C Programming/Print version

C Programming

The current, editable version of this book is available in Wikibooks, the open-content textbooks collection, at http://en.wikibooks.org/wiki/C_Programming

Permission is granted to copy, distribute, and/or modify this document under the terms of the Creative Commons Attribution-ShareAlike 3.0 License.

Contents

Why learn C?

C is the most commonly used programming language for writing operating systems. The first operating system written in C is Unix. Later operating systems like GNU/Linux were all written in C. Not only is C the language of operating systems, it is the precursor and inspiration for almost all of the most popular high-level languages available today. In fact, Perl, PHP, Python and Ruby are all written in C.

By way of analogy, let's say that you were going to be learning Spanish, Italian, French, or Portuguese. Do you think knowing Latin would be helpful? Just as Latin was the basis of all of those languages, knowing C will enable you to understand and appreciate an entire family of programming languages built upon the traditions of C. Knowledge of C enables freedom.

Why C, and not assembly language?

While assembly language can provide speed and maximum control of the program, C provides portability.

Different processors are programmed using different Assembly languages and having to choose and learn only one of them is too arbitrary. In fact, one of the main strengths of C is that it combines universality and portability across various computer architectures while retaining most of the control of the hardware provided by assembly language.

For example, C programs can be compiled and run on the HP 50g calculator (ARM processor), the TI-89 calculator (68000 processor), Palm OS Cobalt smartphones (ARM processor), the original iMac (PowerPC), the Arduino (Atmel AVR), and the Intel iMac (Intel Core 2 Duo). Each of these devices has its own assembly language that is completely incompatible with the assembly language of any other.

Assembly, while extremely powerful, is simply too difficult to program large applications and hard to read or interpret in a logical way. C is a compiled language, which creates fast and efficient executable files. It is also a small "what you see is all you get" language: a C statement corresponds to at most a handful of assembly statements, everything else is provided by library functions.

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 the higher-level library uses. And the pattern continues on and on and on...

Why C, and not another language?

The primary design of C is to produce portable code while maintaining performance and minimizing footprint (CPU time, memory, disk I/O, etc.). This is useful for operating systems, embedded systems or other programs where performance matters a lot ("high-level" interface would affect performance). With C it's relatively easy to keep a mental picture of what a given line really does, because most of the things are written explicitly in the code. C has a big codebase for low level applications. It is the "native" language of UNIX, which makes it flexible and portable. It is a stable and mature language which is unlikely to disappear for a long time and has been ported to most, if not all, platforms.

One powerful reason is memory allocation. Unlike most programming languages, C allows the programmer to write directly to memory. Key constructs in C such as structs, pointers and arrays are designed to structure and manipulate memory in an efficient, machine-independent fashion. In particular, C gives control over the memory layout of data structures. Moreover dynamic memory allocation is under the control of the programmer (which also means that memory deallocation has to be done by the programmer). Languages like Java and Perl shield the programmer from having to worry about memory allocation and pointers. This can be useful since dealing with memory allocation when building a high-level program is a highly error-prone process. However, when dealing with low-level code such as the part of the OS that controls a device, C provides a uniform, clean interface. These capabilities just do not exist in most other languages.

While Perl, PHP, Python and Ruby may be powerful and support many features not provided by default in C, they are not normally implemented in their own language. Rather, most such languages initially relied on being written in C (or another high-performance programming language), and would require their implementation be ported to a new platform before they can be used.

As with all programming languages, whether you want to choose C over another high-level language is a matter of opinion and both technical and business requirements could dictate which language is required.

History

The field of computing as we know it today started in 1947 with three scientists at Bell Telephone Laboratories—William Shockley, Walter Brattain, and John Bardeen—and their groundbreaking invention: the transistor. In 1956, the first fully transistor-based computer, the TX-0, was completed at MIT. The first integrated circuit was created in 1958 by Jack Kilby at Texas Instruments, but the first high-level programming language existed even before then.

"The Fortran project" was originally developed in 1954 by IBM. A shortening of "*The IBM Mathematical **For**mula **Tran**slating System*", the project had the purpose of creating and fostering development of a procedural, imperative programming language that was especially suited to numeric computation and scientific computing. It was a breakthrough in terms of productivity and programming ease (compared to assembly language) and speed (Fortran programs ran nearly as fast as, and in some cases, just as fast as, programs written in assembly). Furthermore, Fortran was written at a high-enough level (and thus was machine independent enough) to become the first widely adopted programming language. The Algorithmic Language (Algol 58) was derived from Fortran in 1958 and evolved into Algol 60 in 1960. The Combined Programming Language (CPL) was then created out of Algol 60 in 1963. In 1967, it evolved into Basic CPL, which was itself, the base for B in 1969. Finally, B, the root of C, was created in 1971.

C was the direct successor of B, a stripped down version of BCPL, created by Ken Thompson at Bell Labs, that was also a compiled language - User's Reference to B (https://www.bell-labs.com/usr/dmr/www/kbman.pdf), used in early internal versions of the UNIX operating system. As noted in Ritchie's C History (https://www.bell-labs.com/usr/dmr/www/chist.html) : "The B compiler on the PDP-7 did not generate machine instructions, but instead 'threaded code', an interpretive scheme in which the compiler's output consists of a sequence of addresses of code fragments that perform the elementary operations. The operations typically — in particular for B — act on a simple stack machine". Thompson and Dennis Ritchie, also working at Bell Labs, improved B and called the result NB. Further extensions to NB created its logical successor, C. Most of UNIX was rewritten in NB, and then C, which resulted in a more portable operating system.

The portability of UNIX was the main reason for the initial popularity of both UNIX and C. 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 then write a C compiler for the new system. Since most of the system utilities were thus written in C, it simply made sense to also write new utilities in C.

The American National Standards Institute began work on standardizing the C language in 1983, and completed the standard in 1989. The standard, ANSI X3.159-1989 "Programming Language C", served as the basis for all implementations of C compilers. The standards were later updated in 1990 and 1999, allowing for features that were either in common use, or were appearing in C++.

What you need before you can learn

Getting Started

The goal of this book is to introduce you to the C programming language. Basic computer literacy is assumed, but no special knowledge is needed.

Before you can start programming in C, you will need a **C compiler**. A compiler is a program that converts C code into executable machine code.[1]

Popular C compilers Include:

Name	Website	Platform	License	Details
Microsoft Visual Studio Express	Visual Studio (http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express)	Windows	Free Version	Powerful and student-friendly version of an industry standard compiler.
Tiny C Compiler (TCC)	tinycc (http://www.tinycc.org)	GNU/Linux, Windows	LGPL	Small, fast and simple compiler.
Clang	clang (http://clang.llvm.org)	GNU/Linux, Windows, Unix, OS X	University of Illinois/NCSA License (http://opensource.org/licenses/UoI-NCSA.php)	A front-end which compiles (Objective) C/C++ using a LLVM backend.
GNU C Compiler	gcc (http://gcc.gnu.org)	GNU/Linux, MinGW(Windows) (http://mingw.org), Unix, OS X.	GPL	The De facto standard. Ships with most Unix systems.

The minimum software requirements to program in C is a text editor, as opposed to a word processor. A plain text Notepad Editor can be used but it does not offer any advanced capabilities such as code completion or debugging. There are many text editors (see List of Text Editors), among the most popular are Notepad++ for Windows, Atom (https://atom.io/), Sublime Text, Vim and Emacs are also available cross-platform. These text editors come with syntax highlighting and line numbers, which makes code easier to read at a glance, and to spot syntax errors.

Though not absolutely needed, many programmers prefer and recommend using an Integrated development environment (**IDE**) instead of a text editor. An IDE is a suite of programs that developers need, combined into one convenient package, usually with a graphical user interface. These programs include a text editor, linker, project management and sometimes bundled with a compiler. 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 through it manually, or alter data as an aid to finding and

correcting programming errors.

For beginners it is recommended not to use an IDE, since it hides most of what is going on. Using the command line builds up familiarity with the toolchain. An IDE may be useful to somebody with programming experience but knows how the IDE works. So as a general guideline: Do not use an IDE unless you know what the IDE does!

Popular IDEs Include:

Name	Website	Platform	License	Details
Eclipse CDT	Eclipse (http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/junor)	Windows, Mac OS X, Linux	Open source	Eclipse IDE for C/C++ developement, a popular open source IDE.
Netbeans	Netbeans (http://netbeans.org)	Cross-platform	CDDL and GPL 2.0	A Good comparable matured IDE to Eclipse.
Anjuta	Anjuta (http://anjuta.org)	Linux	GPL	A GTK+2 IDE for the GNOME desktop environment.
Geany	geany (http://www.geany.org)	Cross-platform	GPL	A lightweight cross-platform GTK+ notepad based on Scintilla, with basic IDE features.
Little C Compiler (LCC)	lcc (http://www.cs.virginia.edu/~lcc-win32)	Windows	Free for non-commercial use	Small open source compiler.
Xcode	Xcode (https://developer.apple.com/xcode)	Mac OS X	Free	Available for free at Mac App Store (https://itunes.apple.com/us/app/xcode/id497799835?ls=1&mt=12).
Pelles C	Pelles C (http://smorgasbordet.com/pellesc)	Windows, Pocket PC	Free	A complete C development kit for Windows.
Dev C++	Dev C++ (http://sourceforge.net/projects/orwelldevcpp/)	Windows	GPL	Updated version of the formerly popular Bloodshed Dev-C++.
Microsoft Visual Studio Express	Visual C++ (http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express)	Windows	Free	A powerful, user friendly version of an industry standard compiler.
CodeLite	CodeLite (http://codelite.org/)	Cross-platform	GPL 2	Free IDE for C/C++ development.
Code::Blocks	Code::Blocks (http://codeblocks.org/)	Cross-platform	GPL 3.0	Built to meet users' most demanding needs. Very extensible and fully configurable.

On **GNU/Linux**, GCC is almost always included automatically.

On **Microsoft Windows**, Dev-C++ is recommended for beginners because it is easy to use, free, and simple to install. Although the initial developer (Bloodshed) hasn't updated it since 2005, a new version appeared in 2011, made by an independent programmer, and is being actively developed.[2] An alternate option for those working only in the Windows environment is the proprietary Microsoft Visual Studio Express which is free of charge and has an excellent debugger.

On **Mac OS X**, the Xcode IDE provides the compilers needed to compile various source files. The newer versions do not include the command line tools. They need to be downloaded via Xcode->Preferences->Downloads.

Footnotes

1. Actually, GCC's(GNU 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), which is not to be confused with **c++** the compiler.
2. http://orwelldevcpp.blogspot.com/

Using a Compiler

Dev-C++

Dev C++ is an Integrated Development Environment(IDE) for the C++ programming language, available from Bloodshed Software (http://www.bloodshed.net/). An updated version is available at Orwell Dev-C++ (http://orwelldevcpp.blogspot.com/).
C++ is a programming language which contains within itself most of the C language, plus extensions. Most C++ compilers will compile C programs, sometimes with a few adjustments (like invoking them with a different name or command line switch). Therefore, you can use Dev C++ for C development.

However, Dev C++ is not the compiler. It is designed to use the MinGW or Cygwin versions of GCC - both of which can be obtained 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 latest version of Dev-C++ is a beta for version 5. However, it still has a significant number of bugs. All the features are there, and it is quite usable. It is 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. (e.g. KDevelop and Anjuta.) Most of the graphical text editors, and other common editors such as *emacs* and *vim*, support syntax highlighting.

GCC

The GNU Compiler Collection (GCC) is a free set of compilers developed by the Free Software Foundation.

Steps for Obtaining the GCC Compiler if You're on GNU/Linux

On **GNU/Linux,** Installing the GNU C Compiler can vary in method from distribution to distribution. (Type in **cc -v** to see if it is installed already.)

- 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 (along with other necessary tools) by using `sudo apt-get install build-essential`, or by using Synaptic. You do not need Universe enabled.
- For Slackware, the package is available on their website (http://www.slackware.com/pb/) - simply download, and type `installpkg gcc-xxxxx.tgz`
- For Gentoo, you should already have GCC 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 -S gcc`.
- If you cannot become root, get the GCC tarball from ftp://ftp.gnu.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.

Steps for Obtaining the GCC Compiler if You're on BSD Family Systems

- For Mac OS X, FreeBSD, NetBSD, OpenBSD, DragonFly BSD, Darwin the port of GNU gcc is available in the base system, or it could be obtained using the ports collection or pkgsrc.

Steps for Obtaining the GCC Compiler if You're on Windows

There are two ways to use GCC on Windows: Cygwin and MinGW. Applications compiled with Cygwin will not run on any computer without Cygwin, so MinGW is recommended. MinGW is simpler to install, and takes less disk space.

To get MinGW, do this:

1. Go to http://sourceforge.net/projects/mingw/ download and save this to your hard drive.
2. Once the download is finished, open it and follow the instructions. You can also choose to install additional compilers, or the tool Make, but these aren't necessary.
3. Now you need to set your PATH. Right-click on "My computer" and click "Properties". Go to the "Advanced" tab and click on "Environment variables". Go to the "System variables" section and scroll down until you see "Path". Click on it, then click "edit". Add ";C:\mingw\bin\" (without the quotes) to the end.
4. To test if GCC works, open a command prompt and type "gcc". You should get the message "gcc: fatal error: no input files compilation terminated.". If you get this message, GCC is installed correctly.

To get Cygwin, do this:

1. Go to http://www.cygwin.com and click on the "Install Cygwin Now" button in the upper right corner of the page.
2. Click "run" in the window that pops up, and click "next" several times, accepting all the default settings.
3. Choose any of the Download sites ("ftp.easynet.be", etc.) when that window comes up; press "next" and the Cygwin installer should start downloading.
4. When the "Select Packages" window appears, scroll down to the heading "Devel" and click on the "+" by it. In the list of packages that now displays, scroll down and find the "gcc-core" package; this is the compiler. Click once on the word "Skip", and it should change to some number like "3.4" etc. (the version number), and an "X" will appear next to "gcc-core" and several other related packages that will now be downloaded.
5. Click "next" and the compiler as well as the Cygwin tools should start downloading; this could take a while. While you're waiting for the installation to finish, download any text-editor designed for programming. While Cygwin does include some, you may prefer doing a web search to find other alternatives. While using a stock text editor is possible, it is not ideal.
6. Once the Cygwin downloads are finished and you have clicked "next", etc. to finish the installation, double-click the Cygwin icon on your desktop to begin the Cygwin "command prompt". Your home directory will automatically be set up in the Cygwin folder, which now should be at "C:\cygwin" (the Cygwin folder is in some ways like a small unix/linux computer on your Windows machine -- not technically of course, but it may be helpful to think of it that way).
7. Type "gcc" at the Cygwin prompt and press "enter"; if "gcc: no input files" or something like it appears you have succeeded and now have the gcc compiler on your computer (and congratulations -- you have also just received your first error message!).

The current stable (usable) version of GCC is 4.9.1 published on 2014-07-16, 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.

Embedded systems

- Most CPUs are microcontrollers in embedded systems, often programmed in C, but most of the compilers mentioned above (except GCC) do not support such CPUs. For specialized compilers that do support embedded systems, see Embedded Systems/C Programming.

Other C compilers

We have a long list of C compilers in a much later section of this Wikibook. *Which of those compilers would be suitable for beginning C programmers, that we should say a few words about getting started with that particular compiler in this section of this Wikibook?*

A taste of C

As with nearly every other programming language learning book, we use the *Hello world* program to introduce you to C.

```c
#include <stdio.h>

int main(void)
{
   puts("Hello, world!");
   return 0;
}
```

This program prints "Hello, world!" and then exits.

And if you want to hold the output and it does not exit, you may use the `getchar();` as following:

```c
#include <stdio.h>

int main(void)
{
   puts("Hello, world!");
   getchar();
   return 0;
}
```

Enter this code into your text editor or IDE, and save it as "hello.c".

Then, presuming you are using GCC, type `gcc -o hello hello.c`. This tells gcc to compile your hello.c program into a form the machine can execute. The '-o hello' tells it to name the compiled program 'hello'.

If you have entered this correctly, you should now see a file called hello. This file is the binary version of your program, and when run should display "Hello, world!"

Here is an example of how compiling and running looks when using a terminal on a unix system. `ls` is a common unix command that will list the files in the current directory, which in this case is the directory `progs` inside the home directory (represented with the special tilde, ~, symbol). After running the `gcc` command, `ls` will list a new file, `hello` in green. Green is the standard color coding of `ls` for executable files.

```
~/progs$ ls
hello.c
~/progs$ gcc -o hello hello.c
~/progs$ ls
hello  hello.c
~/progs$ ./hello
Hello, world!
~/progs$
```

Part-by-part explanation

`#include <stdio.h>` tells the C compiler to find the standard header called <*stdio.h*> and add it 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 which you can use to send messages to a user, or to read input from a user.

`int main(void)` 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 the operating system when it is finished.

`puts("Hello, world!");` is the statement that actually puts the message to the screen. *puts* is a string printing function that is declared in the file *stdio.h* (which is why you had to *#include* that at the start of the program) `puts` automatically prints a newline at the end of the string.

`return 0;` 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.

The empty line is there because it is (at least on UNIX) considered good practice to end a file with a new line. In gcc using the `-Wall -pedantic -ansi` options, if the file does not end with a new line this message is displayed: "warning: no newline at end of file". (The newline isn't shown on the example because MediaWiki automatically removes it)

Intro exercise

Introductory Exercises

On GCC

If you are using a Unix(-like) system, such as GNU/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, If you are using Cygwin.

```
a.exe
```

You should now see your very first C program.

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. The following shows a few examples:

`-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.

All the options are well documented in the manual page (http://gcc.gnu.org/onlinedocs/gcc-4.3.0/gcc/Debugging-Options.html) for GCC.

The Classical "Hello, World!" Program

Tradition dictates that we begin with a very simple program, which simply displays the characters "Hello, World!" on the screen and immediately exits.

```
#include <stdio.h>

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

The task itself sounds like it should be simple enough, and the truth of the matter is that most of the above code is what some refer to as boilerplate code. The line that's doing the work we're interested in is `printf("Hello World!\n");`.

`#include <stdio.h>` is an example of a macro. Loosely speaking, **macros** allow us to tell a part of the compiler - the **preprocessor** - to modify the code we've written before it is compiled. In this case, the `include` macro is retrieving C code that has already been written from the file stdio.h. Files used in this way are called **header files** and are saved with the **.h** extension. In this program, the only thing we needed from `stdio.h` was the `printf` function.

Both `main` and `printf` are examples of C's functions. Although they serve similar purposes, functions are quite different from macros. We'll be committing a great deal of discussion to each later. In computer science, the term **function** tends to be used a bit more loosely than in mathematics. For now, it suffices to say that functions let us define a complex process that we want to reference frequently.

Finally, we consider the last line, `return 0;`. When the operating system executes our program, it's useful to be able to let the OS know whether or not the program succeeded. We do this with an **exit status**, which we send to the operating system by C's `return` statement. In this case, we provide an exit status of "0" to indicate that execution succeeded without error. As our programs grow in complexity, we can use other integers as codes to indicate other types of errors. This style of providing exit statuses is a long standing convention (http://www.gnu.org/software/libc/manual/html_node/Exit-Status.html).

You probably have a few more questions, like *What's up with this* `int argc, char** argv` *business? That's weird.* or *Why is it* `int main` *instead of*

just `main`*?* Don't worry! We'll get to that in short order.

On IDEs

If you are using a proprietary 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.

One can also find opensource IDE's like Eclipse, Netbeans or Qt Creator. The process will be the same as in a proprietary IDE.

Beginning C

Preliminaries

Basic Concepts

Before one gets too deep 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.

Block Structure, Statements, Whitespace, and Scope

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.

So what is in a **block**? Generally, a block consists of executable **statements**.

Before blocks are explained, what is a statement? One way to put it is that statements are the text the compiler will attempt to turn into executable instructions, and the whitespace that surrounds them. An easier way to put it is that statements are bits of code that do things, like this:

```
int i = 6; /* this declares a variable 'i', and sets it to equal 6 */
```

You might have noticed the semicolon at the end of the statement. Statements in C always end with a semicolon (;) character. Leaving off the semicolon is a common mistake that a lot of people make, beginners and experts alike! So until it becomes second nature, be sure to double check

your statements!

Since C is a "free-format" language, several statements can share a single line in the source file, like so:

```
/* this declares the variables 'i', 'test', 'foo', and 'bar'
   note that ONLY 'bar' is set to six! */
int i, test, foo, bar = 6;
```

There are several kinds of statements, and you've seen some of them. Assignment (i = 6;), conditional and flow-control. A substantial portion of this book deals with statement construction.

Now back to blocks. In C, blocks begin with an opening brace "{" and end with a closing brace "}". Blocks can contain other blocks which can contain their own blocks, and so on.

Let's show an example of blocks.

```
int main(void)
{
    /* this is a 'block' */
    int i = 5;

    {
        /* this is also a 'block,' separate from the last one */
        int i = 6;
    }

    return 0;
}
```

Blocks come in handy with readability and scope. You'll learn a little more about scope in a second.

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

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

is the same as

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

which is the same as

```
 printf (
 "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. You can use blocks without a conditional, loop, or other statement to organize your code.

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 is visible. 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.

Let's show some examples, to give a better picture of the idea of scope.

```
int i = 5; /* this is a 'global' variable, it can be accessed from anywhere in the program */

/* this is a function, all variables inside of it
    are "local" to the function. */
int main(void)
{
    int i = 6; /* 'i' now equals 6 */
    printf("%d\n", i); /* prints a '6' to the screen, instead of the global variable of 'i', which is 5 */

    return 0;
}
```

That shows a decent example of local and global, but what about different scopes *inside* of functions? (you'll learn more about functions later, for now, just focus on the "main" part.)

```
/* the main function */
int main(void)
{
    /* this is the beginning of a 'block', you read about those above */

    int i = 6; /* this is the first variable of this 'block', 'i' */

    {
```

```
        /* this is a new 'block', and because it's a different block, it has its own scope */

        /* this is also a variable called 'i', but in a different 'block',
            because it's in a different 'block' than the old 'i', it doesn't affect the old one! */
        int i = 5;
        printf("%d\n", i); /* prints a '5' onto the screen */
    }
    /* now we're back into the old block */

    printf("%d\n", i); /* prints a '6' onto the screen */

    return 0;
}
```

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 return value in the above program is **0**).

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.

Usually, you're free to call a function whatever you wish to. The only restriction is that every executable program needs to have one, and only one, **main** function, which is where the program begins executing.

We will discuss functions in more detail in a later chapter, C Programming/Procedures and functions.

The Standard Library

In 1983, when C was in the process of becoming standardized, the American National Standards Institute (ANSI) formed a committee to establish a standard specification of C known as "ANSI C". That standard specification created a basic set of functions common to each implementation of C, which is referred to as the Standard Library. The Standard Library provides functions for tasks such as input/output, string manipulation, mathematics, files, and memory allocation. The Standard Library does not provide functions that are dependent on specific hardware or operating systems, like graphics, sound, or networking. In the "Hello, World", program, a Standard Library function is used (`printf`) which outputs lines of text

to the standard output stream.

Compiling

Having covered the basic concepts of C programming, we can now briefly discuss the process of *compilation*.

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 that *is* comprehensible to a microprocessor. The ***compiler*** is what takes this code, and translates it into the machine code.

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 references with `#include`, 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.

Preprocessor

The preprocessor provides the ability for the inclusion of header files, macro expansions, conditional compilation, and line control. Many times you will need to give special instructions to your compiler. This is done by inserting preprocessor directives into your code. When you begin compiling your code, a special program called the preprocessor scans the source code and performs simple substitution of tokenized strings for others according to predefined rules. The preprocessor is not a part of the C language.

In C language, all preprocessor directives begin with the hash character (#). You can see one preprocessor directive in the Hello world program introduced in A taste of C:

Example:

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

This directive causes the header to be included into your program. Other directives such as `#pragma` control compiler settings and macros. The result of the preprocessing stage is a text string. You can think of the preprocessor as a non-interactive text editor that prepares your code for the compilation step. The language of preprocessor directives is agnostic to the grammar of C, so the C preprocessor can also be used independently to process other kinds of text files.

Syntax Checking

This step ensures that the code is valid and will sequence into an executable program. Under most compilers, you may get messages or warnings indicating potential issues with your program (such as a statement always being true or false, etc.)

When an error is detected in the program, the compiler will normally report the file name and line that is preventing compilation.

Object Code

The compiler produces a machine code equivalent of the source code that can then be linked into the final program. The code itself can't be executed yet, as it has to complete the linking stage.

It's important to note after discussing the basics that compilation is a "one way street". That is, 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.

Linking

Linking combines the separate object codes into one complete program by integrating libraries and the code and producing either an executable program or a library. Linking is performed by a linker, which is often part of a compiler.

Common errors during this stage are either missing functions, or duplicate functions.

Automation

For large C projects, many programmers choose to automate compilation, both in order to reduce user interaction requirements and to speed up the process by only recompiling modified files.

Most integrated development environments have some kind of project management, which makes such automation very easy. On UNIX-like systems, make and Makefiles are often used to accomplish the same.

Once gcc is installed, it can be called with a list of c source files that have been written but not yet compiled. eg. there is a main.c file that includes some functions described in myfun.h and implemented in myfun_a.c and myfun_b.c , then it is enough to write

```
gcc   main.c myfun_a.c myfun_b.c
```

myfun.h is included in main.c , but if is in a separate header files directory , then that directory can be listed after a "-I " switch.

In larger programs, Makefiles and gnu make program can compile c files into intermediate files ending with suffix .o which can be linked by gcc.

How to compile each object file is usually described in the Makefile with the object file as a label ending with a colon followed by two spaces (tabs are often bad characters) followed by a list of other files that are dependencies, eg. .c files and .o files compiled in another section, and on the next line, the invocation of gcc that is required. typing `man make` or `info make` often gives the information needed to jog the memory on how to use make, and the same goes for gcc, although gcc has a lot of option switches, the main ones being -g to generate debugging for gdb to allow it to show source code during stepping through of the machine code program. gdb has a 'h' command that shows what it can do, and is usually started with 'gdb a.out' if a.out is the anonymous executable machine code file that was compiled by gcc.

Structure and style

C Structure and Style

This is a basic introduction to a good code style 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.

As a beginning programmer, the point of creating structure in the programs' code might not be clear, as the compiler doesn't care about the difference. However, as programs become complex, chances are that writing the program has become a joint effort. (Or others might want to see how it was accomplished.) Therefore, the code is no longer designed purely for a compiler to read.

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

Introduction

In C, programs are composed of statements. These statements are terminated with a semi-colon, and are collected in sections known as functions. By convention, a statement should be kept on its own line, as shown in the example below:

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

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

The following block of code is essentially the same: while it contains exactly the same code, and will compile and execute with the same result, the removal of spacing causes an essential difference, making it harder to read:

```c
#include <stdio.h>
int main(void) {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 white space can create a visual scale of how your code flows, which can be very important when returning to your code when you want to maintain it.

Line Breaks

With minimal line breaks, code is barely readable by humans, and may be hard to debug or understand:

```c
1 #include <stdio.h>
2 int main(void){ int i=0; printf("Hello, World!"); for (i=0; i<1; i++){ printf("\n"); break; } return 0; }
```

Rather than putting everything on one line, it is much more readable to break up long lines so that each statement and declaration goes on its own line. After inserting line breaks, the code will look like this:

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

```
2 int main(void)
3 {
4 int i=0;
5 printf("Hello, World!");
6 for (i=0; i<1; i++)
7 {
8 printf("\n");
9 break;
10 }
11 return 0;
12 }
```

Blank Lines

Blank lines should be used to offset the main components of your code. Use them

- After precompiler declarations.
- After new variables are declared.

Based on these two rules, there should now be two line breaks added.

- After line 1, because line 1 has a preprocessor directive
- After line 5, because line 5 contains a variable declaration

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

The following lines of code have line breaks between functions, but without indentation.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5 int i=0;
6
7 printf("Hello, World!");
8 for (i=0; i<1; i++)
9 {
10 printf("\n");
11 break;
12 }
13 return 0;
14 }
```

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 easier to read, it provides no information about the block structure of the program. Using the tab key can be very helpful now: indentation visually separates paths of execution by moving their starting points to a new column in the line. This simple practice will make it much easier to read and understand code. Indentation follows a fairly simple rule:

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

- All code inside a new block should be indented by one tab[1] more than the code in the previous path.

Based on the code from the previous section, there are two blocks requiring indentation:

- Lines 5 to 13
- Lines 10 and 11

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int i=0;
6
7     printf("Hello, World!");
8     for (i=0; i<1; i++)
9     {
10        printf("\n");
11        break;
12    }
13    return 0;
14 }
```

It is now fairly obvious as to which parts of the program fit inside which blocks. You can tell which parts of the program the coder has intended to loop, and which ones he or she has 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. This indentation makes the structure of your program clear.

Indentation was originally one tab character, or the equivalent of 8 spaces. Research since the original indent size has shown that indents between 2 to 4 characters are easier to read[2], resulting in such tab sizes being used as default in modern IDEs. However, an indent of 8 characters may still be in use for some systems[3].

Comments

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

what specific parts of your code do.

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

```
//Single Line Comments  (added by C99 standard, famously known as c++ style of comments)
```

and

```
/*Multi-Line
Comments*/ (only form of comments supported by C89 standard)
```

Note that Single line comments are a fairly recent addition to C, so some compilers may not support them. A recent version of GCC will have no problems supporting them.

This section is going to focus on the various uses of each form of commentary.

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 5, to explain what 'int i' is going to do
- Line 11, to explain why there is a 'break' keyword.

This will make our program look something like

```
#include <stdio.h>

int main(void) {

    int i=0;                // loop variable.

    printf("Hello, World!");

    for (i=0; i<1; i++) {
        printf("\n");
        break;               //Exits 'for' loop.
    }
```

```
    return 0;
}
```

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 block of code. This can be useful for two reasons: They make your functions easier to understand, and they make it easier to spot errors in code. If you know what a block is *supposed* to do, then it is much easier to find the piece of code that is responsible if an error occurs.

Single-line comments are a new feature, so many C programmers only use multi-line comments.

As an example, suppose we had a program that was designed to print "Hello, World! " a certain number of lines, a specified number of times. 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 before each function, to explain the role, preconditions and postconditions of each function. Always leave the technical details to the individual blocks 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 comment boxes. These

make the comments stand out much more than they would without otherwise. They look like this.

```
/*************************************
 *  This is a multi line comment
 *  That is nearly 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(void)
{
    /*****************************************************************************
     * Function: int main(void)
     * 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 comprehend what the code functions are and how they operate. It also inhibits uncertainty with other like-named functions.

A few programmers add a column of stars on the right side of a block comment:

```
/*************************************
 *  This is a multi line comment       *
 *  that is completely surrounded by a *
 *  cool, starry border!               *
 *************************************/
```

But most programmers don't put any stars on the right side of a block comment. They feel that aligning the right side is a waste of time.

Comments written in source files can be used for documenting source code automatically by using popular tools like Doxygen[4][5]

Links

1. Several programmers recommend "use spaces for indentation. Do not use tabs in your code. You should set your editor to emit spaces when you hit the tab key." [1] (http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml) [2] (http://www.jwz.org/doc/tabs-vs-spaces.html) Other programmers disagree. [3] (http://web.archive.org/20080118165124/diagrammes-modernes.blogspot.com/2006/04/tab-versus-spaces.html) [4] (http://www.derkarl.org/why_to_tabs.html) Regardless of whether you prefer spaces or tabs, make sure you keep it consistent with projects you are working on, because mixing tabs and spaces can cause code to become unreadable.
2. http://www.oualline.com/vim/vim-cook.html#drawing Vim cookbook
3. https://www.kernel.org/doc/Documentation/CodingStyle Linux Kernel Coding Style
4. "Coding Conventions for C++ and Java" (http://www.macadamian.com/index.php?option=com_content&task=view&id=34&Itemid=37) "all the block comments illustrated in this document have no pretty stars on the right side of the block comment. This deliberate choice was made because aligning those pretty stars is a large waste of time and discourages the maintenance of in-line comments.",
5. c2:BigBlocksOfAsterisks,"Code craft" (http://books.google.com/books?id=i4zCzpkrt4sC&pg=PA82&lpg=PA82&dq=programming+comment+block+waste+time+lining+up&source=bl&ots=TUpTMIHBnh&sig=NeZm23WPmvnw2aKMnIRUeQoHmJg&hl=en&ei=pri3SevGIYGyNMn9jd4K&sa=X&oi=book_result&resnum=8&ct=result) by Pete Goodliffe page 82,Falvotech "C Programming Style Guide" (http://www.falvotech.com/content/publications/conventions/c/), Fedora Directory Server Coding Style (http://directory.fedoraproject.org/wiki/Coding_Style)

- Aladdin's C coding guidelines (http://www.cs.wisc.edu/~ghost/doc/AFPL/6.01/C-style.htm) - A more definitive C coding guideline.
- C/C++ Programming Styles (http://www.mycplus.com/category/tutorials/programming-styles/) GNU Coding styles & Linux Kernel Coding style

Error handling

C does not provide direct support for error handling (also known as exception handling). By convention, the programmer is expected to prevent errors from occurring in the first place, and test return values from functions. For example, -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 occur 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 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>        /* perror */
#include <errno.h>        /* errno */
#include <stdlib.h>       /* malloc, free, exit */

int main(void)
{

    /* Pointer to char, requesting dynamic allocation of 2,000,000,000
     * storage elements (declared as an integer constant of type
     * unsigned long int). (If your system has less than 2 GB of memory
     * available, then this call to malloc will fail.)
     */
    char *ptr = malloc(2000000000UL);

    if (ptr == NULL) {
        perror("malloc failed");
        /* here you might want to exit the program or compensate
           for that you don't have 2GB available
         */
    } else {
        /* The rest of the code hereafter can assume that 2,000,000,000
         * chars 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.

Preventing 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.

```
#include <stdio.h> /* for fprintf and stderr */
#include <stdlib.h> /* for exit */
int main( void )
{
    int dividend = 50;
    int divisor = 0;
    int quotient;

    if (divisor == 0) {
        /* Example handling of this error. Writing a message to stderr, and
         * exiting with failure.
         */
        fprintf(stderr, "Division by zero! Aborting...\n");
        exit(EXIT_FAILURE); /* indicate failure.*/
    }

    quotient = dividend / divisor;
    exit(EXIT_SUCCESS); /* indicate success.*/
}
```

Signals

In some cases, the environment may respond to a programming error in C by raising a signal. Signals are events raised by the host environment or operating system to indicate that a specific error or critical event has occurred (e.g. a division by zero, interrupt, and so on.) However, these signals are not meant to be used as a means of error catching; they usually indicate a critical event that will interfere with normal program flow.

To handle signals, a program needs to use the `signal.h` header file. A signal handler will need to be defined, and the signal() function is then called to allow the given signal to be handled. Some signals that are raised to an exception within your code (e.g. a division by zero) are unlikely to allow your program to recover. These signal handlers will be required to instead ensure that some resources are properly cleaned up before the program terminates.

This example creates a signal handler and raises the signal:

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

static void catch_function(int signal) {
    puts("Interactive attention signal caught.");
```

```
}

int main(void) {
    if (signal(SIGINT, catch_function) == SIG_ERR) {
        fputs("An error occurred while setting a signal handler.\n", stderr);
        return EXIT_FAILURE;
    }
    puts("Raising the interactive attention signal.");
    if (raise(SIGINT) != 0) {
        fputs("Error raising the signal.\n", stderr);
        return EXIT_FAILURE;
    }
    puts("Exiting.");
    return 0;
}
```

setjmp

The setjmp function can be used to emulate the exception handling feature of other programming languages. The first call to setjmp provides a reference point to returning to a given function, and is valid as long as the function containing setjmp() doesn't return or exit. A call to longjmp causes the execution to return to the point of the associated setjmp call.

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

jmp_buf test1;

void tryjump()
{
    longjmp(test1, 3);
}

int main (void)
{
    if (setjmp(test1)==0) {
        printf ("setjmp() returned 0.");
        tryjump();
    } else {
        printf ("setjmp returned from a longjmp function call.");
    }
}
```

The values of non-volatile variables may be corrupted when setjmp returns from a longjmp call.

While setjmp() and longjmp() may be used for error handling, it is generally preferred to use the return value of a function to indicate an error, if possible.

Variables

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

It may help to think of variables as a placeholder for a value. You can think of a variable as being equivalent to its assigned value. So, if you have a variable *i* that is **initialized** (set equal) to 4, then it follows that *i + 1* will equal *5*.

Since C is a relatively low-level programming language, before a C program can utilize memory to store a variable it must claim the memory needed to store the values for a variable. This is done by **declaring** variables. Declaring variables is 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 managing and working with variables, it is important to know the *type* of variables and the *size* of these types. Since C is a fairly low-level programming language, 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, every variable declared must be assigned as a certain type of variable.

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 `int`eger 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.

Multiple variables can be declared with one statement, like this:

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

We can also declare *and* assign some content to a variable at the same time.

```
int some_number = 3;
```

This is called *initialization*.

In early versions of C, variables had to be declared at the beginning of a block. In C99 it is allowed to mix declarations and statements arbitrarily – but doing so is not usual, because it is rarely necessary, some compilers still don't support C99 (portability), and it may, because it is uncommon yet, irritate fellow programmers (maintainability).

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, 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[*citation needed*], 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 naming 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.[*citation needed*] Hex numbers are always preceded with *0x*. For now, though, you probably shouldn't be too concerned with hex.

The Four Basic Data Types

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

We will briefly describe them here, then go into more detail in C Programming/Types.

The `int` type

The `int` type stores integers in the form of "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 any one number out of

4294967296 possibilities (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 and j, j here is assigned the literal 5.

The `char` type

The `char` type is capable of holding any member of the execution character set. It stores the same kind of data as an `int` (i.e. integers), but typically has a size of one byte. The size of a byte is specified by the macro `CHAR_BIT` which specifies the number of bits in a char (byte). In standard C it never can be less than 8 bits. A variable of type `char` is most often used to store character data, hence its name. Most implementations use the ASCII character set as the execution character set, but it's best not to know or care about that unless the actual values are important.

Examples of character literals are 'a', 'b', '1', etc., as well as some special characters such as '`\0`' (the null character) and '`\n`' (newline, recall "Hello, World"). Note that the `char` value must be enclosed within single quotations.

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 letter1 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. In short, any single entry that is enclosed within 'single quotes'.

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 displayed. They are surrounded by double quotations (" ", not ' '). An example of a string literal is the "Hello, World!\n" in the "Hello, World" example.

The string literal is assigned to a character **array**, arrays are described later. Example:

```
const char MY_CONSTANT_PEDANTIC_ITCH[] = "learn the usage context.\n";
printf("Square brackets after a variable name means it is a pointer to a string of memory blocks the size of the type of the array element.\n");
```

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 variable (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 a unary operator, not a function.) Its syntax is:

```
sizeof object
sizeof(type)
```

The two expressions above return the size of the object and type specified, in bytes. The return type is `size_t` (defined in the header `<stddef.h>`) which is an unsigned value. Here's an example usage:

```
size_t size;
int i;
size = sizeof(i);
```

`size` will be set to 4, assuming `CHAR_BIT` is defined as 8, and an integer is 32 bits wide. The value of `sizeof`'s result is the number of bytes.

Note that when `sizeof` is applied to a `char`, the result is 1; that is:

```
sizeof(char)
```

always returns 1.

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. Modern C compilers also provide `long long` which is typically an 8 byte integer.

In all of the types described above, one bit is used to indicate the sign (positive or negative) of 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 also may 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, except for certain uses of `char`, and seldom used since all types (except `char`) are signed by default.

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

```
unsigned short int usi;  /* fully qualified -- unsigned short int */
short si;                /* short int */
unsigned long uli;       /* unsigned long int */
```

`const` qualifier

When the **`const`** qualifier is used, the declared variable must be initialized at declaration. It is then not allowed to be changed.

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 value of `pi`.

Note that a Standard conforming compiler must issue a warning if an attempt is made to change a `const` variable - but after doing so the compiler is free to ignore the `const` qualifier.

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 preprocessor will replace all instances of `PRICE_OF_CORN` with 0.99, which the compiler will interpret as the literal `double` 0.99. The preprocessor performs substitution, that is, `PRICE_OF_CORN` is replaced by 0.99 so this means 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 `#define`d words in all capitals, so a programmer will know that this is not a variable that you have declared but a `#define`d macro. It is not necessary to end a preprocessor directive such as `#define` with a semicolon; in fact, some compilers may warn you about unnecessary tokens in your code if you do.

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 to which the variable is intended to be local. 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 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.

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

`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[1][2]) by default. They can be initialized explicitly on declaration to any *constant* value. The initialization is made just once, at compile time.

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

```
#include <stdio.h>

static int j = 0;

void up(void)
{
   /* k is set to 0 when the program starts. The line is then "ignored"
    * for the rest of the program (i.e. k is not set to 0 every time up()
    * is called)
    */
   static int k = 0;
   j++;
   k++;
   printf("up() called.   k= %2d, j= %2d\n", k , j);
}

void down(void)
{
   static int k = 0;
   j--;
   k--;
   printf("down() called. k= %2d, j= %2d\n", k , j);
}

int main(void)
{
   int i;

   /* call the up function 3 times, then the down function 2 times */
   for (i = 0; i < 3; i++)
      up();
```

```
   for (i = 0; i < 2; i++)
      down();

   return 0;
}
```

The j var is accessible by both up and down and retains its value. The k vars also retain their value, but they are two different variables, one in each of their scopes. 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.

Running the program above will produce the following output:

```
up() called.   k=  1, j=  1
up() called.   k=  2, j=  2
up() called.   k=  3, j=  3
down() called. k= -1, j=  2
down() called. k= -2, j=  1
```

Features of `static` variables :

```
   1. Keyword used        - static
   2. Storage             - Memory
   3. Default value       - Zero
   4. Scope               - Local to the block in which it is declared
   5. Lifetime            - Value persists between different function calls
   6. Keyword optionality - Mandatory to use the keyword
```

extern

`extern` is used when a file needs to access a variable in another file that it may not have `#include`d 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.

Features of `extern` variable :

```
   1. Keyword used        - extern
   2. Storage             - Memory
   3. Default value       - Zero
   4. Scope               - Global (all over the program)
   5. Lifetime            - Value persists till the program's execution comes to an end
   6. Keyword optionality - Optional if declared outside all the functions
```

volatile

`volatile` is a special type of 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

`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.

Features of `automatic` variables :

```
1. Keyword used        - auto
2. Storage             - Memory
3. Default value       - Garbage value (random value)
4. Scope               - Local to the block in which it is defined
5. Lifetime            - Value persists while the control remains within the block
6. Keyword optionality - Optional
```

register

`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.

Features of `register` variables :

```
1. Keyword used        - register
2. Storage             - CPU registers (values can be retrieved faster than from memory)
3. Default value       - Garbage value
4. Scope               - Local to the block in which it is defined
5. Lifetime            - Value persists while the control remains within the block
6. Keyword optionality - Mandatory to use the keyword
```

Concepts

- Variables
- Types
- Data Structures
- Arrays

In this section

- C variables
  - C types
  - C arrays

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 parentheses. 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 19 + 31 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
```

```
int main(void)
{
   printf("19+31 is");
```

Better write #include <stdio.h> Use #include "file.h" when the *.h file is in the current work directory.

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("19+31 is %d", 19+31);
```

The placeholder %d literally "holds the place" for the actual number that is the result of adding 19 to 31.

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. Other format specifiers are:

- %d - int (same as %i)
- %ld - long int (same as %li)
- %f - float
- %lf - double
- %c - char
- %s - string
- %x - hexadecimal

A listing of all the format specifiers for `printf` is in the File I/O section.

Tabs and newlines

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

```
 1905
312 +
-----
```

`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\n312 +\n-----\n");
```

or to be a bit more clear, 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("312 +\n");
   printf("-----\n");
   printf("%d", 1905+312);
   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 in 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.)

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);
   printf("You entered: %d\n", a);

   return 0;
}
```

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* ".

If you are trying to input a *string* using *scanf*, you should **not** include the & operator. The code below will not compile.

```
scanf("%s", &a);
```

The correct usage would be:

```
scanf("%s", a);
```

This is because, whenever you use a format specifier for a string (%s), the variable that you use to store the value will be an array and, the array names (in this case - a) themselves point out to their base address and hence, the **address of** operator is not required.

(Although, this is vulnerable to Buffer overflow. fgets() is preferred to scanf()).

Note 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. Some may suggest you use the function fflush(stdin), which may work as desired on some computers, but isn't considered good practice, as you will see later. Doing this has the downfall that if you take your code to a different computer with a different compiler, your code may not work properly.

Links

Back to contents: Beginning C

Simple math

Operators and Assignments

C has a wide range of operators that make simple math easy to handle. The list of operators grouped into precedence levels is as follows:

Primary expressions

An identifier (or variable name) is a primary expression, provided that it has been declared as designating an object (in which case it is an lvalue [a value that can be used as the left side of an assignment expression]) or a function (in which case it is a function designator).

A constant is a primary expression. Its type depends on its form and value.

A string literal is a primary expression.

A parenthesized expression is a primary expression. Its type and value are those of the non-parenthesized expression.

In C11, an expression that starts with `_Generic` followed by (, an initial expression, a list of values of the form *type: expression* where type is either a named type or the keyword default, and) constitute a primary expression. The value is the expression that follows the type of the initial expression or the default if not found.

Postfix operators

First, a primary expression is also a postfix expression. The following expressions are also postfix expressions:

A postfix expression followed by a left square bracket (`[`), an expression, and a right square bracket (`]`) constitutes an invocation of the array subscript operator. One of the expressions shall have type "pointer to object *type*" and the other shall have an integer type; the result type is *type*. Successive array subscript operators designate an element of a multidimensional array.

A postfix expression followed by parentheses or an optional parenthesized argument list indicates an invocation of the function call operator. The value of the function call operator is the return value of the function called with the provided arguments. The parameters to the function are copied on the stack **by value** (or at least the compiler acts as if that is what happens; if the programmer wanted the parameter to be copied by reference, then it is easier to pass the address of the area to be modified by value, then the called function can access the area through the respective pointer). The trend for compilers is to pass the parameters from right to left onto the stack.

A postfix expression followed by a dot (`.`) followed by an identifier selects a member from a structure or union; a postfix expression followed by an arrow (`->`) followed by an identifier selects a member from a structure or union who is pointed to by the pointer on the left-hand side of the expression.

A postfix expression followed by the increment or decrement operators (`++` or `--`) indicates that the variable is to be incremented or decremented as a side effect. The value of the expression is the value of the postfix expression *before* the increment or decrement. These operators only work on integers and pointers.

Unary expressions

First, a postfix expression is a unary expression. The following expressions are all unary expressions:

The increment or decrement operators followed by a unary expression is a unary expression. The value of the expression is the value of the unary expression *after* the increment or decrement. These operators only work on integers and pointers.

The following operators followed by a cast expression are unary expressions:

```
Operator     Meaning
========     =======
   &         Address-of; value is the location of the operand
   *         Contents-of; value is what is stored at the location
   -         Negation
   +         Value-of operator
   !         Logical negation ( (!E) is equivalent to (0==E) )
   ~         Bit-wise complement
```

The keyword `sizeof` followed by a unary expression is a unary expression. The value is the size of the type of the expression in bytes. The expression is not evaluated.

The keyword `sizeof` followed by a parenthesized type name is a unary expression. The value is the size of the type in bytes.

Cast operators

A unary expression is a cast expression.

A parenthesized type name followed by a cast expression is a cast expression. The parenthesized type name has the effect of forcing the cast expression into the type specified by the type name in parentheses. For arithmetic types, this either does not change the value of the expression, or truncates the value of the expression if the expression is an integer and the new type is smaller than the previous type.

An example of casting a float as an int:

```
float pi = 3.141592;
int truncated_pi = (int)pi; // truncated_pi == 3
```

An example of casting a char as an int:

```
char my_char = 'A';
int my_int = (int)my_char; // my_int == 65, which is the ASCII value of 'A'
```

Multiplicative and additive operators

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). (Modulus is not defined for floating-point numbers, but the math library has an `fmod` function.)

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). However, it is always true that for all integer a and nonzero integer b, `((a / b) * b) + (a % b) == a`.

There is no inline operator to do exponentiation (e.g. 5 ^ 2 is **not** 25 [it is 7; ^ is the exclusive-or operator], 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. Multiplicative operators have precedence over additive operators.

```c
#include <stdio.h>

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

    /* while i is less than 5 AND j is less than 5, loop */
    while( (i < 5) && (j < 5) )
    {
        /* postfix increment, i++
         *     the value of i is read and then incremented
         */
        printf("i: %d\t", i++);

        /*
         * prefix increment, ++j
         *     the value of j is incremented and then read
         */
        printf("j: %d\n", ++j);
    }

    printf("At the end they have both equal values:\ni: %d\tj: %d\n", i, j);

    getchar(); /* pause */
    return 0;
}
```

will display the following:

```
i: 0    j: 1
i: 1    j: 2
i: 2    j: 3
i: 3    j: 4
i: 4    j: 5
At the end they have both equal values:
i: 5    j: 5
```

The shift operators (which may be used to rotate bits)

Shift functions are often used in low-level I/O hardware interfacing. Shift and rotate functions are heavily used in cryptography and software floating point emulation. Other than that, shifts can be used in place of division or multiplication by a power of two. Many processors have dedicated function blocks to make these operations fast -- see Microprocessor Design/Shift and Rotate Blocks. On processors which have such blocks, most C compilers compile shift and rotate operators to a single assembly-language instruction -- see X86 Assembly/Shift and Rotate.

shift left

The << operator shifts the binary representation to the left, dropping the most significant bits and appending it with zero bits. The result is equivalent to multiplying the integer by a power of two.

unsigned shift right

The unsigned shift right operator, also sometimes called the logical right shift operator. It shifts the binary representation to the right, dropping the least significant bits and prepending it with zeros. The >> operator is equivalent to division by a power of two for unsigned integers.

signed shift right

The signed shift right operator, also sometimes called the arithmetic right shift operator. It shifts the binary representation to the right, dropping the least significant bit, but prepending it with copies of the original sign bit. The >> operator is not equivalent to division for signed integers.

In C, the behavior of the >> operator depends on the data type it acts on. Therefore, a signed and an unsigned right shift looks exactly the same, but produces a different result in some cases.

rotate right

Contrary to popular belief, it is possible to write C code that compiles down to the "rotate" assembly language instruction (on CPUs that have such an instruction).

Most compilers recognize this idiom:

```
unsigned int x;
unsigned int y;
/* ... */
y = (x >> shift) | (x << (32 - shift));
```

and compile it to a single 32 bit rotate instruction. [3] [4]

On some systems, this may be "#define"ed as a macro or defined as an inline function called something like "rightrotate32" or "rotr32" or "ror32" in a standard header file like "bitops.h". [5]

rotate left

Most compilers recognize this idiom:

```
unsigned int x;
unsigned int y;
/* ... */
y = (x << shift) | (x >> (32 - shift));
```

and compile it to a single 32 bit rotate instruction.

On some systems, this may be "#define"ed as a macro or defined as an inline function called something like "leftrotate32" or "rotl32" in a header file like "bitops.h".

Relational and equality operators

The relational binary operators < (less than), > (greater than), <= (less than or equal), and >= (greater than or equal) operators return a value of 1 if the result of the operation is true, 0 if false.

The equality binary operators == (equals) and != (not equals) operators are similar to the relational operators except that their precedence is lower.

Bitwise operators

The bitwise operators are & (and), ^ (exclusive or) and | (inclusive or). The & operator has higher precedence than ^, which has higher precedence than |.

The values being operated upon must be integral; the result is integral.

One use for the bitwise operators is to emulate bit flags. These flags can be set with OR, tested with AND, flipped with XOR, and cleared with AND NOT. For example:

```
/* This code is a sample for bitwise operations.  */
```

```
#define BITFLAG1    (1)
#define BITFLAG2    (2)
#define BITFLAG3    (4) /* They are powers of 2 */

unsigned bitbucket = 0U;    /* Clear all */
bitbucket |= BITFLAG1;      /* Set bit flag 1 */
bitbucket &= ~BITFLAG2;     /* Clear bit flag 2 */
bitbucket ^= BITFLAG3;      /* Flip the state of bit flag 3 from off to on or
                               vice versa */
if (bitbucket & BITFLAG3) {
  /* bit flag 3 is set */
} else {
  /* bit flag 3 is not set */
}
```

Logical operators

The logical operators are `&&` (and), and `||` (or). Both of these operators produce 1 if the relationship is true and 0 for false. Both of these operators short-circuit; if the result of the expression can be determined from the first operand, the second is ignored. The `&&` operator has higher precedence than the `||` operator.

`&&` is used to evaluate expressions left to right, and returns a 1 if *both* statements are true.

```
 int x = 7;
 int y = 5;
 if(x == 7 && y == 5) {
     ...
 }
```

Here, the && operator checks the left-most expression, then the expression to its right. If there were more than two expressions chained (e.g. x && y && z), the operator would check x first, then y, then continue rightwards to z if neither x or y is zero. Since both statements return true, the && operator returns true, and the code block is executed.

```
   if(x == 5 && y == 5) {
       ...
   }
```

The && operator checks in the same way as before, and finds that the first expression is false. The && operator stops evaluating as soon as it finds a statement to be false, and returns a false.

`||` is used to evaluate expressions left to right, and returns a 1 if *either* of the expressions are true.

```
   /* Use the same variables as before. */
   if(x == 2 || y == 5) { // the || statement checks both expressions, finds that the latter is true, and returns true
       ...
   }
```

The `||` operator here checks the left-most expression, finds it false, but continues to evaluate the next expression. It finds that the next expression returns true, stops, and returns a 1. Much how the `&&` operator ceases when it finds an expression that returns false, the `||` operator ceases when it finds an expression that returns true.

It is worth noting that C does not have Boolean values (true and false) commonly found in other languages. It instead interprets a 0 as false, and any nonzero value as true.

Conditional operators

The ternary `?:` operator is the conditional operator. The expression `(x ? y : z)` has the value of `y` if `x` is nonzero, `z` otherwise.

Example:

```
int x = 0;
int y;
y = (x ? 10 : 6); /* The parentheses are technically not necessary as assignment
                     has a lower precedence than the conditional operator, but
                     it's there for clarity.  */
```

The expression `x` evaluates to 0. The ternary operator then looks for the "if-false" value, which in this case, is 6. It returns that, so `y` is equal to six. Had `x` been a non-zero, then the expression would have returned a 10.

Assignment operators

The assignment operators are `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, and `|=` . The `=` operator stores the value of the right operand into the location determined by the left operand, which must be an lvalue (a value that has an address, and therefore can be assigned to).

For the others, `x op= y` is shorthand for `x = x op (y)` . Hence, the following expressions are the same:

```
   1. x += y     -     x = x+y
```

```
    2. x -= y      -      x = x-y
    3. x *= y      -      x = x*y
    4. x /= y      -      x = x/y
    5. x %= y      -      x = x%y
```

The value of the assignment expression is the value of the left operand after the assignment. Thus, assignments can be chained; e.g. the expression `a = b = c = 0;` would assign the value zero to all three variables.

Comma operator

The operator with the least precedence is the comma operator. The value of the expression `x, y` will evaluate both `x` and `y`, but provides the value of `y`.

This operator is useful for including multiple actions in one statement (e.g. within a for loop conditional).

Here are some small examples of the comma operator:

```
int i, x;      /* Declares two ints, i and x, in one declaration.
                  Technically, this is not the comma operator. */

/* this loop initializes x and i to 0, then runs the loop */
for (x = 0, i = 0; i <= 6; i++) {
    printf("x = %d, and i = %d\n", x, i);
}
```

1. [5] (http://c-faq.com/null/macro.html) - What is NULL and how is it defined?
2. [6] (http://c-faq.com/null/nullor0.html) - NULL or 0, which should you use?
3. GCC: "Optimize common rotate constructs" (http://gcc.gnu.org/ml/gcc-patches/2007-11/msg01112.html)
4. "Cleanups in ROTL/ROTR DAG combiner code" (http://www.mail-archive.com/llvm-commits@cs.uiuc.edu/msg17216.html) mentions that this code supports the "rotate" instruction in the CellSPU
5. "replace private copy of bit rotation routines" (http://archive.is/20130415063059/kerneltrap.org/mailarchive/linux-kernel/2008/4/15/1440064) -- recommends including "bitops.h" and using its rol32 and ror32 rather than copy-and-paste into a new program.

Further math

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. To use these math functions, you must link your program with the math library. For some compilers (including GCC), you must specify the additional

parameter `-lm`[1][2].

The math functions may produce one of two kinds of errors. *Domain errors* occur when the parameters to the functions are invalid, such as a negative number as a parameter to `sqrt` (the square root function). *Range errors* occur when the result of the function cannot be expressed in that particular floating-point type, such as `pow(1000.0, 1000.0)` if the maximum value of a double is around 10^{308}.

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.

The `acosh` functions compute the inverse hyperbolic cosine of the argument. A domain error occurs for arguments less than 1.

The `asinh` functions compute the inverse hyperbolic sine of the argument.

The `atanh` functions compute the inverse hyperbolic tangent of the argument. A domain error occurs if the argument is not in the interval [-1, +1]. A range error may occur if the argument equals -1 or +1.

```
#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 */
float acoshf(float x); /* C99 */
float asinhf(float x); /* C99 */
float atanhf(float x); /* C99 */
double acosh(double x); /* C99 */
double asinh(double x); /* C99 */
double atanh(double x); /* C99 */
long double acoshl(long double x); /* C99 */
```

```
long double asinhl(long double x); /* C99 */
long double atanhl(long double x); /* C99 */
```

Exponential and logarithmic functions

The `exp`, `exp2`, and `expm1` functions

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

The `exp2` functions compute the base-2 exponential function of `x` (2^x). A range error occurs if the magnitude of `x` is too large.

The `expm1` functions compute the base-*e* exponential function of the argument, minus 1. 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 */
float exp2f(float x); /* C99 */
double exp2(double x); /* C99 */
long double exp2l(long double x); /* C99 */
float expm1f(float x); /* C99 */
double expm1(double x); /* C99 */
long double expm1l(long double x); /* C99 */
```

The `frexp`, `ldexp`, `modf`, `scalbn`, and `scalbln` functions

These functions are heavily used in software floating-point emulators, but are otherwise rarely directly called.

Inside the computer, each floating point number is represented by two parts:

- The significand is either in the range [1/2, 1), or it equals zero.
- The exponent is an integer.

The value of a floating point number v is $v = \text{significand} \times 2^{\text{exponent}}$.

The `frexp` functions break the argument floating point number `value` into those two parts, the exponent and significand. After breaking it apart, it stores the exponent in the `int` object pointed to by `ex`, and returns the significand. In other words, the value returned is a copy of the given floating point number but with an exponent replaced by 0. 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. In other words, it returns copy of the given floating point number with the exponent increased by ex. A range error may occur.

The `modf` functions break 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. The `*iptr` is a floating-point type, rather than an "int" type, because it might be used to store an integer like 1 000 000 000 000 000 000 000 which is too big to fit in an int.

The `scalbn` and `scalbln` compute $x \times \texttt{FLT\_RADIX}^n$. `FLT_RADIX` is the base of the floating-point system; if it is 2, the functions are equivalent to `ldexp`.

```
#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 */
float scalbnf(float x, int ex); /* C99 */
double scalbn(double x, int ex); /* C99 */
long double scalbnl(long double x, int ex); /* C99 */
float scalblnf(float x, long int ex); /* C99 */
double scalbln(double x, long int ex); /* C99 */
long double scalblnl(long double x, long int ex); /* C99 */
```

Most C floating point libraries also implement the IEEE754-recommended nextafter(), nextUp( ), and nextDown( ) functions. [10] (http://www.opengroup.org/onlinepubs/009695399/functions/nextafter.html)

The `log`, `log2`, `log1p`, and `log10` functions

The `log` functions compute the base-*e* natural (**not** common) 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 `log1p` functions compute the base-*e* natural (**not** common) logarithm of one plus the argument and return the result. A domain error occurs if the argument is less than -1. A range error may occur if the argument is -1.

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.

The `log2` functions compute the base-2 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 log1pf(float x); /* C99 */
double log1p(double x); /* C99 */
long double log1pl(long double x); /* C99 */
float log10f(float x); /* C99 */
double log10(double x);
long double log10l(long double x); /* C99 */
float log2f(float x); /* C99 */
double log2(double x); /* C99 */
long double log2l(long double x); /* C99 */
```

The `ilogb` and `logb` functions

The `ilogb` functions extract the exponent of `x` as a signed int value. If `x` is zero, they return the value `FP_ILOGB0`; if `x` is infinite, they return the value `INT_MAX`; if `x` is not-a-number they return the value `FP_ILOGBNAN`; otherwise, they are equivalent to calling the corresponding `logb` function and casting the returned value to type `int`. A range error may occur if `x` is zero. `FP_ILOGB0` and `FP_ILOGBNAN` are macros defined in `math.h`; `INT_MAX` is a macro defined in `limits.h`.

The `logb` functions extract the exponent of `x` as a signed integer value in floating-point format. If `x` is subnormal, it is treated as if it were normalized; thus, for positive finite `x`, $1 \leq x \times \texttt{FLT\_RADIX}^{-\texttt{logb}(x)} < \texttt{FLT\_RADIX}$. `FLT_RADIX` is the radix for floating-point numbers, defined in the `float.h` header.

```
#include <math.h>
int ilogbf(float x); /* C99 */
int ilogb(double x); /* C99 */
int ilogbl(long double x); /* C99 */
float logbf(float x); /* C99 */
double logb(double x); /* C99 */
long double logbl(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 positive 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 */
```

The `cbrt` functions

The `cbrt` functions compute the cube root of `x` and return the result.

```
#include <math.h>
float cbrtf(float x); /* C99 */
double cbrt(double x); /* C99 */
long double cbrtl(long double x); /* C99 */
```

The `hypot` functions

The `hypot` functions compute the square root of the sums of the squares of `x` and `y`, without overflow or underflow, and return the result.

```
#include <math.h>
float hypotf(float x, float y); /* C99 */
double hypot(double x, double y); /* C99 */
long double hypotl(long double x, long double y); /* 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 */
```

The `nearbyint`, `rint`, `lrint`, and `llrint` functions

The `nearbyint` functions round their argument to an integer value in floating-point format, using the current rounding direction and without raising the "inexact" floating-point exception.

The `rint` functions are similar to the `nearbyint` functions except that they can raise the "inexact" floating-point exception if the result differs in value from the argument.

The `lrint` and `llrint` functions round their arguments to the nearest integer value according to the current rounding direction. If the result is outside

the range of values of the return type, the numeric result is undefined and a range error may occur if the magnitude of the argument is too large.

```
#include <math.h>
float nearbyintf(float x); /* C99 */
double nearbyint(double x); /* C99 */
long double nearbyintl(long double x); /* C99 */
float rintf(float x); /* C99 */
double rint(double x); /* C99 */
long double rintl(long double x); /* C99 */
long int lrintf(float x); /* C99 */
long int lrint(double x); /* C99 */
long int lrintl(long double x); /* C99 */
long long int llrintf(float x); /* C99 */
long long int llrint(double x); /* C99 */
long long int llrintl(long double x); /* C99 */
```

The `round`, `lround`, and `llround` functions

The `round` functions round the argument to the nearest integer value in floating-point format, rounding halfway cases away from zero, regardless of the current rounding direction.

The `lround` and `llround` functions round the argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction. If the result is outside the range of values of the return type, the numeric result is undefined and a range error may occur if the magnitude of the argument is too large.

```
#include <math.h>
float roundf(float x); /* C99 */
double round(double x); /* C99 */
long double roundl(long double x); /* C99 */
long int lroundf(float x); /* C99 */
long int lround(double x); /* C99 */
long int lroundl(long double x); /* C99 */
long long int llroundf(float x); /* C99 */
long long int llround(double x); /* C99 */
long long int llroundl(long double x); /* C99 */
```

The `trunc` functions

The `trunc` functions round their argument to the integer value in floating-point format that is nearest but no larger in magnitude than the argument.

```
#include <math.h>
float truncf(float x); /* C99 */
```

```
double trunc(double x); /* C99 */
long double truncl(long double x); /* C99 */
```

The `remainder` functions

The `remainder` functions compute the remainder `x` REM `y` as defined by IEC 60559. The definition reads, "When $y \neq 0$, the remainder $r = x$ REM y is defined regardless of the rounding mode by the mathematical reduction $r = x - ny$, where n is the integer nearest the exact value of x/y; whenever $|n - x/y| = ½$, then n is even. Thus, the remainder is always exact. If $r = 0$, its sign shall be that of x." This definition is applicable for all implementations.

```
#include <math.h>
float remainderf(float x, float y); /* C99 */
double remainder(double x, double y); /* C99 */
long double remainderl(long double x, long double y); /* C99 */
```

The `remquo` functions

The `remquo` functions return the same remainder as the `remainder` functions. In the object pointed to by `quo`, they store a value whose sign is the sign of `x`/`y` and whose magnitude is congruent modulo 2^n to the magnitude of the integral quotient of `x`/`y`, where n is an implementation-defined integer greater than or equal to 3.

```
#include <math.h>
float remquof(float x, float y, int *quo); /* C99 */
double remquo(double x, double y, int *quo); /* C99 */
long double remquol(long double x, long double y, int *quo); /* C99 */
```

Error and gamma functions

The `erf` functions compute the error function of the argument $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,dt$; the `erfc` functions compute the complimentary error function of the argument (that is, 1 - erf x). For the `erfc` functions, a range error may occur if the argument is too large.

The `lgamma` functions compute the natural logarithm of the absolute value of the gamma of the argument (that is, $\log_e|\Gamma(x)|$). A range error may occur if the argument is a negative integer or zero.

The `tgamma` functions compute the gamma of the argument (that is, $\Gamma(x)$). A domain error occurs if the argument is a negative integer or if the result cannot be represented when the argument is zero. A range error may occur.

```
#include <math.h>
float erff(float x); /* C99 */
double erf(double x); /* C99 */
long double erfl(long double x); /* C99 */
float erfcf(float x); /* C99 */
double erfc(double x); /* C99 */
long double erfcl(long double x); /* C99 */
float lgammaf(float x); /* C99 */
double lgamma(double x); /* C99 */
long double lgammal(long double x); /* C99 */
float tgammaf(float x); /* C99 */
double tgamma(double x); /* C99 */
long double tgammal(long double x); /* C99 */
```

Further reading

1. [7] (https://stackoverflow.com/questions/1033898/why-do-you-have-to-link-the-math-library-in-c) Why do you have to link the math library in C?
2. [8] (https://stackoverflow.com/questions/5419366/why-do-i-have-to-explicitly-link-with-libm) Why do I have to explicitly link with libm?

Control

Very few 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. [1]

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 expression `(a = b)` sets `a` equal to `b` and subsequently evaluates to `b`; but the expression `(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 simple technique to avoid this kind of bug (in many, not all cases) is to put the constant first. This will cause the compiler to issue an error, if == got misspelled with =.

Note that C does not have a dedicated boolean type as many other languages do. 0 means false and anything else true. So the following are equivalent:

```
if (foo()) {
  // do something
}
```

and

```
if (foo() != 0) {
  // do something
}
```

Often `#define TRUE 1` and `#define FALSE 0` are used to work around the lack of a boolean type. This is bad practice, since it makes assumptions that do not hold. It is a better idea to indicate what you are actually expecting as a result from a function call, as there are many different ways of indicating error conditions, depending on the situation.

```
if (strstr("foo", bar) >= 0) {
  // bar contains "foo"
}
```

Here, `strstr` returns the index where the substring foo is found and -1 if it was not found. Note that this would fail with the `TRUE` definition mentioned in the previous paragraph. It would also not produce the expected results if we omitted the `>= 0`.

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) { // don't do this! it always evaluates to "true"!
  /* 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 < 10) {    // the && means "and"
  /* do some stuff */
}
```

Logical Expressions

a || b
: when EITHER **a** or **b** is true (or both), the result is 1, otherwise the result is 0.

a && b
: when BOTH **a** and **b** are true, the result is 1, otherwise the result is 0.

!a
: when **a** is true, the result is 0, when **a** is 0, the result is 1.

Here's an example of a larger 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 snippet 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 precedes 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. Hexadecimal 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 powers 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.

When the computer reaches the end of the while loop, it always goes back to the while statement at the top of the loop, where it re-evaluates the

controlling condition. If that condition is "true" at that instant -- even if it was temporarily 0 for a few statements inside the loop -- then the computer begins executing the statements inside the loop again; otherwise the computer exits the loop. The computer does not "continuously check" the controlling condition of a while loop during the execution of that loop. It only "peeks" at the controlling condition each time it reaches the `while` at the top of the loop.

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 (inside the loop). 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 initialize 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 an infinite 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);
}
```

Special care should be taken when designing or refactoring the conditional part, especially whether using < or <= , whether start and stop should be corrected by 1, and in case of prefix- and postfix-notations. (On a 100 yards promenade with a tree every 10 yards there are 11 trees.)

```
int i, n = 10;
for (i = 0; i < n; i++)
  printf("%d ", i); // processed n times => 0 1 2 3 ... (n-1)
printf("\n");
for (i = 0; i <= n; i++)
  printf("%d ", i); // processed (n+1) times => 0 1 2 3 ... n
printf("\n");
for (i = n; i--;)
  printf("%d ", i); // processed n times => (n-1) ...3 2 1 0
printf("\n");
for (i = n; --i;)
  printf("%d ", i); // processed (n-1) times => (n-1) ...4 3 2 1
printf("\n");
```

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, and use goto only when necessary. Keep in mind that many, if not most, C style guidelines *strictly forbid* use of **goto**, with 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

```c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int years;

    printf("Enter your age in years : ");
    fflush(stdout);
    errno = 0;
    if (scanf("%d", &years) != 1 || errno)
        return EXIT_FAILURE;
    printf("Your age in days is %d\n", years * 365);
    return 0;
}
```

Further reading

1. C FAQ (http://www.c-faq.com/bool/bool2.html)

Procedures and functions

In C programming, all executable code resides within a **function**. A function is a named block of code that performs a task and then returns control to a caller. Note that other programming languages may distinguish between a "function", "subroutine", "subprogram", "procedure", or "method" -- in C, these are all functions.

A function is often executed (called) several times, from several different places, during a single execution of the program. After finishing a subroutine, the program will branch back (return) to the point after the call.

Functions are a powerful programming tool.

As a basic example, suppose you are writing code to print out the first 5 squares of numbers, do some intermediate processing, then print the first 5 squares again. We could write it like this:

```c
#include <stdio.h>

int main(void)
{
  int i;
  for(i=1; i <= 5; i++)
  {
```

```
    printf("%d ", i*i);
  }
  for(i=1; i <= 5; i++)
  {
    printf("%d ", i*i);
  }
  return 0;
}
```

We have to write the same loop twice. We may want to somehow put this code in a separate place and simply jump to this code when we want to use it. This would look like:

```
#include <stdio.h>

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

int main(void)
{
  Print_Squares();
  Print_Squares();
  return 0;
}
```

This is precisely what functions are for.

More on functions

A function is like a black box. It takes in input, does something with it, 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 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 *g* that gives some kind of answer back to *f* is said to *return* that answer. 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*.

In between 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.

Written in a more concise manner, this code performs exactly the same function as the above:

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

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 if we want the function to return 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. (Actually, one can use the `return` keyword in a procedure to return to the caller before the end of the procedure, but one cannot return a value as if it were a function.)

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.

```
     // don't run this!
void infinite_recursion()
{
    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 keeps its value
       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 int compare( int a, int 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.

Functions from the C Standard Library

While the C language doesn't itself contain functions, it is usually linked with the C Standard Library. To use this library, you need to add an #include directive at the top of the C file, which may be one of the following:

- <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>
- <complex.h>

The functions available are:

<assert.h>	<limits.h>	<signal.h>	<stdlib.h>
▪ assert(int)	▪ (constants only)	▪ int raise(int sig). This ▪ void* signal(int sig, void (*func)(int))	▪ atof(char*), atoi(char*), atol(char*) ▪ strtod(char * str, char ** endptr), strtol(char *str, char **endptr), strtoul(char *str, char **endptr) ▪ rand(), srand() ▪ malloc(size_t), calloc (size_t elements, size_t elementSize), realloc(void*, int) ▪ free (void*) ▪ exit(int), abort() ▪ atexit(void (*func)()) ▪ getenv ▪ system ▪ qsort(void *, size_t number, size_t size, int (*sortfunc)(void*, void*)) ▪ abs, labs ▪ div, ldiv
<ctype.h>	<locale.h>	<stdarg.h>	<string.h>
▪ isalnum, isalpha, isblank ▪ iscntrl, isdigit, isgraph ▪ islower, isprint, ispunct ▪ isspace, isupper, isxdigit ▪ tolower, toupper	▪ struct lconv* localeconv(void); ▪ char* setlocale(int, const char*);	▪ va_start (va_list, ap) ▪ va_arg (ap, (type)) ▪ va_end (ap) ▪ va_copy (va_list, va_list)	▪ memcpy, memmove ▪ memchr, memcmp, memset ▪ strcat, strncat, strchr, strrchr ▪ strcmp, strncmp, strccoll ▪ strcpy, strncpy ▪ strerror ▪ strlen ▪ strspn, strcspn ▪ strpbrk ▪ strstr ▪ strtok ▪ strxfrm
errno.h	**math.h**	**stddef.h**	**time.h**
▪ (errno)	▪ sin, cos, tan	▪ offsetof macro	▪ asctime (struct tm* tmptr)

	■ asin, acos, atan, atan2 ■ sinh, cosh, tanh ■ ceil ■ exp ■ fabs ■ floor ■ fmod ■ frexp ■ ldexp ■ log, log10 ■ modf ■ pow ■ sqrt		■ clock_t clock() ■ char* ctime(const time_t* timer) ■ double difftime(time_t timer2, time_t timer1) ■ struct tm* gmtime(const time_t* timer) ■ struct tm* gmtime_r(const time_t* timer, struct tm* result) ■ struct tm* localtime(const time_t* timer) ■ time_t mktime(struct tm* ptm) ■ time_t time(time_t* timer) ■ char * strptime(const char* buf, const char* format, struct tm* tptr) ■ time_t timegm(struct tm *brokentime)
float.h	**setjmp.h**	**stdio.h**	
■ (constants)	■ int setjmp(jmp_buf env) ■ void longjmp(jmp_buf env, int value)	■ fclose ■ fopen, freopen ■ remove ■ rename ■ rewind ■ tmpfile ■ clearerr ■ feof, ferror ■ fflush ■ fgetpos, fsetpos ■ fgetc, fputc ■ fgets, fputs ■ ftell, fseek	■ fread, fwrite ■ getc, putc ■ getchar, putchar, fputchar ■ gets, puts ■ printf, vprintf ■ fprintf, vfprintf ■ sprintf, snprintf, vsprintf, vsnprintf ■ perror ■ scanf, vscanf ■ fscanf, vfscanf ■ sscanf, vsscanf ■ setbuf, setvbuf ■ tmpnam ■ ungetc

- printf
- full list (http://www.utas.edu.au/infosys/info/documentation/C/CStdLib.html#ctype.h)

Variable-length argument lists

Functions with variable-length argument lists are functions that can take a varying number of arguments. An example in the C standard library is the `printf` function, which can take any number of arguments depending on how the programmer wants to use it.

C programmers rarely find the need to write new functions with variable-length arguments. If they want to pass a bunch of things to a function, they typically define a structure to hold all those things -- perhaps a linked list, or an array -- and call that function with the data in the arguments.

However, you may occasionally find the need to write a new function that supports a variable-length argument list. To create a function that can accept a variable-length argument list, you must first include the standard library header `stdarg.h`. Next, declare the function as you would normally. Next, add as the last argument an ellipsis ("..."). This indicates to the compiler that a variable list of arguments is to follow. For example, the following function declaration is for a function that returns the average of a list of numbers:

```
float average (int n_args, ...);
```

Note that because of the way variable-length arguments work, we must somehow, in the arguments, specify the number of elements in the variable-length part of the arguments. In the `average` function here, it's done through an argument called `n_args`. In the `printf` function, it's done with the format codes that you specify in that first string in the arguments you provide.

Now that the function has been declared as using variable-length arguments, we must next write the code that does the actual work in the function. To access the numbers stored in the variable-length argument list for our `average` function, we must first declare a variable for the list itself:

```
va_list myList;
```

The `va_list` type is a type declared in the `stdarg.h` header that basically allows you to keep track of your list. To start actually using `myList`, however, we must first assign it a value. After all, simply declaring it by itself wouldn't do anything. To do this, we must call `va_start`, which is actually a macro defined in `stdarg.h`. In the arguments to `va_start`, you must provide the `va_list` variable you plan on using, as well as the name of the last variable appearing before the ellipsis in your function declaration:

```
#include <stdarg.h>
float average (int n_args, ...)
{
    va_list myList;
    va_start (myList, n_args);
    va_end (myList);
}
```

Now that `myList` has been prepped for usage, we can finally start accessing the variables stored in it. To do so, use the `va_arg` macro, which pops off the next argument on the list. In the arguments to `va_arg`, provide the `va_list` variable you're using, as well as the primitive data type (e.g. `int`, `char`) that the variable you're accessing should be:

```
#include <stdarg.h>
float average (int n_args, ...)
{
    va_list myList;
    va_start (myList, n_args);

    int myNumber = va_arg (myList, int);
    va_end (myList);
}
```

By popping `n_args` integers off of the variable-length argument list, we can manage to find the average of the numbers:

```
#include <stdarg.h>
float average (int n_args, ...)
{
    va_list myList;
    va_start (myList, n_args);

    int numbersAdded = 0;
    int sum = 0;

    while (numbersAdded < n_args) {
        int number = va_arg (myList, int); // Get next number from list
        sum += number;
        numbersAdded += 1;
    }
    va_end (myList);

    float avg = (float)(sum) / (float)(numbersAdded); // Find the average
    return avg;
}
```

By calling `average (2, 10, 20)`, we get the average of `10` and `20`, which is `15`.

Preprocessor

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. C++ compilers use the same C preprocessor.[1]

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`, located 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>`
- `<fenv.h>`
- `<inttypes.h>`
- `<iso646.h>`
- `<stdbool.h>`
- `<stdint.h>`
- `<tgmath.h>`
- `<wchar.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. The #pragma directive provides a way to request special behavior from the compiler. This directive is most useful for programs

that are unusually large or that need to take advantage of the capabilities of a particular compiler.

Pragmas are used within the source program.

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

1. pragma is usually followed by a single token, which represents a command for the compiler to obey. You should check the software implementation of the C standard you intend on using for a list of the supported tokens. Not surprisingly, the set of commands that can appear in #pragma directives is different for each compiler; you'll have to consult the documentation for your compiler to see which commands it allows and what those commands do.

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 referred as using **include guards**.

#define

WARNING: Preprocessor macros, although tempting, can produce quite unexpected results if not done right. Always keep in mind that macros are textual substitutions done to your source code before anything is compiled. The compiler does not know anything about the macros and never gets to see them. This can produce obscure errors, amongst other negative effects. Prefer to use language features, if there are equivalent (In example use `const int` or `enum` instead of `#define`d constants).

That said, there are cases, where macros are very useful (see the `debug` macro below for an example).

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() were a real function 'x' would now have the value of '-9', but because it was an argument in a macro it was expanded twice and thus has a value of -8.

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 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` or `b` contains expressions, the definition must parenthesize every use of `a,b` in the macro definition, 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 int max(int a, int b) {
  return a>b?a:b
}
```

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

NOTE: The explicit `inline` declaration is not really necessary unless the definition is in a header file, since your compiler can inline functions for you (with gcc this can be done with `-finline-functions` or `-O3`). The compiler is often better than the programmer at predicting which functions are worth inlining. Also, function calls are not really expensive (they used to be).

The compiler is actually free to ignore the `inline` keyword. It is only a hint (except that `inline` is necessary in order to allow a function to be defined in a header file without generating an error message due to the function being defined in more than one translation unit).

(#, ##)

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.

The following trick can be used to turn a numeric constants into string literals

```
#define num2str(x) str(x)
#define str(x) #x
#define CONST 23
```

```
puts(num2str(CONST));
```

This is a bit tricky, since it is expanded in 2 steps. First `num2str(CONST)` is replaced with `str(23)`, which in turn is replaced with `"23"`. This can be useful in the following example:

```
#ifdef DEBUG
#define debug(msg) fputs(__FILE__ ":" num2str(__LINE__) " - " msg, stderr)
#else
#define debug(msg)
#endif
```

This will give you a nice debug message including the file and the line where the message was issued. If DEBUG is not defined however the debugging message will completely vanish from your code. Be careful not to use this sort of construct with anything that has side effects, since this can lead to bugs, that appear and disappear depending on the compilation parameters.

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(void)
{
  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.

One of the few situations where inline functions won't work -- so you are pretty much forced to use function-like macros instead -- is to initialize compile time constants (static initialization of structs). This happens when the arguments to the macro are literals that the compiler can optimize to another literal. [2]

#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.

Programmers use "#error" inside a conditional block, to immediately halt the compiler when the "#if" or "#ifdef" -- at the beginning of the block -- detects a compile-time problem. Normally the compiler skips the block (and the "#error" directive inside it) and the compilation proceeds.

```
#error message
```

#warning

Many compilers support a **#warning** directive. When one is encountered, the compiler emits a diagnostic containing the remaining tokens in the directive.

```
#warning 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, the increment and decrement operators, the address-of operator, and the sizeof operator.

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
```

Useful Preprocessor Macros for Debugging

ANSI C defines some useful preprocessor macros and variables,[3][4] also called "magic constants", include:

__FILE__ => The name of the current file, as a string literal
__LINE__ => Current line of the source file, as a numeric literal
__DATE__ => Current system date, as a string
__TIME__ => Current system time, as a string
__TIMESTAMP__ => Date and time (non-standard)
__cplusplus => undefined when your C code is being compiled by a C compiler; 199711L when your C code is being compiled by a C++ compiler compliant with 1998 C++ standard.
__func__ => Current function name of the source file, as a string (part of C99)
__PRETTY_FUNCTION__ => "decorated" Current function name of the source file, as a string (in GCC; non-standard)

Compile-time assertions

Compile-time assertions can help you debug faster than using only run-time assert() statements, because the compile-time assertions are all tested at compile time, while it is possible that a test run of a program may fail to exercise some run-time assert() statements.

Prior to the C11 standard, some people[5][6][7] defined a preprocessor macro to allow compile-time assertions, something like:

```
#define COMPILE_TIME_ASSERT(pred) switch(0){case 0:case pred:;}

COMPILE_TIME_ASSERT( BOOLEAN CONDITION );
```

The `static_assert.hpp` Boost library defines a similar macro.[8]

Since C11, such macros are obsolete, as `_Static_assert` and its macro equivalent `static_assert` are standardized and built-in to the language.

X-Macros

One little-known usage pattern of the C preprocessor is known as "X-Macros".[9][10][11][12] An X-Macro is a header file or macro. Commonly these use the extension ".def" instead of the traditional ".h". This file contains a list of similar macro calls, which can be referred to as "component macros". The include file is then referenced repeatedly in the following pattern. Here, the include file is "xmacro.def" and it contains a list of component macros of the style "foo(x, y, z)".

```
#define foo(x, y, z) doSomethingWith(x, y, z);
#include "xmacro.def"
#undef foo

#define foo(x, y, z) doSomethingElseWith(x, y, z);
#include "xmacro.def"
#undef foo

(etc...)
```

The most common usage of X-Macros is to establish a list of C objects and then automatically generate code for each of them. Some implementations also perform any `#undef`s they need inside the X-Macro, as opposed to expecting the caller to undefine them.

Common sets of objects are a set of global configuration settings, a set of members of a struct, a list of possible XML tags for converting an XML file to a quickly-traversable tree, or the body of an enum declaration; other lists are possible.

Once the X-Macro has been processed to create the list of objects, the component macros can be redefined to generate, for instance, accessor and/or mutator functions. Structure serializing and deserializing are also commonly done.

Here is an example of an X-Macro that establishes a struct and automatically creates serialize/deserialize functions. For simplicity, this example doesn't account for endianness or buffer overflows.

File **star.def**:

```
EXPAND_EXPAND_STAR_MEMBER(x, int)
```

```
EXPAND_EXPAND_STAR_MEMBER(y, int)
EXPAND_EXPAND_STAR_MEMBER(z, int)
EXPAND_EXPAND_STAR_MEMBER(radius, double)
#undef EXPAND_EXPAND_STAR_MEMBER
```

File **star_table.c**:

```
typedef struct {
  #define EXPAND_EXPAND_STAR_MEMBER(member, type) type member;
  #include "star.def"
  } starStruct;

void serialize_star(const starStruct *const star, unsigned char *buffer) {
  #define EXPAND_EXPAND_STAR_MEMBER(member, type) \
    memcpy(buffer, &(star->member), sizeof(star->member)); \
    buffer += sizeof(star->member);
  #include "star.def"
  }

void deserialize_star(starStruct *const star, const unsigned char *buffer) {
  #define EXPAND_EXPAND_STAR_MEMBER(member, type) \
    memcpy(&(star->member), buffer, sizeof(star->member)); \
    buffer += sizeof(star->member);
  #include "star.def"
  }
```

Handlers for individual data types may be created and accessed using token concatenation ("##") and quoting ("#") operators. For example, the following might be added to the above code:

```
#define print_int(val)    printf("%d", val)
#define print_double(val) printf("%g", val)

void print_star(const starStruct *const star) {
  /* print_##type will be replaced with print_int or print_double */
  #define EXPAND_EXPAND_STAR_MEMBER(member, type) \
    printf("%s: ", #member); \
    print_##type(star->member); \
    printf("\n");
  #include "star.def"
  }
```

Note that in this example you can also avoid the creation of separate handler functions for each datatype in this example by defining the print format for each supported type, with the additional benefit of reducing the expansion code produced by this header file:

```
#define FORMAT_(type) FORMAT_##type
#define FORMAT_int    "%d"
#define FORMAT_double "%g"
```

```
void print_star(const starStruct *const star) {
  /* FORMAT_(type) will be replaced with FORMAT_int or FORMAT_double */
  #define EXPAND_EXPAND_STAR_MEMBER(member, type) \
    printf("%s: " FORMAT_(type) "\n", #member, star->member);
  #include "star.def"
  }
```

The creation of a separate header file can be avoided by creating a single macro containing what would be the contents of the file. For instance, the above file "star.def" could be replaced with this macro at the beginning of:

File **star_table.c**:

```
#define EXPAND_STAR \
  EXPAND_STAR_MEMBER(x, int) \
  EXPAND_STAR_MEMBER(y, int) \
  EXPAND_STAR_MEMBER(z, int) \
  EXPAND_STAR_MEMBER(radius, double)
```

and then all calls to `#include "star.def"` could be replaced with a simple `EXPAND_STAR` statement. The rest of the above file would become:

```
typedef struct {
  #define EXPAND_STAR_MEMBER(member, type) type member;
  EXPAND_STAR
  #undef  EXPAND_STAR_MEMBER
  } starStruct;

void serialize_star(const starStruct *const star, unsigned char *buffer) {
  #define EXPAND_STAR_MEMBER(member, type) \
    memcpy(buffer, &(star->member), sizeof(star->member)); \
    buffer += sizeof(star->member);
  EXPAND_STAR
  #undef  EXPAND_STAR_MEMBER
  }

void deserialize_star(starStruct *const star, const unsigned char *buffer) {
  #define EXPAND_STAR_MEMBER(member, type) \
    memcpy(&(star->member), buffer, sizeof(star->member)); \
    buffer += sizeof(star->member);
  EXPAND_STAR
  #undef  EXPAND_STAR_MEMBER
  }
```

and the print handler could be added as well as:

```
#define print_int(val)    printf("%d", val)
#define print_double(val) printf("%g", val)
```

```
void print_star(const starStruct *const star) {
  /* print_##type will be replaced with print_int or print_double */
  #define EXPAND_STAR_MEMBER(member, type) \
    printf("%s: ", #member); \
    print_##type(star->member); \
    printf("\n");
  EXPAND_STAR
  #undef EXPAND_STAR_MEMBER
}
```

or as:

```
#define FORMAT_(type) FORMAT_##type
#define FORMAT_int    "%d"
#define FORMAT_double "%g"

void print_star(const starStruct *const star) {
  /* FORMAT_(type) will be replaced with FORMAT_int or FORMAT_double */
  #define EXPAND_STAR_MEMBER(member, type) \
    printf("%s: " FORMAT_(type) "\n", #member, star->member);
  EXPAND_STAR
  #undef EXPAND_STAR_MEMBER
  }
```

A variant which avoids needing to know the members of any expanded sub-macros is to accept the operators as an argument to the list macro:

File **star_table.c**:

```
/*
 Generic
 */
#define STRUCT_MEMBER(member, type, dummy) type member;

#define SERIALIZE_MEMBER(member, type, obj, buffer) \
  memcpy(buffer, &(obj->member), sizeof(obj->member)); \
  buffer += sizeof(obj->member);

#define DESERIALIZE_MEMBER(member, type, obj, buffer) \
  memcpy(&(obj->member), buffer, sizeof(obj->member)); \
  buffer += sizeof(obj->member);

#define FORMAT_(type) FORMAT_##type
#define FORMAT_int    "%d"
#define FORMAT_double "%g"

/* FORMAT_(type) will be replaced with FORMAT_int or FORMAT_double */
#define PRINT_MEMBER(member, type, obj) \
  printf("%s: " FORMAT_(type) "\n", #member, obj->member);

/*
```

```
 starStruct
 */

#define EXPAND_STAR(_, ...) \
  _(x, int, __VA_ARGS__) \
  _(y, int, __VA_ARGS__) \
  _(z, int, __VA_ARGS__) \
  _(radius, double, __VA_ARGS__)

typedef struct {
  EXPAND_STAR(STRUCT_MEMBER, )
  } starStruct;

void serialize_star(const starStruct *const star, unsigned char *buffer) {
  EXPAND_STAR(SERIALIZE_MEMBER, star, buffer)
  }

void deserialize_star(starStruct *const star, const unsigned char *buffer) {
  EXPAND_STAR(DESERIALIZE_MEMBER, star, buffer)
  }

void print_star(const starStruct *const star) {
  EXPAND_STAR(PRINT_MEMBER, star)
  }
```

This approach can be dangerous in that the entire macro set is always interpreted as if it was on a single source line, which could encounter compiler limits with complex component macros and/or long member lists.

This technique was reported by Lars Wirzenius[13] in a web page dated January 17, 2000, in which he gives credit to Kenneth Oksanen for "refining and developing" the technique prior to 1997. The other references describe it as a method from at least a decade before the turn of the century.

We discuss X-Macros more in a later section, Serialization and X-Macros.

1. Understanding C++/C Preprocessor
2. David Hart, Jon Reid. "9 Code Smells of Preprocessor Use" (http://qualitycoding.org/preprocessor/). 2012.
3. HP C Compiler Reference Manual (http://docs.hp.com/en/B3901-90003/ch07s04.html)
4. C++ reference: Predefined preprocessor variables (http://www.cppreference.com/wiki/preprocessor/preprocessor_vars)
5. "Compile Time Assertions in C" (http://www.jaggersoft.com/pubs/CVu11_3.html) by Jon Jagger 1999
6. Pádraig Brady. "static assertion" (http://www.pixelbeat.org/programming/gcc/static_assert.html).
7. "ternary operator with a constant (true) value?" (http://stackoverflow.com/questions/27586311/ternary-operator-with-a-constant-true-value).
8. Wikipedia: C++0x#Static assertions
9. Wirzenius, Lars. C Preprocessor Trick For Implementing Similar Data Types (http://liw.iki.fi/liw/texts/cpp-trick.html) Retrieved January 9, 2011.
10. Meyers, Randy (May 2001). "The New C: X Macros". *Dr. Dobb's Journal*. http://www.ddj.com/cpp/184401387. Retrieved 1 May 2008.
11. Beal, Stephan (August 2004). *Supermacros*. http://wanderinghorse.net/computing/papers/#supermacros. Retrieved 27 October 2008.

12. Keith Schwarz. "Advanced Preprocessor Techniques" (http://www.keithschwarz.com/cs106l/spring2009/handouts/080_Preprocessor_2.pdf). 2009. Includes "Practical Applications of the Preprocessor II: The X Macro Trick".
13. Wirzenius, Lars. C Preprocessor Trick For Implementing Similar Data Types (http://liw.iki.fi/liw/texts/cpp-trick.html) Retrieved January 9, 2011.

Libraries

A *library* in C is a group of functions and declarations, exposed for use by other programs. The library therefore consists of an *interface* expressed in a `.h` file (named the "header") and an *implementation* expressed in a `.c` file. This `.c` file might be precompiled or otherwise inaccessible, or it might be available to the programmer. (Note: Libraries may call functions in other libraries such as the Standard C or math libraries to do various tasks.)

The format of a library varies with the operating system and compiler one is using. For example, in the Unix and Linux operating systems, a library consists of one or more *object files*, which consist of object code that is usually the output of a compiler (if the source language is C or something similar) or an assembler (if the source language is assembly language). These object files are then turned into a library in the form of an archive by the *ar* archiver (a program that takes files and stores them in a bigger file without regard to compression). The filename for the library usually starts with "lib" and ends with ".a"; e.g. the *libc.a* file contains the Standard C library and the "libm.a" the mathematics routines, which the linker would then link in. Other operating systems such as Microsoft Windows use a ".lib" extension for libraries and an ".obj" extension for object files. Some programs in the Unix environment such as lex and yacc generate C code that can be linked with the libl and liby libraries to create an executable.

We're going to use as an example a library that contains one function: 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 (http://en.wikipedia.org/wiki/Concatenate) to the letter:

```
-ioptarg
```

or have the argument in a separate argv-element:

```
-i optarg
```

The library also has four declarations that it exports in addition to the function: three integers and a pointer to the optional argument. If the argument does not have an optional argument, the pointer to the optional argument will be null.

In order to parse all these types of arguments, we have written the following "getopt.c" file:

```
#include <stdio.h>              /* for fprintf() and EOF */
#include <string.h>             /* for strchr() */
#include "getopt.h"             /* consistency check */

/* 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 interface would be the following "getopt.h" file:

```
#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 */
```

At a minimum, a programmer has the interface file to figure out how to use a library, although, in general, the library programmer also wrote documentation on how to use the library. In the above case, the documentation should say that the provided arguments `**argv` and `*opts` both

shouldn't be null pointers (or why would you be using the `getopt` function anyway?). Specifically, it typically states what each parameter is for and what return values can be expected in which conditions. Programmers that use a library, are normally not interested in the implementation of the library -- unless the implementation has a bug, in which case he would want to complain somehow.

Both the implementation of the getopts library, and programs that use the library should state `#include "getopt.h"`, in order to refer to the corresponding interface. Now the library is "linked" to the program -- the one that contains the main() function. The program may refer to dozens of interfaces.

In some cases, just placing `#include "getopt.h"` may appear correct but will still fail to link properly. This indicates that the library is not installed correctly, or there may be some additional configuration required. You will have to check either the compiler's documentation or library's documentation on how to resolve this issue.

What to put in header files

As a general rule, headers contain anything that should be exported, or "seen" by the other modules in a program. This includes macro definitions (preprocessor `#define`s); structure, union, and enumeration declarations; typedef declarations; external function declarations; and global variable declarations. In the above `getopt.h` example file, one function (`getopt`) is declared and and four global variables (`optind`, `optopt`, `optarg`, and `opterr`) are also declared. The variables are declared with the storage class specifier `extern` in the header file because that keyword specifies that the "real" variables are stored elsewhere (i.e. the `getopt.c` file) and not within the header file.

The `#ifndef GETOPT_H/#define GETOPT_H` trick is colloquially called **include guards**. This is used so that if the `getopt.h` file were included more than once in a translation unit, the unit would only see the contents once.

Further reading

- C FAQ: "I'm wondering what to put in .c files and what to put in .h files. (What does ".h" mean, anyway?)" (http://c-faq.com/cpp/hfiles.html)
- PIClist thread: "Global variables in projects with many C files." (http://www.piclist.com/techref/postbot.asp?by=time&id=piclist\2007\10\25\073430a&tgt=post)
- "How do I use extern to share variables between source files in C?" (http://stackoverflow.com/questions/1433204/how-do-i-use-extern-to-share-variables-between-source-files-in-c).

Standard libraries

The **C standard library** is a standardized collection of header files and library routines used to implement common operations, such as input/output and character string handling. Unlike other languages (such as COBOL, Fortran, and PL/I) C does not include builtin keywords for these tasks, so

nearly all C programs rely on the standard library to operate.

History

The C programming language previously did not provide any elementary functions, such as I/O operations. Over time, user communities of C shared ideas and implementations to provide those functions. These ideas became common, and were eventually incorporated into the definition of the standardized C programming language. These are now called the **C standard libraries**.

Both Unix and C were created at AT&T's Bell Laboratories in the late 1960s and early 1970s. During the 1970s the C programming language became increasingly popular, with many universities and organizations beginning to create their own variations of the language for their own projects. By the start of the 1980s compatibility problems between the various C implementations became apparent. In 1983 the American National Standards Institute (ANSI) formed a committee to establish a standard specification of C known as "ANSI C". This work culminated in the creation of the so-called **C89** standard in 1989. Part of the resulting standard was a set of software libraries called the **ANSI C standard library**.

Later revisions of the C standard have added several new required header files to the library. Support for these new extensions varies between implementations.

The headers **<iso646.h>**, **<wchar.h>**, and **<wctype.h>** were added with Normative Addendum 1 (hereafter abbreviated as **NA1**), an addition to the C Standard ratified in 1995.

The headers **<complex.h>**, **<fenv.h>**, **<inttypes.h>**, **<stdbool.h>**, **<stdint.h>**, and **<tgmath.h>** were added with **C99**, a revision to the C Standard published in 1999.

Design

The declaration of each function is kept in a header file, while the actual implementation of functions are separated into a library file. The naming and scope of headers have become common but the organization of libraries still remains diverse. The standard library is usually shipped along with a compiler. Since C compilers often provide extra functions that are not specified in ANSI C, a standard library with a particular compiler is mostly incompatible with standard libraries of other compilers.

Much of the C standard library has been shown to have been well-designed. A few parts, with the benefit of hindsight, are regarded as mistakes. The string input functions `gets()` (and the use of `scanf()` to read string input) are the source of many buffer overflows, and most programming guides recommend avoiding this usage. Another oddity is `strtok()`, a function that is designed as a primitive lexical analyser but is highly "fragile" and difficult to use.

ANSI Standard

The ANSI C standard library consists of 24 C header files which can be included into a programmer's project with a single directive. Each header file contains one or more function declarations, data type definitions and macros. The contents of these header files follows.

In comparison to some other languages (for example Java) the standard library is minuscule. The library provides a basic set of mathematical functions, string manipulation, type conversions, and file and console-based I/O. It does not include a standard set of "container types" like the C++ Standard Template Library, let alone the complete graphical user interface (GUI) toolkits, networking tools, and profusion of other functions that Java provides as standard. The main advantage of the small standard library is that providing a working ANSI C environment is much easier than it is with other languages, and consequently porting C to a new platform is relatively easy.

Many other libraries have been developed to supply equivalent functions to that provided by other languages in their standard library. For instance, the GNOME desktop environment project has developed the GTK+ graphics toolkit and GLib, a library of container data structures, and there are many other well-known examples. The variety of libraries available has meant that some superior toolkits have proven themselves through history. The considerable downside is that they often do not work particularly well together, programmers are often familiar with different sets of libraries, and a different set of them may be available on any particular platform.

ANSI C library header files

<assert.h>	Contains the assert macro, used to assist with detecting logical errors and other types of bug in debugging versions of a program.
<complex.h>	A set of functions for manipulating complex numbers. (New with **C99**)
<ctype.h>	This header file contains functions used to classify characters by their types or to convert between upper and lower case in a way that is independent of the used character set (typically ASCII or one of its extensions, although implementations utilizing EBCDIC are also known).
<errno.h>	For testing error codes reported by library functions.
<fenv.h>	For controlling floating-point environment. (New with **C99**)
<float.h>	Contains defined constants specifying the implementation-specific properties of the floating-point library, such as the minimum difference between two different floating-point numbers (_EPSILON), the maximum number of digits of accuracy (_DIG) and the range of numbers which can be represented (_MIN, _MAX).
<inttypes.h>	For precise conversion between integer types. (New with **C99**)
<iso646.h>	For programming in ISO 646 variant character sets. (New with **NA1**)
<limits.h>	Contains defined constants specifying the implementation-specific properties of the integer types, such as the range of numbers which can be represented (_MIN, _MAX).
<locale.h>	For setlocale() and related constants. This is used to choose an appropriate locale.
<math.h>	For computing common mathematical functions -- see Further math or C++ Programming/Code/Standard C Library/Math for details.
<setjmp.h>	setjmp and longjmp, which are used for non-local exits
<signal.h>	For controlling various exceptional conditions
<stdarg.h>	For accessing a varying number of arguments passed to functions.
<stdbool.h>	For a boolean data type. (New with **C99**)
<stdint.h>	For defining various integer types. (New with **C99**)
<stddef.h>	For defining several useful types and macros.
<stdio.h>	Provides the core input and output capabilities of the C language. This file includes the venerable `printf` function.
<stdlib.h>	For performing a variety of operations, including conversion, pseudo-random numbers, memory allocation, process control, environment, signalling, searching, and sorting.
<string.h>	For manipulating several kinds of strings.
<tgmath.h>	For type-generic mathematical functions. (New with **C99**)

<time.h>	For converting between various time and date formats.
<wchar.h>	For manipulating wide streams and several kinds of strings using wide characters - key to supporting a range of languages. (New with **NA1**)
<wctype.h>	For classifying wide characters. (New with **NA1**)

Common support libraries

While not standardized, C programs may depend on a runtime library of routines which contain code the compiler uses at runtime. The code that initializes the process for the operating system, for example, before calling `main()`, is implemented in the C Run-Time Library for a given vendor's compiler. The Run-Time Library code might help with other language feature implementations, like handling uncaught exceptions or implementing floating point code.

The C standard library only documents that the specific routines mentioned in this article are available, and how they behave. Because the compiler implementation might depend on these additional implementation-level functions to be available, it is likely the vendor-specific routines are packaged with the C Standard Library in the same module, because they're both likely to be needed by any program built with their toolset.

Though often confused with the C Standard Library because of this packaging, the C Runtime Library is not a standardized part of the language and is vendor-specific.

Compiler built-in functions

Some compilers (for example, GCC) provide built-in versions of many of the functions in the C standard library; that is, the implementations of the functions are written into the compiled object file, and the program calls the built-in versions instead of the functions in the C library shared object file. This reduces function call overhead, especially if function calls are replaced with inline variants, and allows other forms of optimization (as the compiler knows the control-flow characteristics of the built-in variants), but may cause confusion when debugging (for example, the built-in versions cannot be replaced with instrumented variants).

POSIX standard library

POSIX, (along with the Single Unix Specification), specifies a number of routines that should be available over and above those in the C standard library proper; these are often implemented alongside the C standard library functions, with varying degrees of closeness. For example, glibc implements functions such as fork within libc.so, but before NPTL was merged into glibc it constituted a separate library with its own linker flag. Often, this POSIX-specified function will be regarded as part of the library; the C library proper may be identified as the ANSI or ISO C library.

The following libraries are recognized by POSIX:

c	This option shall make available all interfaces referenced in the System Interfaces volume of POSIX.1-2008, with the possible exception of those interfaces listed as residing in <aio.h>, <arpa/inet.h>, <complex.h>, <fenv.h>, <math.h>, <mqueue.h>, <netdb.h>, <net/if.h>, <netinet/in.h>, <pthread.h>, <sched.h>, <semaphore.h>, <spawn.h>, <sys/socket.h>, pthread_kill(), and pthread_sigmask() in <signal.h>, <trace.h>, interfaces marked as optional in <sys/mman.h>, interfaces marked as ADV (Advisory Information) in <fcntl.h>, and interfaces beginning with the prefix clock_ or time_ in <time.h>. This option shall not be required to be present to cause a search of this library.
l	This option shall make available all interfaces required by the C-language output of lex that are not made available through the -l c option. (The flex program, a clone of lex, uses fl instead of l.)
pthread	This option shall make available all interfaces referenced in <pthread.h> and pthread_kill() and pthread_sigmask() referenced in <signal.h>. An implementation may search this library in the absence of this option.
m	This option shall make available all interfaces referenced in <math.h>, <complex.h>, and <fenv.h>. An implementation may search this library in the absence of this option.
rt	This option shall make available all interfaces referenced in <aio.h>, <mqueue.h>, <sched.h>, <semaphore.h>, and <spawn.h>, interfaces marked as optional in <sys/mman.h>, interfaces marked as ADV (Advisory Information) in <fcntl.h>, and interfaces beginning with the prefix clock_ and time_ in <time.h>. An implementation may search this library in the absence of this option.
trace	This option shall make available all interfaces referenced in <trace.h>. An implementation may search this library in the absence of this option.
xnet	This option shall make available all interfaces referenced in <arpa/inet.h>, <netdb.h>, <net/if.h>, <netinet/in.h>, and <sys/socket.h>. An implementation may search this library in the absence of this option.
y	This option shall make available all interfaces required by the C-language output of yacc that are not made available through the -l c option. (Some clones of yacc, including bison and byacc, include the entire library in the generated file, so it is not necessary to use -l y.)

File IO

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 consists of one or more lines. A line in a text stream consists of zero or more characters plus a terminating new-line character. (The only exception is that in some implementations the last line of a file does not require a terminating new-line character.) 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. (This is the most basic convention, and if it doesn't meet the needs of a text-oriented peripheral

attached to a Unix machine, then the fix-up occurs out at the edges of the system. Nothing in between needs to change.) The string of characters that go into, or come out of a text stream may have to be modified to conform to specific conventions. This results in a possible difference between the data that go into a text stream and the data that come out. For instance, in some implementations when a space-character precedes a new-line character in the input, the space character gets removed out of the output. In general, when the data only consists of printable characters and control characters like horizontal tab and new-line, the input and output of a text stream are equal.

Compared to a text stream, a binary stream is pretty straight forward. A binary stream is an ordered sequence of characters that can transparently record internal data. Data written to a binary stream shall always equal the data that gets read out under the same implementation. Binary streams, however, may have an implementation-defined number of null characters appended to the end of the stream. There are no further conventions which need to be considered.

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.

Standard Streams

When a C program starts its execution the program automatically opens three standard streams named `stdin`, `stdout`, and `stderr`. These are attached for every C program.

The first standard stream is used for input buffering and the other two are used for output. These streams are sequences of bytes.

Consider the following program:

```
/* An example program. */
int main()
{
    int var;
    scanf ("%d", &var); /* use stdin for scanning an integer from keyboard. */
    printf ("%d", var); /* use stdout for printing a character. */
    return 0;
}
/* end program. */
```

By default `stdin` points to the keyboard and `stdout` and `stderr` point to the screen. It is possible under Unix and may be possible under other operating systems to redirect input from or output to a file or both.

`FILE` pointers

The `<stdio.h>` header contains a definition for a type `FILE` (usually via a `typedef`) which is capable of processing all the information needed to

exercise control over 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. Better access to the contents of `FILE` is 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);
```

`fopen` and `freopen` opens the file whose name is in the string pointed to by `filename` and associates a stream with it. Both return a pointer to the object controlling the stream, or if the open operation fails a null pointer. The error and end-of-file indicators are cleared, and if the open operation fails error is set. `freopen` differs from `fopen` in that the file pointed to by `stream` is closed first when already open and any close errors are ignored.

`mode` for both functions points to a string consisting of 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.

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. The behavior of fflush is undefined for input stream.

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 pointer is returned.

Warning: Different operating systems may use different character sequences to represent the end-of-line sequence. For example, some filesystems

use the terminator `\r\n` in text files; `fgets` may 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 (unless the programmer doesn't care about it). Unixes typically use `\n` as its end-of-line sequence, MS-DOS and Windows uses `\r\n`, and Mac OSes used `\r` before OS X.

```
/* A example program that reads from stdin and writes to stdout */
#include <stdio.h>

#define BUFFER_SIZE 100

int main(void)
{
    char buffer[BUFFER_SIZE]; /* a read buffer */
    while( fgets (buffer, BUFFER_SIZE, stdin) != NULL)
    {
         printf("%s",buffer);
    }
    return 0;
}
/* end program. */
```

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.

EOF pitfall

A mistake when using `fgetc`, `getc`, or `getchar` is to assign the result to a variable of type `char` *before* comparing it to `EOF`. The following code fragments exhibit this mistake, and then show the correct approach (using type int):

Mistake

```
char c;
while ((c = getchar()) != EOF)
    putchar(c);
```

Correction

```
int c;
while ((c = getchar()) != EOF)
    putchar(c);
```

Consider a system in which the type `char` is 8 bits wide, representing 256 different values. `getchar` may return any of the 256 possible characters, and it also may return `EOF` to indicate end-of-file, for a total of 257 different possible return values.

When `getchar`'s result is assigned to a `char`, which can represent only 256 different values, there is necessarily some loss of information—when packing 257 items into 256 slots, there must be a collision. The `EOF` value, when converted to `char`, becomes indistinguishable from whichever one of the 256 characters shares its numerical value. If that character is found in the file, the above example may mistake it for an end-of-file indicator; or, just as bad, if type `char` is unsigned, then because `EOF` is negative, it can never be equal to any unsigned `char`, so the above example will not terminate at end-of-file. It will loop forever, repeatedly printing the character which results from converting `EOF` to `char`.

However, this looping failure mode does not occur if the char definition is signed (C makes the signedness of the default char type implementation-dependent),[1] assuming the commonly used `EOF` value of -1. However, the fundamental issue remains that if the `EOF` value is defined outside of the range of the `char` type, when assigned to a `char` that value is sliced and will no longer match the full `EOF` value necessary to exit the loop. On the other hand, if `EOF` is within range of `char`, this guarantees a collision between `EOF` and a char value. Thus, regardless of how system types are defined,

never use `char` types when testing against `EOF`.

On systems where `int` and `char` are the same size (i.e., systems incompatible with minimally the POSIX and C99 standards), even the "good" example will suffer from the indistinguishability of `EOF` and some character's value. The proper way to handle this situation is to check `feof` and `ferror` after `getchar` returns `EOF`. If `feof` indicates that end-of-file has not been reached, and `ferror` indicates that no errors have occurred, then the `EOF` returned by `getchar` can be assumed to represent an actual character. These extra checks are rarely done, because most programmers assume that their code will never need to run on one of these "big `char`" systems. Another way is to use a compile-time assertion to make sure that `UINT_MAX > UCHAR_MAX`, which at least prevents a program with such an assumption from compiling in such a system.

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` function 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. Floating-point numbers are *rounded* to fit the precision; i.e. `printf("%1.1f\n", 1.19);` produces `1.2`.
- 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.

References

1. C99 §6.2.5/15

Beginning exercises

Variables

Naming

1. Can a variable name start with a number?
2. Can a variable name start with a typographical symbol(e.g. #, *, _)?
3. Give an example of a C variable name that would *not* work. Why doesn't it work?

Solution

1. No, the name of a variable must begin with a letter (lowercase or uppercase), or an underscore.
2. Only the underscore can be used.
3. for example, **#nm*rt** is not allowed because # and * are not the valid characters for the name of a variable.

```
#include <stdio.h>
main()
{
    int a, b, c, max;
    clrscr();
    printf("\n enter three numbers ");
    scanf(" %d %d %d ",&a,&b,&c);
    max = a;
    if(max < b)
        max = b;
    if(max < c)
        max = c;
    printf("\n largest=%d \n",max);
    getch();
}
```

Data Types

1. List at least three data types in C
    1. On your computer, how much memory does each require?
    2. Which ones can be used in place of another? Why?
        1. Are there any limitations on these uses?
        2. If so, what are they?
        3. Is it necessary to do anything special to use the alternative?
2. Can the name we use for a data type (e.g. 'int', 'float') be used as a variable?

Solution

- 3 data types : **long int**, **short int**,**float**.
- On my computer :
  - long int : 4 bytes
  - short int : 2 bytes
  - float : 4 bytes
- we can not use 'int' or 'float' as a variable's name.

Assignment

1. How would you assign the value 3.14 to a variable called pi?
2. Is it possible to assign an *int* to a *double*?
    1. Is the reverse possible?

Solution

- The standard way of assigning 3.14 to pi is:

```
double pi;
pi = 3.14;
```

- - Since pi is a constant, good programming convention dictates to make it unchangeable during runtime. Extra credit if you use one of the following two lines:

```
const float pi = 3.14;
#define pi 3.14
```

- Yes, for example :

```
int a = 67;
double b;
b = a;
```

- Yes, but a cast is necessary and the double is truncated:

```
double a=89;
int b;
b = (int) a;
```

Referencing

1. A common mistake for new students is reversing the assignment statement. Suppose you want to assign the value stored in the variable "pi" to another variable, say "pi2":
    1. What is the correct statement?
    2. What is the reverse? Is this a valid C statement (even if it gives incorrect results)?
    3. What if you wanted to assign a constant value (like 3.1415) to "pi2":
        a. What would the correct statement look like?
        b. Would the reverse be a valid or invalid C statement?

Solution

1. `pi2 = pi;`
2. The reverse, `pi = pi2;` is a valid C statement if `pi` is not a constant and `pi2` is initialized.
3. **a**. `pi2 = 3.1415;`
   b. The reverse: `3.1415 = pi2;` is not valid since it is impossible to assign a value to a literal.

Simple I/O

```
#include<stdio.h>
int main(void) {
  int a,b,c;
  a=2058;
  b=270;
  c=a^b;
  printf("%d\n",c);
}
```

String manipulation

1. Write a program that prompts the user for a string, and prints its reverse.

Solution

One possible solution could be:

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

int main(void)
{
    char s[81]; // A string of upto 80 chars + '\0'
    int i;

    puts("Please write a string: ");
    fgets(s, 81, stdin);

    puts("Your sentence in reverse: ");
    for (i= strlen(s)-1; i >= 0; i--)
    {
        if (s[i] == '\n')
            continue; // don't write newline
        else
            putchar(s[i]);
    }
    putchar('\n');
    return 0;
}
```

2. Write a program that prompts the user for a sentence, and prints each word on its own line.

Solution

One possible solution could be:

```c
#define __STDC_WANT_LIB_EXT1__ 1 // Active C11 extended functions (this case is gets_s)
#include<stdio.h>

int slen(const char *); // I don't want to include whole string lib just for get size

int main(void) {
  char s[500];
  printf("%s","Enter a sentence: ");
  if(!gets_s(s,500)) {
    puts("Error!");
    getchar();
    return 0;
  }
  for(int i=0;i<slen(s);i++)
    if(s[i]!=32)
      putchar(s[i]);
    else
```

```
      putchar(10);
  putchar(10);
  getchar();
  return 0;
}

int slen(const char *s) {
  int i;
  for(i=0;s[i]!=0;i++);
  return i;
}
```

Loops

1. Write a function that outputs a right isosceles triangle of height and width *n*, so *n = 3* would look like

```
*
**
***
```

Solution

One possible solution:

```
#include<stdio.h>

int main(void) {
  int n;
  scanf("%d",&n);
  for(int i=0;i<n;i++) {
    for(int j=0;j<=i;j++)
      putchar(42);
    putchar(10);
  }
  while((n=getchar())!=10);
  getchar();
  return 0;
}
```

2. Write a function that outputs a sideways triangle of height *2n-1* and width *n*, so the output for *n = 4* would be:

```
*
**
```

```
***
****
***
**
*
```

Solution

One possible solution:

```c
#include<stdio.h>
// This is the fastest way
int main(void) {
  int n;
  scanf("%d",&n);
  for(int i=0;i<n;i++) {
    for(int j=0;j<=i;j++)
      putchar(42);
    putchar(10);
  }
  for(int i=n-1;i>0;i--) {
    for(int j=0;j<i;j++)
      putchar(42);
    putchar(10);
  }
  while((n=getchar())!=10);
  getchar();
  return 0;
}
```

or like this (all math)

```c
void sideways(int n)
{
    int i=0,j=0;
    for(i=1;i<2*n;i++){
        for(j=1;j<=(n-(abs(n-i)));j++){
        printf("*");
    }
    printf("\n");
    }
}
```

3. Write a function that outputs a right-side-up triangle of height *n* and width *2n-1*; the output for *n = 6* would be:

```
     *
```

```
    ***
   *****
  *******
 *********
***********
```

Solution

One possible solution:

```
void right_side_up(int n)
{
    int x,y;
    for (y= 1; y <= n; y++)
    {
        for (x= 0; x < n-y; x++)
            putchar(' ');
        for (x= (n-y); x < (n-y)+(2*y-1); x++)
            putchar('*');
        putchar('\n');
    }
}
```

Another solution:

```
#include<stdio.h>

int main(void) {
  int n;
  scanf("%d",&n);
  for(int i=0;i<n;i++) {
    for(int j=0;j<n-i-1;j++)
      putchar(32);
    for(int j=0;j<i*2+1;j++)
      putchar(42);
    putchar(10);
  }
  while((n=getchar())!=10);
  getchar();
  return 0;
}
```

Program Flow

1. Build a program where control passes from main to four different functions with 4 calls.

2. Now make a while loop in main with the function calls inside it. Ask for input at the beginning of the loop. End the while loop if the user hits Q

3. Next add conditionals to call the functions when the user enters numbers, so 1 goes to function1, 2 goes to function 2, etc.

4. Have function 1 call function a, which calls function b, which calls function c

5. Draw out a diagram of program flow, with arrows to indicate where control goes

Functions

1. Write a function to check if an integer is negative; the declaration should look like bool is_positive(int i);

2. Write a function to raise a floating point number to an integer power, so for example to when you use it

float a = raise_to_power(2, 3); //a gets 8

float b = raise_to_power(9, 2); //b gets 81

float raise_to_power(float f, int power); //make this your declaration

Math

1. Write a function to calculate if a number is prime. Return 1 if it is prime and 0 if it is not a prime.

Solution

One possible solution using a naïve primality test:

```
// to compile: gcc -Wall prime.c -lm -o prime

#include <math.h>    // for the square root function sqrt()
#include <stdio.h>

int is_prime(int n);

int main()
{
  printf("Write an integer: ");
  int var;
  scanf("%d", &var);
  if (is_prime(var)==1) {
```

```
    printf("A prime\n");
  } else {
    printf("Not a prime\n");
  }
  return 1;
}

int is_prime(int n)
{
  int x;
  int sq= sqrt(n)+1;

  // Checking the trivial cases first
  if ( n < 2 )
    return 0;
  if ( n == 2 || n == 3 )
    return 1;

  // Checking if n is divisible by 2 or odd numbers between 3 and the
  // square root of n.
  if ( n % 2 == 0 )
    return 0;
  for (x= 3; x <= sq; x += 2)
    {
      if ( n % x == 0 )
    return 0;
    }

  return 1;
}
```

Another better solution that doesn't need to include math.h and faster than the one above.

```
#include<stdio.h>

int isPrime(const unsigned long long int);

int main(void) {
  unsigned long long n;
  scanf("%llu",&n);
  printf("%d\n",isPrime(n));
  while((n=getchar())!=10);
  getchar();
  return 0;
}

int isPrime(const unsigned long long int x) {
  if(x<4)
    return x>1;
  if(x%2==0||x%3==0)
    return 0;
  for(unsigned long int i=5;i*i<=x;i+=6)
    if(x%i==0||x%(i+2)==0)
      return 0;
```

```
  return 1;
}
```

2. Write a function to determine the number of prime numbers below n.

3. Write a function to find the square root by using Newton's method.

4. Write functions to evaluate the trigonometric functions:

5. Try to write a random number generator.

6. Write a function to determine the prime number between 2 and 100:

Recursion

Merge sort

1. Write a C program to generate a random integer array with a given length n , and sort it recursively using the Merge sort algorithm.

- The merge sort algorithm is a recursive algorithm .

- sorting a one element array is easy.

- sorting two one-element arrays, requires the merge operation. The merge operation looks at two sorted arrays as lists, and compares the head of the list , and which ever head is smaller, this element is put on the sorted list and the head of that list is ticked off, so the next element becomes the head of that list. This is done until one of the lists is exhausted, and the other list is then copied onto the end of the sorted list.

- the recursion occurs, because merging two one-element arrays produces one two-element sorted array, which can be merged with another two-element sorted array produced the same way. This produces a sorted 4 element array, and the same applies for another 4 element sorted array.

- so the basic merge sort, is to check the size of list to be sorted, and if it is greater than one, divide the array into two, and call merge sort again on the two halves. After wards, merge the two halves in a temporary space of equal size, and then copy back the final sorted array onto the original array.

Solution

One possible solution , after reading online descriptions of recursive merge sort, e.g. Dasgupta :

```
// to compile: gcc -Wall rmergesort.c -lm -o rmergesort
```

```
/*
 ============================================================================
 Name        : rmergesort.c
 Author      : Anon
 Version     : 0.1
 Copyright   : (C)2013 under CC-By-SA 3.0 License
 Description : Recursive Merge Sort, Ansi-style
 ============================================================================
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//const int MAX = 200;
const int MAX = 20000000;

int *b;

int printOff = 0;

// this debugging print out of the array helps to show
// what is going on.
void printArray(char* label, int* a, int sz) {
    int h = sz/ 2;
    int i;

    if (printOff) return;

    printf("\n%s:\n", label);

    for (i = 0; i < h; ++i ) {

        printf("%d%c", a[i],
                    // add in a newline every 20 numbers
                    ( ( i != 0 && i % 20 == 0 )? '\n': ' ' ) );
    }

    printf(" | ");
    for (;i < sz; ++i) {
        printf("%d%c", a[i],
                    ( ( i != 0 && i % 20 == 0 )? '\n': ' ' ) );
    }

    putchar('\n');


}

void mergesort(int* a, int m ) {

    printArray("BEFORE", a, m);

    if (m > 2) {
        // if greater than 2 elements, then recursive
        mergesort(a, m / 2);
        mergesort(a + m / 2, m - m / 2);
```

```
    } else if (m == 2 && a[0] > a[1]) {
        // if exactly 2 elements and need swapping, swap
        int t = a[1];
        a[1] = a[0];
        a[0] = t;
        goto end;

    }

    // 1 or greater than 2 elements which have been recursively sorted

    // divide the array into a left and right array
    // merge into the array b with index l.

    int n = m/2;
    int o = m - n;

    int i = 0, j = n;
    int l = 0;
    // i is left, j is right ;
    // l should equal m the size of the array
    while (i < n) {
        if ( j >= m) {
            // the right array has finished, so copy the remaining left array
            for(; i < n; ++i) {
                b[l++] = a[i];
            }

        // the merge operation is to copy the smaller current element and
        // increment the index of the parent sub-array.
        } else if(  a[i] < a[j] ) {
            b[l++] = a[i++];
        } else {
            b[l++] = a[j++];
        }
    }

    while ( j < m) {
        // copy the remaining right array , if any
        b[l++] = a[j++];
    }

    memcpy(a, b, sizeof(int) * l );

end:
    printArray("AFTER", a, m);

    return;

}

void rand_init(int* a, int n) {
    int i;
    for (i = 0; i < n; ++i ) {
```

```
        a[i] = rand() % MAX;

    }
}

int main(void) {
    puts("!!!Hello World!!!"); /* prints !!!Hello World!!! */

//  int N = 20;
//    int N = 1000;
//    int N = 1000000;
    int N = 100000000;  // still can't make a stack overflow on ubuntu,4GB, phenom
    printOff = 1;

    int *a;

    b = calloc( N, sizeof(int));

    a = calloc( N, sizeof(int));

    rand_init(a, N);

    mergesort(a, N);

    printOff = 0;

    printArray("LAST", a, N);

    free(a);
    free(b);

    return EXIT_SUCCESS;
}


/* Having failed to translate my concept of non-recursive merge sort,
 * I tackled the easier case of recursive merge sort.
 * The next task is to translate the recursive version to a non-recursive
 * version. This could be done by replacing calls to mergesort, with
 * pushes onto a stack of
 * tuples of ( <array start address>, <number of elements to process> )
 */

/* The central idea of merging, is that two sorted lists can be
 * merged into one sorted list, by comparing the top of each list and
 * moving the lowest valued element onto the end of the new list.
 *  The other list which has the higher valued element keeps its top
 *  element unchanged. When a list is exhausted, copy the remaining other list
 *  onto the end of the new list.
 */

/* The recursive part, is to defer any work in sorting an unsorted list,
 * by dividing it into two lists until there is only 1 or two elements,
 * and if there are two elements, sort them directly by swapping if
 * the first element is larger than the second element.
 *
 * After returning from a recursive call, merge the lists, which will
```

```
 * begin with one or two element sorted lists. The result is a sorted list
 * which will be returned to the parent of the recursive call, and can
 * be used for merging.
 */

/* The following is an imaginary discussion about what a programmer
 * might be thinking about when programming:
 *
 * Visualising recursion in terms of a Z80 assembly language, which
 * is similiar to most assembly languages, there is a data stack (DS) and
 * a call stack (CS) pointer, and each recursive call to mergesort
 * pushes the return address , which is the program address of the instruction
 * after the call , onto the stack pointed to by CS and CS is incremented,
 * and the address of the array start and integer which is the subarray length
 * onto the data stack pointed to by DS, which will be incremented twice.
 *
 * If the number of recursive , active calls exceed the allowable space for either the call stack
 * or the data stack, then the program will crash , or a process space protection
 * violation interrupt signal will be sent by the CPU, and the interrupt vector
 * for that signal will jump the processor's current instruction pointer to the
 * interrupt handling routine.
 */
```

Binary heaps

2. **Binary heaps** :

- A binary max-heap or min-heap, is an ordered structure where some nodes are guaranteed greater than other nodes, e.g. the parent vs two children. A binary heap can be stored in an array , where ,

- given a position **i** (the parent) , **i*2** is the left child, and **i*2+1** is the right child.

- (C arrays begin at position 0, but 0 * 2 = 0, and 0 *2 + 1= 1, which is incorrect , so start the heap at position 1, or add 1 for parent-to-child calculations, and subtract 1 for child-to-parent calculations).

- try to model this using with a pencil and paper, using 10 random unsorted numbers, and inserting each of them into a "heapsort" array of 10 elements.
- To insert into a heap, **end-add** and **swap-parent** if higher, until parent higher.
- To delete the top of a heap, move **end-to-top**, and **defer-higher-child** or **sift-down** , until no child is higher.
- try it on a pen and paper the numbers 10, 4, 6 ,3 ,5 , 11.

pen-and-paper-solution

- 10, 4, 6, 3, 5, 11 -> 10
- 4, 6,3, 5, 11 -> 10, 4 : 4 is end-added, no swap-parent because 4 < 10.
- 6, 3, 5, 11 -> 10, 4, 6 : 6 is end-added, no swap-parent because 6 < 10.
- 3, 5, 11 -> 10, 4, 6, 3 : 3 is end-added, 3 is position 4 , divide by 2 = 2, 4 at position 2, no swap-parent because 4 > 3.
- 5 , 11 -> 10, 4, 6, 3 , 5 : 5 is end-added , 5 is position 5, divided by 2 = 2, 4 at position 2, swap-parent as 4 < 5; 5 at position 2, no swap-parent because 5 < 10 at position 1.

- 10 , 5, 6, 3, 4

- 11 -> 10, 5, 6, 3, 4, 11 : 11 is end-added, 11 is position 6, divide by 2 = 3, swap 6 with 11, 11 is position 3, swap 11 with 10, stop as no parent.

- 11, 5, 10, 3, 4, 6

- 11 has children 5, 10 ; 5 has children 3 and 4 ; 10 has child 6. Parent always > child.

- the answer was 11, 5, 10, 3, 4 , 6.
- EXERCISE: Now try removing each top element of 11, 5, 10, 3, 4, 6 , using end-to-top and sift-down (or swap-higher-child) to get the numbers

in descending order.

pen-and-paper-solution

- 11 leaves * , 5, 10, 3, 4, 6 -> 6 , 5, 10, 3, 4 -> **sift-down** -> choose greater child 5 (2*n+0) or 10 (2*n+1) -> is 6 > 10 ? no -> swap 10 and 6 ->

- 10, 5, *6, 3, 4 -> 4 is greatest child as no +1 child. is 6 > 4 ? yes, stop.

- 10 leaves * , 5 , 6 , 3, 4 -> *4, 5, 6, 3 -> is left(0) or right(+1) child greater -> +1 is greater; is 4 > +1 child ? no , swap

- 6,5, *4, 3 -> *4 has no children so stop.

- 6 leaves *, 5, 4, 3 -> *3, 5, 4 -> +0 child is greater -> is 3 > 5 ? no , so swap -> 5, *3, 4 , *3 has no child so stop.

is

- 5 leaves so 3, 4 -> *4, 3 -> +0 child greatest as no right child -> is 4 > 3 ? no , so exit
- 4 leaves 3 .
- 3 leaves *.
- numbers extracted in descending order 11, 10, 6, 5, 4, 3.

- a priority queue allows elements to be inserted with a priority , and extracted according to priority. (This can happen usefully, if the element has a paired structure, one part is the key, and the other part the data. Otherwise, it is just a mechanism for sorting).
- EXERCISE: Using the above technique of insert-back/challenge-parent, and delete-front/last-to-front/defer-higher-child, implement either heap sort or a priority queue.

Dijsktra's algorithm

Dijsktra's algorithm is a searching algorithm using a priority queue. It begins with inserting the start node with a priority value of 0. All other nodes are inserted with priority values of large N. Each node has an adjacency list of other nodes, a current distance to start node, and previous pointer to previous node used to calculate current node. Alternative to an adjacency list, is an adjacency matrix, which needs n x n boolean adjacencies.

The agorithm basically iterates over the priority queue, removing the front node, examining the adjacent nodes, and updating with a distance equal to the sum of the front nodes distance for each adjacent node , and the distance given by the adjacency information for an adjacent node.

After each node's update, the extra operation **"update priority"** is used on that node :

while the node's distance is less than it's parents node (for this priority queue, parents have lesser distances than the children), the node is swapped with the parent.

After this, *while* the node is greater distance than one or more of its children, it is swapped with the least distant child, so the least distant child becomes parent of its greater distant sibling, and parent to the greater distant current node.

With updating the priority, the adjacent node to the current node has a back pointer changed to the current node.

The algorithm ends when the target node becomes the current node removed, and the path to the start node can be recorded in an array by following back pointers, and then doing something like a quick sort partition to reverse the order of the array , to give the shortest path to target node from the start node.

Quick sort

3. Write a C program to recursively sort using the Quick sort partition exchange algorithm.

- you can use the "driver", or the random number test data from Q1. on mergesort. This is "re-use", which is encouraged in general.

- an advantage of reuse is less writing time, debugging time, testing time.

- the concept of partition exchange is that a partition element is (randomly) selected, and every thing that needs sorted is put into 3 equivalance

classes : those elements less than the partition value, the partition element, and everything above (and equal to) the partition element.

- this can be done without allocating more space than one temporary element space for swapping two elements. e.g a temporary integer for integer data.
- However, where the partition element should be using the original array space is not known.
- This is usually implemented with putting the partition on the end of the array to be sorted, and then putting two pointers , one at the start of the array,

and one at the element next to the partition element , and repeatedly scanning the left pointer right, and the right pointer left.

- the left scan stops when an element equal to or greater than the partition is found, and the right scan stops for a smaller element than the partition value,

and these are swapped, which uses the temporary extra space.

- the left scan will always stop if it reaches the partition element , which is the last element; this means the entire array is less than partition value.
- the right scan could reach the first element, if the entire array is greater than the partition , and this needs to be tested for, else the scan doesn't stop.
- the outer loop exits when then left and right pointers cross. Testing for pointer crossing and outer loop exit

should occur before swapping, otherwise the right pointer may be swapping a less-than-partition element previously scanned by the left pointer.

- finally, the partition element needs to be put between the left and right partitions, once the pointers cross.
- At pointer crossing, the left pointer may be stopped at the partition element's last position in the array, and the right pointer not progressed past the

element just before the last element. This happens when all the elements are less than the partition.

- if the right pointer is chosen to swap with the partition, then an incorrect state results where the last element of the left array becomes less than the

partition element value.

- if the left pointer is chosen to swap with the partition, then the left array will be less than the partition, and partition will have swapped with an element with value greater than the partition or the partition itself.

- The corner case of quicksorting a 2 element **out-of-order** array has to be examined.

- The left pointer stops on the first **out of order** element. The right pointer begins on the first **out-of-order** element, but the outer loop exits because this is the leftmost element. The partition element is then swapped with the left pointer's first element, and the two elements are now **in order**.

- In the case of a 2 element **in order** array, the leftmost pointer skips the first element which is less than the partition, and stops on the partition. The right pointer begins on the first element and exits because it is the first position. The pointers have crossed so the outer loop exits. The partition swaps with itself, so the in-ordering is preserved.

- After doing a swap, the left pointer should be incremented and right pointer decremented, so the same positions aren't scanned again, because an endless loop can result (possibly when the left pointer exits when the element is equal to or greater than the partition, and the right element is equal to the partition value). One implementation, Sedgewick, starts the pointers with the left pointer minus one and right pointer

the plus one the intended initial scan positions, and use the pre-increment and pre-decrement operators e.g. (++i, --i) .

Solution

One possible solution , can be to adapt this word sorting use of quicksort to sort integers. Otherwise , an exercise would be to re-write non-generic qsort functions of qsortsimp, partition, and swap for integers.

```
/*
 * qsortsimp.h
 *
 *  Created on: 17/03/2013
 *      Author: anonymous
 */

#ifndef QSORTSIMP_H_
#define QSORTSIMP_H_
#include <stdlib.h>
void qsortsimp( void* a, size_t elem_sz, int len, int(*cmp) (void*,void*) );
void shutdown_qsortsimp();

#endif /* QSORTSIMP_H_ */

//---------------------------------------------------------------------------

/*   qsortsimp.c
 *   author : anonymous
```

```
 */
#include "qsortsimp.h"
#include<stdlib.h>
#include<string.h>

    static void * swap_buf =0;
    static int bufsz = 0;


void swap( void* a, int i, int j, size_t elem_sz) {
    if (i==j)return;
    if (bufsz == 0 || bufsz < elem_sz) {
        swap_buf = realloc(swap_buf, elem_sz);
        bufsz=elem_sz;
    }

    memcpy( swap_buf, a+i*elem_sz, elem_sz);
    memcpy( a+i*elem_sz, a+j*elem_sz, elem_sz);
    memcpy( a+j*elem_sz, swap_buf, elem_sz);
}

void shutdown_qsortsimp() {
    if (swap_buf) {
        free(swap_buf);
    }
}

int partition( void* a, size_t elem_sz, int len, int (*cmp)(void*,void*) ) {

    int i = -1;
    int j = len-1;
    void* v = a + j * elem_sz;

    for(;;) {
        while( (*cmp)(a + ++i * elem_sz , v  ) < 0);
        while ( (*cmp)(v, a + --j * elem_sz) < 0 ) if (j==0) break ;
        if( i>=j)break;
        swap(a, i, j, elem_sz);
    }
    swap( a, i, len-1, elem_sz);
    return i;

}

void qsortsimp( void* a, size_t elem_sz, int len, int(*cmp) (void*,void*) ) {
    if ( len > 2) {
        int p = partition(a, elem_sz, len, cmp);
        qsortsimp( a, elem_sz, p, cmp);
        qsortsimp( a+(p+1)*elem_sz, elem_sz, len - p -1, cmp );
    }

}



//------------------------------------------------------------------------------

/*
```

```
/*
 Name        : words_quicksort.c
 Author      : anonymous
 Version     :
 Copyright   :
 Description : quick sort the words in moby dick in C, Ansi-style
 ============================================================================
 */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include "qsortsimp.h"


void printArray(const char* a[], int n) {
    int i;
    for(i=0; i < n; ++i) {
            if(i!=0 && i% 5 == 0) {
                printf("\n");
            }
            if (i%1000000 ==0) {
                fprintf(stderr,"printed %d words\n", i);
            }
            printf("%s  ", a[i]);

    }
    printf("\n");

}

const int MAXCHARS=250;
char ** wordlist=0;
int nwords=0;
int remaining_block;
const size_t NWORDS_PER_BLOCK = 1000;

//const char* spaces=" \t\n\r";
//inline isspace(const char ch) {
//  int i=0;
//  while(spaces[i]!='\0') {
//      if(spaces[i++] == ch)
//          return 1;
//  }
//  return 0;
//}

void freeMem() {
    int i = nwords;
    while(i > 0 ) {
            free(wordlist[--i]);

    }
    free(wordlist);

}
```

```
static char * fname="~/Downloads/books/pg2701-moby-dick.txt";

void getWords() {

    char buffer[MAXCHARS];
    FILE* f = fopen(fname,"r");
    int state=0;
    int ch;
    int i;
    while ((ch=fgetc(f))!=EOF) {
        if (isalnum(ch) && state==0) {
            state=1;
            i=0;
            buffer[i++]=ch;
        } else if (isalnum(ch)  && i < MAXCHARS-1) {
            buffer[i++]=ch;
        } else if (state == 1) {
            state =0;
            buffer[i++]= '\0';
            char* dynbuf = malloc(i);
            int j;
            for(j=0; j < i; ++j) {
                dynbuf[j] = buffer[j];
            }
            i=0;
            if (wordlist == 0 ) {

                wordlist = calloc(NWORDS_PER_BLOCK, sizeof(char*));
                remaining_block = NWORDS_PER_BLOCK;
            } else if ( remaining_block == 0) {
                wordlist = realloc(wordlist, (NWORDS_PER_BLOCK + nwords)* sizeof(char*));
                remaining_block = NWORDS_PER_BLOCK;
                fprintf(stderr,"allocated block %d , nwords = %d\n", remaining_block, nwords);

            }
            wordlist[nwords++]= dynbuf;
            --remaining_block;
        }

    }
    fclose(f);


}
void testPrintArray() {

    int i;

    for(i=0; i < nwords;++i) {
        printf("%s | ", wordlist[i]);

    }
    putchar('\n');
    printf("stored %d words. \n",nwords);
}

int cmp_str_1( void* a, void *b) {
        int r = strcasecmp( *((char**)a),*((char**)b));
```

```
        return r;
}

int main(int argc, char* argv[]) {
    if (argc > 1) {
        fname = argv[1];
    }
    getWords();
    testPrintArray();

    qsortsimp(wordlist, sizeof(char*), nwords, &cmp_str_1);

    testPrintArray();

    shutdown_qsortsimp();
    freeMem();
    puts("!!!Hello World!!!"); /* prints !!!Hello World!!! */
    return EXIT_SUCCESS;
}
```

In-depth C ideas

Arrays

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 for variables of the same type. 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 (array can be termed as a constant pointer). For now, we will consider just their declaration and their use.

Arrays

C arrays are declared in the following form:

```
type name[number of elements];
```

For example, if we want an array of six integers (or whole numbers), we write in C:

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

For a six character array called *letters*,

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

and so on.

You can also initialize as you declare. Just put the initial elements in curly brackets separated by commas as the initial value:

```
type name[number of elements]={comma-separated values}
```

For example, if we want to initialize an array with six integers, with `0, 0, 1, 0, 0, 0` as the initial values:

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

Though when the array is initialized as in this case, the array dimension may be omitted, and the array will be automatically sized to hold the initial data:

```
int point[]={0,0,1,0,0,0};
```

This is very useful in that the size of the array can be controlled by simply adding or removing initializer elements from the definition without the need to adjust the dimension.

If the dimension is specified, but not all elements in the array are initialized, the remaining elements will contain a value of 0. This is very useful, especially when we have very large arrays.

```
int numbers[2000]={245};
```

The above example sets the first value of the array to 245, and the rest to 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 = point[2];
```

Arrays in C are indexed starting at 0, as opposed to starting at 1. The first element of the array above is `point[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 guarantee bounds checking on array accesses. The compiler may not complain about the following (though the best compilers do):

```
char y;
int z = 9;
char point[6] = { 1, 2, 3, 4, 5, 6 };
//examples of accessing outside the array. A compile error is not always raised
y = point[15];
y = point[-4];
y = point[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 point[-1]. To alleviate indexing problems, the sizeof() expression is commonly used when coding loops that process arrays.

Many people use a macro that in turn uses sizeof() to find the number of elements in an array, a macro variously named "lengthof()",[1] "MY_ARRAY_SIZE()" or "NUM_ELEM()",[2] "SIZEOF_STATIC_ARRAY()",[3] etc.

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

for (ix=0; ix< (sizeof(anArray)/sizeof(short)); ++ix) {
  DoSomethingWith("%d", 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. Good programming practice is to declare a variable *size* , and store the number of elements in the array in it.

size = sizeof(anArray)/sizeof(short)

C also supports multi dimensional arrays (or, rather, arrays of 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 }};
```

The amount of columns must be explicitly stated; however, the compiler will find the appropriate amount of rows based on the initializer list.

There are also weird notations possible:

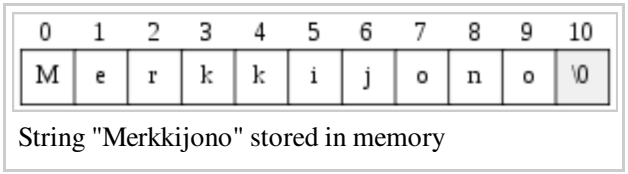
```
int a[100];
int i = 0;
if (a[i]==i[a])
{
  printf("Hello world!\n");
}
```

---

a[i] and i[a] refer 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. C allows a character array to be represented by a character string rather than a list of characters, with the null terminating character automatically added to the end. For example, to store the string "Merkkijono", we would write



String "Merkkijono" stored in memory

```
char string[] = "Merkkijono";
```

or

```
char string[] = {'M', 'e', 'r', 'k', 'k', 'i', 'j', 'o', 'n', 'o', '\0'};
```

In the first example, the string will have a null character automatically appended to the end by the compiler; by convention, library functions expect strings to be terminated by a null character. The latter declaration indicates individual elements, and as such the null terminator needs to be added manually.

Strings do not always have to be linked to an explicit variable. As you have seen already, a string of characters can be created directly as an unnamed string that is used directly (as with the printf functions.)

To create an extra long string, you will have to split the string into multiple sections, by closing the first section with a quote, and recommencing the string on the next line (also starting and ending in a quote):

```
char string[] = "This is a very, very long "
                "string that requires two lines.";
```

While strings may also span multiple lines by putting the backslash character at the end of the line, this method is deprecated.

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

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

This standard string library will allow various tasks to be performed on strings, and is discussed in the Strings chapter.

Further reading

1. Pádraig Brady. "C and C++ notes" (http://www.pixelbeat.org/programming/gcc/c_c++_notes.html).
2. C Programming/Pointers and arrays
3. MINC/Reference/MINC1-volumeio-programmers-reference

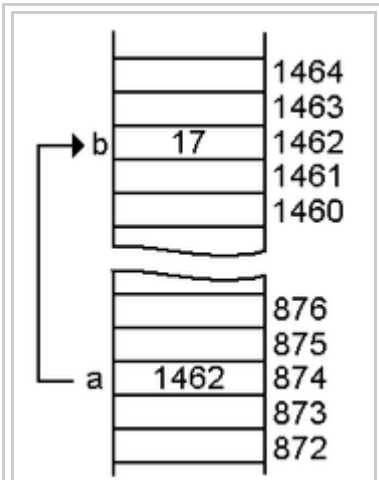
Pointers and arrays

A **pointer** is a value that designates the address (i.e., the location in memory), of some value. Pointers are variables that hold a memory location.

There are four fundamental things you need to know about pointers:

- How to declare them (with the address operator '`&`': `int *pointer = &variable;`)
- How to assign to them (`pointer = NULL;`)
- How to reference the value to which the pointer points (known as *dereferencing*, by using the dereferencing operator '`*`': `value = *pointer;`)
- How they relate to arrays (the vast majority of arrays in C are simple lists, also called "1 dimensional arrays", but we will briefly cover multi-dimensional arrays with some pointers in a later chapter).

Pointers can reference any data type, even functions. We'll also discuss the relationship of pointers with text strings and the more advanced concept of function pointers.



Pointer *a* pointing variable *b*. Note that *b* stores number, whereas *a* stores address of *b* in memory (1462)

Declaring pointers

Consider the following snippet of code which declares two pointers:

```
1  struct MyStruct {
2      int   m_aNumber;
3      float num2;
4  };
5
6 int main()
7 {
8      int *pJ2;
9      struct MyStruct *pAnItem;
10 }
```

Lines 1-4 define a structure. Line 8 declares a variable which points to an `int`, and line 9 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 following, line 1 declares `var1` as a pointer to a long and `var2` as a long and not a pointer to a long. In line 2, `p3` is declared as a pointer to a pointer to an int.

```
1 long  *  var1, var2;
2  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 the address of a variable to a pointer, 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 used to do this.

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

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.

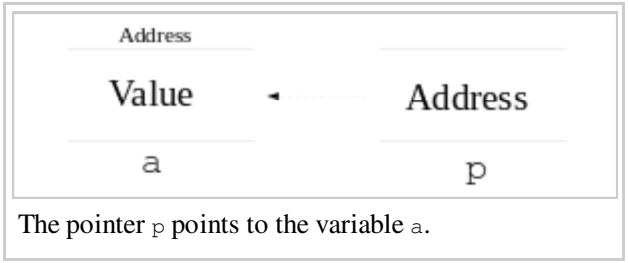
Pointer dereferencing

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 pointer `p` points to the variable `a`.

The expression `bb->m_aNumber` is entirely equivalent to `(*bb).m_aNumber`. They both access the `m_aNumber` 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 explicitly 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; //declaring an integer variable//
  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(*myArray))
float v3;
struct MyStruct *secondArray;
int   someSize;
int   ix;
/* initialization of myArray ... */
v3 = KrazyFunction( myArray, MY_ARRAY_SIZE, 4 );
/* ... */
secondArray = calloc( someSize, sizeof(myArray) );
for (ix=0; ix<someSize; ix++) {
    secondArray[ix].m_aNumber = ix *2;
    secondArray[ix].num2 = .304 * ix * ix;
}
```

Pointers and array names can pretty much be used interchangeably. 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 pointer to the first element of 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 initialization .. */
.
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. Let's 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 a two dimensional 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 a 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 = 6 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 linearA[17];     */
*A[3][-3] = 1.44;         /* refers to linearA[12];
                             negative indices are sometimes useful. But avoid using them as much as possible. */
```

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 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 byte in the string being a zero, or the null character '\0'. 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 initialization 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 */

#define THE_BIGGEST 100

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 );
     } else {
         /* invoke function through function pointer (new style) */
       rv = aFunction(aVariable, dataPointer );
    };
    return rv;
}

typedef struct {
    int    colorSpec;
    char   *phrase;
} 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 preceding 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.

Practical use of function pointers in C

Function pointers are mainly used to reduce the complexity of switch statement. Example with switch statement:

```
#include <stdio.h>
int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
int div(int a, int b);
int main()
{
    int i, result;
    int a=10;
    int b=5;
    printf("Enter the value between 0 and 3 : ");
    scanf("%d",&i);
    switch(i)
    {
        case 0:  result = add(a,b); break;
        case 1:  result = sub(a,b); break;
```

```
        case 2:  result = mul(a,b); break;
        case 3:  result = div(a,b); break;
    }
}
int add(int i, int j)
{
    return (i+j);
}
int sub(int i, int j)
{
    return (i-j);
}
 int mul(int i, int j)
{
    return (i*j);
}
int div(int i, int j)
{
    return (i/j);
}
```

Without using a switch statement:

```
#include <stdio.h>
int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
int div(int a, int b);
int (*oper[4])(int a, int b) = {add, sub, mul, div};
int main()
{
    int i,result;
    int a=10;
    int b=5;
    printf("Enter the value between 0 and 3 : ");
    scanf("%d",&i);
    result = oper[i](a,b);
}
int add(int i, int j)
{
    return (i+j);
}
int sub(int i, int j)
{
    return (i-j);
}
int mul(int i, int j)
{
    return (i*j);
}
int div(int i, int j)
{
    return (i/j);
}
```

Function pointers may be used to create a struct member function:

```
typedef struct
{
    int (*open)(void);
    void (*close)(void);
    int (*reg)(void);
} device;

int my_device_open(void)
{
    /* ... */
}

void my_device_close(void)
{
    /* ... */
}

void register_device(void)
{
    /* ... */
}

device create(void)
{
    device my_device;
    my_device.open = my_device_open;
    my_device.close = my_device_close;
    my_device.reg = register_device;
    my_device.reg();
    return my_device;
}
```

Use to implement this pointer (following code must be placed in library).

```
static struct device_data
{
    /* ... here goes data of structure ... */
};

static struct device_data obj;

typedef struct
{
    int (*open)(void);
    void (*close)(void);
    int (*reg)(void);
} device;

static struct device_data create_device_data(void)
{
    struct device_data my_device_data;
```

```
    /* ... here goes constructor ... */
    return my_device_data;
}

/* here I omit the my_device_open, my_device_close and register_device functions */

device create_device(void)
{
    device my_device;
    my_device.open = my_device_open;
    my_device.close = my_device_close;
    my_device.reg = register_device;
    my_device.reg();
    return my_device;
}
```

Examples of pointer constructs

Below are some example constructs which may aid in creating your pointer.

```
int i;         // integer variable 'i'
int *p;        // pointer 'p' to an integer
int a[];       // array 'a' of integers
int f();       // function 'f' with return value of type integer
int **pp;      // pointer 'pp' to a pointer to an integer
int (*pa)[];   // pointer 'pa' to an array of integer
int (*pf)();   // pointer 'pf' to a function with return value integer
int *ap[];     // array 'ap' of pointers to an integer
int *fp();     // function 'fp' which returns a pointer to an integer
int ***ppp;    // pointer 'ppp' to a pointer to a pointer to an integer
int (**ppa)[]; // pointer 'ppa' to a pointer to an array of integers
int (**ppf)(); // pointer 'ppf' to a pointer to a function with return value of type integer
int *(*pap)[]; // pointer 'pap' to an array of pointers to an integer
int *(*pfp)(); // pointer 'pfp' to function with return value of type pointer to an integer
int **app[];   // array of pointers 'app' that point to pointers to integer values
int (*apa[])[];// array of pointers 'apa' to arrays of integers
int (*apf[])();// array of pointers 'apf' to functions with return values of type integer
int ***fpp();  // function 'fpp' which returns a pointer to a pointer to a pointer to an int
int (*fpa())[];// function 'fpa' with return value of a pointer to array of integers
int (*fpf())();// function 'fpf' with return value of a 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):

```c
/* 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

```c
#define NUM_ELEM(x) (sizeof (x) / sizeof (*(x)))

for( i = 0; i < NUM_ELEM(array); i++ )
{
    /* do something with array[i] */
    ;
}
```

Note that the `sizeof` operator only works on things defined earlier in the same function. The compiler replaces it with some fixed constant number. In this case, the `buffer` was declared as an array of 10 char's earlier in the same function, and the compiler replaces `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 -- an array defined in some other function. For example,

```c
/* broken.c - demonstrates a flaw */

#include <stdio.h>
#include <string.h>
#define NUM_ELEM(x) (sizeof (x) / sizeof (*(x)))

int sum( int input_array[] ){
```

```
  int sum_so_far = 0;
  int i;
  for( i = 0; i < NUM_ELEM(input_array); i++ ) // WON'T WORK -- input_array wasn't defined in this function.
  {
    sum_so_far += input_array[i];
  };
  return( sum_so_far );
}

int main(int argc, char *argv[])
{
  int left_array[] = { 1, 2, 3 };
  int right_array[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
  int the_sum = sum( left_array );
  printf( "the sum of left_array is: %d", the_sum );
  the_sum = sum( right_array );
  printf( "the sum of right_array is: %d", the_sum );

  return 0;
}
```

Unfortunately, (in C and C++) the length of the array cannot be obtained from an array passed in at run time, because (as mentioned above) the size of an array is not stored anywhere. The compiler always replaces sizeof with a constant. This sum() routine needs to handle more than just one constant length of an array.

There are some common ways to work around this fact:

- Write the function to require, for each array parameter, a "length" parameter (which has type "size_t"). (Typically we use sizeof at the point where this function is called).
- Use of a convention, such as a 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 (such as ".length") as well as the array (or a pointer to the first element); similar to the `string` or `vector` classes in C++.

```
/* fixed.c - demonstrates one work-around */

#include <stdio.h>
#include <string.h>
#define NUM_ELEM(x) (sizeof (x) / sizeof (*(x)))

int sum( int input_array[], size_t length ){
  int sum_so_far = 0;
  int i;
  for( i = 0; i < length; i++ )
  {
    sum_so_far += input_array[i];
  };
  return( sum_so_far );
}

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

```
{
  int left_array[] = { 1, 2, 3, 4 };
  int right_array[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
  int the_sum = sum( left_array, NUM_ELEM(left_array) ); // works here, because left_array is defined in this function
  printf( "the sum of left_array is: %d", the_sum );
  the_sum = sum( right_array, NUM_ELEM(right_array) ); // works here, because right_array is defined in this function
  printf( "the sum of right_array is: %d", the_sum );

  return 0;
}
```

It's worth mentioning that sizeof operator has two variations: `sizeof (type)` (for instance: `sizeof (int)` or `sizeof (struct some_structure)`) and `sizeof expression` (for instance: `sizeof some_variable.some_field` or `sizeof 1`).

External Links

- C Reference Card (ANSI) (http://www.digilife.be/quickreferences/QRC/C%20Reference%20Card%20(ANSI)%202.2.pdf)
- "Common Pointer Pitfalls" (http://www.cs.cf.ac.uk/Dave/C/node10.html#SECTION00108000000000000000) by Dave Marshall

Pointer Fun with Binky

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:

- the size of the array must be known beforehand
- 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; `malloc` gives us some space to store the 10 `int`s, and returns the pointer to the first element, which is assigned to that pointer.

Important note! `malloc` does *not* initialize the array; this means that the array may contain random or unexpected values! 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 declaring a pointer for the allocated memory. Often a number of statements exist between the declaration and the call to `malloc`, as follows:

```
int *array = NULL;
printf("Hello World!!!");
/* more statements */
array = malloc(10*sizeof(int)); /* delayed allocation */
/* 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 = malloc(3 * sizeof(int));
if(pt == NULL)
{
   fprintf(stderr, "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 the 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 their 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. For this reason, many embedded systems designers avoid dynamic memory allocation altogether.

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

```
void * realloc ( void * ptr, size_t size );
```

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 );
}
```

free with recursive data structures

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));

// Later: ...

free(temp); // WRONG! don't do this!
```

The statement "`free(temp);`" will **not** free `temp->left`, causing a memory leak. The correct way to free the allocated memory is as follows.

```
free(temp->left);
free(temp->right);
free(temp);
```

Because C does not have a garbage collector, C programmers are responsible for making sure there is a `free()` exactly once for each time there is a `malloc()`. If a tree has been allocated one node at a time, then it needs to be freed one node at a time.

Don't free undefined pointers

Furthermore, using `free` when the pointer in question was never allocated in the first place often crashes or leads to mysterious bugs further along.

To avoid this problem, always initialize pointers when they are declared. Either use `malloc` at the point they are declared (as in most examples in this chapter), or set them to `NULL` when they are declared (as in the "delayed allocation" example in this chapter). [1]

Write constructor/destructor functions

One way to get memory initialization and destruction right is to imitate object-oriented programming. In this paradigm, objects are constructed after raw memory is allocated for them, live their lives, and when it is time for them to be destructed, a special function called a destructor destroys the object's innards before the object itself is destroyed.

For example:

```
#include <stdlib.h> /* need malloc and friends */

/* this is the type of object we have, with a single int member */
typedef struct WIDGET_T {
  int member;
} WIDGET_T;
```

```
/* functions that deal with WIDGET_T */

/* constructor function */
void
WIDGETctor (WIDGET_T *this, int x)
{
  this->member = x;
}

/* destructor function */
void
WIDGETdtor (WIDGET_T *this)
{
  /* In this case, I really don't have to do anything, but
     if WIDGET_T had internal pointers, the objects they point to
     would be destroyed here.  */
  this->member = 0;
}

/* create function - this function returns a new WIDGET_T */
WIDGET_T *
WIDGETcreate (int m)
{
  WIDGET_T *x = 0;

  x = malloc (sizeof (WIDGET_T));
  if (x == 0)
    abort (); /* no memory */
  WIDGETctor (x, m);
  return x;
}

/* destroy function - calls the destructor, then frees the object */
void
WIDGETdestroy (WIDGET_T *this)
{
  WIDGETdtor (this);
  free (this);
}

/* END OF CODE */
```

References

1. "Bug 478901 ... libpng-1.2.34 and earlier might free undefined pointers" (https://bugzilla.mozilla.org/show_bug.cgi?id=478901)

- Memory Management

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.

Syntax

In C, string constants (literals) are surrounded by double quotes (`"`), e.g. `"Hello world!"` and are compiled to an array of the specified `char` values with an additional null terminating character (0-valued) code to mark the end of the string. The type of a string constant is `char []`.

backslash escapes

String literals may not directly in the source code contain embedded newlines or other control characters, or some other characters of special meaning in string.

To include such characters in a string, the backslash escapes may be used, like this:

Escape	Meaning
`\\`	Literal backslash
`\"`	Double quote
`\'`	Single quote
`\n`	Newline (line feed)
`\r`	Carriage return
`\b`	Backspace
`\t`	Horizontal tab
`\f`	Form feed
`\a`	Alert (bell)
`\v`	Vertical tab
`\?`	Question mark (used to escape trigraphs)
`\`*nnn*	Character with octal value *nnn*
`\x`*hh*	Character with hexadecimal value *hh*

Wide character strings

C supports wide character strings, defined as arrays of the type `wchar_t`, 16-bit (at least) values. They are written with an L before the string like this

```
wchar_t *p = L"Hello world!";
```

This feature allows strings where more than 256 different possible characters are needed (although also variable length `char` strings can be used). They end with a zero-valued `wchar_t`. These strings are not supported by the `<string.h>` functions. Instead they have their own functions, declared in `<wchar.h>`.

Character encodings

What character encoding the `char` and `wchar_t` represent is not specified by the C standard, except that the value 0x00 and 0x0000 specify the end of the string and not a character. It the input and output code which are directly affected by the character encoding. Other code should not be too affected. The editor should also be able to handle the encoding if strings shall be able to written in the source code.

There are three major types of encodings:

- One byte per character. Normally based on ASCII. There is a limit of 255 different characters plus the zero termination character.
- Variable length `char` strings, which allows many more than 255 different characters. Such strings are written as normal `char`-based arrays. These encodings are normally ASCII-based and examples are UTF-8 or Shift JIS.
- Wide character strings. They are arrays of `wchar_t` values. UTF-16 is the most common such encoding, and it is also variable-length, meaning that a character can be two `wchar_t`.

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[1] s1, const char * restrict s2);
```

Some people recommend using `strncat()` *or* `strlcat()` *instead of strcat, in order to avoid buffer overflow.*

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[20];
...
int penColor = 3, penThickness = 2;
strcpy(penText, colors[penColor]);
strcat(penText, widths[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)
{
    char ch = 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 != ch)
        s++;
    return (*s == ch) ? (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 characters, 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". (In the previous example, "Alpha2" compares greater than "Alpha12" because '2' comes after '1' in the character set.) 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 */
    ;
```

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);
```

Some people recommend always using `strncpy()` *instead of strcpy, to avoid buffer overflow.*

The `strcpy()` function shall copy the C 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`. There is no value used to indicate an error: if the arguments to `strcpy()` are correct, and the destination buffer is large enough, the function will never fail.

Example:

```
   #include <stdio.h>
   #include <string.h>
   /* ... */
   static const char *penType="round";
   /* ... */
   char penText[20];
   /* ... */
   strcpy(penText, penType);
```

Important: You must ensure that the destination buffer (`s1`) is able to contain all the characters in the source array, including the terminating null byte. Otherwise, `strcpy()` will overwrite memory past the end of the buffer, causing a buffer overflow, which can cause the program to crash, or can be exploited by hackers to compromise the security of the computer.

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')
        ;               /* The body of this loop is left empty. */
```

```
    /* 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)
{
    const 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 s 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);
```

The `strrchr` function is similar to the `strchr` function, except that `strrchr` returns a pointer to the **last** occurrence of `c` within `s` instead of the first.

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, though not completely portable, 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)
{
    const unsigned char *us1 = (const unsigned char *) s1;
    const unsigned char *us2 = (const 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`:

```c
#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 */
}
```

```c
#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 */
}
```

```c
#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 *haystack, const char *needle);
```

The `strstr()` function shall locate the first occurrence in the string pointed to by `haystack` of the sequence of bytes (excluding the terminating null byte) in the string pointed to by `needle`. The function returns the pointer to the matching string in `haystack` or a null pointer if a match is not found. If `needle` is an empty string, the function returns `haystack`.

The following is a public-domain implementation of `strstr`:

```
#include <string.h>
/* strstr */
char *(strstr)(const char *haystack, const char *needle)
{
    size_t needlelen;
    /* Check for the null needle case.  */
    if (*needle == '\0')
        return (char *) haystack;
    needlelen = strlen(needle);
    for (; (haystack = strchr(haystack, *needle)) != NULL; haystack++)
        if (memcmp(haystack, needle, needlelen) == 0)
            return (char *) haystack;
    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 = 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?

Further reading

- A Little C Primer/C String Function Library
- C++ Programming/Code/IO/Streams/string

- Because so many functions in the standard `string.h` library are vulnerable to buffer overflow errors, some people (http://www.and.org/vstr/security) recommend avoiding the `string.h` library and "C style strings" and instead using a dynamic string API, such as the ones listed in the String library comparison (http://www.and.org/vstr/comparison).
- There's a tiny public domain concat() function, which will allocate memory and safely concatenate any number of strings in portable C/C++ code (http://openwall.info/wiki/people/solar/software/public-domain-source-code/concat)

Complex types

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.

Data structures

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.

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 structure;
```

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. For trees or chained lists, we need a pointer to the same data type in the struct. During compilation, the type synonym is not known to the compiler and there will be an error. To avoid this, it is necessary to let the compiler know the name right from the start (Note that the **struct** keyword is used *only* inside the structure! After the declaration, the compiler *knows* that the type synonym refers to a **struct**):

```
typedef struct Mystruct {
  ...
  struct Mystruct *pMystruct
} Mystruct;
```

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

For "register", "volatile", "auto" and "extern", see C Programming/Variables#Other_Modifiers.

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 2300

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);

   memset(&dest, 0, sizeof(dest));                /* zero the struct */
   dest.sin_family = AF_INET;
   dest.sin_addr.s_addr = htonl(INADDR_LOOPBACK); /* set destination IP number - localhost, 127.0.0.1*/
   dest.sin_port = htons(PORTNUM);                /* set destination port number */

   connect(mysocket, (struct sockaddr *)&dest, sizeof(struct sockaddr_in));

   len = recv(mysocket, buffer, MAXRCVLEN, 0);

   /* We have to null terminate the received data ourselves */
   buffer[len] = '\0';

   printf("Received %s (%d bytes).\n", buffer, len);

   close(mysocket);
```

```
    return EXIT_SUCCESS;
}
```

This is the very bare bones of a client; in practice, we would check every function that we call for failure, however, error checking has been left out for clarity.

As you can see, the code mainly revolves around `dest` which is a struct of type `sockaddr_in`. This struct stores 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 irrelevant for now.

```
memset(&dest, 0, sizeof(dest));                /* zero the struct */
dest.sin_family = AF_INET;
dest.sin_addr.s_addr = inet_addr("127.0.0.1"); /* set destination IP number */
dest.sin_port = htons(PORTNUM);                /* set destination port number */
```

Now we get on to the interesting part:

The first line uses `memset()` to zero the struct.

The second line sets the address family. This should be the same value that was passed as the first parameter to `socket()`; for most purposes `AF_INET` will serve.

The third 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 fourth 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 run solely 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 always be used.

Now that all of the preliminary work is done, we can actually make the connection and use it:

```
connect(mysocket, (struct sockaddr *)&dest, sizeof(struct sockaddr_in));
```

This tells our OS to use the socket `mysocket` to create a connection to the machine specified in `dest`.

```
len = recv(mysocket, buffer, MAXRCVLEN, 0);
```

Now this receives up to `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 the buffer, so we need to do it ourselves with `buffer[len] = '\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 then this is quite easy. All you have to do is use the `send()` function, which uses the same parameters as `recv()`. If in our previous example `buffer` had the text we wanted to send and its length was stored in `len` we would write `send(mysocket, buffer, len, 0)`. `send()` returns the number of bytes that were sent. It is important to remember that `send()`, for various reasons, may not be able to send all of the bytes, so it is important to check that its return value is equal to the number of bytes you tried to send. 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 2300

int main(int argc, char *argv[])
{
    char* msg = "Hello World !\n";

    struct sockaddr_in dest; /* socket info about the machine connecting to us */
    struct sockaddr_in serv; /* socket info about our server */
    int mysocket;            /* socket used to listen for incoming connections */
    socklen_t socksize = sizeof(struct sockaddr_in);

    memset(&serv, 0, sizeof(serv));           /* zero the struct before filling the fields */
    serv.sin_family = AF_INET;                /* set the type of connection to TCP/IP */
    serv.sin_addr.s_addr = htonl(INADDR_ANY); /* set our address to any interface */
    serv.sin_port = htons(PORTNUM);           /* set the server port number */

    mysocket = socket(AF_INET, SOCK_STREAM, 0);
```

```
    /* bind serv information to mysocket */
    bind(mysocket, (struct sockaddr *)&serv, sizeof(struct sockaddr));

    /* start listening, allowing a queue of up to 1 pending connection */
    listen(mysocket, 1);
    int consocket = accept(mysocket, (struct sockaddr *)&dest, &socksize);

    while(consocket)
    {
        printf("Incoming connection from %s - sending welcome\n", inet_ntoa(dest.sin_addr));
        send(consocket, msg, strlen(msg), 0);
        close(consocket);
        consocket = accept(mysocket, (struct sockaddr *)&dest, &socksize);
    }

    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 received on the port specified in the `sockaddr_in` should be handled by our specified socket.

The `listen()` function then tells our program to start listening using the given 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. If there is no connection waiting on the queue the program waits until a connection is received. The `accept()` function returns another socket. This socket is essentially a "session" socket, and can be used solely for communicating with connection we took off the queue. The original socket (`mysocket`) continues to listen on the specified port for further connections.

Once we have "session" socket we can handle it in the same way as with the client, using `send()` and `recv()` 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 `fork()` off separate processes, or use threads, to handle the connections.

Useful network functions

```
int gethostname(char *hostname, size_t size);
```

The parameters are a pointer to an array of chars and the size of that array. If possible, it finds the hostname 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 stores it in a `hostent` struct. The most useful part of a `hostent` structure is the `(char**) h_addr_list` field, which is a null terminated array of the 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 and would rather just use a UDP connection, then you can just replace `SOCK_STREAM` with `SOCK_DGRAM` in your call to `socket()` and use the result 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 specifying who you are communicating with.

How do I check for errors?

The functions `socket()`, `recv()` and `connect()` all return -1 on failure and use errno for further details.

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.

- Dynamic multidimensional arrays can be allocated by first allocating an array of pointers, and then allocating subarrays and storing their addresses in the array of pointers.[2] (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 (http://www.eskimo.com/~scs/C-faq/q6.16.html).

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.

- Another useful way to create a dynamic multi-dimensional array is to flatten the array and index manually. For example, a 2-dimensional array with sizes x and y has x*y elements, therefore can be created by

```
int dynamic_multi_array[x*y];
```

The index is slightly trickier than before, but can still be obtained by y*i+j. You then access the array with

```
static_multi_array[i][j];
```

```
dynamic_multi_array[y*i+j];
```

Some more examples with higher dimensions:

```
int dim1[w];
int dim2[w*x];
int dim3[w*x*y];
int dim4[w*x*y*z];

dim1[i]
dim2[w*j+i];
dim3[w*(x*i+j)+k] // index is k + w*j + w*x*i
dim4[w*(x*(y*i+j)+k)+l] // index is w*x*y*i + w*x*j + w*k + l
```

Note that w*(x*(y*i+j)+k)+l is equal to w*x*y*i + w*x*j + w*k + l, but uses fewer operations (see Horner's Method (http://en.wikipedia.org/wiki/Horner%27s_method)). It uses the same number of operations as accessing a static array by dim4[i][j][k][l], so should not be any slower to use.

The advantage to using this method is that the array can be passed freely between functions without knowing the size of the array at compile time (since C sees it as a 1-dimensional array, although some way of passing the dimensions is still necessary), and the entire array is contiguous in memory, so accessing consecutive elements should be fast. The disadvantage is that it can be difficult at first to get used to how to index the elements.

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: [3]

```
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 */
    *s=NULL;           /* zero the argument */
}
```

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))`
   - Dangerous, remember it replaces the text in verbatim. Suppose your code is `square (x++)`, after the macro invocation will x be incremented by 2
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.

Further reading

1. [9] (http://www.oracle.com/technetwork/server-storage/solaris10/cc-restrict-139391.html)The restrict keyword
2. Adam N. Rosenberg. [http://www.the-adam.com/adam/rantrave/st02.pdf "A Description of One Programmer's Programming Style Revisited"]. 2001. p. 19-20.
3. comp.lang.c FAQ list: "Why isn't a pointer null after calling free?" (http://c-faq.com/malloc/ptrafterfree.html) mentions that "it is often useful to set [pointer variables] to NULL immediately after freeing them".

There are a huge number of C style guidelines.

- "C and C++ Style Guides" (http://www.chris-lott.org/resources/cstyle/) by Chris Lott lists many popular C style guides.
- The Motor Industry Software Reliability Association (MISRA) publishes "MISRA-C: Guidelines for the use of the C language in critical systems". (Wikipedia: MISRA C; [11] (http://www.misra-c.com/)).

C and beyond

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").[1]
- 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

- GNU C: Extensions to the C Language (http://gcc.gnu.org/onlinedocs/gcc-4.0.2/gcc/C-Extensions.html#C-Extensions)
- C/C++ interpreter Ch extensions to the C language for scripting (http://www.softintegration.com/support/faq/general.html#4)
- SDCC: Storage Class Language Extensions (http://sdcc.sourceforge.net/doc/sdccman.html/node56.html)

1. comp.lang.c FAQ list: Question 2.6 (http://c-faq.com/struct/structhack.html): "C99 introduces the concept of a flexible array member, which allows the size of an array to be omitted if it is the last member in a structure, thus providing a well-defined solution."

Mixing languages

Assembler

See Embedded Systems/Mixed C and Assembly Programming

Cg

Make the main program (for CPU) in C, which loads and run the Cg program (for GPU).[1][2][3]

Header Files

Add to C program :[4]

```
#include <Cg/cg.h> /* To include the core Cg runtime API into your  program */
#include <Cg/cgGL.h>  /* to include the OpenGL-specific Cg runtime API */
```

Minimal program

- by bobobobo[5]

Java

Using the Java native interface (JNI), Java applications can call C libraries.

See also

- Java_Programming/Keywords/native

Perl

To mix Perl and C, we can use XS. XS is an interface description file format used to create an extension interface between Perl and C code (or a C library) which one wishes to use with Perl.

The basic procedure is very simple. We can create the necessary subdirectory structure by running "h2xs" application (e.g. "h2xs -A -n Modulename"). This will create - among others - a Makefile.PL, a .pm Perl module and a .xs XSUB file in the subdirectory tree. We can edit the .xs file by adding our code to that, let's say:

```
void
hello()
  CODE:
    printf("Hello, world!\n");
```

and we can successfully use our new command at Perl side, after running a "perl Makefile.PL" and "make".

Further details can be found on the perlxstut (http://perldoc.perl.org/perlxstut.html) perldoc (http://perldoc.perl.org) page.

Python

Here can be found some details about extending Python with modules written in C (http://docs.python.org/3/extending/extending.html). You might read about Cython (http://cython.org/) and Pyrex (http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/) as well, that makes easier to create modules in C, translating a Python-like code into C.

Using the Python ctypes (https://docs.python.org/2/library/ctypes.html) module, one can write C code directly into Python.

For further reading

- Embedded Systems/Mixed C and Assembly Programming

References

1. Lesson: 47 from NeHe Productions (http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=47)
2. Cg Bumpmapping by Razvan Surdulescu at GameDev (http://www.gamedev.net/reference/articles/article1903.asp)

3. [http://www.fusionindustries.com/default.asp?page=cg-hlsl-faq | Cg & HLSL Shading Language FAQ by Fusion Industries]
4. http://http.developer.nvidia.com/CgTutorial/cg_tutorial_appendix_b.html NVidia Cg tutorial. Appendix B. The Cg Runtime
5. Absolutely minimal CG program for good fundamentals understanding (http://bobobobo.wordpress.com/2008/10/05/cg-1/)

Code library

The following is an implementation of the Standard C99 version of `<assert.h>`:

```
/* assert.h header */
#undef assert
#ifdef NDEBUG
#define assert(_Ignore) ((void)0)
#else
void _Assertfail(char *, char *, int, char *);
#define assert(_Test) ((_Test)?((void)0):_Assertfail(#_Test,__FILE__,__LINE__,__func__))
#endif
/* END OF FILE */
```

```
/* xassertfail.c -- _Assertfail function */
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
void
_Assertfail(char *test, char *filename, int line_number, char *function_name)
{
   fprintf(stderr, "Assertion failed: %s, function %s, file %s, line %d.",
           test, function_name, filename, line_number);
   abort();
}
/* END OF FILE */
```

Computer Programming

The following articles are C adaptations from articles of the Computer programming book.

Statements

A **statement** is a command given to the computer that instructs the computer to take a specific action, such as display to the screen, or collect input. A computer program is made up of a series of statements.

```
printf ("Hi there!");
```

```
printf ("Hi there!");
printf ("Strange things are afoot...");
```

In C, a statement can be any of the following:

Labeled Statements

A statement can be preceded by a label. Three types of labels exist in C.

A simple identifier followed by a colon (`:`) is a label. Usually, this label is the target of a `goto` statement.

Within `switch` statements, `case` and `default` labeled statements exist. A statement of the form

`case` *constant-expression* `:` *statement*

indicates that control will pass to this statement if the value of the control expression of the `switch` statement matches the value of the *constant-expression*. A statement of the form

`default` `:` *statement*

indicates that control will pass to this statement if the control expression of the `switch` statement does not match any of the *constant-expressions* within the `switch` statement. If the `default` statement is omitted, the control will pass to the statement following the `switch` statement.

Compound Statements

A *compound statement* is the way C groups multiple statements into a single statement. It consists of multiple statements and declarations within braces (i.e. `{` and `}`). In the ANSI C Standard of 1989-1990, a compound statement contained an optional list of declarations followed by an optional list of statements; in more recent revisions of the Standard, declarations and statements can be freely interwoven through the code. The body of a function is also a compound statement by rule.

Expression Statements

An *expression statement* consists of an optional expression followed by a semicolon (`;`). If the expression is present, the statement may have a value. If no expression is present, the statement is often called the *null statement*.

The `printf` function calls are expressions, so the statements above are expression statements.

Selection Statements

Three types of selection statements exist in C:

`if (` *expression* `)` *statement*

In this type of if-statement, the sub-statement will only be executed iff the expression is non-zero.

`if (` *expression* `)` *statement* `else` *statement*

In this type of if-statement, the first sub-statement will only be executed iff the expression is non-zero; otherwise, the second sub-statement will be executed. Each `else` matches up with the closest unmatched `if`, so that the following two snippets of code are not equal:

```
if (expression)
  if (secondexpression) statement1;
else
  statement2;
```

```
if (expression)
  {
    if (secondexpression) statement1;
  }
else
  statement2;
```

because in the first, the `else` statement matches up with the if statement that has `secondexpression` for a control, but in the second, the braces force the `else` to match up with the if that has `expression` for a control.

Switch statements are also a type of selection statement. They have the format

`switch (` *expression* `)` *statement*

The statement here is usually compound and it contains case-labeled statements and optionally a default-labeled statement.

Iteration Statements

C has three kinds of iteration statements. The first is a while-statement with the form

`while (` *expression* `)` *statement*

The substatement of a while runs repeatedly as long as the control expression evaluates to non-zero at the beginning of each iteration. If the control expression evaluates to zero the first time through, the substatement may not run at all.

The second is a do-while statement of the form

`do` *statement* `while (` *expression* `) ;`

This is similar to a while loop, except that the controlling expression is evaluated at the end of the loop instead of the beginning and consequently the sub-statement must execute at least once.

The third type of iteration statement is the for-statement. In ANSI C 1989, it has the form

`for (` $expression_{opt}$ `;` $expression_{opt}$ `;` $expression_{opt}$ `)` *statement*

In more recent versions of the C standard, a declaration can substitute for the first expression. The *opt* subscript indicates that the expression is optional.

The statement

```
for (e1; e2; e3)
  s;
```

is the rough equivalent of

```
{
  e1;
  while (e2)
    {
      s;
      e3;
    }
}
```

except for the behavior of `continue` statements within `s`.

The `e1` expression represents an initial condition; `e2` a control expression; and `e3` what to happen on each iteration of the loop. If `e2` is missing, the expression is considered to be non-zero on every iteration, and only a `break` statement within `s` (or a call to a non-returning function such as `exit` or `abort`) will end the loop.

Jump Statements

C has four types of jump statements. The first, the `goto` statement, is used sparingly and has the form

`goto` *identifier* `;`

This statement transfers control flow to the statement labeled within the given identifier.

The second, the break statement, with the form

```
break ;
```

is used within iteration statements and `switch` statements to pass control flow to the statement following the while, do-while, for, or switch.

The third, the continue statement, with the form

```
continue ;
```

is used within the substatement of interation statements to transfer control flow to the place just before the end of the substatement. In `for` statements the iteration expression will then be executed before the controlling expression is evaluated.

The fourth type of jump statement is the return statement with the form

`return` $expression_{opt}$ `;`

This statement returns from the function. If the function return type is `void`, the function may not return a value; otherwise, the expression represents the value to be returned.

C Reference Tables

This section has some tables and lists of C entities.

Reference Tables

List of Keywords

ANSI C (C89)/ISO C (C90) keywords:

- **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**

Very old compilers may not recognize some or all of the C89 keywords **const**, **enum**, **signed**, **void**, **volatile**, as well as any later standards' keywords.

Keywords added to ISO C99 (supported in most new compilers):

- **_Bool**
- **_Complex**
- **_Imaginary**
- **inline**
- **restrict** (http://en.wikipedia.org/wiki/Restrict)

Keywords added to ISO C11 (supported only in some newer compilers):

- **alignof**
- **_Alignas**
- **_Atomic**
- **_Generic**
- **_Noreturn**
- **_Static_assert**
- **_Thread_local**

Although not technically a keyword, C99-capable preprocessors/compilers additionally recognize the special preprocessor operator **_Pragma**, which acts as an alternate form of the **#pragma** directive that can be used from within macro expansions. For example, the following code will cause some

compilers (incl. GCC, Clang) to emit a diagnostic message:

```
#define EMIT_MESSAGE(str)    EMIT_PRAGMA(message(str))
#define EMIT_PRAGMA(content) _Pragma(#content)
EMIT_MESSAGE("Hello, world!")
```

Some compilers use a slight variant syntax; in particular, MSVC supports **__pragma** instead of **_Pragma**.

Specific compilers may also—in a non-standards-compliant mode, or with additional syntactic markers like **__extension__**—treat some other words as keywords, including **asm**, **cdecl**, **far**, **fortran**, **huge**, **interrupt**, **near**, **pascal**, or **typeof**. However, they typically allow these keywords to be overridden by declarations when operating in standards-compliant modes (e.g., by defining a variable named **typeof**), in order to avoid introducing incompatibilities with existing programs. In order to ensure the compiler can maintain access to extension features, these compilers usually have an additional set of proper keywords beginning with two underscores (__). For example, GCC treats **asm**, **__asm**, and **__asm__** somewhat identically, but the latter two are always guaranteed to have the expected meaning since they can't be overridden.

Many of the newly introduced keywords—namely, those beginning with an underscore and capital letter like **_Noreturn** or **_Imaginary**—are intended to be used only indirectly in most situations. Instead, the programmer should prefer the use of standard headers such as `<stdbool.h>` or `<stdalign.h>`, which typically use the preprocessor to establish an all-lower-case variant of the keyword (e.g., "`complex`" or "`noreturn`"). These headers serve the purpose of enabling C and C++ code, as well as code targeting different compilers or language versions, to interoperate more cleanly. For example, by including `<stdbool.h>`, the tokens "`bool`", "`true`", and "`false`" can be used identically in either C99 or C++ without having to explicitly use **_Bool** in C99 or **bool** in C++.

See also the list of reserved identifiers (http://publib.boulder.ibm.com/infocenter/comphelp/v7v91/topic/com.ibm.vacpp7a.doc/language/ref/clrc02reserved_identifiers.htm).

C character sets

Programs written in C can read and write any character set.

The source program text for those programs, however, is usually limited to the following "basic C source character set":

- Lowercase and uppercase letters: a–z A–Z
- Decimal digits: 0–9
- Graphic characters: ! " # % & ' ( ) * + , - . / : ; < = > ? [\ ] ^ _ { | } ~
- Whitespace characters: space, horizontal tab, vertical tab, form feed, newline.

(In file containing the source program text, the end of a line of text is often not a single newline character, but C compilers treat the end of each line as if it were a single newline character).

Some coding standards mandate that only the above 96 characters are allowed in source code files.[1]

Practically all compilers allow the 3 remaining printable 7-bit ASCII characters '$', '@', and '`'; at least in string literals, but they are not entirely portable. Many compilers allow literal multibyte Unicode characters in string literals, but they are not entirely portable.

A few of the basic C source characters must be encoded to represent them in a string or character constant:

\\

\"

\'

\n

\t

\f

\v

Most compilers allow an even wider "basic C execution character set" to be represented in string literals and otherwise used in at run time: all the characters from the basic C source character set, along with

\r

\a

\b

\xhh , with each h replaced by a hexadecimal character, to portably represent arbitrary bytes (including \x00, the NULL byte)

\uhhhh or \Uhhhhhhhh encoding, with each h replaced by a hexadecimal character, to portably represent Unicode characters.

1. "Joint strike fighter air vehicle C++ coding standards" (http://stroustrup.com/JSF-AV-rules.pdf). section 4.4.2 Character Sets.

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>

Very old compilers may not include some or all of the following headers:

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>

Headers added to ISO C (C11) (Supported only in new compilers):

- <stdalign.h>
- <stdatomic.h>
- <stdnoreturn.h>
- <threads.h>
- <uchar.h>

Table of Operators

Operators in the same row of this table have the same **precedence** and the order of evaluation is decided by the **associativity** (*left-to-right* or *right-to-left*). Operators closer to the top of this table 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]`	
`.`	member access operator *for an object of struct/union type or a reference to it*	`obj.member`	
`->`	member access operator *for a pointer to an object of struct/union type*	`ptr->member`	
	Unary Operators		
`!`	logical not operator	`!eof_reached`	Right to left
`~`	bitwise not operator	`~mask`	
`+ -`[1]	unary plus/minus operators	`-num`	
`++ --`	post-increment/decrement operators	`num++`	
`++ --`	pre-increment/decrement operators	`++num`	
`&`	address-of operator	`&data`	
`*`	indirection operator	`*ptr`	
`sizeof`	sizeof operator *for expressions*	**`sizeof`** `123`	
`sizeof``()`	sizeof operator *for types*	**`sizeof`** `(`**`int`**`)`	
`(`*`type`*`)`	cast operator	`(`**`float`**`)i`	
	Multiplicative Operators		
`* / %`	multiplication, division and modulus operators	`celsius_diff * 9.0 / 5.0`	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		
`||`	see Logical Expressions	`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	Alternative 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 the integral data-type used normally.	**signed**, **signed int**

		■ 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`
`intmax_t`	the maximum width supported by the platform	▪ Can store integers in the range -(1 << n-1)+1 ~ (1 << n-1)-1, with 'n' the width of intmax_t. ▪ Used by the "j" length modifier in the C Programming/File IO#Formatted output functions: the printf family of functions.	—
`uintmax_t`	same as `intmax_t`	▪ Can store integers in the range 0 ~ (1 << n)-1, with 'n' the width of uintmax_t.	—

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 **char**	■ Enumerations are a 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).

Compilers

For a brief introduction to setting up and using some of the more beginner-friendly compilers and IDEs, see Using a Compiler.

Gratis (or with a gratis version)

- Ch_interpreter (http://www.softintegration.com) - The software works in Windows, Linux, Mac OS X, Freebsd, Solaris, AIX and HP-UX. The Ch Standard Edition is free for noncommercial use.
- Interactive C (http://www.botball.org/educational-resources/ic.php).
  - target platform: Handy Board (Freescale 68HC11); Lego RCX
- CINT is an interpreter for C and C++ code, included in the data-analysis package ROOT. The CINT interpreter is licensed under the X11/MIT license. (https://root.cern.ch/drupal/content/cint).
- PicoC (https://github.com/zsaleeba/picoc)
- Extensible Interactive C (EiC) (http://sourceforge.net/projects/eic/)
- lcc-win32 (http://www.cs.virginia.edu/~lcc-win32) - Software copyrighted by Jacob Navia. It is free for non-commercial use. Windows (98/ME /XP/2000/NT).
- GNU Compiler Collection (http://gcc.gnu.org) - GNU Compiler Collection. GNU General Public License / GNU Lesser General Public License.
  - MinGW (http://www.mingw.org/) provides GCC for Windows
- clang (LLVM) (http://clang.llvm.org/) - Almost everywhere but Windows
- Open Watcom (http://www.openwatcom.org) Open Source development community to maintain and enhance the Watcom C/C++ and Fortran cross compilers and tools. Version 1.4 released in December 2005.
  - **Host Platforms:** Win32 systems (IDE and command line), 32-bit OS/2 (IDE and command line), DOS (command line), and Windows 3.x (IDE)
  - **Target Platforms:** DOS (16-bit), Windows 3.x (16-bit), OS/2 1.x (16-bit), Extended DOS, Win32s, Windows 95/98/Me, Windows NT/2000/XP, 32-bit OS/2, and Novell NLMs
  - **Experimental / Development:** Linux, BSD, *nix, PowerPC, Alpha AXP, MIPS, and Sparc v8
- Tiny C Compiler

- Portable C Compiler (http://pcc.ludd.ltu.se) - Portable C Compiler. BSD Style License(s).
- Small Device C Compiler (SDCC)
  - target platforms: Intel 8051-compatibles; Freescale (Motorola) HC08; Microchip PIC16 and PIC18.
- FpgaC. Target platform: FPGA hardware via XNF or VHDL files.
- C compilers for many digital signal processors (DSPs), many of them free, are listed in the comp.dsp FAQ (http://www.bdti.com/faq/3.htm).
- Microsoft Visual C++ (http://msdn.microsoft.com/visualc) - Free (partially limited) version available (Express edition)

Paid

- Intel C Compiler (http://software.intel.com/en-us/intel-compilers) - Windows, Linux, Mac, QNX, and embedded C/C++ compilers. Optimized for Intel 32-bit and 64-bit CPUs.
- Impulse C - Target platform: FPGA hardware via Hardware Description Language (HDL) files.

Retrieved from "https://en.wikibooks.org/w/index.php?title=C_Programming/Print_version&oldid=3070100"

- This page was last modified on 8 April 2016, at 12:16.
- Text is available under the Creative Commons Attribution-ShareAlike License.; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy.