Data Types in C

Recall, there are 3 basic data types: **Integer**, **Floating Point** and **Character**.

INTEGER data types:

- **int**
- **long int**
- **short int**
- **unsigned int**
- **unsigned long int**
- **unsigned short int**

The "int" data type:

Variables that are declared type "**int**" (or integer) are those that will contain *whole numbers*. That is, numbers without decimal places. To declare a variable to be of type integer, you simply use the abbreviation "**int**" before the variable name.

For example:

```
void main(void)
{
    int  value1;
    int  total = 0;
```

In the above code segment, 2 variables are declared to be of type **int**: namely **value1** and **total**. Notice that not only is variable **total** declared to be of type integer, it is also being assigned the value zero. This is a valid C statement, and is called "Initialization during declaration". Sometimes you want your variable to contain an initial value before it is used (such as keeping a running total), so we have the ability to initialize during declaration.

When outputting (printing to screen) the value of an integer, you use the "*format specifier*" **%i** in your **printf** statement. You can also use an "older" format specifier for integers known as **%d**. There is a slight difference between the two, however, in all of the programs for this class, and in most "real world" programs, you can interchange the 2 without any problem. We will discuss these differences later in the course.

So, for example, (as seen in a previous programming example), the following segment of code will declare, assign, and output a variable of type **int:**

```
Void main(void)
{
   int  num;
   num = 2000;
   printf ("The number is %i. Goodbye!\n", num);
   getchar( );
}
```

The output of the above program segment would be:

```
The number is 2000. Goodbye!
```

Now, to spice thing up a bit. Most machines (computers) will reserve 2-bytes (16-bits) of memory for each integer variable. If this is the case, the range of numbers that your computer can store in 16-bits is: -32767 to 32767. If you had a program that needed to work with numbers larger than that (say, in the millions), then declaring your variable to be of type **int** to hold this very large number would not work. You need a "bigger" integer. So, C provides a variation of the **int** data type called: ***long***.

<u>The “long int” data type</u>:

Values that are declared type "**long int**" or simply "**long**" are those that will contain *large whole numbers*.

For example:

```
long  grand_total;
grand_total = 1000000;
printf ("The Grand Total is: %li\n", grand_total);
```

In the above examples, variable **grand_total** is declared to be of type **long integer**, and is assigned the value of 1 million. Also, when outputting the value of a long integer, you use the "*format specifier*" **%li** (or **%ld**) in your **printf** statement.

The output of the above segment of code is:

```
The Grand Total is: 1000000
```

Most machines will reserve 4-bytes (32-bits) of memory for each long integer variable. If this is the case, the range of numbers that your computer can store in 32-bits is: -2,147,483,647 to 2,147,483,647. (BTW - These "sizes" are defined on page 433 of your textbook.) If you had a program that needed to work with numbers even larger than that (say, 3 billion), then declaring your variable to be of type **long** to hold this very large number would not work. You need an even "bigger" integer. So, indeed, C provides a way to handle this dilemma. We will discuss it below in the section titled: **unsigned long**. But first, I want to backtrack a bit and mention what we do if we know that a value that we plan to use will be very "small". That it will probably not exceed 100. Well, why reserve 4 bytes (**long)** or even 2 bytes (**int**)of memory for that variable, when a single byte will do. C provides a data type which is *half* the size of an **int**, and that is type ***short***.

The “short int” data type:

Values that are declared type "**short int**" or simply **"short"** are those that will contain *very small whole numbers*.

For example:

```
short  age;
age = 25;
printf ("You are %hi years old.\n", age);
```

In the above examples, variable **age** is declared to be of type short integer, and is assigned the value of 25. Also, when outputting the value of a short integer, you use the "*format specifier*" **%hi** (or **%hd**) in your **printf** statement.

The output of the above segment of code is:

```
You are 25 years old.
```

Some machines will reserve 1-byte (8-bits) of memory for each short integer variable. If this is the case, the range of numbers that your computer can store in 8-bits of memory is: -127 to 127. Some machines do not like to work with "odd" memory addresses (a single byte at a time), so will allocate 2-bytes even for a **short** integer variable. (This is transparent to the program, as you will see when we present some additional examples.) In this case, the size of data type **short int** is the same as that of a regular **int**: 2-bytes of memory per variable.

The "unsigned int" data type:

Values that are declared type "**unsigned int**" are those that will contain *positive whole numbers*. That is, numbers without decimal places AND numbers that are *greater than or equal to zero*. You use this data type if you know your values will always be positive, and you need to work with numbers slightly higher than 32,767. With a regular **int**, you get 16-bits to work with, but only 15 are usable for your number. The last bit is used as the "sign-bit". If zero, your number is positive, if 1 your number is negative. (Programmers do not typically concern themselves with how data is represented in memory, but I think you should know what is going on.) When you declare a variable to be of type unsigned int, you are freeing-up that 16th bit for data. So, your range of valid values for this type of variable is: **0 - 65,535**.

To declare a variable to be of type unsigned integer, you use "**unsigned int**" before the variable name.

For example:

```
unsigned int   value1;
```

When outputting the value of an unsigned integer, you simply use the "*format specifier*" **%u** in your **printf** statement.

As you guessed it, we also have an ***unsigned short integer*** and and ***unsigned long integer***. At this time, I will simply summarize (in table form) below the various integer data types, their format specifiers for the printf statement, and their range of values. I think you get the gist of this by now.

Refer to chapter 3 for additional details (Table 3.1 on page 28).

Data type	**Format Specifier**	**Range of values**
short int	%hi (or %hd)	-127 to 127
int	%i (or %d)	-32767 to 32767
long	%li (or %ld)	-2,147,483,647 to 2,147,483,647
unsigned short int	%hu	0 to 255
unsigned int	%u	0 to 65,535
unsigned long int	%lu	0 to 4,294,967,295

Now we move on to "Floating Point" Data Types. Luckily, for your sake, there are only 3 types of floating point data variables.

FLOATING POINT data types

- **float**
- **double**
- **long double (???)**

The "float" data type:

Floating point data is data that is numeric, but contains a *decimal place*. Another term for this type of data is: **real**. Some examples of floating point data include: 3.3, 3.0, 271.67452, -.001, etc.

To declare a variable to be of type floating point, you simply use the abbreviation "**float**" before the variable name.
For example:

```
void main(void)
{
    float  value;
    float  sum = 0.0;
```

In the above code segment, 2 variables are declared to be of type **float**: namely **value** and **sum**. Notice once again, that I initialized variable sum to be equal to 0.0 during declaration. An important point here is that I used 0.0 instead of 0 because my goal, to ensure that my C program is 100% portable, is that the data type of information on the right side of the assignment operator (**=**), is the same as the data type on the left side of the assignment operator. And, since variable **sum** is of type **float**, the value zero should also be of type float, hence **0.0**.

For data of type float, C will maintain 7 decimal places of accuracy for you. That is, if you set variable value above equal to: 10.123456789, C would only be able to store: 10.1234567. The remaining 2 decimal places would be inaccurate.

When outputting (printing to screen) the value of a floating point number, you use the *"format specifier*" **%f** in your **printf** statement.
So, for example, the following segment of code will declare, assign, and output a variable of type **float:**

```
void main(void)
{
   float  num;
   num = 25.26;
   printf ("The number is %f. Goodbye!\n", num);
   getchar( );
}
```

The output of the above program segment would be:

```
The number is 25.260000. Goodbye!
```

<u>Controlling the number of decimal places displayed</u>.

Why am I showing 6 decimal places? It has nothing to do with the "accuracy" that I mentioned above. What is going on here is that C will *assume* that you want to see 6 decimal places in all of you floating point output, when you use %f. This is not always the case. So, I need to inform C that I only want to see 2 (for example) decimal places in my output. I do this by putting a "**.2**" (point 2) between the **%** and the **f** in the format specifier. For example, to display the above number with only 2 decimal places, my **printf** would look like:

```
   printf ("The number is %.2f. Goodbye!\n", num);
                                ^ Notice the .2f here
```

Now, I would see as my output**:**

```
The number is 25.26. Goodbye!
```

What if I wanted to see only **1** decimal place using the above variable (**num**). My **printf** statement would look like:

```
    printf ("The number is %.1f. Goodbye!\n", num);
                                   ^ Notice the .1f here
```

And my output would be: (pay attention here -- it's not what you think...)

```
The number is 25.3. Goodbye!
```

Notice, the output was not 25.2, but 25.3. That is because C will *round-up* if the number of places you are outputting are less than the accuracy that C has maintained for you, and if the next decimal number is 5 or higher. So, in this case, since variable num is of type float, C automatically maintains 7 decimal places of accuracy. The number is stored as: 25.2600000 (7 decimal places of accuracy), however you are only asking for 1 decimal place. C will round up for you here, because the second decimal place (the number 6) is greater than 5.

One last thing here, recall, C will maintain up to 7 decimal places of accuracy with a float variable. If you opted to output *more than* 7 decimal places of your variable of type float, you will get data beyond the 7th decimal place displayed, it would just be bogus. (That's an official computer term by the way - bogus.)

For example:

```
float  results;
results = -345.123456789;
printf ("The results are: %.12f\n", results);
```

Your output would be:

```
The results are: -345.123456720699
```

Notice the accuracy was maintained up through the 7th decimal place, and the last 5 decimal places output in this example are, well, as I put it earlier... bogus!

Now, let's say you need more than just 7 decimal places of accuracy. C provides another floating point data type which will give about 16 decimal places of accuracy. It is type **double** described below. I know what you're thinking (did I tell you I was psychic?!?!), you're thinking that "**double** is to **float**" as "**long** is to **int**". Oh... you weren't thinking that? You were thinking about a bagel and cream cheese? Oh well, my psychic powers aren't too precise. I'm still working on them. ☺

For the 2 or 3 of you who *did* notice the similarities between **double** and **long**, you are right on track! Good for you.

<u>The "double" data type</u>:

Values that are declared type "**double**" contain *floating point numbers with lots of accuracy*.

The format specifier for variables declared as type double is: **%f**. Yes. That's right. %f. It is *the same format specifier as for variables declared as type float*. Although C will maintain 16 decimal places of accuracy for variables of type float, if you use **%f** as your format specifier in your printf statement, C will once again display 6 decimal places (by default).

For example:

```
double  results;
results = -345.123456789;
printf ("The results are: %f\n", results);
printf ("The results are: %.2f\n", results);
printf ("The results are: %.12f\n", results);
```

The output would be:

```
The results are: -345.123457
The results are: -345.12
The results are: -345.123456789000
```

Notice on the first line of output, the 6 decimal places output were: .123457, not .123456. The reason is because you are asking C to output less precision than C has maintained for you, so C will round-up if needed. Since the number 7 (following the number 6) is greater than 5 then the number 6 is rounded up to 7, in this case.

The "long double" data type:

Values that are declared type "**double**" contain *floating point numbers with lots and lots of accuracy (at least you would hope so)*.

The textbook mentions a data type called: **long double**. It is supposed to provide more precision than the double, but this is typically not the case. Most machines cannot provide more than 16 decimal places of accuracy, so the long double qualifier in front of the variable name is typically ignored by the computer, and simply type double is used. If you want to give it a try, the format specifier for long double is: **%lf** (or **%Lf**).

A tidbit about Scientific Notation:

One last comment regarding floating point numbers is that you can represent (output) floating point numbers using "*Scientific Notation*". If you use the **%e** or **%E** format specifier in the **printf** statement on a variable of type floating point (or double), C will output your results using scientific notation. Please read page 27 for more information regarding scientific notation.

Next, we try something completely different. A non-numeric data type called **char**.

<u>CHARACTER data type</u>

- **Char**

Not all C programs are numeric-only programs. More often than not, you will have to represent data that is non-numeric in your program. When we think of non-numeric data, we often think of "strings" of data. Such as someone's name. However, C does not provide a "**string**" data type. Instead, C provides a more basic "character" data type (called **char**), where within a variable of type **char**, you can store a single character. If you want to work with "string data" (i.e., more than a single character in the data), then you must group individual characters together to form your string. This is done using "arrays" which is covered in Week 7 of this course. For now, we will discuss the most basic non-numeric data type called: **char**.

<u>The “char” data type</u>:

Values that are declared type "**char**" contain *a single character.* The *range of values* of data of type **char** is: *any character on your keyboard* (and then some)! The *format specifier* for data of type **char** is **"%c"**. To assign a character variable an initial value, you must enclose the single character in **single quotes**. However, the single quotes will not be output during your printf statement. For example:

```
char   answer;
answer = 'Y';
printf ("Your answer is: %c\n", answer);
```

The output of the above program segment would be:

```
Your answer is: Y
```

Notice that we enclosed the character **Y** in single quotes in the program, however the quotes were not output in the **printf** statement. The reason we need to do this is because if we didn't, then C would assume that we were assigning variable **answer** equal to *variable* **Y** (because **Y** is a valid variable name), and we would get a compiler error that we have not yet declared variable **Y**!

That's all I have regarding data types. If you have any questions at all regarding data types, please post them in the discussion board for Week 1. I realize that programming concepts can seem fairly abstract, and I want to be sure that you have a thorough understanding of each-and-every concept so far. We will be using all that we have learned so far in all future examples and programs. So, I want to be sure that you "get it".

Next, we will discuss the type of "arithmetic" (math) you can perform in C!