Introduction to C

By Alex Allain

This tutorial is designed to be a stand-alone introduction to C, even if you've never programmed before. However, because C++ is a more modern language, if you're not sure if you should learn C or C++, I recommend the C++ tutorial instead, which is also designed for people who have never programmed before. Nevertheless, if you do not desire some of C++'s advanced features or simply wish to learn C instead of C++, then this tutorial is for you!

Getting set up - finding a C compiler

The very first thing you need to do, before starting out in C, is to make sure that you have a compiler. What is a compiler, you ask? A compiler turns the program that you write into an **executable** that your computer can actually understand and run. If you're taking a course, you probably have one provided through your school. If you're starting out on your own, your best bet is to use Code::Blocks with MinGW. If you're on Linux, you can use gcc, and if you're on Mac OS X, you can use XCode. If you haven't yet done so, go ahead and get a compiler set up--you'll need it for the rest of the tutorial.

Intro to C

Every full C program begins inside a function called "main". A function is simply a collection of commands that do "something". The main function is always called when the program first executes. From main, we can call other functions, whether they be written by us or by others or use built-in language features. To access the standard functions that comes with your compiler, you need to include a header with the #include directive. What this does is effectively take everything in the header and paste it into your program. Let's look at a working program:

```
#include <stdio.h>
int main()
{
    printf( "I am alive!  Beware.\n" );
    getchar();
    return 0;
}
```

Let's look at the elements of the program. The #include is a "preprocessor" directive that tells the compiler to put code from the header called stdio.h into our program before actually creating the executable. By including header files, you can gain access to many different functions--both the printf and getchar functions are included in stdio.h.

The next important line is int main(). This line tells the compiler that there is a function named main, and that the function returns an integer, hence int. The "curly braces" ({ and }) signal the beginning and end of functions and other code blocks. If you have programmed in Pascal, you will know them as BEGIN and END. Even if you haven't programmed in Pascal, this is a good way to think about their meaning.

The printf function is the standard C way of displaying output on the screen. The quotes tell the compiler that you want to output the literal string as-is (almost). The '\n' sequence is actually treated as a single character that stands for a newline (we'll talk about this later in more detail); for the time being, just remember that there are a few sequences that, when they appear in a string literal, are actually not displayed literally by printf and that '\n' is one of them. The actual effect of '\n' is to move the cursor on your screen to the next line. Notice the semicolon: it tells the compiler that you're at the end of a command, such as a function call. You will see that the semicolon is used to end many lines in C.

The next command is getchar(). This is another function call: it reads in a single character and waits for the user to hit enter before reading the character. This line is included because many compiler environments will open a new console window, run the program, and then close the window before you can see the output. This command keeps that window from closing because the program is not done yet because it waits for you to hit enter. Including that line gives you time to see the program run.

Finally, at the end of the program, we return a value from main to the operating system by using the return statement. This return value is important as it can be used to tell the operating system whether our program succeeded or not. A return value of 0 means success.

The final brace closes off the function. You should try compiling this program and running it. You can cut and paste the code into a file, save it as a .c file, and then compile it. If you are using a command-line compiler, such as Borland C++ 5.5, you should read the compiler instructions for information on how to compile. Otherwise compiling and running should be as simple as clicking a button with your mouse (perhaps the "build" or "run" button).

You might start playing around with the printf function and get used to writing simple C programs.

Explaining your Code

Comments are critical for all but the most trivial programs and this tutorial will often use them to explain sections of code. When you tell the compiler a section of text is a comment, it will ignore it when running the code, allowing you to use any text you want to describe the real code. To create a comment in C, you surround the text with /* and then */ to block off everything between as a comment. Certain compiler environments or text editors will change the color of a commented area to make it easier to spot, but some will not. Be certain not to accidentally comment out code (that is, to tell the compiler part of your code is a comment) you need for the program.

When you are learning to program, it is also useful to comment out sections of code in order to see how the output is affected.

Using Variables

So far you should be able to write a simple program to display information typed in by you, the programmer and to describe your program with comments. That's great, but what about interacting with your user? Fortunately, it is also possible for your program to accept input.

But first, before you try to receive input, you must have a place to store that input. In programming, input and data are stored in variables. There are several different types of variables; when you tell the compiler you are declaring a variable, you must include the data type along with the name of the variable. Several basic types include char, int, and float. Each type can store different types of data.

A variable of type char stores a single character, variables of type int store integers (numbers without decimal places), and variables of type float store numbers with decimal places. Each of these variable types - char, int, and float - is each a keyword that you use when you declare a variable. Some variables also use more of the computer's memory to store their values.

It may seem strange to have multiple variable types when it seems like some variable types are redundant. But using the right variable size can be important for making your program efficient because some variables require more memory than others. For now, suffice it to say that the different variable types will almost all be used!

Before you can use a variable, you must tell the compiler about it by declaring it and telling the compiler about what its "type" is. To declare a variable you use the syntax <variable type> <name of variable>;. (The brackets here indicate that your replace the expression with text described within the brackets.) For instance, a basic variable declaration might look like this:

```
int myVariable;
```

Note once again the use of a semicolon at the end of the line. Even though we're not calling a function, a semicolon is still required at the end of the "expression". This code would create a variable called myVariable; now we are free to use myVariable later in the program.

It is permissible to declare multiple variables of the same type on the same line; each one should be separated by a comma. If you attempt to use an undefined variable, your program will not run, and you will receive an error message informing you that you have made a mistake.

Here are some variable declaration examples:

```
int x;
int a, b, c, d;
char letter;
float the_float;
```

While you can have multiple variables of the same type, you cannot have multiple variables with the same name. Moreover, you cannot have variables and functions with the same name.

A final restriction on variables is that variable declarations must come before other types of statements in the given "code block" (a code block is just a segment of code surrounded by { and }). So in C you must declare all of your variables before you do anything else:

Wrong

```
#include <stdio.h>
int main()
{
    /* wrong!  The variable declaration must appear first */
    printf( "Declare x next" );
    int x;
```

```
    return 0;
}
```

Fixed

```
#include <stdio.h>
int main()
{
    int x;
    printf( "Declare x first" );

    return 0;
}
```

Reading input

Using variables in C for input or output can be a bit of a hassle at first, but bear with it and it will make sense. We'll be using the scanf function to read in a value and then printf to read it back out. Let's look at the program and then pick apart exactly what's going on. You can even compile this and run it if it helps you follow along.

```
#include <stdio.h>

int main()
{
    int this_is_a_number;

    printf( "Please enter a number: " );
    scanf( "%d", &this_is_a_number );
    printf( "You entered %d", this_is_a_number );
    getchar();
    return 0;
}
```

So what does all of this mean? We've seen the #include and main function before; main must appear in every program you intend to run, and the #include gives us access to printf (as well as scanf). (As you might have guessed, the io in stdio.h stands for "input/output"; std just stands for "standard.") The keyword int declares this_is_a_number to be an integer.

This is where things start to get interesting: the scanf function works by taking a string and some variables modified with &. The string tells scanf what variables to look for: notice that we have a string containing only "%d" -- this tells the scanf function to read in an integer. The second argument of scanf is the variable, sort of. We'll learn more about what is going on later, but the gist of it is that scanf needs to know where the variable is stored in order to change its value. Using & in front of a variable allows you to get its location and give that to scanf instead of the value of the variable. Think of it like giving someone directions to the soda aisle and letting them go get a coca-cola instead of fetching the coke for that person. The & gives the scanf function directions to the variable.

When the program runs, each call to scanf checks its own input string to see what kinds of input to expect, and then stores the value input into the variable.

The second printf statement also contains the same '%d'--both scanf and printf use the same format for indicating values embedded in strings. In this case, printf takes the first argument after the string, the variable this_is_a_number, and treats it as though it were of the type specified by the "format specifier". In this case, printf treats this_is_a_number as an integer based on the format specifier.

So what does it mean to treat a number as an integer? If the user attempts to type in a decimal number, it will be truncated (that is, the decimal component of the number will be ignored) when stored in the variable. Try typing in a sequence of characters or a decimal number when you run the example program; the response will vary from input to input, but in no case is it particularly pretty.

Of course, no matter what type you use, variables are uninteresting without the ability to modify them. Several operators used with variables include the following: *, -, +, /, =, ==, >, <. The * multiplies, the / divides, the - subtracts, and the + adds. It is of course important to realize that to modify the value of a variable inside the program it is rather important to use the equal sign. In some languages, the equal sign compares the value of the left and right values, but in C == is used for that task. The equal sign is still extremely useful. It sets the value of the variable on the left side of the equals sign equal to the value on the right side of the equals sign. The operators that perform mathematical functions should be used on the right side of an equal sign in order to assign the result to a variable on the left side.

Here are a few examples:

```
a = 4 * 6; /* (Note use of comments and of semicolon) a is 24 */
```

```
a = a + 5; /* a equals the original value of a with five added to it */
a == 5     /* Does NOT assign five to a. Rather, it checks to see if a equals 5.*/
```

The other form of equal, ==, is not a way to assign a value to a variable. Rather, it checks to see if the variables are equal. It is extremely useful in many areas of C; for example, you will often use == in such constructions as conditional statements and loops. You can probably guess how < and > function. They are greater than and less than operators.

For example:

```
a < 5  /* Checks to see if a is less than five */
a > 5  /* Checks to see if a is greater than five */
a == 5 /* Checks to see if a equals five, for good measure */
```

If you're having some trouble following the tutorial, try some expert help.
Quiz yourself
Next: If Statements
Back to C Tutorial Index

Want something in hard copy? I highly recommend C Programming: A Modern Approach, 2nd Edition: