Beginner’s C++
Tutorial 02

Josh ‘Faceless J’

Welcome

Welcome to the second installment of my ‘*Beginner’s C++*’ tutorial series. Last time we talked about creating your first application and how to output text to the screen. However, that on its own isn’t very useful. So in this latest tutorial, we will be covering reading input from the keyboard and using variables. So, let’s get to it!

Introducing; Variables!

First off, you need to make a new project. If you can’t remember how to do this, refer back to Tutorial 1. As we did last time, we shall start off the actual coding by looking at the entire source code for this part of the tutorial.

```
#include <stdio.h>

int main ()
{
     char c;
     int i;
     c = 'A';
     i = 66;
     printf("%c %i\n", c, i);
     printf("Press enter to continue...\n");
     scanf("(space)*");
     return 0;
}
```

The framework for this code is the same as in Tutorial 1. If you need to review it, go back and have a read-through again. Before we continue, run this program so you can see what it will output.

Looking through the code, you will notice some new instructions. The first two you will notice are:

```
char c;
int i;
```

These create variables of the type `char`, which hold characters, and `int`, which hold integers, or whole numbers. These variable types are called primitive types, since they are built into the C++ language.

Other primitive types which you will use commonly include:
`bool` – A boolean value. True or False.
`float` – A floating point number. Has a decimal component.
`double` – Same as a float, except more precise.
I will not go into them in detail just yet.

Also, with the names I picked for the variables were just for an example. You could pick something like:

```
char myCharVariableWithAStupidlyLongName_12;
```

However, there is not much point using a name this long, since you will have to type it every time you wish to use it. You should choose something succinct that will describe what it will hold.

So, now that we have some containers to hold stuff, we can fill them:

```
c = 'A';
i = 66;
```

You will remember in Tutorial 1, I mentioned that apostrophes represent single characters. This is the situation I was describing. A `char` variable can only hold a single character. If you were to use the line `c = “A”;` instead, you would get an error from the compiler. In VC++, the error would look something like this:

```
error C2440: '=' : cannot convert from 'const char [2]' to
'char'
```

The exact details of why this is will be covered in a future tutorial that deals with arrays.

Anyway, back to the code. The `=` operator assigns the right hand value to the left hand variable. The first line here will assign the variable `c` the value of `A`. The second line will put a `66` in the variable `i`.

Now, to print out these variables to the console, we use the `printf` function, like we did in Tutorial 1:

```
printf("%c %i\n", c, i);
```

Now, this looks a bit confusing. As I mentioned in the last tutorial, the `"%c %i\n"` is the format string. The `%` signs and the letter after them (placeholder) tell the computer to insert a variable there. Depending on the letter you choose will depend on what sort of variable the computer will use. There are many placeholders that can be used, along with some nifty little tricks, however, I will only list the most common ones here:
`%c` – A char
`%i` – An integer
`%f` – A float
`%s` – A character string
After the first comma is a list of variables. These are the variables, in order, that will be used by the placeholders. So, the `%c` will be replaced by the char value of `c`, and `%i` will be replaced by the int value of `i`.

This instruction will print out what is in container `c`, a space, what is in container `i` and then an end line. As you would expect, the output is:

```
A 66
Press enter to continue . . .
```

Easy as pie, eh? Let's move onto the next section.

Variable Arithmetic

It's all well and good to be able to print out variables, but what if you want to change variable during the running of the program. They are, after all, called *variables*. Well, I'll show you how to do exactly that in this section. Copy the following source code, and compile and run it:

```
#include <stdio.h>

int main ()
{
	char c;
	int i, j = 5;
	c = 'A';
	i = 66;
	i = i + j;
	printf("%c %i\n", c, i);
	printf("Press enter to continue...\n");
	scanf("(space)*");
	return 0;
}
```

What is this comma I've used in the line `int i, j = 5;`? When declaring variables of the same type, you can simply separate them by a comma, rather than having to type:

```
int i;
int j = 5;
```

Also, you will notice I assigned a value to the integer container `j` when it was created. This again, is to with the laziness of programmers.

The main line of code I wish to show off in this section is:

```
i = i + j;
```

As you will guess, it adds `i` (66) and `j` (5) and will obviously result in 71. However, you may be wondering why this does not result in a circular reference (Since, when this line completes, `i` = 71, but `i = i + j`, so it would become 76 and so forth). The reason for this is that the computer will calculate the right hand side entirely and will end up (in the case of `int`s) with a single number. This number is then assigned to the container, `i`. So, now, when the program goes to print out `i`, it will print 71 instead of 66.

Now, for a really interesting instruction. Replace the line:

```
i = i + j;
```

with:

```
c = c + j;
```

Surely this wouldn't work, would it? How can you add a number and a letter? This leads me onto the next topic, type casting.

It's an int, no, it's a char, no, it's both!

As just mentioned, this section will cover type casting. Now, there are two types of casting. Implicit and explicit. Let's discuss implicit casting first.

Implicit casts, like `c = c + j;`, just happen. The compiler handles them. However, very few types implicitly cast to one another. It just so happens that `char`s and `int`s do. But this raises the question, how can integers and characters be equated and operated on like this? It is simple really. In Windows and Linux, and a lot of other operating systems, characters actually represent ASCII (Or unicode) letters. ASCII is used to represent letters in a computer system. The letter A is the number 65. It also worth noting that upper and lower case are different, in that, a = 97. All the other letters continue on from these numbers, ie, B = 66, b = 98 and so on. Basically what is happening in the line `c = c + j;` is the computer takes the integer values, adds them, and converts it back to a letter. In this case, it is 65 + 5 = 70. If you ran the program with the new line, you would know that 70 is equals F. The same thing happens if you were to use arithmetic on two characters.

Now, the other sort of casting is explicit. This means you tell the computer to perform a type cast, and is done using the following syntax:

```
c = (int)c + j;
```

You use a set of parentheses around the type to convert to (in this case, an `int`) directly in front of the variable (or constant) to convert.

The best way to see this in action is to change the `printf` line to the following:

```
printf("%c %i\n", i, c);
```

When you run the program, you will get this output:

```
B 65
Press any key to continue . . .
```

This printed out the integer value of `c` and what character `i` matched to. (Note, I did not use an explicit type cast here, because it is not needed due to the implicit casting for chars and ints)

One last note to make about type casting, is that it is usually advisable to use it whenever you're not sure of the output. This is best explained by an example:

```
int i = 7, j = 2, k;
double l;
k = i / j;
l = i / j;
```

We all know that 7 / 2 is 3.5. However, if you were to output `k` and `l`, they would both be 3. This is obvious for `k`, since it is an `int`, however, `l` is a `double` (a decimal number), so it should 3.5 not 3. The reason is that the output of a calculation like:

`int / int` is an `int`

and so, the `double l` is being used to hold an `int`, which is pefectly reasonable (Since an `int` can be turned into a `double` by adding .0 to the end).

To get the correct output from the division, at least one (or both) of the numerator or denominator need to be `double`s or `float`s. This is done through explicit type casting if they are both integers. So, to get the correct output, the line:

```
l = i / j;
```

would become:

```
l = (double)i / j;
```

Running this new line will make `l` = 3.5, which is correct. You could do the same thing to the `k = i / j;` line however, there is no point, since it converts back to an `int`, because `k` is an `int`.

Last but not least, keyboard input

I realise this tutorial is getting a bit lengthy, so I'll try to make this last section as short and sweet as possible. We already know about `printf`, which outputs stuff to the screen. I have briefly touched on `scanf`, however I will go into more depth now. Try out this code:

```
#include <stdio.h>

int main ()
{
	char c;
	printf("Enter a letter:\n");
	scanf("%c", &c);
	printf("The letter you entered is: %c\n", c);
	printf("Press enter to continue...\n");
	flushall();
	scanf("(space)*");
	return 0;
}
```

The new line is `scanf("%c", &c);` which introduces something new. That is, the use of an ampersand (&) in front of the variable c. This will be discussed later in the tutorial on pointers. Just know that this allows scanf to modify the value of c. two new things.

The `scanf` call will only stop when you press enter. However, it will only get the first letter in this case. It will put the letter it got into the `char` container c. It then outputs the character to the screen. Tthe rest will go into the input buffer. This is a problem (or nifty little feature, depends what you need it for). Enter, `flushall();`. This function flushes the input buffer. You may be wondering why, even if the user enters 1 letter the the second `scanf` skips right through without the `flushall()` call. This is because on the end of the buffer is sneaky little '`\n`' character (because you pressed enter to input the letter). So this '`\n`' is put straight into the second `scanf`, causing the program to finish. `flushall()` removes the '`\n`'.

You may be wondering what will happen if the user types a number instead of a character. Well, the ASCII code has values for every key on the keyboard, so the character digits 0-9 are represented by other numbers (‘0’ = 48). So, if you wanted to perform numerical arithmetic with character value, rather than the ASCII value, what can you do? It is simple really. You subtract ‘0’ from the character. Say the user entered ‘1’. To get this number, you can use:

```
i = c - '0';
```

Where `i` is an `int` and `c` is a `char` (‘1’). This will work out what ‘1’ (49) – ‘0’ (48) will be. Low and behold, 49 – 48 = 1 which is the number we wanted.

This tutorial introduced a few new concepts to you. I suggest you spend a while playing around with them. Remember, and I know this sounds corny and is overused, but that is only because it is true, practice makes perfect!