Variables in C - An introduction.

This section of notes will introduce the concept of program variables in C.

Variable: A location in memory where a value can be stored for use by a program.

We have all used "memory" before, with our calculators! It is the same idea with computer programs. The only difference is that with calculators, we have just one memory area to store our calculated value for later use.

So, we simply press the "memory +" button (or something like that) on our calculator and it stores the current value in memory. The value can later be retrieved by pressing the "memory" button on our calculators. With computer programs the idea is the same. That is, we have calculated values that we want stored while the program is running, so the program can make use of the values throughout program execution. The main difference between computer programs and calculators is that with computer programs, we may have *several* different calculated values that we want stored (and later retrieved).

In order for the program to keep track of values to be stored in memory, we (the programmers) assign a "name" to a memory location that will hold our calculated value. These named memory locations are known as "variables". We (programmers) do not need to know exactly *where* in memory these variables are being reserved, we simply have to know how to access them. We access values of variables by using the variable's *name*.

Not only do programmers need to specify a name for a variable, we also need to indicate the "*type*" of data the variable will contain. That is, we need to specify whether the variable (memory location) will contain numeric data, or non-numeric data (character or string data). If it will contain numeric data, we need to specify if the number will be a whole number (integer or "**int**") or if it will be a number with a decimal place (real or "**float**"). C is a "strongly typed" language, like Pascal (unlike Basic or FORTRAN). Although data typing can seem quite tedious, it ensures program and data integrity. In other words: it's good for you!!! ☺

Note: The value of variables can change as the program executes (hence the word "variable"), however, the name and data type of the variable cannot change.

I think it is a good time to introduce an example to help explain these concepts. The next program in the textbook will be used as an example and is shown below.

PROGRAM 2.4

```
1.  #include <stdio.h>
2.  void main(void)
3.  {
4.      int sum;
5.
6.      sum = 50 + 25;
7.      printf ("The sum of 50 and 25 is %i\n", sum);
8.      getchar( );
9.  }
```

When the above program is compiled, linked, and then executed, the following is output:

The sum of 50 and 25 is 75

Again, let's *parse* the C code above, so we thoroughly understand the new concepts introduced here.

Lines 1, 2, 3, 8, and 9: Nothing new.

These 5 lines were discussed in previous notes. Refer to previous notes for more details if needed.

Line 4: **int sum;**

This statement indicates a very important concept in C. That is, that you must **declare** any variable that you want to use in your program.

Declaring a variable: It tells the compiler the **name** and the **data type** of the variable that you want to create. (We will find out why data typing is so important in Week 3.) Variables must be declared before they are used. Variable declarations typically appear in the beginning of the function block, following the begin brace.

WARNING: In ANSI C all declarations of variables must precede all other programming statements.

This would be wrong:

```
#include <stdio.h>
  void main(void)
  {
      printf("Welcome to my program\n\n\n");
      int sum;
      sum = 50 + 25;
      printf ("The sum of 50 and 25 is %i\n", sum);
```

```
    getchar( );
}
```

In the correct above, variable **sum** is declared to be of type **int**. That is, a location in memory will be reserved to hold an integer (whole number) value while the program executes. The programmer can store information in the memory location known as **sum**, and can retrieve information from that memory location, as long as it is of type **int** (or a whole number). We will discuss the different data types in great detail in Week 3.

During declaration, a location in memory (variable) is reserved to hold whatever the program wants to eventually store in it. There is no "initial value" automatically assigned to the variable. It is not automatically initialized to zero, or any other value. If you want the variable to contain valid data, you must store that data into the variable. We store data in a variable by "assigning" the variable a value. (Shown below in Line 6.)

Also notice that during declaration, the variable's type is listed first, then the variable's name, followed by a semicolon. Variable names should be descriptive. There are several variable naming rules and conventions, which are discussed below. One final note here, a program can have many variables declared. This is also illustrated during Week 3.

Line 5: (this is a blank line)

I know what you're thinking. You think that I've had too much coffee. Well, for your information, I put a blank line in the program on purpose. :-) Blank lines are very important in programs; not for what they do (they actually do nothing), but for how they look. Blank lines help to make your program more "readable". There's that "R" word again! We typically will put a blank line following our variable declarations so to keep them visually separate from the processing code.

Line 6: sum = 50 + 25;

This line is the heart of this program. It illustrates the concept of "assigning values to variables". This particular program adds 2 numbers (notice, both integers) 50 and 25, and stores the result in variable **sum**. In C, we use the **assignment operator** "**=**" for storing results into variables. The statement is actually executed (by the CPU) from the right side of the assignment operator to the left. So, 50 and 25 are added, and the result of that calculation is then assigned to (stored in) the memory location that we have designated as variable **sum**.

At this point, the program is able to access that calculated value simply be referencing the variable's name (**sum**).

One point to be made here, and will be made again throughout this course, and that is that the data type of the information on the right side of an assignment operator should be the same as the data type of the variable on the left side of an assignment operator. This ensures portability of code. In our example, I pointed out that both of the values on the right side of the assignment operator are whole numbers (**int**s), and the variable on the left side (variable **sum**) is also of type **int**. We will discuss this in more detail during Week 3.

Line 7: printf ("The sum of 50 and 25 is %i\n", sum);

Okay, here's where it gets a bit tricky. (Just a bit though. I know you can handle it!)
We have visited the **printf** statement before, but we now see 2 different components. First of all, there is a very odd **"%i**" before the \n. Also, the string ends at the end quote, but then there is a comma and a variable name (sum). Hmmmm. Are they related you ask. The answer is YES!!! Let me explain how.

First of all let's discuss the **%i**. The **printf** function will output everything it finds between the quotation marks until it reaches a backslash, in which case it processes the escape sequence (discussed earlier). Another character that will temporarily halt the output of a **printf** statement is the percent sign. The **percent sign** tells the printf statement that a "**value**" is to be displayed at that particular spot in the output. The "**i**" after the percent sign tells **printf** that the **type of data** to be displayed is an "integer". The combination of the percent sign and the letter "i" (**%i**) is known as a format specifier. It specifies the format (type) of the data to be displayed at that point in the output string.

So, we have a format specifier of **int**, and **printf** is now ready to output some value of type "**int**", but *where* does **printf** find that particular value. Well, it looks at the parameter (argument) following the end quotes of the string being displayed. (When you have more than one argument being passed to a function, they are listed one after the other, separated by commas. - More on this later.) So, the **printf** function looks for a comma and then finds the name of the integer variable where the value to be output is located (variable **sum**, in this example).

At this point the value stored in variable **sum** is retrieved by the **printf** function and output in the exact spot that the format specifier (**%i**) is located in the output string. Then **printf** continues outputting any additional characters that appear in the output string. In our example above, only the "newline character" (**\n**) is output following the value of variable **sum**.

So, to refresh your memory, the **printf** statement looked like:

printf("The sum of 50 and 25 is %i\n", sum);

And the output of this **printf** statement looks like:

The sum of 50 and 25 is 75

If I wanted an exclamation point to be output at the end of the statement above (because, this stuff is so exciting!), then I would have to tell **printf** just that. The **printf** statement would now look like:

printf("The sum of 50 and 25 is %i!\n", sum);
^ notice the exclamation point here

And the output of the **printf** statement now looks like:

The sum of 50 and 25 is 75!
^notice the exclamation point in output.

So, you can see how very tedious, yet comforting all of this is, knowing that the computer will do what you tell it to do. Nothing more, nothing less. (Is it my imagination, or have we heard this before?!?!) ☺

Now, let's ensure our knowledge of variables by reviewing one more example. The following program builds on the previous program, using more than one variable in the program.

PROGRAM 2.5

```
#include <stdio.h>
void main(void)
{
     int value1, value2, sum;

    value1 = 50;
    value2 = 25;

    sum = value1 + value2;
```

```
    printf ("The sum of %i and %i is %i\n", value1, value2, sum);
    getchar( );
}
```

When the above program is compiled, linked, and then executed, the following is output:

The sum of 50 and 25 is 75

Notice, the output is the same as the previous program. But, there are some very obvious differences in the source code.

First of all, 3 integer variables are being declared in this program: **value1**, **value2**, and **sum**. Note that you can declare each variable on its own "line" in the program as follows:

```
int  value1;
int  value2;
int  sum;
```

instead of: **int value1, value2, sum;**

This is a programmer's personal preference. Either method is acceptable.

Here 3 memory locations are being reserved to store 3 separate integer values.

Now, in the following line:

value1 = 50;

we *assign* the integer value **50** to integer variable **value1**.

With the line: **value2 = 25;** we *assign* integer value **25** to integer variable **value2**.

Finally, with **sum = value1 + value2**; we *add* the value of variable **value1** (50) to the value of variable **value2** (25) and *assign* the results of that addition to variable **sum** (75).

Now, the **printf** statement. Don't be afraid of it. Its bark is bigger than its byte! ☺

The **printf** looks as follows:

printf("The sum of %i and %i is %i\n", value1, value2, sum);

If you take this one step at a time, you see how it follows all of the concepts discussed so far. That is, the **printf** statement will continue to output characters to the display until it comes across a percent sign or a backslash. In this case here, it will reach a percent sign first (**%i**). Printf will temporarily halt its output and "look" for an argument following the output string (hopefully containing the name of an integer variable).

Notice the first argument in this **printf** statement following the output string is in fact variable **value1**. So, the value of variable **value1** will be displayed at the spot that the *first* **%i** is located in the output string.

Next, **printf** will continuing outputting characters until it reaches the *second* **%i**. It will look for a second argument following the text string, and finds variable **value2**. It grabs the value of variable **value2** and outputs that at the location of the second %i.

Finally, **printf** continues outputting characters and finds the *third* **%i**. In this spot, **printf** will find variable **sum** as the next value to be displayed. Then, the **\n** (newline) is output. Whew!

The last program shown is my preferred way of doing the problem*. The shortest way is not necessarily the best right now:*

NOT Desirable:

```
#include <stdio.h>
void main(void)
{
        printf ("The sum of %i and %i is %i\n", 50, 25, 50+25);

        getchar( );
}
```

Although the result is the same, and the program is correct, this kind of shortcut undermines the purpose of the exercise - which is to learn to store data in variables, compute with the assignment statement, and print the data contained in variable locations. *I personally do not like computations within **printf** statements, even though allowed in C.* **Please do not submit program #1 in this manner.**

RULES FOR FORMING VARIABLE NAMES

- Variable names in C must begin with either a letter or an underscore.

- Variable names in C may contain letters, underscores or digits (0-9).
- Variable names in C should not exceed 31 characters.

Well, those are the rules. However, one can truly run-a-muck (?) with only the 3 rules listed above. Keep in mind, C is case-sensitive, so variable name "sum" would be different than "Sum". That is, if in your program you declared variable: **int sum;** and then later in your program you made the following assignment: **Sum = 50 + 25;** you would get a compiler error telling you that variable "Sum" is not declared. And you may get an additional another compiler warning telling you that variable "sum" is not used!

So, with that in mind, we have a few programming conventions (standards) that we use when forming variable names. They help maintain the readability as well as the correctness of the program.

STANDARDS FOR FORMING VARIABLE NAMES

- Use only lower-case letters in variable names, along with digits and underscores if desired.
- Do not begin your variable name with an underscore. Begin with a character.
- If you have multi-word variable names, do not group it all together as a single word.

Example: If you want a variable to contain the "grand total" of all numbers, **do not** call your variable: **grandtotal**. Instead, use one of the following 2 standards:

You could name your variable: "**grand_total**" or "**grandTotal**" . In the first case (grand_total), you put an underscore between the 2 words. In the second case (grandTotal), you make all subsequent words with a capital first letter, and skip the underscore.

Whichever standard you choose, be sure to use it throughout your program. Do not mix different standards here, because the inconsistency could reduce the readability.

The textbook does a good job explaining these concepts. Please refer to the programming examples in Chapter 2 of your textbook for additional information and/or clarity.

The next topic in this chapter relates to **"Commenting Your Code".** It is a very good programming habit to get into from the start. (And essential to maximize the points you get on your programs!) ☺