority of arrays in C are simple lists, also called "1 dimensional arrays". We will briefly cover multi-dimensional arrays in a later chapter.

Declaring pointers

Consider the following snippet of code which declares two pointers:

```
struct MyStruct {
    int   m_aNumber;
    float num2;
};

int             * pJ2;
struct MyStruct * pAnItem;
```

The first four lines define a structure. The next line declares a variable which points to an `int`, and the bottom line declares a variable which points to something with structure MyStruct. So to declare a variable as something which points to some type, rather than contains some type, the asterisk (`*`) is placed before the variable name.

In the first of the following lines of code, `var1` is a pointer to a long while `var2` is a long and not a pointer to a long. In the second line `p3` is declared as a pointer to a pointer to an int.

```
long  *  var1, var2;
int   ** p3;
```

Pointer types are often used as parameters to function calls. The following shows how to declare a function which uses a pointer as an argument. Since C passes function arguments by value, in order to allow a function to modify a value from the calling routine, a pointer to the value must be passed. Pointers to structures are also used as function arguments even when nothing in the struct will be modified in the function. This is done to avoid copying the complete contents of the structure onto the stack. More about pointers as function arguments later.

```
int MyFunction( struct MyStruct *pStruct );
```

Assigning values to pointers

So far we've discussed how to declare pointers. The process of assigning values to pointers is next. To assign a pointer the address of a variable, the & or 'address of' operator is used.

```
int   myInt;
int  *pPointer;
struct MyStruct   dvorak;
struct MyStruct  *pKeyboard;

pPointer = &myInt;
pKeyboard = &dvorak;
```

Here, pPointer will now reference myInt and pKeyboard will reference dvorak.

Pointers can also be assigned to reference dynamically allocated memory. The malloc() and calloc() functions are often what are used to do this.

```
#include <stdlib.h>
.
.
struct MyStruct *pKeyboard;
.
.
pKeyboard = malloc(sizeof(struct MyStruct));
.
.
```

The malloc function returns a pointer to dynamically allocated memory (or NULL if unsuccessful). The size of this memory will be appropriately sized to contain the MyStruct structure.

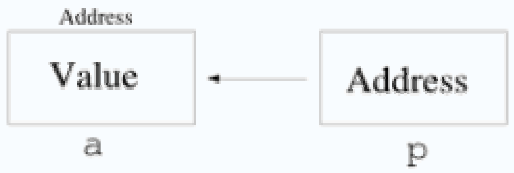
The following is an example showing one pointer being assigned to another and of a pointer being assigned a return value from a function.

```
static struct MyStruct val1, val2, val3, val4;
.
.
struct MyStruct *ASillyFunction( int b )
{
   struct MyStruct *myReturn;

   if (b == 1) myReturn = &val1;
   else if (b==2) myReturn = &val2;
   else if (b==3) myReturn = &val3;
   else myReturn = &val4;

   return myReturn;
}
.
.
struct MyStruct *strPointer;
int     *c, *d;
int     j;
.
.
c = &j;                           /* pointer assigned using & operator */
```

```
d = c;                           /* assign one pointer to another     */

strPointer = ASillyFunction( 3 ); /* pointer returned from a function. */
```

When returning a pointer from a function, do not return a pointer that points to a value that is local to the function or that is a pointer to a function argument. Pointers to local variables become invalid when the function exits. In the above function, the value returned points to a static variable. Returning a pointer to dynamically allocated memory is also valid.

Pointer dereferencing



The pointer `p` points to the variable `a`.

To access a value to which a pointer points, the `*` operator is used. Another operator, the `->` operator is used in conjunction with pointers to structures. Here's a short example.

```
int   c, d;
int   *pj;
struct MyStruct astruct;
struct MyStruct *bb;

c = 10;
pj = &c;             /* pj points to c */
d = *pj;             /* d is assigned the value to which pj points, 10 */
pj = &d;             /* now points to d */
*pj = 12;            /* d is now 12 */

bb = &astruct;
(*bb).m_aNumber = 3;  /* assigns 3 to the m_aNumber member of astruct */
bb->num2 = 44.3;      /* assigns 44.3 to the num2 member of astruct   */
*pj = bb->m_aNumber;  /* eqivalent to d = astruct.m_aNumber;          */
```

The expression `bb->mem` is entirely equivalent to `(*bb).mem`. They both access the `mem` element of the structure pointed to by `bb`. There is one more way of dereferencing a pointer, which will be discussed in the following section.

When dereferencing a pointer that points to an invalid memory location, an error often occurs which results in the program terminating. The error is often reported as a segmentation error. A common cause of this is failure to initialize a pointer before trying to dereference it.

C is known for giving you just enough rope to hang yourself, and pointer dereferencing is a prime example. You are quite free to write code that accesses memory outside that which you have explicity requested from the system. And many times, that memory may appear as available to your program due to the vagaries of system memory allocation. However, even if 99 executions allow your program to run without fault, that 100th execution may be the time when your "memory pilfering" is caught by the system and the program fails. Be careful to ensure that your pointer offsets are within the bounds of allocated memory!

The declaration `void *somePointer;` is used to declare a pointer of some nonspecified type. You can assign a value to a void pointer, but you must cast the variable to point to some specified type before you can dereference it. Pointer arithmetic is also not valid with `void *` pointers.

Pointers and Arrays

Up to now, we've carefully been avoiding discussing arrays in the context of pointers. The interaction of pointers and arrays can be confusing but here are two fundamental statements about it:

- A variable declared as an array of some type acts as a pointer to that type. When used by itself, it points to the first element of the array.
- A pointer can be indexed like an array name.

The first case often is seen to occur when an array is passed as an argument to a function. The function declares the parameter as a pointer, but the actual argument may be the name of an array. The second case often occurs when accessing dynamically allocated memory. Let's look at examples of each. In the following code, the call to calloc() effectively allocates an array of struct MyStruct items.

```
float KrazyFunction( struct MyStruct *parm1, int p1size, int bb )
{
  int ix;
  for (ix=0; ix<p1size; ix++) {
     if (parm1[ix].m_aNumber == bb )
         return parm1[ix].num2;
  }
  return 0.0f;
}
.
.
struct MyStruct myArray[4];
#define MY_ARRAY_SIZE (sizeof(myArray)/sizeof(struct MyStruct))
float v3;
struct MyStruct *secondArray;
int   someSize;
int   ix;
.
/* initialization of myArray ... */
.
v3 = KrazyFunction( myArray, MY_ARRAY_SIZE, 4 );
.
secondArray = calloc( someSize, sizeof(struct MyStruct));
for (ix=0; ix<someSize; ix++) {
    secondArray[i].m_aNumber = ix *2;
    secondArray[i].num2 = .304 * ix * ix;
}
```

Pointers and array names can pretty much be used interchangably. There are exceptions. You cannot assign a new pointer value to an array name. The array name will always point to the first element of the array. In the function `KrazyFunction` above, you could however assign a new value to parm1, as it is just a copy of the value for myArray. It is also valid for a function to return a pointer to one of the array elements from an array passed as an argument to a function. A function should never return a pointer to a local variable, even though the compiler will probably not complain.

When declaring parameters to functions, declaring an array variable without a size is equivalent to declaring a pointer. Often this is done to emphasize the fact that the pointer variable will be used in a manner equivalent to an array.

```
/* two equivalent function definitions */
int LittleFunction( int *paramN );
int LittleFunction( int paramN[] );
```

Now we're ready to discuss pointer arithmetic. You can add and subtract integer values to/from pointers. If myArray is declared to be some type of array, the expression `*(myArray+j)`, where j is an integer, is equivalent to `myArray[j]`. So for instance in the above example where we had the expression secondArray[i].num2, we could have written that as `*(secondArray+i).num2` or more simply `(secondArray+i)->num2`.

Note that for addition and subtraction of integers and pointers, the value of the pointer is not adjusted by the integer amount, but is adjusted by the amount multiplied by the size (in bytes) of the type to which the pointer refers. One pointer may also be subtracted from another, provided they point to elements of the same array (or the position just beyond the end of the array). If you have a pointer that points to an element of an array, the index of the element is the result when the array name is subtracted from the pointer. Here's an example.

```
struct MyStruct someArray[20];
struct MyStruct *p2;
int idx;

.
/* array initialiation .. */
.
for (p2 = someArray; p2 < someArray+20;  ++p2) {
   if (p2->num2 > testValue) break;
}
idx = p2 - someArray;
```

You may be wondering how pointers and multidimensional arrays interact. Lets look at this a bit in detail. Suppose A is declared as a two dimensional array of floats (`float A[D1][D2];`) and that pf is declared a pointer to a float. If pf is initialized to point to A[0][0], then *(pf+1) is equivalent to A[0][1] and *(pf+D2) is equivalent to A[1][0]. The elements of the array are stored in row-major order.

```
float A[6][8];
float *pf;
pf = &A[0][0];
*(pf+1) = 1.3;   /* assigns 1.3 to A[0][1] */

*(pf+8) = 2.3;   /* assigns 2.3 to A[1][0] */
```

Let's look at a slightly different problem. We want to have an two dimensonal array, but we don't need to have all the rows the same length. What we do is declare an array of pointers. The second line below declares A as an array of pointers. Each pointer points to an float. Here's some applicable code:

```
float  linearA[30];
float *A[6];

A[0] = linearA;              /*  5 - 0 = 5 elements in row           */
A[1] = linearA + 5;          /* 11 - 5 = 6 elements in row  */
A[2] = linearA + 11;         /* 15 - 11 = 4 elements in row */
A[3] = linearA + 15;         /* 21 - 15 = 5 elements        */
A[4] = linearA + 21;         /* 25 - 21 = 4 elements        */
A[5] = linearA + 25;         /* 30 - 25 = 5 elements        */

A[3][2] = 3.66;          /* assigns 3.66 to linear[17];     */
A[3][-3] = 1.44;         /* refers to linear[12];
                            negative indices are sometimes useful. */
```

We also note here something curious about array indexing. Suppose myArray is an array and idx is an integer value. The expression myArray[idx] is equivalent to idx[myArray]. The first is equivalent to *(myArray+idx), and the second is equivalent to *(idx+myArray). These turn out to be the same, since the addition is commutative.

Pointers can be used with preincrement or post decrement, which is sometimes done within a loop, as in the following example. The increment and decrement applies to the pointer, not to the object to which the pointer refers. In other words, *pArray++ is equivalent to *(pArray++).

```
long  myArray[20];
long  *pArray;
int  i;

/* Assign values to the entries of myArray */
pArray = myArray;
for (i=0; i<10; ++i) {
  *pArray++ = 5 + 3*i + 12*i*i;
  *pArray++ = 6 + 2*i + 7*i*i;
}
```

Pointers in Function Arguments

Often we need to invoke a function with an argument that is itself is a pointer. In many instances, the variable is itself a parameter for the current function and may be a pointer to some type of structure. The ampersand character is not needed in this circumstance to obtain a pointer value, as the variable is itself a pointer. In the example below, the variable `pStruct`, a pointer, is a parameter to function `FunctTwo,` and is passed as an argument to `FunctOne`. The second parameter to `FunctOne` is an int. Since in function `FunctTwo, mValue` is a pointer to an int, the pointer must first be dereferenced using the * operator, hence the second argument in the call is `*mValue`. The third parameter to function `FunctOne` is a pointer to a long. Since `pAA` is itself a pointer to a long, no ampersand is needed when it is used as the third argument to the function.

```
int FunctOne( struct SomeStruct *pValue, int iValue, long *lValue )
{
   /*  do some stuff ... */
   . . .
   return 0;
}
int FunctTwo( struct someStruct *pStruct, int *mValue )
{
   int j;
   long  AnArray[25];
   long *pAA;

   pAA = &AnArray[13];
   j = FunctOne( pStruct, *mValue, pAA );
   return j;
}
```

Pointers and Text Strings

Historically, text strings in C have been implemented as arrays of characters, with the last character in the string being a zero, or the NULL character. Most C implementations come with a standard library of functions for manipulating strings. Many of the more commonly used functions expect the strings to be null terminated strings of characters. To use these functions requires the inclusion of the standard C header file "string.h".

A statically declared, initialized string would look similar to the following:

```
static const char *myFormat = "Total Amount Due: %d";
```

The variable `myFormat` can be viewed as an array of 21 characters. There is an implied null character ('\0') tacked on to the end of the string after the 'd' as the 21st item in the array. You can also initialize the individual characters of the array as follows:

```
static const char myFlower[] = { 'P', 'e', 't', 'u', 'n', 'i', 'a', '\0' };
```

An initialized array of strings would typically be done as follows:

```
static const char *myColors[] = {
   "Red", "Orange", "Yellow", "Green", "Blue", "Violet" };
```

The initilization of an especially long string can be split across lines of source code as follows.

```
static char *longString = "Hello. My name is Rudolph and I work as a reindeer "
  "around Christmas time up at the North Pole.  My boss is a really swell guy."
  " He likes to give everybody gifts.";
```

The library functions that are used with strings are discussed in a later chapter.

Pointers to Functions

C also allows you to create pointers to functions. Pointers to functions can get rather messy. Declaring a typedef to a function pointer generally clarifies the code. Here's an example that uses a function pointer, and a void * pointer to implement what's known as a callback. The `DoSomethingNice` function invokes a caller supplied function `TalkJive` with caller data. Note that `DoSomethingNice` really doesn't know anything about what `dataPointer`refers to.

```
typedef  int (*MyFunctionType)( int, void *);      /* a typedef for a function pointer */

int DoSomethingNice( int aVariable, MyFunctionType aFunction, void *dataPointer )
{
    int rv = 0;
    if (aVariable < THE_BIGGEST) {
       /* invoke function through function pointer (old style) */
       rv = (*aFunction)(aVariable, dataPointer );

       /* invoke function through function pointer (new style) */
       rv = aFunction(aVariable, dataPointer );
    }
    return rv;
}
.
.
struct sDataINeed {
    int    colorSpec;
    char   *phrase;
 }
typedef struct sDataINeed  DataINeed;
.
int TalkJive( int myNumber, void *someStuff )
{
    /* recast void * to pointer type specifically needed for this function */
    DataINeed *myData = someStuff;
    /* talk jive. */
    return 5;
}
.
.
static DataINeed  sillyStuff = { BLUE, "Whatcha talkin 'bout Willis?" };

DoSomethingNice( 41, &TalkJive,  &sillyStuff );
```

Some versions of C may not require an ampersand preceeding the `TalkJive` argument in the `DoSomethingNice` call. Some implementations may require specifically casting the argument to the `MyFunctionType` type, even though the function signature exacly matches that of the typedef.

Function pointers can be useful for implementing a form of polymorphism in C. First one declares a structure having as elements function pointers for the various operations to that can be specified polymorphically. A second base object structure containing a pointer to the previous structure is also declared. A class is defined by extending the second structure with the data specific for the class, and static variable of the type of the first structure, containing the addresses of the functions that are associated with the class. This type of polymorphism is used in the standard library when file I/O functions are called.

A similar mechanism can also be used for implementing a state machine in C. A structure is defined which contains function pointers for handling events that may occur within state, and for functions to be invoked upon entry to and exit from the state. An instance of this structure corresponds to a state. Each state is initialized with pointers to functions appropriate for the state. The current state of the state machine is in effect a pointer to one of these states. Changing the value of the current state pointer effectively changes the current state. When some event occurs, the appropriate function is called through a function pointer in the current state.

Examples of pointer constructs

Find below some examples of pointer constucts which may will help you creating your needed pointer.

```
int i;         // integer variable
int *p;        // pointer to integer variable
int a[];       // array of integer
int f();       // function with returnvalue integer
int **pp;      // pointer to pointer to integer
int (*pa)[];   // pointer to an array of integer
int (*pf)();   // pointer to a function with returnvalue integer
int *ap[];     // array of pointers to integer
int *fp();     // function, which returns a pointer to an integer
int ***ppp;    // pointer to a pointer to a pointer to integer
int (**ppa)[]; // pointer to a pointer to an array of integer
int (**ppf)(); // pointer to a pointer to a function with returnvalue integer
int *(*pap)[]; // pointer to an array of pointers to integer
int *(*pfp)(); // pointer to function with returnvalue pointer to integer
int **app[];   // array of pointer to pointer to integer
int (*apa[])[];// array of pointers to array of integer
int (*apf[])();// array of pointers to functions with returnvalue integer
int ***fpp();  // function with returnvalue pointer to pointer to pointer to int
int (*fpa())[];// function with returnvalue pointer to array of integers
int (*fpf())();// function with returnvalue pointer to function, which returns an integer
```

sizeof

The sizeof() operator is often used to refer to the size of a static array declared earlier in the same function.

To find the end of an array (example from wikipedia:Buffer_overflow):

```
/* better.c - demonstrates one method of fixing the problem */


#include <stdio.h>

#include <string.h>

int main(int argc, char *argv[])
{
  char buffer[10];
  if (argc < 2)
  {
    fprintf(stderr, "USAGE: %s string\n", argv[0]);
    return 1;
  }
  strncpy(buffer, argv[1], sizeof(buffer));
  buffer[sizeof(buffer) - 1] = '\0';
  return 0;
}
```

To iterate over every element of an array, use

```
#define NUM_ELEM(x) (sizeof (x) / sizeof (*(x)))

for( i = 0; i < NUM_ELEM(array); i++ )
{
    /* do something with array[i] */
    ;
}
```

Note that the use of `sizeof()` above is only a convenience syntax, and works only because `buffer` was declared as an array of 10 char's earlier in the function, and the compiler can thus replace `sizeof(buffer)` with the number 10 at compile time (equivalent to us hard-coding 10 into the code in place of `sizeof(buffer)`). The information about the length of `buffer` is not actually stored anywhere in memory (unless we keep track of it separately) and cannot be programmatically obtained at run time from the array/pointer itself.

Often a function needs to know the size of an array it was given. Unfortunately, (in C and C++) this is not possible, because (as mentioned above) the size of an array is not stored anywhere.

There are 4 common ways to work around this fact:

- Write the function to require, for each array parameter, a "length" parameter. (Typically we use sizeof() at the point where this function is called).
- Use a convention such as null-terminated string to mark the end of the array.
- Instead of passing raw arrays, pass a structure that includes the length of the array (".length") as well as the array (or a pointer to the first element); similar to the `string` or `vector` classes in C++.

Memory Management

In C, you have already considered creating variables for use in the program. You have created some arrays for use, but you may have already noticed some limitations:

- that the size of the array must be known beforehand
- that the size of the array cannot be changed in the duration of your program

Dynamic memory allocation in C is a way of circumventing these problems.

Malloc

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
void free(void *ptr);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
```

The C function `malloc` is the means of implementing dynamic memory allocation. It is defined in stdlib.h or malloc.h, depending on what operating system you may be using. Malloc.h contains only the definitions for the memory allocation functions and not the rest of the other functions defined in stdlib.h. Usually you will not need to be so specific in your program, and if both are supported, you should use <stdlib.h>, since that is ANSI C, and what we will use here.

The corresponding call to release allocated memory back to the operating system is `free`.

When dynamically allocated memory is no longer needed, `free` should be called to release it back to the memory pool. Overwriting a pointer that points to dynamically allocated memory can result in that data becoming inaccessible. If this happens frequently, eventually the operating system will no longer be able to allocate more memory for the process. Once the process exits, the operating system is able to free all dynamically allocated memory associated with the process.

Let's look at how dynamic memory allocation can be used for arrays.

Normally when we wish to create an array we use a declaration such as

```
int array[10];
```

Recall `array` can be considered a pointer which we use as an array. We specify the length of this array is 10 `int`s. After array[0], nine other integers have space to be stored consecutively.

Sometimes it is not known at the time the program is written how much memory will be needed for some data. In this case we would want to dynamically allocate required memory after the program has started executing. To do this we only need to declare a pointer, and invoke malloc when we wish to make space for the elements in our array, *or*, we can tell malloc to make space when we first initialize the array. Either way is acceptable and useful.

We also need to know how much an int takes up in memory in order to make room for it; fortunately this is not difficult, we can use C's builtin `sizeof` operator. For example, if `sizeof(int)` yields 4, then one `int` takes up 4 bytes. Naturally, `2*sizeof(int)` is how much memory we need for 2 `int`s, and so on.

So how do we malloc an array of ten `int`s like before? If we wish to declare and make room in one hit, we can simply say

```
int *array = malloc(10*sizeof(int));
```

We only need to declare the pointer to the array, `malloc` gives us some space to store the 10 `int`s afterward.

Important note! `malloc` does *not* initialize the array! Like creating arrays without dynamic allocation, the programmer must initialize the array with sensible values before using it. Make sure you do so, too. (*See later the function `memset` for a simple method.)*

It is not necessary to immediately call malloc after declairing a pointer for the allocated memory. Often a number of statements exist between the declaration and the call to malloc, as follows:

```
int *array;
printf("Hello World!!!");
/* more statements */
array = malloc(10*sizeof(int));
/* use the array */
```

Error checking

When we want to use `malloc`, we have to be mindful that the pool of memory available to the programmer is *finite*. As such, we can conceivably run out of memory! In this case, `malloc` will return `NULL`. In order to stop the program crashing from having no more memory to use, one should always check that malloc has not returned `NULL` before attempting to use the memory; we can do this by

```
int *pt;
pt = malloc(3 * sizeof(int));
if(pt == NULL)
{
   printf("Out of memory, exiting\n");
   exit(1);
}
```

Of course, suddenly quitting as in the above example is not always appropriate, and depends on the problem you are trying to solve and the architecture you are programming for. For example if program is a small, non critical application that's running on a desktop quitting may be appropriate. However if the program is some type of editor running on a desktop, you may want to give the operator the option of saving his tediously entered information instead of just exiting the program. A memory allocation failure in an embedded processor, such as might be in a washing machine, could cause an automatic reset of the machine.

The `calloc` function

The `calloc` function allocates space for an array of items and initilizes the memory to zeros. The call `mArray = calloc( count, sizeof(struct V))` allocates `count` objects, each of whose size is sufficient to contain an instance of the structure `struct V`. The space is initialized to all bits zero. The function returns either a pointer to the allocated memory or, if the allocation fails, `NULL`.

The `realloc` function

The `realloc` function changes the size of the object pointed to by `ptr` to the size specified by `size`. The contents of the object shall be unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the object is indeterminate. If `ptr` is a null pointer, the `realloc` function behaves like the `malloc` function for the specified size. Otherwise, if `ptr` does not match a pointer earlier returned by the `calloc`, `malloc`, or `realloc` function, or if the space has been deallocated by a call to the `free` or `realloc` function, the behavior is undefined. If the space cannot be allocated, the object pointed to by `ptr` is unchanged. If `size` is zero and `ptr` is not a null pointer, the object pointed to is freed. The `realloc` function returns either a null pointer or a pointer to the possibly moved allocated object.

The `free` function

Memory that has been allocated using `malloc, realloc, or calloc` must be released back to the system memory pool once it is no longer needed. This is done to avoid perpetually allocating more and more memory, which could result in an eventual memory allocation failure. Memory that is not released with `free` is however released when the current program terminates on most operating systems. Calls to `free` are as in the following example.

```
int *myStuff = malloc( 20 * sizeof(int));
if (myStuff != NULL ) {
  /* more statements here */
  /* time to release myStuff */
  free( myStuff );
}
```

It should be noted that `free` is neither intelligent nor recursive. The following code that depends on the recursive application of free to the internal variables of a struct does not work.

```
typedef struct BSTNode
{
   int value;
   struct BSTNode* left;
   struct BSTNode* right;
} BSTNode;
// Later: ...
BSTNode* temp = (BSTNode*) calloc(1, sizeof(BSTNode));
temp->left = (BSTNode*) calloc(1, sizeof(BSTNode));
free(temp);
```

`free` temp will *not* free temp->left.

Furthermore, using `free` when the pointer in question was never allocated in the first place often crashes or leads to mysterious bugs further along. ← Memory management | Complex types →

Strings

A **string** in C is merely an array of characters. The length of a string is determined by a terminating null character: `'\0'`. So, a string with the contents, say, `"abc"` has four characters: `'a'`, `'b'`, `'c'`, and the terminating null character.

The terminating null character has the value zero.

The `<string.h>` Standard Header

Because programmers find raw strings cumbersome to deal with, they wrote the code in the `<string.h>` library. It represents not a concerted design effort but rather the accretion of contributions made by various authors over a span of years.

First, three types of functions exist in the string library:

- the `mem` functions manipulate sequences of arbitrary characters without regard to the null character;
- the `str` functions manipulate null-terminated sequences of characters;
- the `strn` functions manipulate sequences of non-null characters.

The more commonly-used string functions

The nine most commonly used functions in the string library are:

- `strcat` - concatenate two strings
- `strchr` - string scanning operation

- `strcmp` - compare two strings
- `strcpy` - copy a string
- `strlen` - get string length
- `strncat` - concatenate one string with part of another
- `strncmp` - compare parts of two strings
- `strncpy` - copy part of a string
- `strrchr` - string scanning operation

The `strcat` function

`char *strcat(char * restrict s1, const char * restrict s2);`

The `strcat()` function shall append a copy of the string pointed to by `s2` (including the terminating null byte) to the end of the string pointed to by `s1`. The initial byte of `s2` overwrites the null byte at the end of `s1`. If copying takes place between objects that overlap, the behavior is undefined. The function returns `s1`.

This function is used to attach one string to the end of another string. It is imperative that the first string (`s1`) have the space needed to store both strings.

Example:

```
#include <stdio.h>
#include <string.h>
...
static const char *colors[] = {"Red","Orange","Yellow","Green","Blue","Purple" };
static const char *widths[] = {"Thin","Medium","Thick","Bold" };
...
char *penText;
...
penColor = 3; penThickness = 2;
strcpy(penText, colors[penColor]);
strcat(penText, colors[penThickness]);
printf("My pen is %s\n", penText); // prints 'My pen is GreenThick'
```

Before calling `strcat()`, the destination must currently contain a null terminated string or the first character must have been initialized with the null character (e.g. `penText[0] = '\0';`).

The following is a public-domain implementation of `strcat`:

```
#include <string.h>
/* strcat */
char *(strcat)(char *restrict s1, const char *restrict s2)
{
    char *s = s1;
    /* Move s so that it points to the end of s1.  */
    while (*s != '\0')
        s++;
    /* Copy the contents of s2 into the space at the end of s1.  */
    strcpy(s, s2);
    return s1;
}
```

The `strchr` function

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

The `strchr()` function shall locate the first occurrence of `c` (converted to a `char`) in the string pointed to by `s`. The terminating null byte is considered to be part of the string. The function returns the location of the found character, or a null pointer if the character was not found.

This function is used to find certain characters in strings.

At one point in history, this function was named `index`. The `strchr` name, however cryptic, fits the general pattern for naming.

The following is a public-domain implementation of `strchr`:

```
#include <string.h>
/* strchr */
char *(strchr)(const char *s, int c)
{
    /* Scan s for the character.  When this loop is finished,
       s will either point to the end of the string or the
       character we were looking for.  */
    while (*s != '\0' && *s != (char)c)
         s++;
    return ( (*s == c) ? (char *) s : NULL );
}
```

The `strcmp` function

`int strcmp(const char *s1, const char *s2);`

A rudimentary form of string comparison is done with the strcmp() function. It takes two strings as arguments and returns a value less than zero if the first is lexographically less than the second, a value greater than zero if the first is lexographically greater than the second, or zero if the two strings are equal. The comparison is done by comparing the coded (ascii) value of the chararacters, character by character.

This simple type of string comparison is nowadays generally considered unacceptable when sorting lists of strings. More advanced algorithms exist that are capable of producing lists in dictionary sorted order. They can also fix problems such as strcmp() considering the string "Alpha2" greater than "Alpha12". What we're saying is, don't use this `strcmp()` alone for general string sorting in any commercial or professional code.

The `strcmp()` function shall compare the string pointed to by `s1` to the string pointed to by `s2`. The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type `unsigned char`) that differ in the strings being compared. Upon completion, `strcmp()` shall return an integer greater than, equal to, or less than 0, if the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2`, respectively.

Since comparing pointers by themselves is not practically useful unless one is comparing pointers within the same array, this function lexically compares the strings that two pointers point to.

This function is useful in comparisons, e.g.

```
if (strcmp(s, "whatever") == 0) /* do something */
    ;
```

Because the type of string comparison done by `strcmp()` is rather simple minded, when it is used to sort lists of strings, the lists are often not sorted as one would expect. It is generally considered unacceptable to use strcmp in commercial software for general sorting of lists of strings. Better comparison algorithms exist which result in string lists being sorted in the order they would appear in a dictionary, and can deal with problems such as making sure that `"Alpha2"` compares as less than `"Alpha12"`. (In the previous example, `"Alpha2"` compares greater than `"Alpha12"` because `'2'` comes after `'1'` in the character set.)

The collating sequence used by `strcmp()` is equivalent to the machine's native character set. The only guarantee about the order is that the digits from `'0'` to `'9'` are in consecutive order.

The following is a public-domain implementation of `strcmp`:

```
#include <string.h>
/* strcmp */
int (strcmp)(const char *s1, const char *s2)
{
    unsigned char uc1, uc2;
    /* Move s1 and s2 to the first differing characters
       in each string, or the ends of the strings if they
       are identical.  */
    while (*s1 != '\0' && *s1 == *s2) {
        s1++;
        s2++;
    }
    /* Compare the characters as unsigned char and
       return the difference.  */
    uc1 = (*(unsigned char *) s1);
    uc2 = (*(unsigned char *) s2);
    return ((uc1 < uc2) ? -1 : (uc1 > uc2));
}
```

The `strcpy` function

```
char *strcpy(char *restrict s1, const char *restrict s2);
```

The `strcpy()` function shall copy the string pointed to by `s2` (including the terminating null byte) into the array pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined. The function returns `s1`. No value is used to indicate an error.

Example:

```
   #include <stdio.h>
   #include <string.h>
   ...
   static const char *penType="round";
   ...
   char penText[20];
   ...
   strcpy(penText, penType);
```

Important: When you call this function, you must ensure that the destination is able to contain all the characters in the source array. Not doing so can have very serious consequences including compromising the security and integrity of your entire computer. This is also true with some of the other functions such as `strcat()`. This is very unlikely and in most cases a problem will simply result in the program crashing, or not functioning correctly. But this problem has been at the root of many computer security problems that you may have read or heard about in recent times.

The `s1` pointer must pointer to a buffer with enough space to store the string pointed to by `s2`, or undefined behavior may result, including (but not limited to) monkeys flying out of your nose.

This function is used to copy one string to another, as direct assignment of pointers can be unwieldingly tricky.

The following is a public-domain implementation of `strcpy`:

```
#include <string.h>
/* strcpy */
char *(strcpy)(char *restrict s1, const char *restrict s2)
{
    char *dst = s1;
    const char *src = s2;
    /* Do the copying in a loop.  */
    while ((*dst++ = *src++) != '\0')
        ;
    /* Return the destination string.  */
    return s1;
}
```

The `strlen` function

```
size_t strlen(const char *s);
```

The `strlen()` function shall compute the number of bytes in the string to which `s` points, not including the terminating null byte. It returns the number of bytes in the string. No value is used to indicate an error.

The following is a public-domain implementation of `strlen`:

```
#include <string.h>
/* strlen */
size_t (strlen)(const char *s)
{
    char *p = s;
    /* Loop over the data in s.  */
    while (*p != '\0')
        p++;
    return (size_t)(p - s);
}
```

The `strncat` function

`char *strncat(char *restrict s1, const char *restrict s2, size_t n);`

The `strncat()` function shall append not more than `n` bytes (a null byte and bytes that follow it are not appended) from the array pointed to by `s2` to the end of the string pointed to by `s1`. The initial byte of `s2` overwrites the null byte at the end of `s1`. A terminating null byte is always appended to the result. If copying takes place between objects that overlap, the behavior is undefined. The function returns `s1`.

The following is a public-domain implementation of `strncat`:

```
#include <string.h>
/* strncat */
char *(strncat)(char *restrict s1, const char *restrict s2, size_t n)
{
    char *s = s1;
    /* Loop over the data in s1.  */
    while (*s != '\0')
        s++;
    /* s now points to s1's trailing null character, now copy
       up to n bytes from s2 into s1 stopping if a null character
       is encountered in s2.
       It is not safe to use strncpy here since it copies EXACTLY n
       characters, NULL padding if necessary.  */
    while (n != 0 && (*s = *s2++) != '\0') {
        n--;
        s++;
    }
    if (*s != '\0')
        *s = '\0';
    return s1;
}
```

The `strncmp` function

```
int strncmp(const char *s1, const char *s2, size_t n);
```

The `strncmp()` function shall compare not more than `n` bytes (bytes that follow a null byte are not compared) from the array pointed to by `s1` to the array pointed to by `s2`. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type `unsigned char`) that differ in the strings being compared. See `strcmp` for an explanation of the return value.

This function is useful in comparisons, as the `strcmp` function is.

The following is a public-domain implementation of `strncmp`:

```
#include <string.h>
/* strncmp */
int (strncmp)(const char *s1, const char *s2, size_t n)
{
    unsigned char uc1, uc2;
    /* Nothing to compare?  Return zero.  */
    if (n == 0)
        return 0;
    /* Loop, comparing bytes.  */
    while (n-- > 0 && *s1 == *s2) {
        /* If we've run out of bytes or hit a null, return zero
           since we already know *s1 == *s2.  */
        if (n == 0 || *s1 == '\0')
            return 0;
        s1++;
        s2++;
    }
    uc1 = (*(unsigned char *) s1);
    uc2 = (*(unsigned char *) s2);
    return ((uc1 < uc2) ? -1 : (uc1 > uc2));
}
```

The `strncpy` function

```
char *strncpy(char *restrict s1, const char *restrict s2, size_t n);
```

The `strncpy()` function shall copy not more than `n` bytes (bytes that follow a null byte are not copied) from the array pointed to by `s2` to the array pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined. If the array pointed to by `s2` is a string that is shorter than `n` bytes, null bytes shall be appended to the copy in the array pointed to by `s1`, until `n` bytes in all are written. The function shall return s1; no return value is reserved to indicate an error.

It is possible that the function will **not** return a null-terminated string, which happens if the `s2` string is longer than `n` bytes.

The following is a public-domain version of `strncpy`:

```
#include <string.h>
/* strncpy */
```

```
char *(strncpy)(char *restrict s1, const char *restrict s2, size_t n)
{
    char *dst = s1;
    const char *src = s2;
    /* Copy bytes, one at a time.  */
    while (n > 0) {
        n--;
        if ((*dst++ = *src++) == '\0') {
            /* If we get here, we found a null character at the end
               of s2, so use memset to put null bytes at the end of
               s1.  */
            memset(dst, '\0', n);
            break;
        }
    }
    return s1;
}
```

The `strrchr` function

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

`strrchr` is similar to `strchr`, except the string is searched right to left.

The `strrchr()` function shall locate the last occurrence of `c` (converted to a `char`) in the string pointed to by `s`. The terminating null byte is considered to be part of the string. Its return value is similar to `strchr`'s return value.

At one point in history, this function was named `rindex`. The `strrchr` name, however cryptic, fits the general pattern for naming.

The following is a public-domain implementation of `strrchr`:

```
#include <string.h>
/* strrchr */
char *(strrchr)(const char *s, int c)
{
    const char *last = NULL;
    /* If the character we're looking for is the terminating null,
       we just need to look for that character as there's only one
       of them in the string.  */
    if (c == '\0')
        return strchr(s, c);
    /* Loop through, finding the last match before hitting NULL.  */
    while ((s = strchr(s, c)) != NULL) {
        last = s;
        s++;
    }
    return (char *) last;
}
```

The less commonly-used string functions

The less-used functions are:

- `memchr` - Find a byte in memory
- `memcmp` - Compare bytes in memory
- `memcpy` - Copy bytes in memory
- `memmove` - Copy bytes in memory with overlapping areas
- `memset` - Set bytes in memory
- `strcoll` - Compare bytes according to a locale-specific collating sequence
- `strcspn` - Get the length of a complementary substring
- `strerror` - Get error message
- `strpbrk` - Scan a string for a byte
- `strspn` - Get the length of a substring
- `strstr` - Find a substring
- `strtok` - Split a string into tokens
- `strxfrm` - Transform string

Copying functions

The `memcpy` function

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

The `memcpy()` function shall copy `n` bytes from the object pointed to by `s2` into the object pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined. The function returns `s1`.

Because the function does not have to worry about overlap, it can do the simplest copy it can.

The following is a public-domain implementation of `memcpy`:

```
#include <string.h>
/* memcpy */
void *(memcpy)(void * restrict s1, const void * restrict s2, size_t n)
{
    char *dst = s1;
    const char *src = s2;
    /* Loop and copy.  */
    while (n-- != 0)
        *dst++ = *src++;
    return s1;
}
```

The `memmove` function

```
void *memmove(void *s1, const void *s2, size_t n);
```

The `memmove()` function shall copy `n` bytes from the object pointed to by `s2` into the object pointed to by `s1`. Copying takes place as if the `n` bytes from the object pointed to by `s2` are first copied into a temporary array of `n` bytes that does not overlap the objects pointed to by `s1` and `s2`, and then the `n` bytes from the temporary array are copied into the object pointed to by `s1`. The function returns the value of `s1`.

The easy way to implement this without using a temporary array is to check for a condition that would prevent an ascending copy, and if found, do a descending copy.

The following is a public-domain implementation of `memmove`:

```
#include <string.h>
/* memmove */
void *(memmove)(void *s1, const void *s2, size_t n)
{
   /* note: these don't have to point to unsigned chars */
   char *p1 = s1;
   const char *p2 = s2;
   /* test for overlap that prevents an ascending copy */
   if (p2 < p1 && p1 < p2 + n) {
       /* do a descending copy */
       p2 += n;
       p1 += n;
       while (n-- != 0)
           *--p1 = *--p2;
   } else
       while (n-- != 0)
           *p1++ = *p2++;
   return s1;
}
```

Comparison functions

The `memcmp` function

```
int memcmp(const void *s1, const void *s2, size_t n);
```

The `memcmp()` function shall compare the first `n` bytes (each interpreted as `unsigned char`) of the object pointed to by `s1` to the first `n` bytes of the object pointed to by `s2`. The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type `unsigned char`) that differ in the objects being compared.

The following is a public-domain implementation of `memcmp`:

```
#include <string.h>
/* memcmp */
int (memcmp)(const void *s1, const void *s2, size_t n)
{
    unsigned char *us1 = (unsigned char *) s1;
    unsigned char *us2 = (unsigned char *) s2;
    while (n-- != 0) {
        if (*us1 != *us2)
            return (*us1 < *us2) ? -1 : +1;
        us1++;
        us2++;
    }
    return 0;
}
```

The `strcoll` and `strxfrm` functions

```
int strcoll(const char *s1, const char *s2);
```

```
size_t strxfrm(char *s1, const char *s2, size_t n);
```

The ANSI C Standard specifies two locale-specific comparison functions.

The `strcoll` function compares the string pointed to by `s1` to the string pointed to by `s2`, both interpreted as appropriate to the `LC_COLLATE` category of the current locale. The return value is similar to `strcmp`.

The `strxfrm` function transforms the string pointed to by `s2` and places the resulting string into the array pointed to by `s1`. The transformation is such that if the `strcmp` function is applied to the two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the `strcoll` function applied to the same two original strings. No more than `n` characters are placed into the resulting array pointed to by `s1`, including the terminating null character. If `n` is zero, `s1` is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined. The function returns the length of the transformed string.

These functions are rarely used and nontrivial to code, so there is no code for this section.

Search functions

The `memchr` function

```
void *memchr(const void *s, int c, size_t n);
```

The `memchr()` function shall locate the first occurrence of `c` (converted to an `unsigned char`) in the initial `n` bytes (each interpreted as `unsigned char`) of the object pointed to by `s`. If `c` is not found, `memchr` returns a null pointer.

The following is a public-domain implementation of `memchr`:

```
#include <string.h>
/* memchr */
void *(memchr)(const void *s, int c, size_t n)
{
    const unsigned char *src = s;
    unsigned char uc = c;
    while (n-- != 0) {
        if (*src == uc)
            return (void *) src;
        src++;
    }
    return NULL;
}
```

The `strcspn`, `strpbrk`, and `strspn` functions

```
size_t strcspn(const char *s1, const char *s2);

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

size_t strspn(const char *s1, const char *s2);
```

The `strcspn` function computes the length of the maximum initial segment of the string pointed to by `s1` which consists entirely of characters **not** from the string pointed to by `s2`.

The `strpbrk` function locates the first occurrence in the string pointed to by `s1` of any character from the string pointed to by `s2`, returning a pointer to that character or a null pointer if not found.

The `strspn` function computes the length of the maximum initial segment of the string pointed to by `s1` which consists entirely of characters from the string pointed to by `s2`.

All of these functions are similar except in the test and the return value.

The following are public-domain implementations of `strcspn`, `strpbrk`, and `strspn`:

```
#include <string.h>
/* strcspn */
size_t (strcspn)(const char *s1, const char *s2)
{
    const char *sc1;
    for (sc1 = s1; *sc1 != '\0'; sc1++)
        if (strchr(s2, *sc1) != NULL)
            return (sc1 - s1);
    return sc1 - s1;            /* terminating nulls match */
}
#include <string.h>
/* strpbrk */
char *(strpbrk)(const char *s1, const char *s2)
{
    const char *sc1;
    for (sc1 = s1; *sc1 != '\0'; sc1++)
        if (strchr(s2, *sc1) != NULL)
            return (char *)sc1;
    return NULL;                /* terminating nulls match */
}
#include <string.h>
/* strspn */
size_t (strspn)(const char *s1, const char *s2)
{
    const char *sc1;
    for (sc1 = s1; *sc1 != '\0'; sc1++)
        if (strchr(s2, *sc1) == NULL)
            return (sc1 - s1);
    return sc1 - s1;            /* terminating nulls don't match */
}
```

The `strstr` function

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

The `strstr()` function shall locate the first occurrence in the string pointed to by `s1` of the sequence of bytes (excluding the terminating null byte) in the string pointed to by `s2`. The function returns the pointer to the matching string in `s1` or a null pointer if a match is not found. If `s2` is an empty string, the function returns `s1`.

The following is a public-domain implementation of `strstr`:

```
#include <string.h>
/* strstr */
char *(strstr)(const char *s1, const char *s2)
{
    size_t s2len;
    /* Check for the null s2 case.  */
    if (*s2 == '\0')
        return (char *) s1;
    s2len = strlen(s2);
    for (; (s1 = strchr(s1, *s2)) != NULL; s1++)
        if (strncmp(s1, s2, s2len) == 0)
            return (char *) s1;
    return NULL;
}
```

The `strtok` function

`char *strtok(char *restrict s1, const char *restrict delimiters);`

A sequence of calls to `strtok()` breaks the string pointed to by `s1` into a sequence of tokens, each of which is delimited by a byte from the string pointed to by `delimiters`. The first call in the sequence has `s1` as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by `delimiters` may be different from call to call.

The first call in the sequence searches the string pointed to by `s1` for the first byte that is not contained in the current separator string pointed to by `delimiters`. If no such byte is found, then there are no tokens in the string pointed to by `s1` and `strtok()` shall return a null pointer. If such a byte is found, it is the start of the first token.

The `strtok()` function then searches from there for a byte (or multiple, consecutive bytes) that is contained in the current separator string. If no such byte is found, the current token extends to the end of the string pointed to by `s1`, and subsequent searches for a token shall return a null pointer. If such a byte is found, it is overwritten by a null byte, which terminates the current token. The `strtok()` function saves a pointer to the following byte, from which the next search for a token shall start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The `strtok()` function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

Because the `strtok()` function must save state between calls, and you could not have two tokenizers going at the same time, the Single Unix Standard defined a similar function, `strtok_r()`, that does not need to save state. Its prototype is this:

```
char *strtok_r(char *s, const char *delimiters, char **lasts);
```

The `strtok_r()` function considers the null-terminated string `s` as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string `delimiters`. The argument lasts points to a user-provided pointer which points to stored information necessary for `strtok_r()` to continue scanning the same string.

In the first call to `strtok_r()`, `s` points to a null-terminated string, `delimiters` to a null-terminated string of separator characters, and the value pointed to by `lasts` is ignored. The `strtok_r()` function shall return a pointer to the first character of the first token, write a null character into `s` immediately following the returned token, and update the pointer to which `lasts` points.

In subsequent calls, `s` is a null pointer and `lasts` shall be unchanged from the previous call so that subsequent calls shall move through the string `s`, returning successive tokens until no tokens remain. The separator string `delimiters` may be different from call to call. When no token remains in `s`, a NULL pointer shall be returned.

The following public-domain code for `strtok` and `strtok_r` codes the former as a special case of the latter:

```
#include <string.h>
/* strtok_r */
char *(strtok_r)(char *s, const char *delimiters, char **lasts)
{
    char *sbegin, *send;
    sbegin = s ? s : *lasts;
    sbegin += strspn(sbegin, delimiters);
    if (*sbegin == '\0') {
        *lasts = "";
        return NULL;
    }
    send = sbegin + strcspn(sbegin, delimiters);
    if (*send != '\0')
        *send++ = '\0';
    *lasts = send;
    return sbegin;
}
/* strtok */
char *(strtok)(char *restrict s1, const char *restrict delimiters)
{
    static char *ssave = "";
    return strtok_r(s1, delimiters, &ssave);
}
```

Miscellaneous functions

These functions do not fit into one of the above categories.

The `memset` function

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

The `memset()` function converts `c` into `unsigned char`, then stores the character into the first `n` bytes of memory pointed to by `s`.

The following is a public-domain implementation of `memset`:

```
#include <string.h>
/* memset */
void *(memset)(void *s, int c, size_t n)
{
    unsigned char *us = s;
    unsigned char uc = c;
    while (n-- != 0)
        *us++ = uc;
    return s;
}
```

The `strerror` function

```
char *strerror(int errorcode);
```

This function returns a locale-specific error message corresponding to the parameter. Depending on the circumstances, this function could be trivial to implement, but this author will not do that as it varies.

The Single Unix System Version 3 has a variant, `strerror_r`, with this prototype:

```
int strerror_r(int errcode, char *buf, size_t buflen);
```

This function stores the message in `buf`, which has a length of size `buflen`.

examples

To determine the number of characters in a string, the `strlen()` function is used.

```
   #include <stdio.h>
   #include <string.h>
   ...
   int length, length2;
   char *turkey;
   static char *flower= "begonia";
   static char *gemstone="ruby ";
   length = strlen(flower);
   printf("Length = %d\n", length); // prints 'Length = 7'
   length2 = strlen(gemstone);
   turkey = (char *)malloc( length + length2 + 1);
   if (turkey) {
   strcpy( turkey, gemstone);
   strcat( turkey, flower);
```

```
  printf( "%s\n", turkey); // prints 'ruby begonia'
  free( turkey );
  }
```

Note that the amount of memory allocated for 'turkey' is one plus the sum of the lengths of the strings to be concatenated. This is for the terminating null character, which is not counted in the lengths of the strings.

Exercises

1. The string functions use a lot of looping constructs. Is there some way to portably unravel the loops?
2. What functions are possibly missing from the library as it stands now?

← Strings | Networking in UNIX →

Complex Types

< Programming:C

In the section C types we looked at some basic types. However **C complex types** allow us greater flexibility in managing data in our C program.

Complex data types

A data structure ("struct") contains multiple pieces of data. Each piece of data (called a "member") can be accessed by the name of the variable, followed by a '.', then the name of the member. (Another way to access a member is using the member operator '->'). The member variables of a struct can be of any data type and can even be an array or a pointer.

Pointers

Pointers are variables that don't hold the actual data. Instead they point to the memory location of some other variable. For example,

```
  int *pointer = &variable;
```

defines a pointer to an `int`, and also makes it point to the particular integer contained in `variable`.

The '*' is what makes this an integer pointer. To make the pointer point to a different integer, use the form

```
pointer = &sandwitches;
```

Where & is the *address of* operator. Often programmers set the value of the pointer to NULL like this:

```
pointer = 0;
```

which tells us that the pointer isn't currently pointing to any real location.

Additionally, to dereference (access the thing being pointed at) the pointer, use the form:

```
value = *pointer;
```

Structs

A data structure contains multiple pieces of data. One defines a data structure using the `struct` keyword. For example,

```
struct mystruct
{
    int int_member;
    double double_member;
    char string_member[25];
} variable;
```

`variable` is an instance of `mystruct`. You can omit it from the end of the **`struct`** declaration and declare it later using:

```
struct mystruct variable;
```

It is often common practice to make a *type synonym* so we don't have to type "struct mystruct" all the time. C allows us the possibility to do so using a **`typedef`** statement, which aliases a type:

```
typedef struct
{
  ...
} Mystruct;
```

The **`struct`** itself has no name (by the absence of a name on the first line), but it is aliased as `Mystruct`. Then you can use

```
Mystruct variable;
```

Note that it is commonplace, and good style to capitalize the **first letter** of a type synonym. However in the actual definition we need to give the struct a *tag* so we can refer to it: we may have a *recursive data structure* of some kind.

Unions

The definition of a union is similar to that of a struct. The difference between the two is that in a struct, the members occupy different areas of memory, but in a union, the members occupy the same area of memory. Thus, in the following type, for example:

```
union {
    int i;
    double d;
} u;
```

The programmer can access either `u.i` or `u.d`, but not both at the same time. Since `u.i` and `u.d` occupy the same area of memory, modifying one modifies the value of the other, sometimes in unpredictable ways.

The size of a union is the size of its largest member.

Type modifiers

`register` is a hint to the compiler to attempt to optimise the storage of the given variable by storing it in a register of the computer's CPU when the program is run. Most optimising compilers do this anyway, so use of this keyword is often unnecessary. In fact, ANSI C states that a compiler can ignore this keyword if it so desires -- and many do. Microsoft Visual C++ is an example of an implementation that completely ignores the *register* keyword.

`volatile` is a special type modifier which informs the compiler that the value of the variable may be changed by external entities other than the program itself. This is necessary for certain programs compiled with optimisations - if a variable were not defined `volatile` then the compiler may assume that certain operations involving the variable were safe to optimise away when in fact they aren't. *volatile* is particularly relevant when working with embedded systems (where a program may not have complete control of a variable) and multi-threaded applications.

`auto` is a modifier which specifies an "automatic" variable that is automatically created when in scope and destroyed when out of scope. If you think this sounds like pretty much what you've been doing all along when you declare a variable, you're right: all declared items within a block are implicitly "automatic". For this reason, the *auto* keyword is more like the answer to a trivia question than a useful modifier, and there are lots of very competent programmers that are unaware of its existence.

`extern` is used when a file needs to access a variable in another file that it may not have #included directly. Therefore, *extern* does not actually carve out space for a new variable, it just provides the compiler with sufficient information to access the remote variable.

← Complex types | Common practices →

Networking in UNIX

Network programming under UNIX is relatively simple in C.

This guide assumes you already have a good general idea about C, UNIX and networks.

A simple client

To start with we'll look at one of the simplest things you can do, initialize a stream connection and receive a message from a remote server.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define MAXRCVLEN 500
#define PORTNUM 2343

 int main(int argc, char *argv[])
{
 char buffer[MAXRCVLEN+1]; /* +1 so we can add null terminator */
 int len, mysocket;
 struct sockaddr_in dest;

 mysocket = socket(AF_INET, SOCK_STREAM, 0);

 dest.sin_family = AF_INET;
 dest.sin_addr.s_addr = inet_addr("127.0.0.1"); /* Sets destination IP number */
 dest.sin_port = htons (PORTNUM); /* Sets destination port number */
 memset(&(dest.sin_zero), '\0', 8); /* Zeroes rest of struct */

 connect(mysocket, (struct sockaddr *)&dest,sizeof(struct sockaddr));

 len=recv(mysocket, buffer, MAXRCVLEN, 0);

 buffer[len]='\0'; /* We have to null terminate the received data ourselves */

 printf("Rcvd: %s",buffer);
 close(mysocket); /* Close the file descriptor when finished */
 return EXIT_SUCCESS;
}
```

This is the very bare bones of a client; in practice, we would check every function that we called for failure, however for clarity, error checking the code has been left out for now.

As you can see, the code mainly resolves around "dest" which is a struct of type sockaddr_in; in this struct we store information about the machine we want to connect to.

```
mysocket = socket(AF_INET, SOCK_STREAM, 0);
```

The socket function tells our OS that we want a file descriptor for a socket which we can use for a network stream connection , what the parameters mean is mostly irrelevent for now.

```
dest.sin_family = AF_INET;
dest.sin_addr.s_addr = inet_addr("127.0.0.1"); /*Sets destination IP number*/
dest.sin_port = htons(PORTNUM); /*Sets destination port number*/
memset(&(dest.sin_zero), '\0', 8); /*Zeroes rest of struct*/
```

Now we get on to the interesting part,

The first line sets the address family, this should be the same as was used in the socket function, for most purposes AF_INET will serve.

The second line is where we set the IP of the machine we need to connect to. The variable dest.sin_addr.s_addr is just an integer stored in Big Endian format, but we don't have to know that as the inet_addr function will do the conversion from string into Big Endian integer for us.

The third line sets the destination port number, the htons() function converts the port number into a Big Endian short integer. If your program is going to be solely run on machines which use Big Endian numbers as default then dest.sin_port = 21; would work just as well, however for portability reasons htons() should be used.

The fourth line uses memset to zero the rest of the struct.

Now that's all the preliminary work done, now we can actually make the connection and use it,

```
connect(mysocket, (struct sockaddr *)&dest,sizeof(struct sockaddr));
```

This tells our OS to use the socket mysocket to create a connection to the machine specified in dest.

```
inputlen=recv(mysocket, buffer, MAXRCVLEN, 0);
```

Now this receives upto MAXRCVLEN bytes of data from the connection and stores them in the buffer string. The number of characters received is returned by recv(). It is important to note that the data received will not automatically be null terminated when stored in buffer hence we need to do it ourselves with buffer[inputlen]='\0'.

And that's about it !

The next step after learning how to receive data is learning how to send it, if you've understood the previous section than this is quite easy, all you have to do is used the the send() function which uses the same parameters as receive. If in our previous example "buffer" had the text we wanted to send and its length was stored in "len" we would do send(mysocket, buffer, len, 0), send() returns the number of bytes that were sent. It is important to remember send() for various reasons may not be able to send all of the bytes so it is imporant to check that this value is equal to the number of bytes you were sending (in most cases this can be resolved by resending the unsent data).

A simple server

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

 #define PORTNUM 2343

 int main(int argc, char *argv[])
{
 char buffer[]="Hello World !\n";
 int socksize; /*we use this to store the size of out sockaddr_in struct */
 int mysocket; /* The socket we use to listen with */
 int consocket; /* Once we have a connection we get a "session" socket to use */
 struct sockaddr_in dest;  /* Info on the machine connecting to us */
 struct sockaddr_in serv; /* Info on our server */

 socksize = sizeof(struct sockaddr_in);
  mysocket = socket(AF_INET, SOCK_STREAM, 0);
  serv.sin_family = AF_INET;
 serv.sin_addr.s_addr = INADDR_ANY; /* Set the IP to our IP (given by INADDR_ANY) */
 serv.sin_port = htons(PORTNUM); /*Sets server port number*/
 memset(&(dest.sin_zero), '\0', 8); /*Zeroes rest of struct*/
  bind(mysocket, (struct sockaddr *)&serv,sizeof(struct sockaddr)); /*bind serv
information to mysocket */
 listen(mysocket,1); /* start listening, allow a queue of upto 1 pending connection*/

 while(1)
 {
  consocket = accept(mysocket, (struct sockaddr *)&dest, &socksize); /* Get data - the
program waits here for a connection*/
  printf("Incoming connection from %s - sending welcome\n",inet_ntoa(dest.sin_addr));
  send(consocket, buffer, strlen(buffer), 0);
  close(consocket);
 }

 close(mysocket);
 return EXIT_SUCCESS;
}
```

Superficially this is very similar to the client, the first important difference is that rather than creating a sockaddr_in with information about the machine we're connecting to we create it with information about the server, and then we "bind" it to the socket. This allows the machine to know the data recieved on the port specified in the sockaddr_in should be handled by our specified socket.

The listen command then tells our program to start listening using that socket, the second parameter of listen() allows us to specify the maximum number of connections that can be queued. Each time a connection is made to the server it is added to the queue, we take connections from the queue using the accept() function.

Then we have the core of the server code, we use the accept() function to take a connection from the queue, if there is no connection waiting on the queue the function causes the program to wait until a connection is received. The accept() function returns us another socket this is essentially a "session" socket solely for communicating with connection we took of the queue. The original socket (mysocket) continues to listen on the prespecified port for further connections.

Once we have "session" socket we can treated it in the same way as with the client using send() and receive() to handle data transfers.

Note that this server can only accept one connection at a time, if you want to simultaneously handle multiple clients then you'll need to use forking off seperate processes to handle the connections.

Useful network functions

```
int gethostname(char *hostname, size_t size);
```

This function takes as parameter a pointer to an array of chars and the size of that array, if possible it then finds out the hostname of the localhost and stores it in the array. On failure it returns -1.

```
struct hostent *gethostbyname(const char *name);
```

This function obtains information about a domain name and stored it in a hostent struct, in the hostent structure the most useful part is the (char**) h_addr_list field, this is a null terminated array of IP addresses associated with that domain. The field h_addr is a pointer to the first IP address in the h_addr_list array. Returns NULL on failure.

FAQs

What about stateless connections ?

If you don't want to exploit the properties of TCP in your program but rather just have to use a UDP protocol then you can just switch "SOCK_STREAM" in your socket call for "SOCK_DGRAM" and use it in the same way. It is important to remember that UDP does not guarantee delivery of packets and order of delivery, so checking is important.

If you want to exploit the properties of UDP, then you can use sendto() and recvfrom() which operate like send() and recv() except you need to provide extra parameters specifiying who you are communicating with.

You should have more info on correcting a failed condition. I have a network connection that is lost. I was looking for the recovery methods. Maybe some information on "netstat -a | grep ##.##.##.##" would help. Thanks

How do I check for errors ?

The functions socket(), recv() and connect() all return -1 on failure and use errno for further details. ← Networking in UNIX | Language extensions →

Common Practices

With its extensive use, a number of common practices and conventions have evolved to help avoid errors in C programs. These are simultaneously a demonstration of the application of good software engineering principles to a language and an indication of the limitations of C. Although few are used universally, and some are controversial, each of these enjoys wide use.

Dynamic multidimensional arrays

Although one-dimensional arrays are easy to create dynamically using malloc, and fixed-size multidimensional arrays are easy to create using the built-in language feature, dynamic multidimensional arrays are trickier. There are a number of different ways to create them, each with different tradeoffs. The two most popular ways to create them are:

- They can be allocated as a single block of memory, just like static multidimensional arrays. This requires that the array be *rectangular* (i.e. subarrays of lower dimensions are static and have the same size). The disadvantage is that the syntax of declaration the pointer is a little tricky for programmers at first. For example, if one wanted to create an array of ints of 3 columns and `rows` rows, one would do

```
int (*multi_array)[3] = malloc(rows * sizeof(int[3]));
```

(Note that here `multi_array` is a pointer to an array of 3 ints.)
Because of array-pointer interchangeability, you can index this just like static multidimensional arrays, i.e. `multi_array[5][2]` is the element at the 6th row and 3rd column.

- They can be allocated by first allocating an array of pointers, and then allocating subarrays and storing their addresses in the array of pointers (this approach is also known as an Iliffe vector). The syntax for accessing elements is the same as for multidimensional arrays described above (even though they are stored very differently). This approach has the advantage of the ability to make ragged arrays (i.e. with subarrays of different sizes). However, it also uses more space and requires more levels of indirection to index into, and can have worse cache performance. It also requires many dynamic allocations, each of which can be expensive.

For more information, see the comp.lang.c FAQ, question 6.16.

In some cases, the use of multi-dimensional arrays can best be addressed as an array of structures. Before user-defined data structures were available, a common technique was to define a multi-dimensional array, where each column contained different information about the row. This approach is also frequently used by beginner programmers. For example, columns of a two-dimensional character array might contain last name, first name, address, etc.

In cases like this, it is better to define a structure that contains the information that was stored in the columns, and then create an array of pointers to that structure. This is especially true when the number of data points for a given record might vary, such as the tracks on an album. In these cases, it is better to create a structure for the album that contains information about the album, along with a dynamic array for the list of songs on the album. Then an array of pointers to the album structure can be used to store the collection.

Constructors and destructors

In most object-oriented languages, objects cannot be created directly by a client that wishes to use them. Instead, the client must ask the class to build an instance of the object using a special routine called a constructor. Constructors are important because they allow an object to enforce invariants about its internal state throughout its lifetime. Destructors, called at the end of an object's lifetime, are important in systems where an object holds exclusive access to some resource, and it is desirable to ensure that it releases these resources for use by other objects.

Since C is not an object-oriented language, it has no built-in support for constructors or destructors. It is not uncommon for clients to explicitly allocate and initialize records and other objects. However, this leads to a potential for errors, since operations on the object may fail or behave unpredictably if the object is not properly initialized. A better approach is to have a function that creates an instance of the object, possibly taking initialization parameters, as in this example:

```
struct string {
    size_t size;
    char *data;
};
struct string *create_string(const char *initial) {
    assert (initial != NULL);
    struct string *new_string = malloc(sizeof(*new_string));
    if (new_string != NULL) {
        new_string->size = strlen(initial);
        new_string->data = strdup(initial);
    }
    return new_string;
}
```

Similarly, if it is left to the client to destroy objects correctly, they may fail to do so, causing resource leaks. It is better to have an explicit destructor which is always used, such as this one:

```
void free_string(struct string *s) {
    assert (s != NULL);
    free(s->data);  /* free memory held by the structure */
    free(s);        /* free the structure itself */
}
```

It is often useful to combine destructors with *#Nulling freed pointers*.

Sometimes it is useful to hide the definition of the object to ensure that the client does not allocate it manually. To do this, the structure is defined in the source file (or a private header file not available to users) instead of the header file, and a forward declaration is put in the header file:

```
struct string;
struct string *create_string(const char *initial);
void free_string(struct string *s);
```

Nulling freed pointers

As discussed earlier, after `free()` has been called on a pointer, it becomes a dangling pointer. Worse still, most modern platforms cannot detect when such a pointer is used before being reassigned.

One simple solution to this is to ensure that any pointer is set to a null pointer immediately after being freed:

```
free(p);
p = NULL;
```

Unlike dangling pointers, a hardware exception will arise on many modern architectures when a null pointer is dereferenced. Also, programs can include error checks for the null value, but not for a dangling pointer value. To ensure it is done at all locations, a macro can be used:

```
#define FREE(p)   do { free(p); (p) = NULL; } while(0)
```

(To see why the macro is written this way, see *#Macro conventions*.) Also, when this technique is used, destructors should zero out the pointer that they are passed, and their argument must be passed by reference to allow this. For example, here's the destructor from *#Constructors and destructors* updated:

```
void free_string(struct string **s) {
    assert(s != NULL  &&  *s != NULL);
    FREE((*s)->data);  /* free memory held by the structure */
    FREE(*s);          /* free the structure itself */
}
```

Unfortunately, this idiom will not do anything to any other pointers that may be pointing to the freed memory. For this reason, some C experts regard this idiom as dangerous due to creating a false sense of security.

Macro conventions

Because preprocessor macros in C work using simple token replacement, they are prone to a number of confusing errors, some of which can be avoided by following a simple set of conventions:

1. Placing parentheses around macro arguments wherever possible. This ensures that, if they are expressions, the order of operations does not affect the behavior of the expression. For example:
   - Wrong: `#define square(x) x*x`
   - Better: `#define square(x) (x)*(x)`
2. Placing parentheses around the entire expression if it is a single expression. Again, this avoids changes in meaning due to the order of operations.
   - Wrong: `#define square(x) (x)*(x)`
   - Better: `#define square(x) ((x)*(x))`
3. If a macro produces multiple statements, or declares variables, it can be wrapped in a **do** { ... } **while**(0) loop, with no terminating semicolon. This allows the macro to be used like a single statement in any location, such as the body of an if statement, while still allowing a semicolon to be placed after the macro invocation without creating a null statement. Care must be taken that any new variables do not potentially mask portions of the macro's arguments.
   - Wrong: `#define FREE(p) free(p); p = NULL;`
   - Better: `#define FREE(p) do { free(p); p = NULL; } while(0)`
4. Avoiding using a macro argument twice or more inside a macro, if possible; this causes problems with macro arguments that contain side effects, such as assignments.
5. If a macro may be replaced by a function in the future, considering naming it like a function.

C and beyond

← Common practices | Mixing languages →

Language Extensions

Most C compilers have one or more "extensions" to the standard C language, to do things that are inconvenient to do in standard, portable C.

Some examples of language extensions:

- in-line assembly language
- interrupt service routines
- variable-length data structure (a structure whose last item is a "zero-length array").
- re-sizeable multidimensional arrays
- various "#pragma" settings to compile quickly, to generate fast code, or to generate compact code.
- bit manipulation, especially bit-rotations and things involving the "carry" bit
- storage alignment

- Arrays whose length is computed at run time.

External links

- C: Extensions to the C Language
- SDCC: Storage Class Language Extensions

← Language extensions | Code library →

- **auto**
- **break**
- **case**
- **char**
- **const**
- **continue**
- **default**
- **do**
- **double**
- **else**
- **enum**
- **extern**
- **float**
- **for**
- **goto**
- **if**
- **int**
- **long**
- **register**
- **return**
- **short**
- **signed**
- **sizeof**
- **static**
- **struct**
- **switch**
- **typedef**
- **union**
- **unsigned**
- **void**
- **volatile**
- **while**

Keywords added to ISO C (C99) (Supported only in new compilers):

- **_Bool**
- **_Complex**
- **_Imaginary**
- **inline**
- **restrict**

Specific compilers may (in a non-standard-compliant mode) also treat some other words as keywords, including **asm**, **cdecl**, **far**, **fortran**, **huge**, **interrupt**, **near**, **pascal**, **typeof**.

Very old compilers may not recognize some or all of the C89 keywords **const**, **enum**, **signed**, **void**, **volatile** as well as the C99 keywords.

See also the list of reserved identifiers.

List of Standard Headers

ANSI C (C89)/ISO C (C90) headers:

- assert.h
- ctype.h
- errno.h
- float.h
- limits.h
- locale.h
- math.h
- setjmp.h
- signal.h
- stdarg.h
- stddef.h
- stdio.h
- stdlib.h
- string.h
- time.h

Headers added to ISO C (C94/C95) in Amendment 1 (AMD1):

- iso646.h
- wchar.h
- wctype.h

Headers added to ISO C (C99) (Supported only in new compilers):

- complex.h
- fenv.h
- inttypes.h
- stdbool.h
- stdint.h
- tgmath.h

Very old compilers may not include some or all of the C89 headers iso646.h, locale.h, wchar.h, wctype.h, nor the C99 headers.

Table of Operators

Operators in the same group have the same **precedence** and the order of evaluation is decided by the **associativity** (*left-to-right* or *right-to-left*). Operators in a preceding group have *higher* precedence than those in a subsequent group.

Operators	Description	Example Usage	Associativity
	Postfix operators		
`()`	function call operator	`swap (x, y)`	Left to right
`[]`	array index operator	`arr [i]`	Left to right
`.`	member access operator *for an object of class/union type or a reference to it*	`obj.member`	Left to right
`->`	member access operator *for a pointer to an object of class/union type*	`ptr->member`	Left to right
	Unary Operators		
`!`	logical not operator	`!eof_reached`	Right to left
`~`	bitwise not operator	`~mask`	Right to left
`+ -`[1]	unary plus/minus operators	`-num`	Right to left
`++ --`	post-increment/decrement operators	`num++`	Right to left
`++ --`	pre-increment/decrement operators	`++num`	Right to left
`&`	address-of operator	`&data`	Right to left
`*`	indirection operator	`*ptr`	Right to left
`sizeof`	sizeof operator *for expressions*	**`sizeof`** `123`	Right to left
`sizeof``()`	sizeof operator *for types*	**`sizeof`** `(`**`int`**`)`	Right to left
`(`*`type`*`)`	cast operator	`(`**`float`**`)i`	Right to left
	Multiplicative Operators		
`* / %`	multiplication, division and modulus operators	`celsius_diff * 9 / 5`	Left to right

Additive Operators			
`+ -`	addition and subtraction operators	`end - start + 1`	Left to right
Bitwise Shift Operators			
`<<`	left shift operator	`bits << shift_len`	Left to right
`>>`	right shift operator	`bits >> shift_len`	
Relational Inequality Operators			
`< > <= >=`	less-than, greater-than, less-than or equal-to, greater-than or equal-to operators	`i < num_elements`	Left to right
Relational Equality Operators			
`== !=`	equal-to, not-equal-to	`choice != 'n'`	Left to right
Bitwise And Operator			
`&`		`bits & clear_mask_complement`	Left to right
Bitwise Xor Operator			
`^`		`bits ^ invert_mask`	Left to right
Bitwise Or Operator			
`|`		`bits | set_mask`	Left to right
Logical And Operator			
`&&`		`arr != 0 && arr->len != 0`	Left to right
Logical Or Operator			
`||`		`arr == 0 || arr->len == 0`	Left to right
Conditional Operator			
`?:`		`size != 0 ? size : 0`	Right to left

Assignment Operators			
`=`	assignment operator	`i = 0`	Right to left
`+= -= *= /=` `%= &= |= ^=` `<<= >>=`	shorthand assignment operators (*`foo`* `op=` *`bar`* *represents* *`foo`* `=` *`foo`* `op` *`bar`*)	`num /= 10`	
Comma Operator			
`,`		`i = 0, j = i + 1, k = 0`	Left to right

Table of Operators Footnotes

[1]Very old compilers may not recognize the unary + operator.

Table of Data Types

Type	Size in Bits	Comments	Alternate Names
Primitive Types in ANSI C (C89)/ISO C (C90)			
`char`	≥ 8	▪ **`sizeof`** gives the size in units of **`char`**s. These "C bytes" need not be 8-bit bytes (though commonly they are); the number of bits is given by the `CHAR_BIT` macro in the `limits.h` header. ▪ Signedness is implementation-defined. ▪ Any encoding of 8 bits or less (e.g. ASCII) can be used to store characters. ▪ Integer operations can be performed portably only for the range 0 ~ 127. ▪ All bits contribute to the	—

		value of the **char**, i.e. there are no "holes" or "padding" bits.	
`signed char`	same as `char`	▪ Characters stored like for type `char`. ▪ Can store integers in the range -127 ~ 127 portably[1].	—
`unsigned char`	same as `char`	▪ Characters stored like for type `char`. ▪ Can store integers in the range 0 ~ 255 portably.	—
`short`	≥ 16, ≥ size of `char`	▪ Can store integers in the range -32767 ~ 32767 portably[2]. ▪ Used to reduce memory usage (although the resulting executable may be larger and probably slower as compared to using `int`.	`short int`, `signed short`, `signed short int`
`unsigned short`	same as `short`	▪ Can store integers in the range 0 ~ 65535 portably. ▪ Used to reduce memory usage (although the resulting executable may be larger and probably slower as compared to using `int`.	`unsigned short int`
`int`	≥ 16, ≥ size of `short`	▪ Represents the "normal" size of data the processor deals with (the word-size); this is	`signed`, `signed int`

		the integral data-type used normally. ▪ Can store integers in the range -32767 ~ 32767 portably[2].	
`unsigned int`	same as **`int`**	▪ Can store integers in the range 0 ~ 65535 portably.	**`unsigned`**
`long`	≥ 32, ≥ size of **`int`**	▪ Can store integers in the range -2147483647 ~ 2147483647 portably[3].	**`long int`**, **`signed long`**, **`signed long int`**
`unsigned long`	same as **`long`**	▪ Can store integers in the range 0 ~ 4294967295 portably.	**`unsigned long int`**
`float`	≥ size of **`char`**	▪ Used to reduce memory usage when the values used do not vary widely. ▪ The floating-point format used is implementation defined and need not be the IEEE single-precision format. ▪ **`unsigned`** cannot be specified.	—
`double`	≥ size of **`float`**	▪ Represents the "normal" size of data the processor deals with; this is the floating-point data-type used normally. ▪ The floating-point format used is implementation defined and need not be the	—

		IEEE double-precision format. ▪ **`unsigned`** cannot be specified.	
`long double`	≥ size of **`double`**	▪ **`unsigned`** cannot be specified.	—

Primitive Types added to ISO C (C99)

`long long`	≥ 64, ≥ size of **`long`**	▪ Can store integers in the range -9223372036854775807 ~ 9223372036854775807 portably[4].	**`long long int`**, **`signed long long`**, **`signed long long int`**
`unsigned long long`	same as **`long long`**	▪ Can store integers in the range 0 ~ 18446744073709551615 portably.	**`unsigned long long int`**

User Defined Types

`struct`	≥ sum of size of each member	▪ Said to be an *aggregate type*.	—
`union`	≥ size of the largest member	▪ Said to be an *aggregate type*.	—
`enum`	≥ size of	▪ Enumerations are a	—

	`char`	separate type from **`int`**s, though they are mutually convertible.	
`typedef`	same as the type being given a name	▪ **`typedef`** has syntax similar to a storage class like **`static`**, **`register`** or **`extern`**.	—
	Derived Types[5]		
`type*` (pointer)	≥ size of **`char`**	▪ `0` always represents the null pointer (an address where no data can be placed), irrespective of what bit sequence represents the value of a null pointer. ▪ Pointers to different types may have different representations, which means they could also be of different sizes. So they are not convertible to one another. ▪ Even in an implementation which guarantess all data pointers to be of the same size, function pointers and data pointers are in general incompatible with each other. ▪ For functions taking variable number of arguments, the arguments passed must be of appropriate type, so even `0` must be cast	—

		to the appropriate type in such function-calls.	
`type [integer[6]]` (array)	≥ *integer* × size of `type`	▪ The brackets (`[]`) *follow* the identifier name in a declaration. ▪ In a declaration which also initializes the array (including a function parameter declaration), the size of the array (the *integer*) can be omitted. ▪ `type []` is not the same as `type*`. Only under some circumstances one can be converted to the other.	—
`type (comma-delimited list of types/declarations)` (function)	—	▪ Functions declared without any storage class are **`extern`**. ▪ The parentheses (`()`) *follow* the identifier name in a declaration, e.g. a 2-arg function pointer: `int (* fptr) (int arg1, int arg2)`.	—

Table of Data Types Footnotes

[1] -128 can be stored in two's-complement machines (i.e. most machines in existence). Very old compilers may not recognize the **`signed`** keyword.

[2] -32768 can be stored in two's-complement machines (i.e. most machines in existence). Very old compilers may not recognize the **`signed`** keyword.

[3] -2147483648 can be stored in two's-complement machines (i.e. most machines in existence). Very old compilers may not recognize the **`signed`** keyword.

[4] -9223372036854775808 can be stored in two's-complement machines (i.e. most machines in existence).

[5] The precedences in a declaration are:

`[]`, `()` (left associative) — Highest

`*` (right associative) — Lowest

[6] The standards do NOT place any restriction on the size/type of the integer, it's implementation dependent. The only mention in the standards is a reference that an implementation may have limits to the maximum size of memory block which can be allocated, and as such the limit on integer will be size_of_max_block/sizeof(type).