Program Structure and Flow

Programming a computer may seem difficult, but it is a logical, step by step process. Every application has three sections: input, process, output. As you may expect, each of those three steps can take on different forms. The input may come from the user actively entering data. Or the input may be read from a pre-existing file. Or it may come from randomly created variables, or simply assigned to the applications variables at compile time. The middle part, processing, is where much of the work normally occurs in any program. The input data may be sorted on an attribute, then a report is created by selecting records with certain characteristics. Output can be an answer on the command line, a stack of paper in your printer's tray, or an active graphic display.

There are only three types of program paths: sequential, iterative, or bifurcative with an if then statement. Every line of code, in even complex programs, can be described with these three paths. Sequential flow is just as it sounds, one command followed by another, then by another. Iterative program flow is when a command, or a series of commands, need to be repeated for a set number of times. The third type of flow, if then, is when a variable is asked its value while the answer determines the path.

In the C/C++ language, iteration can use any of three command structures: a for loop, a do while loop, or a while loop.

```
for (i=0; i<25; i++)   {...}
```

is a for loop on the integer value i. All the commands between the opening brace and the closing brace are processed for i = 0 to i = 24. The commands between the braces can use i as in index variable or you can just process those commands 25 times because you want to.

The while loop has two variations:

```
do  {...}  while ( some statement is true )
```

or

```
while ( some statement is true )  {...}
```

Depending on the logic of your application, you place the test after the command list, or before it.

You can write any application using these three types of program flow. The problem you want to solve determines how you order your commands, in any program you write. However, we can write a template, and some pseudo code, to get you started:

input

get input from the user, from a file, or from within the code

process

massage the data into a form which can be used to write a report

output

display the massaged data on the screen, print the output on paper, or save the report in a file

Writing code is really that simple. Once you have the structure of a program set up, writing the commands to do what you want is just a matter of thinking and typing. The structure is necessary, as is a list of libraries of the functions you will use in a program, any variables which may be used, and a main() function. The function main() is how the operating system and the application connect to each other. The OS handles the keyboard, graphics, memory, files, and many other tasks. Your application simply calls those processes, and exchanges information.

A simple example would be good.

```
// This is my application code
#include <stdio.h>

// global variables here

int main(void)
   {
   int j;

   for (j = 0; j<25; j++)
      {
      printf("This is line %d\n", j);
      }
   return (0);
   }
```

The printf() function is one way to print output to the computer's display. Your application needs to add the library stdio.h, for the compiler to know how to use the printf() function. That is why the line #include <stdio.h> is near the very beginning of your application code. Next comes the main() function. The void inside the parentheses means main() does not require any input to work. The int before main() declares the return value of the function. If main() functions properly, it should return a zero telling us there was no error. The return(0) command tells the OS, and anyone else, there was no problem with running our app.

Everything inside of the brace immediately after main(void), and the very last brace, is contained in your program. The variable int j is declared next, so it can be used later in our for loop. A standard for loop is used, with j set to zero, looping until j is equal to 25. j++ tells the loop to add one to j, each time around the loop. The brace right after the for loop command, and the closing brace right before the return(0) statement, contain your loop's code. In this instance a sentence is printed to the screen, “This is line 0” followed by “This is line 1” and so on up to “This is line 24” as you iterate through the for loop. Once the loop is done, you send the return(0) command and hit the closing brace, which ends your program and returns control to the command line.

You may have noticed j, the index for the for loop, starts at zero and grows from there. There is a simple reason why programmers start counting with zero. Often you are indexing a structure located in memory. Think of the index variable looking at various spots in memory, as the index increases. When the index is equal to zero you are looking at the base location of the structure. As you increase the index you're looking at the next, then the next, then the next location in memory. So using zero as your index reference simply means you are looking at the beginning, with no offset. Then the index points to 1, offsetting your memory location by the size of your structure. Zero means no offset, while any other value gives us the amount to offset from the very beginning. If we had started with one, we would be wasting the first chunk of memory. So zero means look here, while one means look at the next memory chunk whose address has been offset by one unit of memory.

Our simple example did not show the two ways you can order your program. You can put the main() function right after the comments, #includes, #defines, and your function declarations. Or you can put the main() function at the end of the program file, with the function definitions preceding it. If you place your main() function first you MUST declare each function you use, either in one of the #include files, or explicitly before you can enter main(). If you put main() first, and don’t declare your functions, your compiler will not know anything about them and give you a lot of errors. But if you place the function definitions before your main() function there is no need to explicitly declare them, since that is done implicitly from their definition.

I write my main file either way and change the order as the program matures. It is nice to have the function you are currently debugging close to the top, be it the main() function, or one of your own functions. That way you can get to work on it quickly when you start the day. Once I have thoroughly worked out the kinks, I will write a declaration, and put the function toward the end of the file. When you have built up a common list of functions, along with their declarations, you can export them to their own file. Then you add a #include statement to use those functions in your main file.

Building your own libraries is very useful. You know everything in the code since you wrote it and debugged it. You know it works for the same reason. Collecting it into one file, so you can access it with an #include statement, shortens your main file considerably. This makes for a cleaner view of the system, and lets you follow the logic more easily. Once you are sure the code is mature, you can create a .dll from it in Windows, or a .so or .a file in Linux. Archived libraries (*.a) are statically linked when you compile your code. Shared libraries (*.so) are dynamically linked during runtime. Under Windows the .dlls are also linked dynamically at runtime, or can be statically linked at compile time.