Some Computer Preliminaries

Before we get started, let's look at some basic components that play a major role in a computer's ability to run programs:

1) Central Processing Unit: The "brains" of the computer that actually execute all the instructions the computer runs.

2) Main Memory: All running programs and data they use must be stored here. This is only active when the computer is physically on.

3) External Memory: This is more permanent memory, such as a hard-drive, floppy disk, CD, etc. In order for any programs stored on these medium to run, they have to be transferred into main memory.

4) Operating System: Provides an interface between the machine and the user and allocates the computer's resources.

5) Compiler: A "translator" which converts a program written in a high-level computer language into machine language, a set of 0s and 1s understood by the computer.

6) Application Programs: Programs intended to be used by end-users on a computer.

How everything fits together

When you boot up your computer, the first major thing that occurs, is that the operating system gets loaded into main memory.

This program (the operating system), is responsible for providing the user with a nice interface, and responding to the requests of the user.

For example, if the user double clicks on the Netscape icon, this invokes the computer to find the Netscape program stored on the hard drive and load it into main memory. Once the program is loaded into main memory, if no other program needs to be run, then it becomes the current program running. Any data this program needs to execute must also be loaded into main memory. All executable programs are stored in machine language, which is understood by the machine.

In this class, our goal will be to learn how to write some simple programs in a high-level language, C, that can be compiled into executable programs.

The vast majority of the work you all will do for this class will follow the cycle pictured on page 3 of the text. This process can be listed as follows:

1) Edit your program
2) Compile your program
3) Execute your program
4) Go to step 1, if you are not satisfied with your results.

The key here is to note that changing your program doesn't automatically change how it runs. Compilation is necessary.

Our first C Program

```
/* Arup Guha
   My First C Program
   8/26/04
   COP 3223 Rocks! */

#include <stdio.h>

int main(void) {

  printf("COP 3223 Rocks!\n");
  return 0;
}
```

Let's take this program apart line by line:

```
/* Arup Guha
   My First C Program
   8/26/04
   COP 3223 Rocks! */
```

This is a comment, which is ignored by the compiler. The purpose of a comment is to help the reader identify key information about the program. The first comment in a program, the header comment, should identify the author of the program, the date the program was written, as well as a brief description of the program. We will talk about other types of comments later. The computer knows that this portion of the code is a comment because it starts with /*. Everything after these two characters until the two characters */ are encountered is ignored by the compiler and treated like a comment.

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

This line is a directive to include a standard C header file. There are some pre-written functions in C that we commonly use. All the functions that control standard input and output are included in the file stdio.h. When we include this file, we are allowed to use these functions. These functions allow us to print information to the screen and read in information from the user. Nearly all of your programs will include this line. (There are many other files which can be included in for your programs as well.)

```
int main(void) {
```

This line signifies the beginning of the function main. All code in C resides in functions. You can think of functions as all being separate little programs. main is a special function. It is the only function that automatically EXECUTES. A program may be made up on many functions, but the computer only DIRECTLY executes main. In order for the computer to execute any other function, main must give instructions to call those functions. All functions are signified by a return type, (which is int for this function), a name (which is main for this function), and a paramter list in parentheses (which is void for this function). After this listed information, the curly brace { signifies the beginning of the function. A corresponding curly brace } ends the definition of the function.

```
printf("COP 3223 Rocks!\n");
```

This line calls a function printf which resides in stdio.h. The way this function works is that it prints out any string contained in between two sets of double quotes. However, it does not print all the characters above as seen to the screen. There are a few characters which have special codes, called escape sequences. These characters all start with a backslash: \. The code \n stands for a newline character. That simply tells the computer to advance the cursor to print information to the screen to the left hand side of the new line. This line will simply print:

```
COP 3223 Rocks!
```

to the screen and advance the cursor to the next line so that the next piece of information printed will start printing directly below the C instead of right after the !.

```
return 0;
```

Each non-void function must return a value of the proper type. For most functions except for main, this return value is meaningful. But for the function main, this value is of little significance and is only included so that the program has proper syntax.

```
}
```

This simply signifies the end of the function main.

Side Note about Main and Sequential Execution

For your first few weeks, your programs will only comprise of one function: main. When your compiled program executes, it will simply execute each instruction in main in order, and finish when it hits the ending curly brace for main. One of the key concepts of this class is understanding sequential execution. Namely, each program specifies exact instructions for the computer to execute in a particular order without ambiguity. When writing your programs, you must take this into account. You must be precise about the statements you ask the computer to execute and exactly which order you want them to execute.

Variables

Computers are powerful because they can manipulate data. However, the computer must have a standardized manner of storing information. Primitive data types native to the C language aid this standardization. In particular, in order to store data in a C computer program, you must store it one of the primitive data types given by C. Of these data types, the most common ones you will use are:

int, char, float, double

A type specifies what will be stored.

For example, an int stores an integer in between -2^{31} and $2^{31}-1$. Typically, most applications don't compute integers outside of this range, so you can mostly think of an int as storing an integer.

A char stores a single character. A character includes any key on the keyboard, along with a few "special" characters that are not necessarily printable. Incidentally, all characters are stored inside the computer as integers in between 0 and 255. The numeric value of a character is known as its ascii value. We'll talk more about these later. Strings, which we will often use, technically are not primitive data types, but are rather several char variables strung together.

float and double store decimal numbers to varying precision. floats store decimal numbers to about 6 or 7 digits of precision, while doubles store decimal numbers to about 13 digits of precision.

How to use a variable in a program

In order to use a variable in a program, you must first declare the variable. Once you declare it, typically, you should initialize the value of the variable. These two things can be done in one step. Here is an example of a variable declaration:

```
int feet_in_mile;
```

After this line is executed, the computer allocates space for an integer variable. The name given to that space is `feet_in_mile`. Pictorially, we have something as follows:

`feet_in_mile`

Based on this line of code, the variable (the box) may store any value. We can initialize the value as follows:

```
feet_in_mile = 0;
```

This tells the computer to take the value on the right of the =, and store it in the variable to the left of the =.
Note that all statements in C end in a semicolon. (Occasionally, you will see situations that seem to contradict this rule, but in fact, in those situations, the structure in question turns out not to be a single statement.)

The picture after this statement is as follows:

Once a variable has a value, it can be used in statements for computational purposes. First we will show a sample program that uses variables, then we will discuss common practices with variables, assignment statements and arithmetic expressions. (These are the separate components of the following program.)

```
/* Arup Guha
   My Second C Program
   8/26/04
   Computes the number of feet in a mile */

#include <stdio.h>

int main(void) {
  int feet_in_mile, yards_in_mile;
  int feet_in_yard;

  yards_in_mile = 1760;
  feet_in_yard = 3;
  feet_in_mile = yards_in_mile*feet_in_yard;
  printf("Mile = %d Feet.\n", feet_in_mile);
  return 0;
}
```

The first two lines of the program declare all the variables for the function main. All variables inside of a function must be declared at the beginning of the function. Thus, after these two lines, our pictures looks like:

`feet_in_mile`	`yards_in_mile`	`feet_in_yard`

Note, these boxes may NOT be contiguous in the computer's memory. I have only drawn them this way because it's easier for me to do so.

The next two lines initialize the variables `yards_in_mile` and `feet_in_yard`. Here is the picture after these two lines are executed:

`feet_in_mile`	`yards_in_mile`	`feet_in_yard`
	1760	**3**

The next line is an assignment statement. (Technically speaking, so are the two previous lines we just went over.) The syntax of an assignment statement is that a variable is always on the left hand side of an equal sign, and an arithmetic expression is always on the right hand side, followed by a semicolon. The order of execution is as follows:

1) The value of the arithmetic expression on the right is determined using the current values of the variables in this expression. Note: There may also be constants in this expression.

2) This value is then stored in the variable on the left hand side.

Using this procedure, we evaluate the expression:

```
yards_in_mile*feet_in_yard
```

This evaluates to 1760*3, using the current values of each of the respective variables. Simplifying this numerical expression we get 5280. Finally, this value is stored in `feet_in_mile`.

The picture after this step is as follows:

`feet_in_mile`	`yards_in_mile`	`feet_in_yard`
5280	**1760**	**3**

Finally, the last line prints out some information to the screen. Up until now, we only talked about how to print characters to the screen. But now, we would like to print the *value* of a variable to the screen. In order to do this, we must specify the format of the variable. %d is the format for an integer variables. A list of format codes is given in the text on page 16. (Note: The code for a float is %f, and a double is %lf.) This code tells the computer what *type* of variable will be printed. After the ending double quote, the actual variable itself must be specified. This is done by first typing a comma, followed by the name of the variable. The printf function call must then be closed with a close parethesis.

The end result of running this line the statement:

```
Mile = 5280 Feet.
```

will print to the screen.