

## Chapter 5: The Basics of a C Program

It's time to start writing C programs! The good news is that you can start writing programs that do useful things right away.

You are going to write your C programs using vi as your text editor. Your programs will all end in .c, so typically you will start working on your programs with a Unix command such as:

```
vi windchill.c
```

### What Every C Program Needs:

Every C program needs to start with the following two lines:

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

All these two lines do is attach (or "include") the commands and functions from two other files (stdio.h and math.h) to your program when it is time to compile your program. These "header" files contain information that every C program needs for its input-output and mathematical functions to work properly.

Execution of the C program begins with the next required block of code, which looks like this:

```
main () {
}
```

Everything the program does needs to be between the opening and closing curly braces of the "main" function.

In fact, that could be it! You really could just have a program that looks like this:

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

main () {
}
```

And what would this program do? Absolutely nothing! Since that isn't terribly interesting, let's learn a few more commands before we learn how to make this program run.

## Writing Output to the Screen

It is a grand old tradition in computer programming that your first program should just print the words "Hello World!" to the computer screen. And who am I to argue with tradition?

The command that writes text to the screen is `printf`. The `printf` command is a "function" in C--that means that it requires some information in order to work properly. In the case of the `printf` function, the information that it needs is the text that you want to print to the screen. Here is an example of the `printf` command in action:

```
printf ("Hello World!\n");
```

This is the `printf` command in its purest form, so we should spend a few moments examining it.

- Notice that the information that the `printf` function needed (i.e, "Hello World!\n") is inside parentheses. All functions work this way.
- Notice that the `printf` command ends with a semicolon. All C commands end in a semicolon--that's how the compiler knows when it has reached the end of a command.
- Notice that the line printed to the screen ends with a `\n`. This is the "newline" character, and it tells the computer to move to the next line after printing the text.

The `\n` character can be easy to forget when you are creating your programs. Consider the following examples:

Example 1:

```
printf ("Hello\n");  
printf ("World!\n");
```

Example 1 output:

```
Hello  
World!
```

Example 2:

```
printf ("Hello \nWorld!\n");
```

Example 2 output:

```
Hello  
World!
```

Example 3:

```
printf ("Hello ");  
printf ("World!\n");
```

Example 3 output:

```
Hello World!
```

I cannot stress enough the importance of understanding the use of the `printf` function--and especially the use of the `\n` character. Make sure that you understand the three examples provided!

## A First Program in C

Our first little program in C should be a "Hello World!" program--and I promise that this is the first, last and only time in the entire course that we do something cheesy like this just because the Computer Science textbooks say that we should! So, create the following text in `vi` and call the file `hello.c`:

```
#include <stdio.h>  
#include <math.h>  
  
main () {  
    printf ("Hello World!\n");  
}
```

Once you have this text entered in `vi`, go back to the Unix command line (i.e., write and quit in `vi`). It's time to compile your first program.

## Compiling

You've probably heard the expression that someone was going to "compile a program" or that his or her computer program "wouldn't compile", but you probably didn't know what that meant before this.

Computers are incredibly stupid machines. They know practically nothing. They know how to *do* practically nothing. Everything that they know how to do has to be done in incredibly tiny steps. For example, even a ridiculously simple command such as this one:

```
printf ("Hello World!\n");
```

is much too complicated for the computer to process directly. Rather, this seemingly simple command gets broken down into literally dozens of astonishingly tiny steps that you and I wouldn't have even thought of, such as moving the cursor to the bottom of the window, scrolling the window up one line, clearing out the bottom line before writing the text, loading the appropriate font, drawing the letter "H", moving the cursor a space to the right, drawing the letter "e", and so on. The compiler "compiles" all of the little tiny

"baby steps" that the computer is going to use to complete the task that you asked it to do in the C program.

Computers work in a separate computer language called "machine language". The actual operations and calculations that the computer performs in machine language are incredibly tedious--there might be literally thousands of small, machine language steps involved in something as seemingly simple as the computation of  $\cos(45^\circ)$ .

That's where languages like C come in. The computer doesn't know anything about C--the "printf" function is *not* something the computer knows. But the *compiler*, on the other hand, knows how to translate C commands like "printf" into the necessary and tedious machine language commands which the computer *does* know. (Make sure that you understand what a compiler does!)

Therefore, in order for your C program to work, you are going to have to compile it, and this is done in Unix with the gcc command (which stands for "GNU C Compiler"):

```
gcc -lm hello.c
```

(The -lm part of the command "links to the math libraries" in C. This particular program doesn't need those libraries, but most C programs will, so it's a good habit to always include that "flag".)

Assuming that everything is okay in your program--and that's a big assumption--after your program compiles you will have a file in your directory called a.out. By default, gcc creates this file, which is the "executable" file for this C program. You can run the program then by typing:

```
a.out
```

If everything worked, you should see the following output:

```
Hello World!
```

But, sadly, in life few things work the first time, and this will probably be no exception. The compiler can only successfully compile your program if there are no "syntax" errors in it. Every line must be perfect. That doesn't mean that the program has to necessarily *work*--it just means that every individual line of the program has to contain no errors whatsoever. Fortunately, if there are errors, gcc is more than willing to report those errors to you.

Often, gcc will create a huge list of errors, sometimes literally hundreds of errors. In general, you should look into the very *first* error that gcc reports, fix it, and try compiling again. (Often after the first error, gcc will get confused and report errors in lines that are fine, so starting with the first error is a good strategy.) Finding the errors in the original program can be frustrating, but gcc helps as much as possible by telling you the line that the error occurs on, and even suggesting what the problem might be. (Remember, you can use vi to hop right to a specific line of the program; if gcc reports that there is an error on line 87, in vi's command mode type 87G and your cursor will move to the 87th line.)

## Variables

A program that could only print text on the screen wouldn't be terribly interesting. Most useful programs need to be able to work with information, perform calculations, and so on. These operations will be computed on "variables".

Everyone in this class has taken lots of math courses, and so we are all generally familiar with the concept of working with variables. We all know that variables are like named boxes that can contain some number. Variables in C can have any name that you want: `x`, `temperature`, `TemperatureInCelsius`, whatever. (The names of the variables *are* case-sensitive, so be careful with the use of capitalization!)

Before you use a variable, you have to "declare" the variable. Variable declaration helps the gcc compiler know that the word it is looking at is a variable. For example, suppose you wanted to work with a variable called "pressure". The gcc compiler wouldn't have the slightest idea what the word "pressure" means if the variable weren't declared. (Therefore, gcc would claim that there was a syntax error.)

When you declare a variable, you have to give it a "type". In math classes, a variable like `x` can hold any number that you want it to hold. In C programming, however, there are different variable types, and you have to use the right type. The two most common variable types are:

- `int`--the variable is going to be an integer
- `float`--the variable is going to be a real number (i.e, a "floating point" number)

The reason the computer needs to know the type of the variable is that integers and floating point numbers are stored differently in memory. In general, these two types of variables can't be used together easily. For example, you can't add an integer to a float, divide a float by an integer, etc. That might seem really restricting, but you'd be surprised how rarely this is a problem. In practice, we tend to use these two types of variables in two different ways--integers are for counting, and floats are for measurements and calculations.

Variables are declared at the beginning of a function, which for now means that they should be declared right after the `main()` function begins. They need to be declared before any line of the program is executed. Here is a simple program with some examples of the proper use of variables:

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

main () {
    int i,j;
    float x,y,z;

    x = 1.5;
    y = 2.;
    z = x + y;

    i = 3;
    j = i * 2;
```

```
}
```

Notice the declaration of variables in this program--it occurs right after the main() line, before any of the operations of the program take place. You can see that the variables i and j were declared to be integers, while the variables x, y and z were declared to be floats. Inside the program, x was set to 1.5. Notice that y was set to a value of 2.0. (Actually, y was set to a value of "2.", which is the same thing.) The float "2." and the integer "2" are not the same thing--remember, floats and integers are stored differently in memory! Similarly, notice that the integer i was set to "3", and then j was set to "i \* 2"--not "i \* 2.0"!

## Mathematical Operations

All of the standard arithmetic operations work in C: + - \* /. To raise a number to some power, you use the pow function. For example,  $5.4^3$  would be written:

```
pow(5.4, 3)
```

Other important mathematical functions include square root (sqrt), sine (sin), cosine (cos), tangent (tan), exponential (exp), and natural logarithm (log). Parentheses are used just as they are in math classes.

The same "order of operations" applies in C as it does in math:

- First evaluate exponents.
  - Then evaluate multiplication and division.
  - Then evaluate addition and subtraction.
- And, of course, parentheses trumps the order of operations--evaluate the innermost sets of parentheses first.

Therefore, according to the order of operations in C:

```
2.0 + 4.0 * 5.0
```

is equal to 22.0,

```
(2.0 + 4.0 ) * 5.0
```

is equal to 30.0, and

```
2.0 + 5.0 * pow (3.0, 1.0 + 1.0)
```

is equal to 47.0. Make sure that you understand the order of operations.

## Formatting Output with printf

Most of your computer programs are going to need to print some output to the screen. Most of the time, this is going to involve *formatting* your output—determining what the output should look like on the screen.

Suppose that we wanted to print the current wind speed. Let's assume that we are working with the wind speed as an integer. A simple program that would do this would look like this:

```
int wspd;

wspd = 10;

printf ("The wind speed is %d knots.\n", wspd);
```

By now, none of this should seem too mysterious until you get to the printf function. Here we see that the printf function now has two “arguments”, separated by a comma. The first argument is a string that reads “The wind speed is %d knots.\n”, and the second argument is the wind speed variable, wspd. The %d is a *format character* in the string; when the program runs, the %d will be replaced with the value of wspd. The resulting output looks like this:

```
The wind speed is 10 knots.
```

The %d format is specifically for integers. Another important format symbol is %f, which is used for floating point numbers:

```
float wspd;

wspd = 10.0;

printf ("The wind speed is %f knots.\n", wspd);
```

Here we see that now I'm using wspd as a floating point number, so I need to print this variable using the %f format. However, exactly what you'll see on the screen is a little bit unpredictable when you just use the %f format. On some machines, you would see this:

```
The wind speed is 10. knots.
```

On others:

```
The wind speed is 10.0 knots.
```

Neither of which is too bad, but some computers would produce this output:

```
The wind speed is 10.000000000000 knots.
```

which is considerably less pleasant. To maintain control of what the output is going to look like, you'll need to specify the number of digits that you want to see in the floating point number. Here is an example of code that does this:

```
float wspd;  
  
wspd = 10.0;  
  
printf ("The wind speed is %4.1f knots.\n",wspd);
```

(The changes in the code have been underlined.) This code would produce the following output:

```
The wind speed is 10.0 knots.
```

Why the difference? It's all about the "%4.1f" format that I used. The "4" means that the floating point number should be four characters wide (including the decimal point!), and the 1 means that there should be one digit after the decimal point. This is a very powerful means of controlling the appearance of your output. If I had used the %10.6f format, the output would look like this:

```
The wind speed is 10.000000 knots.
```

And if I had used the %3.0f format, it would have looked like this:

```
The wind speed is 10. knots.
```

## Using a makefile

Keeping track of how to compile your programs can be fairly tricky. Not only is there the fairly unintuitive "gcc" command, but there are flags that are needed like -lm and -lX11, which are very easy to forget (note: we'll learn about the -lX11 flag when we learn about graphics in C). Additionally, problems will arise as you create programs that might have libraries of functions, which will be stored in a separate file than the main program. All of this can be solved using a UNIX invention called the makefile.

When you write a makefile, you are establishing rules about when and how to recompile your programs. The name of your makefile should be "makefile", and you will create the makefile in vi, just as you do programs. Once you have a working makefile, you can easily recompile your programs by just typing "make" and the name of the program. Here is an example of a simple makefile:

```
windchill: windchill.c  
gcc -lm windchill.c -o windchill
```

The first line is telling us that we are establishing rules for recompiling a program called "windchill". After the colon, we will list the files that the program "windchill" is dependent on. In this case, the only file that contains code for the windchill program windchill.c

Whenever you type:

```
make windchill
```

the makefile will check to see if there are any changes in the windchill.c program. If there have been changes, it will execute the compile command shown on the following line. (Important note: for reasons that have never been clear to me, there needs to be a TAB--not a space--before the gcc command.) In this case, the gcc compiler will compile the windchill.c program, link it to the math libraries (-lm), and write the binary executable output as "windchill".

The true usefulness of a makefile is revealed when the code for your programs is broken up into several different files. For example, later this semester you will have a group of functions that compute various meteorological parameters. I will encourage you to put these functions in a separate file called metcalc.c. Similarly, there will be a number of graphical functions in a separate file, which we'll suppose that we will call mygraphics.c. A useful makefile in this situation would look something like this:

```
1      metcalc.o: metcalc.c
2          gcc -lm metcalc.c -o -c metcalc.c
3
4      mygraphics.o: mygraphics.c
5          gcc -lm -lX11 -c mygraphics.c -o mygraphics.o
6
7      wxmap: wxmap.c mygraphics.o metcalc.o
8          gcc -lm -lX11 wxmap.c mygraphics.o metcalc.o -o wxmap
```

This makefile would help you compile a program called "wxmap". If you type:

```
make wxmap
```

at the command line, the makefile will check line 7 and see that the wxmap program is dependent on the code in wxmap.c, but also uses the mygraphics.o and metcalc.o files. These files are libraries of C code (which *you* wrote!) that have already been compiled. As long as there haven't been any changes to mygraphics.c and metcalc.c (notice the .c), the makefile will compile wxmap using line 8. But, let's say that there have been changes to the metcalc.c library--maybe you found a bug and are fixing the code. The makefile will notice that there is a change, and first jump to line 1, which tells us that the compiled binary code metcalc.o depends on metcalc.c. By compiling metcalc.c (line 2) into metcalc.o, we now have a completely up-to-date version of the metcalc.c libraries compiled and ready to go. Now execution can pass to the rule on line 8, completing the compilation of wxmap using the new and improved version of metcalc.c.

The benefits of using a makefile are many. For one thing, it's just a lot easier to only set up the compilation of your programs once, and then just keep using the make command. (Typing the entire gcc command with all of its flags time after time would be very error-prone and tedious.) Additionally, it's more efficient. If the code for metcalc.c has already been compiled and hasn't changed since the last time it was compiled, there's no reason for the computer to waste its time compiling it again. None of the programs that we write in this course are all that big, so compiling them doesn't take the computer all that much work. However, imagine what it takes to compile a very large computer program, such as Microsoft Word! Somewhere in Redmond, Washington, someone has a computer with a makefile that describes how to compile Word from its source code. Word is probably millions of lines long, contained in probably hundreds of "libraries". Even a big computer probably needs a great deal of time to completely recompile Word from scratch. On the other hand, there is probably little reason for the guys at Microsoft to routinely recompile much of Word--the functions that control features like underlining and printing and spell-checking have been working for 15 years or more, so surely no one is making changes to these functions anymore. Therefore, the makefile for Word presumably only recompiles the parts of the code that have changed, just as your makefile will.

I'm not going to make you use a makefile (no pun intended). There will be no assignments in which I'm asking to see your makefile. On the other hand, compiling your programs is going to be a lot easier if you do. For more information about makefiles, search the internet or see me.

### Assignment 3: Your first C program

Write a C program that performs the following calculations and prints the results:

- Convert 15.3°C to Fahrenheit.
- Convert 43.1°F to Celsius.
- Compute the saturation vapor pressure in millibars for an air temperature of 303K.
- Compute the potential temperature (in Kelvin) of air that has a temperature of 303K and a pressure of 900mb.
- Compute the potential temperature in the previous problem to Celsius and Fahrenheit.
- Convert a wind speed of 15 knots to MPH.

(All of the formulas that you need are in the “Common Meteorological Calculations” appendix to this book.)

To receive full credit, the program must be “documented”.

Completing the assignment shown above is worth 85%. To receive a grade higher than 85%, the student will need to “impress me”. Format your output neatly. Document your program particularly well and neatly.

This assignment is worth 5 points.