

Objective: -

- Compiling A Single-Source "C" Program
- Compiling A Single-Source "C++" Program
- Basic Data Types and Operators
- C keywords.
- Global and Local variables

2. Compiling "C" Programs On Unix Systems - gcc/g++

1. Preface
2. Compiling A Single-Source "C" Program
3. Running The Resulting Program
4. Creating Debug-Ready Code
5. Creating Optimized Code
6. Getting Extra Compiler Warnings
7. Compiling A Single-Source "C++" Program
8. Compiling A Multi-Source "C" Program
9. Getting a Deeper Understanding - Compilation Steps

Preface

This details below tries to give the reader basic knowledge in compiling C and C++ programs on a Unix system. If you've no knowledge as to how to compile C programs under Unix (for instance, you did that until now on other operating systems), you'd better read this tutorial first, and then write a few programs before you try to get to gdb, makefiles or C libraries.

If you're already familiar with that, it's recommended to learn about makefiles, and then go and learn other C programming topics and practice the usage of makefiles, before going on to read about C libraries. This last issue is only relevant to larger projects, while makefiles make sense even for a small program composed of but a few source files.

As a policy, we'll stick with the basic features of programming tools mentioned here, so that the information will apply to more than a single tool version. This way, you might find the information here useful, even if the system you're using does not have the GNU tools installed.

In this lovely tutorial, we'll deal with compilation of a C program, using the compiler directly from the command line. It might be that you'll eventually use a

more sophisticated interface (an IDE - Integrated Development Environment) of some sort, but the common denominator you'll always find is the plain command line interface. Further more, even if you use an IDE, it could help you understand how things work "behind the scenes". We'll see how to compile a program, how to combine several source files into a single program, how to add debug information and how to optimize code.

Compiling A Single-Source "C" Program

The easiest case of compilation is when you have all your source code set in a single file. This removes any unnecessary steps of synchronizing several files or thinking too much. Lets assume there is a file named 'single_main.c' that we want to compile. We will do so using a command line similar to this:

```
cc single_main.c
```

Note that we assume the compiler is called "cc". If you're using a GNU compiler, you'll write 'gcc' instead. If you're using a Solaris system, you might use 'acc', and so on. Every compiler might show its messages (errors, warnings, etc.) differently, but in all cases, you'll get a file 'a.out' as a result, if the compilation completed successfully. Note that some older systems (e.g. SunOs) come with a C compiler that does not understand ANSI-C, but rather the older 'K&R' C style. In such a case, you'll need to use gcc (hopefully it is installed), or learn the differences between ANSI-C and K&R C (not recommended if you don't *really* have to), or move to a different system.

You might complain that 'a.out' is a too generic name (where does it come from anyway? - well, that's a historical name, due to the usage of something called "a.out format" for programs compiled on older Unix systems). Suppose that you want the resulting program to be called "single_main". In that case, you could use the following line to compile it:

```
cc single_main.c -o single_main
```

Every compiler I've met so far (including the glorious gcc) recognized the '-o' flag as "name the resulting executable file 'single_main'".

Running The Resulting Program

Once we created the program, we wish to run it. This is usually done by simply typing its name, as in:

single_main

However, this requires that the current directory be in our PATH (which is a variable telling our Unix shell where to look for programs we're trying to run). In many cases, this directory is not placed in our PATH. Aha! - we say. Then lets show this computer who is smarter, and thus we try:

```
./single_main
```

This time we explicitly told our Unix shell that we want to run the program from the current directory. If we're lucky enough, this will suffice. However, yet one more obstacle could block our path - file permission flags.

When a file is created in the system, it is immediately given some access permission flags. These flags tell the system who should be given access to the file, and what kind of access will be given to them. Traditional Unix systems use 3 kinds of entities to which they grant (or deny) access: The user which owns the file, the group which owns the file, and everybody else. Each of these entities may be given access to read the file ('r'), write to the file ('w') and execute the file ('x').

Now, when the compiler created the program file for us, we became owners of the file. Normally, the compiler would make sure that we get all permissions to the file - read, write and execute. It might be, thought that something went wrong, and the permissions are set differently. In that case, we can set the permissions of the file properly (the owner of a file can normally change the permission flags of the file), using a command like this:

```
chmod u+rwx single_main
```

This means "the user ('u') should be given ('+') permissions read ('r'), write ('w') and execute ('x') to the file 'single_main'. Now we'll surely be able to run our program. Again, normally you'll have no problem running the file, but if you copy it to a different directory, or transfer it to a different computer over the network, it might loose its original permissions, and thus you'll need to set them properly, as shown above. Note too that you cannot just move the file to a different computer an expect it to run - it has to be a computer with a matching operating system (to understand the executable file format), and matching CPU architecture (to understand the machine-language code that the executable file contains).

Finally, the run-time environment has to match. For example, if we compiled the program on an operating system with one version of the standard C library, and we try to run it on a version with an incompatible standard C library, the program might crush, or complain that it cannot find the relevant C library. This is especially true for systems that evolve quickly (e.g. Linux with libc5 vs. Linux with libc6), so beware.

Creating Debug-Ready Code

Normally, when we write a program, we want to be able to debug it - that is, test it using a debugger that allows running it step by step, setting a break point before a given command is executed, looking at contents of variables during program execution, and so on. In order for the debugger to be able to relate between the executable program and the original source code, we need to tell the compiler to insert information to the resulting executable program that'll help the debugger. This information is called "debug information". In order to add that to our program, lets compile it differently:

```
cc -g single_main.c -o single_main
```

The '-g' flag tells the compiler to use debug info, and is recognized by mostly any compiler out there. You will note that the resulting file is much larger than that created without usage of the '-g' flag. The difference in size is due to the debug information. We may still remove this debug information using the strip command, like this:

```
strip single_main
```

You'll note that the size of the file now is even smaller than if we didn't use the '-g' flag in the first place. This is because even a program compiled without the '-g' flag contains some symbol information (function names, for instance), that the strip command removes. You may want to read strip's manual page (man strip) to understand more about what this command does.

Creating Optimized Code

After we created a program and debugged it properly, we normally want it to compile into an efficient code, and the resulting file to be as small as possible. The compiler can help us by optimizing the code, either for speed (to run faster), or for space (to occupy a smaller space), or some combination of the two. The basic way to create an optimized program would be like this:

```
cc -O single_main.c -o single_main
```

The '-O' flag tells the compiler to optimize the code. This also means the compilation will take longer, as the compiler tries to apply various optimization algorithms to the code. This optimization is supposed to be conservative, in that it ensures us the code will still perform the same functionality as it did when compiled without optimization (well, unless there are bugs in our compiler).

Usually can define an optimization level by adding a number to the '-O' flag. The higher the number - the better optimized the resulting program will be, and the slower the compiler will complete the compilation. One should note that because optimization alters the code in various ways, as we increase the optimization level of the code, the chances are higher that an improper optimization will actually alter our code, as some of them tend to be non-conservative, or are simply rather complex, and contain bugs. For example, for a long time it was known that using a compilation level higher than 2 (or was it higher than 3?) with gcc results bugs in the executable program. After being warned, if we still want to use a different optimization level (lets say 4), we can do it this way:

```
cc -O4 single_compile.c -o single_compile
```

And we're done with it. If you'll read your compiler's manual page, you'll soon notice that it supports an almost infinite number of command line options dealing with optimization. Using them properly requires thorough understanding of compilation theory and source code optimization theory, or you might damage your resulting code. A good compilation theory course (preferably based on "the Dragon Book" by Aho, Sethi and Ulman) could do you good.

Getting Extra Compiler Warnings

Normally the compiler only generates error messages about erroneous code that does not comply with the C standard, and warnings about things that usually tend to cause errors during runtime. However, we can usually instruct the compiler to give us even more warnings, which is useful to improve the quality of our source code, and to expose bugs that will really bug us later. With gcc, this is done using the '-W' flag. For example, to get the compiler to use all types of warnings it is familiar with, we'll use a command line like this:

```
cc -Wall single_source.c -o single_source
```

This will first annoy us - we'll get all sorts of warnings that might seem irrelevant. However, it is better to eliminate the warnings than to eliminate the usage of this flag. Usually, this option will save us more time than it will cause us to waste, and if used consistently, we will get used to coding proper code without thinking too much about it. One should also note that some code that works on some architecture with one compiler, might break if we use a different compiler, or a different system, to compile the code on. When developing on the first system, we'll never see these bugs, but when moving the code to a different platform, the bug will suddenly appear. Also, in many cases we eventually will want to move the code to a new system, even if we had no such intentions initially.

Note that sometimes '-Wall' will give you too many errors, and then you could try to use some less verbose warning level. Read the compiler's manual to learn about the various '-W' options, and use those that would give you the greatest benefit. Initially they might sound too strange to make any sense, but if you are (or when you will become) a more experienced programmer, you will learn which could be of good use to you.

Compiling A Single-Source "C++" Program

Now that we saw how to compile C programs, the transition to C++ programs is rather simple. All we need to do is use a C++ compiler, in place of the C compiler we used so far. So, if our program source is in a file named 'single_main.cc' ('cc' to denote C++ code. Some programmers prefer a suffix of 'C' for C++ code), we will use a command such as the following:

```
g++ single_main.cc -o single_main
```

Or on some systems you'll use "CC" instead of "g++" (for example, with Sun's compiler for Solaris), or "aCC" (HP's compiler), and so on. You would note that with C++ compilers there is less uniformity regarding command line options, partially because until recently the language was evolving and had no agreed standard. But still, at least with g++, you will use "-g" for debug information in the code, and "-O" for optimization.

Compiling A Multi-Source "C" Program

So you learned how to compile a single-source program properly (hopefully by now you played a little with the compiler and tried out a few examples of your own). Yet, sooner or later you'll see that having all the source in a single file is rather limiting, for several reasons:

- As the file grows, compilation time tends to grow, and for each little change, the whole program has to be re-compiled.
- It is very hard, if not impossible, that several people will work on the same project together in this manner.
- Managing your code becomes harder. Backing out erroneous changes becomes nearly impossible.

The solution to this would be to split the source code into multiple files, each containing a set of closely-related functions (or, in C++, all the source code for a single class).

There are two possible ways to compile a multi-source C program. The first is to use a single command line to compile all the files. Suppose that we have a program whose source is found in files "main.c", "a.c" and "b.c" (found in directory "multi-source" of this tutorial). We could compile it this way:

```
cc main.c a.c b.c -o hello_world
```

This will cause the compiler to compile each of the given files separately, and then link them all together to one executable file named "hello_world". Two comments about this program:

1. If we define a function (or a variable) in one file, and try to access them from a second file, we need to declare them as external symbols in that second file. This is done using the C "extern" keyword.
2. The order of presenting the source files on the command line may be altered. The compiler (actually, the linker) will know how to take the relevant code from each file into the final program, even if the first source file tries to use a function defined in the second or third source file.

The problem with this way of compilation is that even if we only make a change in one of the source files, all of them will be re-compiled when we run the compiler again.

In order to overcome this limitation, we could divide the compilation process into two phases - compiling, and linking. Lets first see how this is done, and then explain:

```
cc -c main.c
cc -c a.c
cc -c b.c
cc main.o a.o b.o -o hello_world
```

The first 3 commands have each taken one source file, and compiled it into something called "object file", with the same names, but with a ".o" suffix. It is the "-c" flag that tells the compiler only to create an object file, and not to generate a final executable file just yet. The object file contains the code for the source file in machine language, but with some unresolved symbols. For example, the "main.o" file refers to a symbol named "func_a", which is a function defined in file "a.c". Surely we cannot run the code like that. Thus, after creating the 3 object files, we use the 4th command to link the 3 object files into one program. The linker (which is invoked by the compiler now) takes all the symbols from the 3 object files, and links them together - it makes sure that when "func_a" is invoked from the code in object file "main.o", the function code in object file "a.o" gets executed. Further more, the linker also links the standard C library into the program, in this case, to resolve the "printf" symbol properly.

To see why this complexity actually helps us, we should note that normally the link phase is much faster than the compilation phase. This is especially true when doing optimizations, since that step is done before linking. Now, let's assume we change the source file "a.c", and we want to re-compile the program. We'll only need now two commands:

```
cc -c a.c
cc main.o a.o b.o -o hello_world
```

In our small example, it's hard to notice the speed-up, but in a case of having few tens of files each containing a few hundred lines of source-code, the time saving is significant; not to mention even larger projects.

Getting a Deeper Understanding - Compilation Steps

Now that we've learned that compilation is not just a simple process, let's try to see what is the complete list of steps taken by the compiler in order to compile a C program.

1. Driver - what we invoked as "cc". This is actually the "engine", that drives the whole set of tools the compiler is made of. We invoke it, and it begins to invoke the other tools one by one, passing the output of each tool as an input to the next tool.
2. C Pre-Processor - normally called "cpp". It takes a C source file, and handles all the pre-processor definitions (#include files, #define macros, conditional source code inclusion with #ifdef, etc.) You can invoke it separately on your program, usually with a command like:

```
cc -E single_source.c
```

Try this and see what the resulting code looks like.

3. The C Compiler - normally called "cc1". This is the actual compiler, that translates the input file into assembly language. As you saw, we used the "-c" flag to invoke it, along with the C Pre-Processor, (and possibly the optimizer too, read on), and the assembler.
4. Optimizer - sometimes comes as a separate module and sometimes as the found inside the compiler module. This one handles the optimization on a representation of the code that is language-neutral. This way, you can use the same optimizer for compilers of different programming languages.

5. Assembler - sometimes called "as". This takes the assembly code generated by the compiler, and translates it into machine language code kept in object files. With gcc, you could tell the driver to generate only the assembly code, by a command like:

```
cc -S single_source.c
```

6. Linker-Loader - This is the tool that takes all the object files (and C libraries), and links them together, to form one executable file, in a format the operating system supports. A Common format these days is known as "ELF". On SunOS systems, and other older systems, a format named "a.out" was used. This format defines the internal structure of the executable file - location of data segment, location of source code segment, location of debug information and so on.

As you see, the compilation is split in to many different phases. Not all compiler employs exactly the same phases, and sometimes (e.g. for C++ compilers) the situation is even more complex. But the basic idea is quite similar - split the compiler into many different parts, to give the programmer more flexibility, and to allow the compiler developers to re-use as many modules as possible in different compilers for different languages (by replacing the preprocessor and compiler modules), or for different architectures (by replacing the assembly and linker-loader parts).

Basic Data Types and Operators

The *type* of a variable determines what kinds of values it may take on. An *operator* computes new values out of old ones. An *expression* consists of variables, constants, and operators combined to perform some useful computation. In this chapter, we'll learn about C's basic types, how to write constants and declare variables of these types, and what the basic operators are.

As Kernighan and Ritchie say, "The type of an object determines the set of values it can have and what operations can be performed on it." This is a fairly formal, mathematical definition of what a type is, but it is traditional (and meaningful). There are several implications to remember:

1. The "set of values" is finite. C's int type can not represent *all* of the integers; its float type can not represent *all* floating-point numbers.
2. When you're using an object (that is, a variable) of some type, you may have to remember what values it can take on and what operations you can perform on it. For example, there are several operators which play with the binary (bit-level) representation of integers, but these operators are not meaningful for and may not be applied to floating-point operands.

3. When declaring a new variable and picking a type for it, you have to keep in mind the values and operations you'll be needing.

In other words, picking a type for a variable is not some abstract academic exercise; it's closely connected to the way(s) you'll be using that variable.

2.1 Types

There are only a few basic data types in C. The first ones we'll be encountering and using are:

- `char` a character
- `int` an integer, in the range -32,767 to 32,767
- `long int` a larger integer (up to +-2,147,483,647)
- `float` a floating-point number
- `double` a floating-point number, with more precision and perhaps greater range than `float`

If you can look at this list of basic types and say to yourself, "Oh, how simple, there are only a few types, I won't have to worry much about choosing among them," you'll have an easy time with declarations. (Some masochists wish that the type system were more complicated so that they could specify more things about each variable, but those of us who would rather not have to specify these extra things each time are glad that we don't have to.)

The ranges listed above for types `int` and `long int` are the guaranteed minimum ranges. On some systems, either of these types (or, indeed, any C type) may be able to hold larger values, but a program that depends on extended ranges will not be as portable. Some programmers become obsessed with knowing exactly what the sizes of data objects will be in various situations, and go on to write programs which depend on these exact sizes. Determining or controlling the size of an object is occasionally important, but most of the time we can sidestep size issues and let the compiler do most of the worrying.

(From the ranges listed above, we can determine that type `int` must be at least 16 bits, and that type `long int` must be at least 32 bits. But neither of these sizes is exact; many systems have 32-bit ints, and some systems have 64-bit long ints.)

You might wonder how the computer stores characters. The answer involves a *character set*, which is simply a mapping between some set of characters and some set of small numeric codes. Most machines today use the ASCII character set, in which the letter A is represented by the code 65, the ampersand & is represented by the code 38, the digit 1 is represented by the code 49, the space character is represented by the code 32, etc. (Most of the time, of course, you have *no* need to know or even worry about these particular code values; they're

automatically translated into the right shapes on the screen or printer when characters are printed out, and they're automatically generated when you type characters on the keyboard. Eventually, though, we'll appreciate, and even take some control over, exactly when these translations--from characters to their numeric codes--are performed.) Character codes are usually small--the largest code value in ASCII is 126, which is the ~ (tilde or circumflex) character. Characters usually fit in a byte, which is usually 8 bits. In C, type char is defined as occupying one byte, so it is usually 8 bits.

Most of the simple variables in most programs are of types int, long int, or double. Typically, we'll use int and double for most purposes, and long int any time we need to hold integer values greater than 32,767. As we'll see, even when we're manipulating individual characters, we'll usually use an int variable, for reasons to be discussed later. Therefore, we'll rarely use individual variables of type char; although we'll use plenty of arrays of char.

C keywords.

The following list shows all the ANSI defined C keywords. I have included [sizeof](#) because it looks like a keyword and it keeps the table below tidy....

auto	break	case	char	const	continue		do
double	else	enum	extern		for	goto	if
int	long	register	return		signed	sizeof	static
struct	switch	typedef		unsigned		volatile	while

Examples:

auto - storage class

auto is the default storage class for local variables.

```
{
    int Count;
    auto int Month;
}
```

The example above defines two variables with the same storage class. auto can only be used within functions, i.e. local variables.

register - Storage Class

register is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
    register int Miles;
}
```

Register should only be used for variables that require quick access - such as counters. It should also be noted that defining 'register' goes not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register - depending on hardware and implementation restrictions.

static - Storage Class

static is the default storage class for [global variables](#). The two variables below (**count** and **road**) both have a static storage class.

```
static int Count;
int Road;

main()
{
    printf("%d\n", Count);
    printf("%d\n", Road);
}
```

'static' can also be defined within a function. If this is done, the variable is initialised at compilation time and retains its value between calls. Because it is initialised at compilation time, the initialisation value must be a constant. This is serious stuff - tread with care.

```
void Func(void)
{
    static Count=1;
}
```

Here is an [example](#)

There is one very important use for 'static'. Consider this bit of code.

```
char *Func(void);

main()
{
```

```
char *Text1;
Text1 = Func();
}

char *Func(void)
{
    char Text2[10]="martin";
    return(Text2);
}
```

'Func' returns a pointer to the memory location where 'Text2' starts BUT Text2 has a storage class of [auto](#) and will disappear when we exit the function and could be overwritten by something else. The answer is to specify:

```
static char Text[10]="martin";
```

The storage assigned to 'Text2' will remain reserved for the duration of the program.

static functions

static functions are functions that are only visible to other functions in the same file. Consider the following code.

funcs.c	
<pre>#include main() { Func1(); Func2(); }</pre>	<pre> /***** * * Function declarations (prototypes). * *****/ static void Func1(void); /* Func2 is visible to all functions. */ void Func2(void); /***** * * Function definitions * *****/</pre>

```
void Func1(void)
{
    puts("Func1 called");
}

/*****/

void Func2(void)
{
    puts("Func2 called");
}
```

If you attempted to compile this code with the following command,

```
gcc main.c funcs.c
```

it will fail with an error similar to.....

```
undefined reference to `Func1'
```

Because 'Func1' is declared as *static* and cannot be 'seen' by 'main.c'.

Notes:

For some reason, *static* has different meanings in in different contexts.

1. When specified on a function declaration, it makes the function local to the file.
2. When specified with a variable inside a function, it allows the vairable to retain its value between calls to the function. See [static variables](#).

It seems a little strange that the same keyword has such different meanings....

extern - storage Class

extern defines a global variable that is visable to ALL object modules. When you use 'extern' the variable cannot be initalized as all it does is point the variable name at a storage location that has been previously defined.

Source 1

Source 2

```
extern int count;                int count=5;

write()                          main()
{                                {
    printf("count is %d\n", count);    write();
}
```

Global and Local variables.

Local variables

Local variables must always be [defined](#) at the top of a block.

C++ has changed the rules regarding where you can define a local variable. [Click here](#) for the low down.

When a local variable is [defined](#) - it is not initialised by the system, you must initialise it yourself.

A local variable is [defined](#) inside a **block** and is only visible from within the block.

```
main()
{
    int i=4;
    i++;
}
```

When execution of the block starts the variable is available, and when the block ends the variable 'dies'.

A local variable is visible within nested blocks unless a variable with the same name is [defined](#) within the nested block.

```
main()
{
    int i=4;
    int j=10;

    i++;
}
```

```

        if (j > 0)
        {
            printf("i is %d\n",i);          /* i defined in
'main' can be seen */
        }

        if (j > 0)
        {
            int i=100;                      /* 'i' is defined
and so local to
                                           * this block */
            printf("i is %d\n",i);
        }                                  /* 'i' (value 100)
dies here */

        printf("i is %d\n",i);             /* 'i' (value 5) is
now visable. */
    }

```

Global variables

C++ has enhanced the use of [global variables](#).

Global variables ARE initialised by the system when you [define](#) them!

Data Type	Initialser
int	0
char	'\0'
float	0
pointer	NULL

In the next example **i** is a global variable, it can be seen and modified by **main** and any other functions that may reference it.

```

int i=4;

main()
{
    i++;
}

```

Now, this example has **global** and **Internal** variables.

```

int i=4;          /* Global definition */

main()
{
    i++;          /* global variable */
    func
}

func()
{
    int i=10;     /* Internal declaration */
    i++;          /* Internal variable */
}

```

i in **main** is global and will be incremented to 5. **i** in **func** is internal and will be incremented to 11. When control returns to **main** the internal variable will die and any reference to **i** will be to the global.

static variables can be 'seen' within all functions in this source file. At link time, the static variables defined here will not be seen by the object modules that are brought in.

Example:

```

int counter = 0;          /* global because we are outside
                           all blocks. */
int func(void);

main()
{
    counter++;           /* global because it has not been
                           declared within this block */
    printf("counter is %2d before the call to func\n", counter);

    func();              /* call a function. */
    printf("counter is %2d after the call to func\n", counter);
}

int func(void)
{
    int counter = 10;     /* local. */
    printf("counter is %2d within func\n", counter);
}

```

C Data types.

Variable definition

C has a concept of '*data types*' which are used to [define](#) a variable before its use.

The definition of a variable will assign storage for the variable and define the type of data that will be held in the location.

So what data types are available?

int	float	double	char	void	enum
---------------------	-----------------------	------------------------	----------------------	----------------------	----------------------

Please note that there is not a boolean data type. C does not have the traditional view about logical comparison, but thats another story.

Recent C++ compilers do have a [boolean](#) datatype.

int - data type

int is used to define integer numbers.

```
{
    int Count;
    Count = 5;
}
```

float - data type

float is used to define floating point numbers.

```
{
    float Miles;
    Miles = 5.6;
}
```

double - data type

double is used to define BIG floating point numbers. It reserves twice the storage for the number. On PCs this is likely to be 8 bytes.

```
{
    double Atoms;
    Atoms = 2500000;
}
```

char - data type

char defines characters.

```
{
    char Letter;
    Letter = 'x';
}
```

Modifiers

The three data types above have the following modifiers.

- short
- long
- signed
- unsigned

The modifiers define the amount of storage allocated to the variable. The amount of storage allocated is not cast in stone. ANSI has the following rules:

```
short int <= int <= long int
float <= double <= long double
```

What this means is that a 'short int' should assign less than or the same amount of storage as an 'int' and the 'int' should be less or the same bytes than a 'long int'. What this means in the real world is:

	Type	Bytes	Bits	Range
(32kb)	short int	2	16	-32,768 -> +32,767
(64Kb)	unsigned short int	2	16	0 -> +65,535
(4Gb)	unsigned int	4	32	0 -> +4,294,967,295

	int	4	32	-2,147,483,648 -> +2,147,483,647
(2Gb)				
	long int	4	32	-2,147,483,648 -> +2,147,483,647
(2Gb)				
	signed char	1	8	-128 -> +127
	unsigned char	1	8	0 -> +255
	float	4	32	
	double	8	64	
	long double	12	96	

These figures only apply to today's generation of PCs. Mainframes and midrange machines could use different figures, but would still comply with the rule above.

You can find out how much storage is allocated to a data type by using the [sizeof](#) operator.

Qualifiers

- [const](#)
- [volatile](#)

The *const* qualifier is used to tell C that the variable value can not change after initialisation.

```
const float pi=3.14159;
```

pi cannot be changed at a later time within the program.

Another way to define constants is with the [#define](#) preprocessor which has the advantage that it does not use any storage (but who counts bytes these days?).

Constants

A *constant* is just an immediate, absolute value found in an expression. The simplest constants are decimal integers, e.g. 0, 1, 2, 123. Occasionally it is useful to specify constants in base 8 or base 16 (octal or hexadecimal); this is done by prefixing an extra 0 (zero) for octal, or 0x for hexadecimal: the constants 100, 0144, and 0x64 all represent the same number. (If you're not using these non-decimal constants, just remember not to use any leading zeroes. If you accidentally write 0123 intending to get one hundred and twenty three, you'll get 83 instead, which is 123 base 8.)

We write constants in decimal, octal, or hexadecimal for our convenience, not the compiler's. The compiler doesn't care; it always converts everything into binary internally, anyway. (There is, however, no good way to specify constants in source code in binary.)

A constant can be forced to be of type long int by suffixing it with the letter L (in upper or lower case, although upper case is strongly recommended, because a lower case l looks too much like the digit 1).

A constant that contains a decimal point or the letter e (or both) is a floating-point constant: 3.14, 10., .01, 123e4, 123.456e7. The e indicates multiplication by a power of 10; 123.456e7 is 123.456 times 10 to the 7th, or 1,234,560,000. (Floating-point constants are of type double by default.)

We also have constants for specifying characters and strings. (Make sure you understand the difference between a character and a string: a character is exactly one character; a string is a set of zero or more characters; a string containing one character is distinct from a lone character.) A character constant is simply a single character between single quotes: 'A', '.', '%'. The numeric value of a character constant is, naturally enough, that character's value in the machine's character set. (In ASCII, for example, 'A' has the value 65.)

A *string* is represented in C as a sequence or array of characters. (We'll have more to say about arrays in general, and strings in particular, later.) A string constant is a sequence of zero or more characters enclosed in double quotes: "apple", "hello, world", "this is a test".

Within character and string constants, the backslash character \ is special, and is used to represent characters not easily typed on the keyboard or for various reasons not easily typed in constants. The most common of these "character escapes" are:

\n	a "newline" character
\b	a backspace
\r	a carriage return (without a line feed)
\'	a single quote (e.g. in a character constant)
\"	a double quote (e.g. in a string constant)
\\	a single backslash

For example, "he said \"hi\"" is a string constant which contains two double quotes, and \" is a character constant consisting of a (single) single quote. Notice once again that the character constant 'A' is very different from the string constant "A".

Declarations

Informally, a *variable* (also called an *object*) is a place you can store a value. So that you can refer to it unambiguously, a variable needs a name. You can think of the variables in your program as a set of boxes or cubbyholes, each with a label giving its name; you might imagine that storing a value ``in" a variable consists of writing the value on a slip of paper and placing it in the cubbyhole.

A *declaration* tells the compiler the name and type of a variable you'll be using in your program. In its simplest form, a declaration consists of the type, the name of the variable, and a terminating semicolon:

```
char c;  
int i;  
float f;
```

You can also declare several variables of the same type in one declaration, separating them with commas:

```
int i1, i2;
```

Later we'll see that declarations may also contain *initializers*, *qualifiers* and *storage classes*, and that we can declare *arrays*, *functions*, *pointers*, and other kinds of data structures.

The placement of declarations is significant. You can't place them just anywhere (i.e. they cannot be interspersed with the other statements in your program). They must either be placed at the beginning of a function, or at the beginning of a brace-enclosed block of statements (which we'll learn about in the next chapter), or outside of any function. Furthermore, the placement of a declaration, as well as its storage class, controls several things about its *visibility* and *lifetime*, as we'll see later.

You may wonder *why* variables must be declared before use. There are two reasons:

1. It makes things somewhat easier on the compiler; it knows right away what kind of storage to allocate and what code to emit to store and manipulate each variable; it doesn't have to try to intuit the programmer's intentions.
2. It forces a bit of useful discipline on the programmer: you cannot introduce variables willy-nilly; you must think about them enough to pick appropriate types for them. (The compiler's error messages to you, telling you that you apparently forgot to declare a variable, are as often helpful as they are a

nuisance: they're helpful when they tell you that you misspelled a variable, or forgot to think about exactly how you were going to use it.)

Although there are a few places where declarations can be omitted (in which case the compiler will assume an implicit declaration), making use of these removes the advantages of reason 2 above, so I recommend always declaring everything explicitly.

Most of the time, I recommend writing one declaration per line. For the most part, the compiler doesn't care what order declarations are in. You can order the declarations alphabetically, or in the order that they're used, or to put related declarations next to each other. Collecting all variables of the same type together on one line essentially orders declarations by type, which isn't a very useful order (it's only slightly more useful than random order).

A declaration for a variable can also contain an initial value. This *initializer* consists of an equals sign and an expression, which is usually a single constant:

```
int i = 1;
int i1 = 10, i2 = 20;
```

Variable Names

Within limits, you can give your variables and functions any names you want. These names (the formal term is "identifiers") consist of letters, numbers, and underscores. For our purposes, names must begin with a letter. Theoretically, names can be as long as you want, but extremely long ones get tedious to type after a while, and the compiler is not required to keep track of extremely long ones perfectly. (What this means is that if you were to name a variable, say, supercalafragalisticespialidocious, the compiler might get lazy and pretend that you'd named it supercalafragalisticespialidocio, such that if you later misspelled it supercalafragalisticespialidociouz, the compiler wouldn't catch your mistake. Nor would the compiler necessarily be able to tell the difference if for some perverse reason you *deliberately* declared a second variable named supercalafragalisticespialidociouz.)

The capitalization of names in C is significant: the variable names `variable`, `Variable`, and `VARIABLE` (as well as silly combinations like `variAble`) are all distinct.

A final restriction on names is that you may not use *keywords* (the words such as `int` and `for` which are part of the syntax of the language) as the names of variables or functions (or as identifiers of any kind).

Expressions and Operators

One reason for the power of C is its wide range of useful operators. An operator is a function which is applied to values to give a result. You should be familiar with operators such as +, -, /.

Arithmetic operators are the most common. Other operators are used for comparison of values, combination of logical states, and manipulation of individual binary digits. The binary operators are rather low level for so are not covered here.

Operators and values are combined to form expressions. The values produced by these expressions can be stored in variables, or used as a part of even larger expressions.

-
- Assignment Statement
 - Arithmetic operators
 - Type conversion
 - Comparison
 - Logical Connectors
 - Summary

Assignment Statement

The easiest example of an expression is in the assignment statement. An expression is evaluated, and the result is saved in a variable. A simple example might look like

$$y = (m * x) + c$$

This assignment will save the value of the expression in variable y.

Arithmetic operators

Here are the most common arithmetic operators

- + Addition
- Subtraction
- * Multiplication
- / Division
- % Modulo Reduction (Remainder from integer division)

*, / and % will be performed before + or - in any expression. Brackets can be used to force a different order of evaluation to this. Where division is performed between two integers, the result will be an integer, with remainder discarded.

Modulo reduction is only meaningful between integers. If a program is ever required to divide a number by zero, this will cause an error, usually causing the program to crash.

Here are some arithmetic expressions used within assignment statements.

```
velocity = distance / time;
```

```
force = mass * acceleration;
```

```
count = count + 1;
```

C has some operators which allow abbreviation of certain types of arithmetic assignment statements.

Shorthand	Equivalent
<code>i++;</code> or <code>++i;</code>	<code>i = i + 1;</code>
<code>i--;</code> or <code>--i;</code>	<code>i = i - 1;</code>

These operations are usually very efficient. They can be combined with another expression.

```
x = a * b++; is equivalent to x = a * b;  
b = b + 1;
```

Versions where the operator occurs before the variable name change the value of the variable before evaluating the expression, so

```
x = --i * (a + b); is equivalent to i = i - 1;  
x = i * (a + b);
```

These can cause confusion if you try to do too many things on one command line. You are recommended to restrict your use of ++ and - to ensure that your programs stay readable.

Another shorthand notation is listed below

Shorthand	Equivalent
<code>i += 10;</code>	<code>i = i + 10;</code>
<code>i -= 10;</code>	<code>i = i - 10;</code>
<code>i *= 10;</code>	<code>i = i * 10;</code>
<code>i /= 10;</code>	<code>i = i / 10;</code>

These are simple to read and use.

Type conversion

You can mix the types of values in your arithmetic expressions. char types will be treated as int. Otherwise where types of different size are involved, the result will usually be of the larger size, so a float and a double would produce a double result. Where integer and real types meet, the result will be a double.

There is usually no trouble in assigning a value to a variable of different type. The value will be preserved as expected except where;

- The variable is too small to hold the value. In this case it will be corrupted (this is bad).
- The variable is an integer type and is being assigned a real value. The value is rounded down. This is often done deliberately by the programmer.

Values passed as function arguments must be of the correct type. The function has no way of determining the type passed to it, so automatic conversion cannot take place. This can lead to corrupt results. The solution is to use a method called casting which temporarily disguises a value as a different type.

eg. The function sqrt finds the square root of a double.

```
int i = 256;  
int root
```

```
root = sqrt( (double) i);
```

The cast is made by putting the bracketed name of the required type just before the value. (double) in this example. The result of sqrt((double) i); is also a double, but this is automatically converted to an int on assignment to root.

Questions: -

1. What are steps of Compiling A Single-Source "C" Program?
2. What are steps of Compiling A Single-Source "C++" Program?
3. What are Basic Data Types and Operators keywords, Global and Local variables?