Getting set up – Programming language C

C is a programming language of many different dialects, similar to the way that each spoken language has many different dialects. In C, dialects don't exist because the speakers live in the North or South. Instead, they're there because there are many different compilers that support slightly different features. There are several common compilers: in particular, Borland C++, Microsoft C++, and GNU C. There are also many front-end environments for the different compilers--the most common is Dev-C++ around GNU's G++ compiler. Some, such as GCC, are free, while others are not. Please see the compiler listing for more information on how to get a compiler and set it up. You should note that if you are programming in C on a C++ compiler, then you will want to make sure that your compiler attempts to compile C instead of C++ to avoid small compatibility issues in later tutorials.

Each of these compilers is slightly different. Each one should support the ANSI standard C functions, but each compiler will also have nonstandard functions (these functions are similar to slang spoken in different parts of a country). Sometimes the use of nonstandard functions will cause problems when you attempt to compile source code (the actual C code written by a programmer and saved as a text file) with a different compiler. These tutorials use ANSI standard C and should not suffer from this problem; fortunately, since C has been around for quite a while, there shouldn't be too many compatibility issues except when your compiler tries to create C++ code.

If you don't yet have a compiler, It is strongly recommended finding one now. A simple compiler is sufficient for our use, but make sure that you do get one in order to get the most from these tutorials. The page linked above, compilers, lists compilers by operating system.

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>
 #include <conio.h>
int main()
{
  printf( "I am alive!  Beware.\n" );
  getch();
  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 headers called stdio.h and conio.h into our program before actually creating the executable code. By including header files, you can gain access to many different functions-- the printf( ) function is included in stdio.h, but the getch( ) function is included in conio.h. The semicolon is part of the syntax of C. It tells the compiler that you're at the end of a command. You will see later that the semicolon is used to end most commands in C.

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. 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. Again, notice the semicolon: it is added onto the end of all lines, such as function calls, in C.
The next command is getch( ). This is another function call: it reads in a single character of key passed by user. 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 any button. 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.cpp 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.
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 = a + 5; /* a equals the original value of a with five added to it
*/

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