LESSON 2
VARIABLES, OPERATORS, EXPRESSIONS, AND USER INPUT

PROF. JOHN P. BAUGH
PROFJPBAUGH@GMAIL.COM
PROFJPBAUGH.COM

CONTENTS

INTRODUCTION 3

Assumptions 3

2.1 – Variables and Data Types 3

2.1.1 – Numeric Data Types: Integer, Float, and Double 4

2.1.2 – Characters 5

2.1.3 – Boolean Variables 6

2.2 – Constants 7

2.3 – Arithmetic Operators 8

2.4 – User Input 10

2.5 – Summary 12

Exercises 12

INTRODUCTION

In this lesson, we begin formalizing our understanding of fundamental programming concepts. C++ and languages like it make extensive use of variables, perform operations on these variables and form expressions from them. We also explore how to obtain input from the **user** (the person running your program.)

ASSUMPTIONS

I assume that you've read through and are comfortable with the material in the previous lesson. You should at least have a good idea about how to start Visual Studio 2010 and create a new project and a source file. Since this is still early in your C++ and IDE usage education, it is perfectly okay to refer back to the material in lesson 1. Eventually, most of the tasks will become second nature.

So, I assume before you start this chapter that you know how to print a simple line of text (such as "Hello World" or your name) to the console using `cout` statements, and that you are comfortable with what `endl` does. If you are not, go back to lesson 1, and also make sure you can do the exercises at the end of that chapter.

2.1 – VARIABLES AND DATA TYPES

In this section, we explore various **data types** that are built-in to the C++ language. Later on in this book, we'll see that you can use many different complex data types or even create your own. But the fundamental data types that are provided with C++ are used extensively.

To understand data types better, we must also understand the concept of a variable. A **variable** is a location reserved for data in memory. The name of the variable, called its **identifier** is used to refer to the storage location. This storage location may have a known value, or the value may not be known immediately.

As an example of data types and variables, consider the following:

```
int myNum = 5;
```

The first part of the above statement, `int`, indicates to the compiler that it should reserve memory for an **integer** variable, meaning the variable can hold whole numbers and their opposites (e.g., 10, 4565, -45, 0, 99, -33.) The identifier of the reserved memory is to be `myNum`. This is the name to which we will refer when we wish to access the variable. The last part, `= 5;` is an **assignment statement**, followed by the semi-colon to delimit the statement. Since in this example, the initial value of the variable is assigned, we call this process **initializing the variable**.

Note that in C++, all statements must end with a semi-colon. This is a rule of C++'s grammar, called the syntax. The **syntax** of a programming language consists of the rules for forming **expressions** in that language, which are any combination of the symbols of that language. On the other hand, the meaning of the expressions is known as the **semantics** of the language.

Another language, like Java, might have identical syntax in certain situations (as is the case with declaring and assigning the value 5 to an integer named `myNum`.) Other languages, like ***Visual Basic*** might have a statement such as:

```
Dim myNum as Integer = 5
```

Notice that the declaration of a variable is quite different in Visual Basic, but it should be clear what this is doing. Note the absence of the semi-colon as well. The syntax is different, but the essential semantics of the statement is the same.

Note that you can also declare a variable and define its value at a later time, such as the following:

```
int myNum;
myNum = 5;
```

Here, we've declared the variable `myNum` on one line, and then defined its value on the next line.

The fundamental data types that we'll take a look at now are:

- Integer
- Float
- Double
- Char
- Boolean

2.1.1 – NUMERIC DATA TYPES: INTEGER, FLOAT, AND DOUBLE

The fundamental or **primitive** numeric data types in C++ are integers, floating point numbers, and doubles. The syntax for these data types of these values are as follows:

Syntax	Description
int	Integers are whole numbers and their opposites. E.g., 45, 5000, 2400, -1234
float	Real numbers up to around 7 digits. E.g., 400.56, 435.5, 2.3, 3.14159
double	Real numbers up to around 15 digits. Same as float, but with higher precision available

Let's consider an example with integers being added to produce a result. We'll more formally treat operations upon integers later, but for now, let's consider this simple example to see integers in action.

```
1   //Lesson 2.1:  Simple Numeric Example
2   #include <iostream>
3   using namespace std;
4
5   int main()
6   {
7          int a;
8          int b;
9          int result;
10
11         a = 15;
12         b = 20;
13
14         result = a + b;
15
```

```
16        cout<<"A is: "<<a<<endl;
17        cout<<"B is: "<<b<<endl;
18        cout<<"The result of A + B is: "<<result<<endl;
19
20        return 0;
21    }
```

Code 2.1-1: Simple Numeric Example

Observing the code sample ***Code 2.1-1***, we can see that we declared three integers on lines 7 – 9, namely `a`,`b`, and `result`. Then, on line 11, we assign the value 15 to `a`, and then assign the value 20 to `b` on line 12. On line 14, we perform arithmetic on variables `a` and `b`, and store the result in the variable `result`.

On lines 16-18 we print out a text string indicating what we're printing, and then the values of `a`, `b`, and `result`, respectively.

If you build and run the program, you will see the following output:

```
A is: 15
B is: 20
The result of A + B is: 35
Press any key to continue . . .
```

Output for Code 2.1-1

This code prints out essentially what is expected, namely, it echoes the values of the variables `a` and `b`, and then prints out the `result`.

2.1.2 – CHARACTERS

The two character types that we are interested in are the following:

Syntax	Description
char	Character. Represents a single character from the ASCII characters. A variable of type char is 8 bits (1 byte) in size. E.g., 'A', 'b', '1'
wchar_t	Wide character. Represents a single character, and typically supports a larger character set than standard ASCII. Often, this is the Unicode set. However, the size of wchar_t is compiler-specific. It can be as small as 8 bits, but is often 16 bits or 32 bits depending on the compiler and platform.

We will focus more on the `char` type for our purposes here, but essentially everything we do with `char` can also be done with `wchar_t`.

Let us consider the following example:

```
1    //Lesson 2.1:  Character data type
```

```
2   #include <iostream>
3   using namespace std;
4
5   int main()
6   {
7          char myChar = 'A';
8          char anotherChar = 'J';
9
10         cout<<"The first character is : "<<myChar<<" and the second is "
11                <<anotherChar<<endl;
12
13         return 0;
14  }
```

Code 2.1-2: Character data

The output is as follows:

```
The first character is : A and the second is J
Press any key to continue . . .
```

Output for Code 2.1-2

Notice that although the code to print the characters is on two separate lines (namely, lines 10 and 11), the actual text that is printed to the console is on one line. Why is this?

We are able to break a cout statement onto different lines as long as we don't put a semicolon until the very end, and as long as we have individual **string literals** without a break in the double quotes. For example, if you hit enter before the double quotation mark at the end of line 10 and moved it to line 11, a compiler error would result.

It should be noted that "under the hood", characters in C++ are actually integers. They are values in the ASCII table.

2.1.3 – BOOLEAN VARIABLES

Variables of type **`bool`**, called **Boolean variables**, named after the mathematician George Boole (1815 – 1864), can only contain one of two values, namely, `true` or `false`. Note that the syntax in C++ uses lowercase `bool`, but when referring to the full name or speaking of variables that contain truth values, we capitalize Boolean, because Boolean is derived from a proper name.

This is used extensively by programs where decisions must be made based on certain criteria. In fact, nearly all large programs written in languages like C++ will contain a great deal of decisions (do I take this code path, or the other?) We will cover this more extensively when we discuss control structures.

In C++, the value true is equal to the integer 1, and the value false is equal to 0. In this manner, `true` and `false` act similarly to named constants which we will explore later in this lesson.

Let's consider a simple example of Boolean variables in action.

```
1   //Lesson 2.1:  Boolean data
2   #include <iostream>
3   using namespace std;
```

```
4
5   int main()
6   {
7          bool truthVal1 = true;
8          bool truthVal2 = false;
9
10         cout<<truthVal1<<endl;
11         cout<<truthVal2<<endl;
12
13         cout<<endl;
14         cout<<boolalpha<<truthVal1<<endl;
15         cout<<truthVal2<<endl;
16
17         return 0;
18  }
```

Code 2.1-3: Boolean variables

The output is thus:

```
1
0

true
false
Press any key to continue . . .
```

Output for Code 2.1-3

Careful note should be made of the output here. Notice on lines 10 and 11, the resultant output is a 1 and 0, respectively. This is because, as I stated earlier, `true` is 1, and `false` is 0. However, often you want to print out the actual words true and false to the console. This can be done with a special **flag**, `boolalpha` which causes the output stream to print the strings "true" and "false" instead of the integer equivalent. Notice that we only place the flag once, on line 14, and it still affects subsequent printing of Boolean data, such as on line 15. This is because `boolalpha` is a flag that is said to be **sticky**. It must be explicitly reset using `noboolalpha` in order to return the printing to the default behavior (printing 0s and 1s.)

2.2 – CONSTANTS

In some cases, we may have variables that we don't want to change during the program. In this case, we use **constants**, using the keyword const.

An example might be a sales tax, a constant numeric value like pi, or some other value that we don't want to be changed during execution of the program.

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6          const double PI = 3.14159;
7
8          cout<<"Pi is : "<<PI<<endl;
```

```
9
10        return 0;
11    }
```

Code 2.2-1: Constant data

In Code 2.2-1, note that we use the keyword `const` before the data type (`double`) of our variable `PI`. Note that I used capital letters for the identifier of the constant. This is not a syntax requirement (the compiler won't flag an error if you don't use all capital letters), but it is what we call a **naming convention**. It is useful to name constants with all capital letters so that throughout the program, if you or another programmer sees a variable name and it is all caps, you know immediately that it's a named constant. The output for this is trivial, so let's move on and consider what happens if we try to change the value of this constant.

Let's see what happens if we try to change the value of PI:

```
1     #include <iostream>
2     using namespace std;
3
4     int main()
5     {
6           const double PI = 3.14159;
7           PI = 3.1415926;
8           cout<<"Pi is : "<<PI<<endl;
9
10          return 0;
11    }
```

Code 2.2-2: Trying to change constant data

Line 7 will cause there to be a compiler error when we try to build the program. In Visual Studio 2010, we would receive the following:

```
error C3892: 'PI' : you cannot assign to a variable that is const
```

This error is quite clear. It essentially tells us we can't modify a variable that is `const`. This may seem to be a bit of an annoyance, but can actually be quite useful. ***Code 2.2-2*** may seem fairly trivial, but in large programs, we might forget and might try to modify a constant when we shouldn't. By declaring a variable as const, we enforce maintaining its value the way it was declared.

Note that it is necessary to initialize the value of a `const` when you declare the variable. With non-constant variables, you can assign a value on the line in which it is declared, or later on, but by the nature of a constant variable, you obviously must assign it when you first declare it, as we do on line 6 in Code 2.2-1.

2.3 – ARITHMETIC OPERATORS

In this section, we discuss **operators** which allow us to perform various operations on variables. Specifically, in this section we'll focus on **arithmetic operators** which allow us to perform – you guessed it – arithmetic on variables and values. We'll look at operators involved in logical decisions and comparison in a later chapter. We've seen a simple example earlier in this chapter when we added two integers together. C++ allows for much more than just addition (which is very fortunate!)

The operators that we're concerned with are **binary** operators, which means that they take two **operands**. Operands are just the variables or values that the operators act upon. Consider the following:

Operation to perform	Operator in C++	Description	C++ Expression
Addition	+	Sums its operands	a + 7
Subtraction	-	Subtracts the second operand from the first operand, returning the difference	27 – c
Multiplication	*	Multiplies the operands, returning the product	2 * PI * r
Division	/	Divides the first operand by the second operand and returns the quotient	30 / 2
Modulus	%	Divides the first operand by the second operand and returns the remainder (residue)	5 % 3

The only arithmetic operator you may not be extremely familiar with is **modulus**. This returns the remainder after integer arithmetic. In algebra, and symbol for modulus is typically just mod, as in the following exampls:

5 mod 2 = 1 since 5 / 2 = 2, *with a remainder of 1*

10 mod 4 = 2 since 10 / 4 = 2, *with a remainder of 2*

16 mod 2 = 0, since 16 / 2 = 8, *with a remainder of 0*

It should be very clear that with modulus, we are more concerned with the remainder from the division problem, not the quotient.

You can string these operators, and the standard order of operation is used, that is:

Parenthesis, (Exponents), Multiplication/Division, Addition/Subtraction

Note that Modulus would actually fall in with Multiplication/Division, so you may have to come up with a new mnemonic device. Maybe instead of **Please Excuse My Dear Aunt Sally**, it could be **Please Excuse My Dear Maniacal Aunt Sally**.

Let's look at an example with the operators used extensively:

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6          int num1 = 10;
7          int num2 = 3;
8
9          cout<<"num1 + num2 = "<<num1+num2<<endl;
10         cout<<"num1 - num2 = "<<num1-num2<<endl;
11         cout<<"num1 * num2 = "<<num1*num2<<endl;
```

```
12          cout<<"num1 / num2 = "<<num1/num2<<endl;
13          cout<<"num1 % num2 = "<<num1%num2<<endl;
14
15          cout<<"num1 + 7 * num2 - 3 = "<<num1 + 7 * num2 - 3<<endl;
16
17          return 0;
18  }
```

Code 2.3-1: Operators and operands in action

The example ***Code 2.3-1*** shows each of the operators in action on two integers, num1 and num2 on lines 9 – 13. Then, we mix operators on line 15 to show that you can use larger mathematical expressions. The output is thus:

```
num1 + num2 = 13
num1 - num2 = 7
num1 * num2 = 30
num1 / num2 = 3
num1 % num2 = 1
num1 + 7 * num2 - 3 = 28
Press any key to continue . . .
```

Output for Code 2.3-1

2.4 – USER INPUT

We've been doing a whole lot of printing to the console lately. In fact, it should be pretty familiar to you by this point (assuming you've been following along in your IDE.) However, programs would be pretty boring, and not always very useful if the user couldn't interact with them at all. User input comes in many different forms, ranging from the mouse clicking and dragging, to keyboard typing, to pushing down the left thumbstick or right trigger on an Xbox 360 controller.

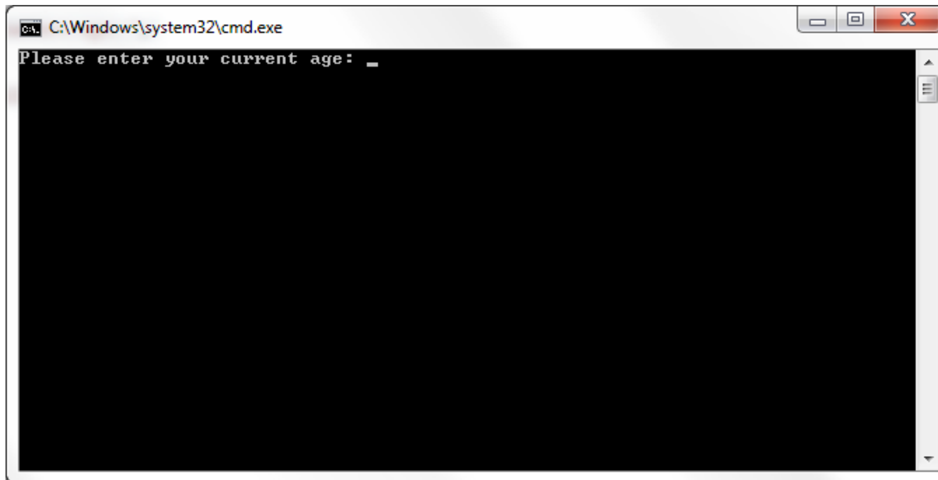
For our entry into user input in C++, we're going to be concerned with input from the keyboard, the standard input device on almost all personal computers.

The best way to learn how to obtain input from the user is with an example:

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6           int age;
7
8           cout<<"Please enter your current age: ";
9           cin>>age;
10          cout<<endl;
11          cout<<"In 10 years, you will be "<<age + 10<<" years old."<<endl;
12
13          return 0;
14  }
```

Code 2.4-1: User input

Most of what we see in ***Code 2.4-1*** is pretty typical. But there is one major difference. Line 9 has a new variable, `cin`, pronounce "see in" which stands for "console in". Notice also that the symbols following the `cin` variable are greater than signs, instead of less than signs (as is the case with `cout`.) These two greater than signs are called the **stream extraction operator**, because data is being extracted from the standard input stream and placed into a variable. In this case, the variable the data will be placed in is `age`.



Screenshot of the when Code 2.4-1 is initially run

We can see in the above screenshot that where the `cin` statement occurs, the program is actually waiting for us to enter input from the keyboard. Once we do so, and then hit enter, the variable `age` will have that value stored and the rest of the program will execute.

The entire output, assuming I enter the value 20 as the age, is as follows:

```
Please enter your current age: 20

In 10 years, you will be 30 years old.
Press any key to continue . . .
```

Output for Code 2.4-1 with 20 as the input from keyboard

Now we have our first real *interactive* program. The user enters some value as input, and a calculation is performed and printed to the user.

Before you fall asleep, or possibly wonder why we're dealing with so much math-related stuff, it is important to note that a very large number of programs – from embedded systems in cars and appliances, to video games on all sorts of platforms, require many mathematical calculations to be performed constantly. So, mathematics and interaction with the user are both crucial topics in all levels of programming. Although it is useful to be good with mathematics when you go into computer science and software engineering, it's not necessary to be a mathematical whiz. In fact, as long as you can translate mathematical equations into code, the code will perform the calculations for you. This is good, because even the world's greatest mathematicians, geniuses and savants cannot perform mathematical calculations as fast as a computer program can.

2.5 – SUMMARY

We accomplished quite a bit in this chapter. We explored how to declare variables and how to use them. Arithmetic operators were also discussed, and we saw how we can directly write mathematical expressions in `cout` statements, or store resultant values in variables.

We learned about integers, real numbers, characters, and Boolean variables. We learned that characters as essentially just integers "under the hood". And, finally, we learned how to obtain input from the user via the `cin` variable, which uses the standard input device, which is typically the keyboard.

EXERCISES

1. Write a program that solves for y, given a slope (m), a value for x, and the y-intercept (b). Hint: The famous equation, in slope-intercept form is: $y = mx + b$. Ask the user for m, x, and b and then print the resultant y value.

2. Write a program that finds the circumference of a circle, given the radius (entered by the user via the keyboard.) The equation for circumference is: $C = 2\pi r$. Hint: Use a named constant for PI = 3.14159.

3. Write a program that finds the area of a square, when the user enters the length of a side. Hint: A = side * side.