Running a Program

Let us begin as we always do, by writing the "Hello World" program. In C, just like in Java, all programs start on the first line of the function called `main`. Here is the code for the program in Java:

```java
public class HelloProgram{
    public static void main(String[] args){
        System.out.println("Hello World!");
    }
}
```

Since everything in Java happens within a class, due to the object-oriented nature of the language, we must create a class for our main method to be written into. By contrast, C is not an object-oriented language, so no classes are created or used. Here is the same example in C:

```c
#include<stdio.h>
int main(){
    printf("Hello World!");
}
```

On the first line of the code, we have

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

This imports information from the standard input/output library. C is a very low-level language, designed to work very near to the computer. The language doesn't contain any function definitions on its own, so every function that you call has to be defined in the file where it is called. Compilers for C will include a set of libraries, which contain definitions for many useful functions. In this case, we wanted to print a message, and we can do so with the formatted print function `printf`, defined in the standard library `stdio.h`. The compiler will actually copy and paste the contents of the file `stdio.h` right where the import directive is, so the definition of `printf` will end up in the file.

Compiling

The extension for C source code files is `.c`. For example, the code above might be written into a file named `hello.c`. To compile C source code into an executable file, we will use the GNU C Compiler, which is accessed through the terminal with the command `gcc`. For our program `hello.c`, the format of the command in the terminal is

```
gcc <options> hello.c
```

where we can include options to customize the output of the compiler. If we don't include any options, then the file will compile and produce an executable output named `a.out`, which is the default name.

A common option is to customize the name of the output, which can be done with the option `-o filename`. For example, if we want to compile the file `hello.c` and have it create the executable program named `myProgram`, then we would use the following command in the terminal:

```
gcc -o myProgram hello.c
```

Note that in a Linux terminal the order of the arguments is actually not important in most cases, so

```
gcc hello.c -o myProgram
```

will work just as well. It is important though that `-o` and `myProgram` be kept together though, since they form a single option.

To run the compiled program, you will just run it directly in the terminal with the name of the program. For example,

```
        ./myProgram
```

will run the program we compiled above. Notice that the `./` specifies that the program we are running is in the directory where the terminal is located. This is to differentiate the program from the system commands, like `ls` and `cd`.

Data Types and Variables

Recall that in Java, there are eight primitive data types. There are four integer types `byte`, `short`, `int` and `long`, two floating point number types, `float` and `double`, as well as the character type `char` and the boolean type `bool`. The sizes of these data types are defined by the Java VM and because of this are uniform everywhere the programs are run.

Integer Types

C defines the following signed integer types:

```
char    short    int    long    long long
```

Since different systems have different resource constraints, the sizes of the types are not heavily regulated. The standard rules are that the `char` data type is 1 byte, and the sizes must be non-decreasing in the order above. Aside from that, the exact sizes of each data type is dictated by the machine that the code is compiled on.

As in Java, if the size of the integer data type is n bits, then the range of the numbers which can be represented by the data type is all of the integers x in the range

$$-2^{n-1} \leq x \leq 2^{n-1} - 1$$

For example, if the `short` data type is 2 bytes (=16 bits), then the range of integers x that can be represented is

$$-2^{15} \leq x \leq 2^{15} - 1$$

or

$$-32,768 \leq x \leq 32,767$$

.

Unsigned Integer Types

The integer types defined above are also called the **signed integer types** in C. The reason for this is that C also has **unsigned integer types**. To declare an unsigned integer type, we just use the keyword `unsigned` in front of any of the integer types.

The key difference is what numbers can be represented by an unsigned integer type. For an unsigned integer type of n bits, the integers x which can be stored are in the range

$$0 \leq x \leq 2^n - 1.$$

For example, if an `unsigned short` is 2 bytes (=16 bits), then it can represent the integers x in the range

$$0 \le x \le 65535$$

Notice, this means that instead of representing negative numbers, the range of the positive numbers has doubled. This is quite useful in extending the positive range for values that you know can't be negative.

The sizeof() Operator

Since we may not know what the size of a data type is on a particular system, C provides a compile-time operator called `sizeof`, which when applied to a datatype, is replaced by the size of that data type. Since the size of data types are naturally non-negative, the value given by the `sizeof` operator is represented by an unsigned integer type.

The following code will print out the sizes of the integer data types:

```
        #include<stdio.h>

        int main(){
            printf("char: %ud\n", sizeof(char));
            printf("short: %ud\n", sizeof(short));
            printf("int: %ud\n", sizeof(int));
            printf("long: %ud\n", sizeof(long));
            printf("long long: %ud\n", sizeof(long long));
        }
```

Notice the "`%ud`" in the `printf` statements. These are the format specifiers for unsigned integers. We will discuss the format specifiers of `printf` in more detail later.

Floating Point Data Types

There are three types of floating point numbers which can be represented in C:

`float`   `double`   `long double`

As with the integer, the sizes of the data types depends on the machine that the code is compiled on. Due to the way that floating point numbers are encoded, there are no signed and unsigned variations. All floating point numbers are signed.

Variables and Arrays

C is a strongly typed language, just like Java, which means we must declare variables and their types before we use them. The declaration syntax for primitive variables is very much the same as in Java. For example, if we want to declare in `int` variable named var, then we can do with without an initialization:

```
int var;
```

or with an initialization:

```
int var = 5;
```

Declaring arrays is quite a bit different. In Java, arrays are treated similar to objects. You create an array with the `new` operator and then attach it to a declared array variable of the same type.

An example of the creation of a character array of length 100 in Java is:

```
char[] arr = new char[100];
```

In C, there are two ways to construct an array, and you may view it as two different kinds of arrays. The simpler of the two is called a **static array**. There is no `new` operator in C because there are no objects to build. The creation of the static character array of length 100 is done as follows:

```
char arr[100];
```

The syntax is much simpler, and this line will declare the variable `arr` and create the array all in one step. The downside here is that the variable name get tied to the array itself, and it isn't possible to resize a static array after it's created. By comparison, in Java if we want `arr` to refer to a larger array, we could simply use `new` to construct a larger array and then assign the new array to `arr`. We will see another way to construct arrays in C later, with a similar dynamic nature.

Character Arrays and Strings

The way strings are handled in Java, as is the case with most things, is as an object, with data fields and methods. You never have to think about how the strings are being stored internally, just that they are part of the String class definition. Here is an example of storing and interacting with a string in Java:

```
public static void main(String[] args){
    String myStr = "Hello World!";
    System.out.println("String: " + myStr);
    System.out.println("Length: " + myStr.length());
}
```

There is no String data type in C. Instead, strings are stored directly in character arrays. This means that the length of the array use for storage and the length of the string being stored can (and often are) different. In particular, the length of the array should be at least as long as the string, so we have enough room to store the string, but it can be larger.

```
int main(){
    char myString[100] = ``Hello World!'';
}
```

In the above example, the character array has length 100. What is the length of the string "Hello World!"? After we store it in `myString`, how does the computer know how many of the 100 characters are part of the String?

In Java, String has an attribute called `length`, so for example in the case of "Hello World!", the length would be 12, and from that we know that the string uses indices 0 through 11.

Arrays and strings in C don't have a length attribute, since they aren't objects. Instead, every string has an extra, hidden character, called the null character, and this null character marks the end of the string. The character literal for the null characters is `\0`, and it has ASCII value 0. Therefore, the total length of the string "Hello World!" in C is 13, which accounts for the 12 visible characters and the one hidden null character on the end.

Try the following example:

```
#include<stdio.h>
int main(){
    char hello_string[12] = ``Hello World!'';
    char another_string[100] = ``Here is another string.'';

    printf(``%s\n'', hello_string);
}
```

Here we didn't reserve enough space in `hello_string` to store the null character on the end. As a result, what is printed will be:

```
Hello World!Here is another string.
```

That is, `printf` doesn't see the null character, and so it doesn't know to stop printing characters and start printing characters from a completely separate string!

If you run the above example again, with the array size of `hello_string` set to 13, you will see the proper behavior from `printf` because in that case, the null character is stored.

Constants

There are four different kinds of constants in C: literals, symbolic constants, stored constants, and enumerations.

Literals

Literals are the constant values that we type directly into the code. For example, in the statement

```
x = 5;
```

The 5 is an integer literal. There are four major kinds of literals: integer, floating point, character and string.

Integer Literals

An integer literal by default has the size of an int, as long as its value can be represented by an `int`. If it can't, then it will be interpreted as a `long`, a `long long` or an `unsigned long`, depending on its size.

To force the interpretation as a long, you can add an `l` or `L` as a suffix to the value. For example

```
long x = 123l;
```

will interpret 123 as a long integer in the line. This isn't strictly necessary, as 123 will be cast as a long anyway to fit it into the variable `x`, but it is a good practice to not make the compiler guess at the proper datatype, especially when using literals in arithmetic expressions.

Similarly, the suffix `ll` on an integer literal will denote a `long long`, and a `u` will denote an unsigned integer. We can also combine the unsigned character with either of the longs: `ul` and `ull` represent unsigned long and unsigned long long, respectively.

Floating Point Literals

A numerical literal is interpreted as a floating point literal if the decimal point (`.`) is present. By default, the type of a floating point literal is a `double` (a double-precision floating point number). To manually designate the literal as a `float` (single-precision floating point value), we can add the suffix `f` or `F` to the end of the value.

Character Literals

A character literal is any ASCII-defined character between a pair of single quote marks. For example:

```
char c = 'A';
```

sets `c` to the value 65. Remember that the char datatype is an integer type, so the data being stored is that of an integer. In fact, the above line is equivalent to

```
char c = 65;
```

However, since we don't want to have to remember and type the ASCII values directly into the code, the character literals make our code a lot easier to write and nicer to look at.

In addition to all of the printable characters, there are escape sequences for several special characters.

`\0`	the null character (ends a string)
`\n`	newline
`\r`	carriage return
`\t`	tab
`\v`	vertical tab
`\f`	newpage (formfeed)
`\a`	audible alert
`\'`	single quote mark
`\"`	double quote mark
`\\`	backslash
`\?`	question mark

String Literals

A string literal is a string of characters between a pair of quotation marks. The literal always automatically includes the null character at the end of the string. Any of the ASCII encoded characters can be included in a string literal, including all of the special characters in the table above.

Defined Constants

A **symbolic constant** is a way to give names to meaningful constants without storing them with variables. To define a symbolic constant, we use the syntax

```
#define NAME value
```

and all instances of `NAME` within your code will be replaced by the specified value. This happens at compile time, so the best way to visualize it is that the compiler will copy and paste the value into your code wherever `NAME` is. For example, the following example will construct a character array of length 100:

```
#define SIZE 100
int main(){
    char arr[SIZE];
}
```

It is a good practice to name your constants like this for two reasons. The first is that it makes your code easier to read, especially if the names reflect the intention of how the constant is used.

The second reason is that it makes your code easier to modify. In the above example, if we want to change the size of the declared array, we simply need to update the value for `SIZE`. While that might not seem to impressive in this simple example, imagine if we have a much larger code base where we declare arrays of size equal to 100 at several locations in the code. Tracking down and manually updating each one can be tedious and error-prone. It would be much easier to use the constant to simplify updates.

Declared Constants

A declared constant is a variable in memory which is read-only. What that means is that after the variable has been initialized, it cannot be changed. To declare a variable constant, we just need to add the modifier `const` to the declaration. For example

```
const float PI = 3.141592;
```

declares the constant variable `PI` and sets its value to 3.141592. Any attempt to assign a new value to PI will be stopped with a compiler error.

Unlike a symbolic constant, a declared constant puts the value in memory at runtime, and it can be referenced by any parts of the code which are in the variable's scope. The `const` modifier can be applied to any declared type.

Enumerations

Enumerations allow us to define a set of integer constants all at once, and is often syntactically simpler than defining each constant one at a time. It can be especially useful to give numerical encodings to non-numerical concepts. For example:

```
#include<stdio.h>
enum{ SUN, MON, TUE, WED, THU, FRI, SAT};

int main(){
    int day = WED;

    if (day == SAT || day == SUN)
        printf("It is the weekend.\n");
    else
        printf("It is a weekday.\n");
}
```

In this example, we can see the basic syntax of an `enum`. We use the `enum` keyword, followed by a pair of braces with a comma separated list of constant names inside. It is a statement in the code, so it ends with a semi-colon.

By default, the first name on the list (`SUN` in our example) is assigned the value 0. Then each name is assigned a value incremented from the first (`MON` is 1, `TUE` is 2, etc.).

Also notice that in the example, the values don't really matter that much. It is clear from the names and the way we are referencing them in the code what the intended effect is. This is the power of encoding things as numerical values!

We can manually assign values to the names in enum if we like

```
enum{ AUG = 8, SEPT = 9, OCT = 10, NOV = 11, DEC = 12};
```

Or we can partially specify the values:

```
enum{ AUG = 8, SEPT, OCT, NOV, DEC};
```

In this case, the value of the first unspecified constant will be incremented from the previous value. (In the above example, SEPT will be set to 9.)

Conversions

There is often a need to transform data of one type to another type. For example, changing an integer into a floating point number, or a string representation of an integer into its value.

Auto-Casting

Most conversions of the primitive types are done by auto-casting. This is somewhat similar to Java, except that in C, wider datatypes will be auto-cast to narrower types. For example in Java the following code

```
        public static void main(String[] args){
            int x = 3.0;
        }
```

will generate an error during compilation. To fix it and force the casting to occur, we would add a manual cast:

```
        public static void main(String[] args){
            int x = (int) 3.0;
        }
```

However, in C, the following code will work without any errors:

```
        int main(){
            int x = 3.0;
        }
```

In fact, auto-casting will occur with any of the integer, unsigned integer, and floating point data types. Be careful, while it is convenient to not have to manually cast, it can also be a pain when a bug occurs because of an unintentional cast causing the corruption of values. For example,

```
int x = 3.5;
```

will cause the value to change from 3.5 to 3.

Auto-casting also occurs when the operands to an operator have different data types. In this case, the operand with the narrower data type will be cast to the same type and the operand with the wider type. In the following example,

```
10 + 2.5
```

we have an `int` literal and a `double` literal. The double is a floating point type, and is therefore wider than an integer type. Therefore, the 10 will be auto-cast to a double before the addition takes place. The result will be the double 12.5.

Remember that characters are an integer type, and is treated in the same way as the other integers with respect to auto-casting. In the following statement

```
char c = 'A' + 2;
```

we have an addition operator between a character type (a 1-byte integer) and an int type, which is wider. Before the addition, the 'A' will be auto-cast to an `int` type. Then the addition will occur between two int values. Finally, the result will be auto-cast back to a `char` so that it can be stored in the variable `c`. In the end, `c` will contain the value 67, (which is the same as `'C'`).

Conversions with Strings

When reading a document that contains a number, we interpret the number immediately in our heads, without ever thinking about it. In reality, what is written on the document is a sequence of symbols that together form a string which represents the value.

This also occurs in computer programs. Anytime we type a value out, we are providing a string with numerical characters, which together form a single value. So it is very necessary to have a way to convert numbers (both integers and floating point numbers) back and forth between a string representation of the number and the actual value. As an example, the string

```
char myString[9] = "3.141592";
```

is clearly an approximation of π. However, it is just a sequence of characters, so if we looked at the values stored in the char array, it is the ASCII integers for `3` then `.` then `1`, etc. We could write a program which takes a string representing a floating point number as an argument and then return that floating point number. However, C provides versions of these conversion functions in the standard library `stdlib.h`. The names of these conversion functions are

`double atof(const char *string)`	converts `string` to `double`
`int atoi(const char *string)`	converts `string` to `int`
`long atol(const char *string)`	converts `string` to `long`

Note that the `const char *string` input to these functions is just a way to specify a character array. We will learn about the format later, but for now we can ignore it. Using the function is simple, as in the following example.

```
#include<stdlib.h>
int main(){
    char pi_arr[9] = "3.141592";
    double pi = atof(pi_arr);
}
```

The result is that the variable `pi` will contain the value 3.141592.

Converting from numerical values to string representations is easiest done with the `printf()` family of functions. For example

```
printf("%d", 150);
```

will print the string representation of the integer 150 to the terminal (standard out). There are other versions of `printf` which will print the string to a character array or a file, which we will introduce later.

Functions

The header of functions are much simpler than that of methods in Java. Because methods are part of a class, each method needs to have a few modifiers to specify its relationship with the class and/or any generated objects.

In C, the header of a function only needs to include the return type of the function, the name of the function and the list of formal parameters. Here is a function which squares its input:

```c
double square(double num){
    return num * num;
}
```

Calling a function is syntactically the same as calling a method, except that you can just directly call the function by its name. There is no need to reference it from an object or class.

```c
int main(){
    double x = square(10.0);
}
```

However, the compiler for C is a one-pass compiler. This means that the compiler only scans the source code file once as it is compiling. If a function is called before it is defined, then the compiler won't have the information that defines the function at the time the call is made. For example, putting the previous two examples together, if we compile the following code:

```c
int main(){
    double x = square(10.0);
}

double square(double num){
    return num * num;
}
```

the compiler will generate an error. There are two ways to fix this. The first is to write the definition of the `square()` function before it is called in the `main()` function:

```c
double square(double num){
    return num * num;
}

int main(){
    double x = square(10.0);
}
```

This code will compile correctly.

However, there is another way to fix the issue which doens't require you to write the functions in a specific order. You can declare functions! To do so, you write the header for the function, and then end it with a semi-colon, instead of an open brace.

```c
double square(double);

int main(){
    double x = square(10.0);
}

double square(double num){
    return num * num;
}
```

This code will also compile correctly. Notice that in the parameter list of the declaration, the variable name is not specified. Since we are just declaring the function, we can just specify the data types and don't need to give parameter names until the definition. This makes sense because when we define a function, we will then need to access the parameters, so they need a name, but when we declare a function, we don't need to interact with the parameters.

All parameters in C functions are passed by value. This means that any changes the function makes to the input will not be preserved after the function returns. By comparison, in Java all of the primitive data types are passed by value, and arrays and objects data types are passed by reference. While that sounds a bit limiting, we will learn how functions can manipulate data in a different way later, when we discuss pointers.

The main() Method

Up to this point, the header that we have used when writing the main method has been

```
int main()
```

Just like in Java, (and most other programming languages), we may want the user of the program to pass in some arguments when running our program. These arguments are collected as an array of strings.

In Java, the header for the main method is

```
public static void main(String[] args)
```

We again encounter the fact that Java treats strings as objects and keeps track of the length of arrays as an attribute. Since C does not do this, then we need to manually be told the length of an array. This is done with an integer argument in main called `argc` (the ‘c’ is for count).

We should also remember that there is no string data type, so to store a single string, we use a character array. In order to store an array of strings, we will need an array of character arrays. In other words, we need a 2D character array. That is essentially what `argv` is, but you can still think about it, and interact with it, as an array of strings.

Putting all of this together, the general header for a main method that accepts arguments is

```
int main(int argc, char *argv[])
```

For the second time, we see an asterisk in the declaration of a function, and we will continue to ignore it for now. An alternate way to declare `argv`, which will have the exact same effect is

```
int main(int argc, char **argv)
```

All that is important is that `argv[0]`, `argv[1]`, etc are strings. Here is a program which will print the arguments that it receives:

```c
#include<stdio.h>
int main(int argc, char *argv[]){
    for(int i = 0; i < argc; i++){
        printf("argv[%d] = %s\n", i, argv[i]);
    }
}
```

Let’s say we compile this into a program named argList. Then we can run this program with

```
$ ./argList arg1 arg2 arg3
```

The result will be displayed as

```
arg[0] = ./argList
arg[1] = arg1
arg[2] = arg2
arg[3] = arg3
```

Notice that the index 0 is always the program name. The argv array will split exactly what you type into the terminal into separate words separated by white space.

Standard Input and Standard Output

When a program is running and it prints something to the terminal, it is outputting to the **standard output**. Later (much later) you will learn that standard output can be redirected to other places besides the terminal. For now anytime you see a reference to standard output, you should just interpret that to mean the terminal.

Similarly, when a program prompts the user for input in the terminal, it is reading from **standard input**. There is also a third channel, called **standard error**, which like standard output prints to the terminal by default, but as the name suggests, programs use this to print out any error messages.

Functions which allow programs to interact with the standard in and out are defined in the `stdio.h` library.

The printf() Function

We have already printed some messages in examples above, and the function we hav used is the **formatted print function**, `printf`. Formatted strings are very useful for building strings from literals, and various data types, in a very compact way.

The simplest way to think about a format string is as a string literal with "fill in the blanks" called format specifiers. Format specifiers start with a % character, and then a letter to indicate what type of data will fill in the blank. For example, the format string

```
"integer: %d\n"
```

contains one format specifier `%d`, which will take an integer and put it here in the string. Everything else in the format string is a regular character and will be interpreted the same way as in an ordinary string literal.

Obviously, if we have a format string with one or more format specifiers, then we need to specify the data to fill in for those specifiers. In `printf()`, the first argument is a format string, and then the rest of the arguments are the data to fill in for the format specifiers, matched in a left-to-right manner. An example call to `printf()` is

```
printf("integer: %d\n", 5)
```

The result of this command will be that the program prints the string:

```
integer: 5
```

to standard out.

There are format specifiers for each type of data:

`%d`	integer
`%l`	long integer
`%ll`	long long integer
`%u`	unsigned integer
`%f`	floating point number
`%c`	character
`%s`	string

There are also some format specifiers for special output types

`%o`	unsigned integer in octal notation
`%x, %X`	unsigned integer in hexadecimal notation
`%e, %E`	floating point number in exponential notation
`%g, %G`	floating point number, variable notation

The `%l` format specifier can be used on its own to indicate an `long int` or used as a modifier to the other integer specifiers. For example `%lu` is an `unsigned long`, and `%lx` is an long integer in hexadecimal notation.

In order to print the percent symbol itself, you just use `%%`. This isn't a format specifier that expects an input. Here is a program, which takes two integers as inputs, divides the first by the second and prints the quotient and remainder.

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

        int main(int argc, char *argv[]){
            int num = atoi(argv[1]);
            int denom = atoi(argv[2]);

            printf("%d / %d: quotient = %d, remainder = %d\n",
                num, denom, num/denom, num % denom);
        }
```

Note that there is no error checking for the inputs here. If no inputs are specified, the program will crash, and if something other than integers are given, the behavior is unspecified.

We can also control how the format specifier is displayed in several ways. For example, for a floating point number, we may want to control how many digits are displayed after the decimal point.

The way to control the minimum width of the field is by putting a number between the `%` and the format specifier. For example

```
printf("%10d is an integer", 300);
```

This will pad the 300 with spaces so that its width is 10, and it will right-justify the number within those 10 spaces. To left justify the number, simply add the negative sign in front of the number

```
printf("%-10d is an integer", 300);
```

To specify the precision (for example the number of digits of a floating point number printed after the period), add a decimal point and a number after it to the format specifier. For example,

```
printf("PI is %.2f", 3.141592);
```

This will round the number to two decimal points (3.14 in this case). The default precision is 6 decimal places, if left unspecified.

The sprintf() Function

The `printf()` function is great for printing formatted strings to standard out. However, you may also want to create formatted strings in situations where you don't want to immediately print them to terminal. In that case, `sprintf()` will allow you to store the string in a character array.

The first argument of `sprintf()` should be a character array that is large enough to store the formatted string that will be produced. The second argument is the format string, and then the additional arguments are for filling in the format string. Here is an example of using `sprintf()` to store a formatted string:

```c
        #include<stdio.h>

        int main(){
            char buf[100];
            sprintf(buf, "PI is %.2f", 3.141592);
        }
```

The result is that the character array `buf` will contain the string “PI is 3.14”.

The scanf() Function

In order to read input from standard in, C has a function called `scanf()`, which has a very similar format as `printf()`. Of course the difference is that we expect to receive some text input from the user and will want to interpret it as a particular data type. For instance, we might ask the user to input an integer. The user will enter the string representation of the integer, and we will want to interact with its value. We could read the input in as a string, and then use the `atoi()` function that we previously described to translate it, but we have another option. The function `scanf()` can read input from the user and format it to a specific data type.

The first argument for `scanf()` is an format string, just like `printf()`. This time, you should specify the format that you expect the user to enter their input. For example, if you expect a user to enter a date in the format “month/day/year” where month, day and year are each integers, you might use the following call to `scanf()` to receive that data.

```c
        #include<stdio.h>

        int main(){
            int day, month, year;

            printf("Please enter a date in the format mm/dd/yyyy: ");
            scanf("%d/%d/%d", &month, &day, &year);
        }
```

After `scanf()` is called in the above example, the day, month, and year will be set to the integer values entered by the user.

Here we come across another oddity that we will just accept for the time being, and explain later. In the above example, the 2nd, 3rd and 4th arguments are the names of the variables which will store the values in the format string of `scanf()` from left to right. The format of those arguments have an ampersand (`&`) prefixed to the names of the variables. This is intentional and necessary in order to `scanf()` to work.

Of course, we can prompt the user for simpler inputs as well. For example, if I just want a single floating point number to be entered, I can call

```
scanf("%f", &var);
```

where this time `var` is the name of a floating point variable that was previously declared.

The getchar() and putchar() Functions

As an alternative to the complexity of using formatted strings for input and output, there are two functions which are much simpler.

For reading from the standard input, the function `getchar()` is as simple as it gets. The header for get char is

```
int getchar(void)
```

It takes no arguments, and returns the character that is read from input. You might immediately be concerned that a function which reads in one character from input can't be that useful, as most users will likely input many characters when prompted. The way to understand this is to view `getchar()` as receiving one character at a time. For example, if the user is prompted for a number and they enter

```
3.1415
```

the first time that getchar() is called, it will prompt the user and then return the character `'3'`. The second time it is called it will return the character `'.'` that was already entered. The third time, `'1'` will be returned, and so on. When the user is done entering their input, they can signal this by entering `EOF`. This is done in the terminal with `ctrl+d`. The constant `EOF` is defined in the `stdio.h` file.

Here is a sample program, which calls `getchar()` in a loop until it received an `EOF` and stores the characters in a character array.

```c
#include<stdio.h>

int main(){
    char arr[100];
    char c;
    int i = 0;

    while ((c = getchar()) != EOF){
        arr[i++] = c;
    }
    arr[i] = '\0';

    printf("Input: %s\n", arr);
}
```

One minor detail here is that after the loop terminates, we manually add the null character `'\0'` to the end of the string in the character array. This wasn't necessary in previous examples when an entire string is being copied into the array at once from a string literal or from `scanf()` because the null character was automatically included in these situations. However, here we are just copying individual characters into the array ourselves, so there is no automatic inclusion of anything.

There is also a simple function to write to standard out one character at a time, called `putchar()`. Its header is

```
int putchar(int)
```

The function takes one argument, which is the character to be written. The return value will the same as the input if it worked correctly, or will be `EOF` in case there is an error. We could modify the previous example, to replace the `printf()` with `putchar()`.

```c
#include<stdio.h>

int main(){
    char arr[100];
    char c;
    int i = 0;

    while ((c = getchar()) != EOF){
        arr[i++] = c;
    }
    arr[i] = '\0';

    for (int i = 0; arr[i] != '\0'; i++){
        putchar(arr[i]);
    }
}
```

In the last `for` loop, we are now using the null character ourselves to determine where to stop printing characters.

Logic

A lot of the interactions with logic is very similar between Java and C. All of the comparison operators are the same and have the same syntax, as do the logical connectives. The biggest difference is in how true and false is interpreted.

True and False

In Java, there is a boolean data type, which has two possible values, designated by the keywords `true` and `false`. This is another abstraction created by the control that Java has by filtering everything through the Java Virtual Machine.

The CPU in your computer doesn't have any data types called true and false, and therefore, C doesn't have the boolean data type. Instead, true and false are encoded as integers directly in a simple way:

0 is false

$\neq 0$ is true.

Therefore the condition in the conditional statement

```
if(1)
```

will be interpreted as true, and so will the condition in

```
if(-3)
```

but the condition in

```
if(0)
```

will be interpreted as false.

A lot of the time, you won't have to think about the encoding of true and false. However, sometimes you can use it in creative ways. For example, we can simplify the loop that prints a string using `putchar()` to the single line:

```
while(putchar(arr[i++]));
```

Recall that putchar will echo the character that was output, so when it gets to the end of the string, it will output the null character `'\0'` and return it as well. Since the null character has the ASCII encoding of 0, that means that `putchar()` returns 0 in this case, and since 0 is false, the while loops will end.