Tutorial 03 - Fundamental Datatypes and Operations

## Introduction

You've already used variables a little bit and the Integer data type, so this tutorial takes a deeper dive into variables and the Fundamental Data Types, and how to use them with arithmetic operations.

This tutorial will also introduce you to the basics of input and output in C++ programs, so you can write out the values in variables and read input from the keyboard into them.

## Data Representation

As covered before, everything in a computer is stored as sequences of 1s and 0s, becoming sequences of bytes, which combine further to become larger numbers. These numbers are categorized into several basic categories, called **data types**, which determine the kind of value they may hold and how they are treated in memory. Several data types are "built in" as part of the original C language; several more types were added in C++, and even more types have been added as the C++ standard has grown. C and C++ share the very powerful feature of **definable data types** which allows your programs (or software your programs build on) to create new types for bespoke requirements.

Some languages do not have data types, such as Python. These languages interpret the type based on the value they hold and how it's used. C/C++ don't work that way, because that interpretation step can take up valuable processing time and encourage sloppy practices that lead to program errors. The compiler helps to enforce the type rules so that everything in the running program works as it should, without performance issues.We'll start with the **Fundamental** types, which are useful for most basic operations.

## Variables

Variables are used to temporarily store information of all types: numbers, letters, words, lists of student IDs, website URLs… whatever you need in your program. Temporary can mean anything from "just until the next operation" to "until the program stops running" – it depends on how you declare that variable and ***scope*** which we'll go over in detail later.

Variables are easy to work with and fast, so don't worry about having too many variables and slowing your program down – what's important is that the program is clear to you.

### Declaring a Variable

Before a variable can be used it must be ***declared***. This informs the compiler that you want to use a new "symbol" so it will then recognize it in the future, and know how to work with it. A simple declaration looks like:

int my\_int;

which defines an **integer** variable called my\_int . The variable my\_int can now be used, such as by giving it a value:

my\_int = 0;

Assigning the value also serves to **define** the variable, which ensures that the compiler knows what the variable is and that it assigns an appropriate chunk of memory for it. In this case, that chunk of memory will have the value 0 as well.

### Declaring and Defining Variables in a Single Statement

If you want to you can declare and define variables at the same time, which saves you some typing and is also a good practice: your code is clearer and the variable always has an appropriate value.

int my\_int;

my\_int = 0;

can be written as:

int my\_int = 0;

When a new variable is declared its value is **indeterminate**, which is technical for "probably garbage". It is essential that you get in the habit of assigning values to variables immediately – ideally when they are declared. This prevents bugs from arising by using *uninitialized variables*:

int my\_int = 0;

int my\_second\_int; // danger – not initialized!

int my\_third\_int; // danger – not initialized!

my\_third\_int = my\_first\_int + my\_second\_int;

The value of my\_third\_int is totally unpredictable here. The compiler will generate lots of warnings about variables if it detects possible problems. Pay attention to them!

### Variable Declaration vs Definition

The declaration gives the compiler basic information about a symbol: its type and its name. The *definition* also provides all the memory location of that symbol, so the variable is connected to real storage.

The compiler only needs to have a declaration for something in order to compile a .cpp file. If a symbol is declared but never defined, you will have linker errors stating there are "undefined symbols".

## Fundamental Data Types

So far you have used string literals such as "Hello World!", integer literals such as 25, and basic integer variables. There are several other data types that you can use in C/C++, these are called **fundamental data types** as they are the most basic kind of variables that can be used in C/C++ and are built into the language.

Note that the # Bits column is determined by several factors. These are the sizes for the data types

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **# Bits** | **Notes** | **Value Range** |
| bool | ?? | size unspecified | true or false; 0 or 1 |
| char | 8 | ASCII table | -127 to 127 |
| unsigned char | 8 |  | 0 to 255 |
| short | 16 | Half Integer size | –32,768 to 32,767 |
| unsigned short | 16 | Half Integer size | 0 to 65,535 |
| int | 32 | By default optimal size for CPU | –2,147,483,648 to 2,147,483,647 |
| unsigned int | 32 | By default optimal size for CPU | 0 to 4,294,967,295 |
| float | 32 | Same # bits as integer | 3.4E +/- 38 (7 digits) |
| long | 32 | Integer size or bigger | –2,147,483,648 to 2,147,483,647 |
| unsigned long | 32 | Integer size or bigger | 0 to 4294967295 |
| double | 64 | 2 x Integer size | 1.7E +/- 308 (15 digits) |
| long long | 64 | Bigger than long | –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |

You can check the size of any data type using the sizeof() operator. sizeof() returns the number of bytes (or technically octets) in the data type or variable you pass in.

int integer\_size;

integer\_size = sizeof(int);

cout << "The size of an integer is " << integer\_size << " bytes";

On a standard Windows PC, the printed result of this chunk of code is 4, because an integer is 4 bytes or octets in length.

### Integer

A bit more on integers - it's important to understand that an integer can never hold a fractional value, and when machine code is generated to do math operations on integers, any fractions are simply discarded – so you can never assume that division with integers will produce totally accurate values.

int result

result = (7 / 2) + (7 / 2);

The simple code above just divides 7 by 2 twice, adds the results and stores it in an integer. You might think the result would be 3.5 + 3.5 = 7. But no – because 7 and 2 are both integers, the result of each division is 3 – the .5 remainder is discarded immediately so result is 6. We'll get into how to work around this when necessary, but the general solution is to use the correct data type for the job, which in this case would be a float.

### Floating Point Numbers

There are two types of **floating point numbers**, declared as floatordouble, which have both a whole and a fractional part. The difference between the two types is that double uses **8 bytes** per variable instead of **4** bytes for floats, and this means doubles can store bigger and more accurate fractional numbers. You will normally use floats as they take up less space and are much faster than using a double; you only need a double for very accurate calculations. Here's some code using float:

float float\_pi;

float\_pi = 3.14159265358979f;

cout << "The value of pi is " << float\_pi << endl;

Here float\_pi is of type float, and 3.14159265358979f is a **floating point literal**. The 'f' tells the compiler that the literal is a float as opposed to a double. It isn’t absolutely necessary to do this (using modern compilers), but forgetting the 'f' causes a warning during compilation.

cout automatically detects that it is outputting a floating point number and displays it accordingly. If you try this code you might notice that it doesn't output the full fractional value, but instead just 3.14159. This is because what is called "default precision" limits cout to using only the first 6 significant digits of the decimal portion. This can be increased, but don’t worry about that for now.

Be aware that when using cout to display floating point values, very small and very large might be displayed a form of **scientific notation**, e.g. 0.000000123 might be shown as 1.23e-007 (this means 1.23 x 10-7). The format of the number output is dependent on its size and the precision cout has been told to print.

### Character

Character variables are declared as char. They are 1 byte in size, and contain a number that represents a single character (e.g. 'A', 'z', '3', '#', etc.). As mentioned previously, 1 byte can store 256 different values. The number that is stored corresponds to the **ASCII** (pronounced 'Asskey', means American Standard Code for Information Interchange) value of the character. Use the following code to declare and define a character variable:

char my\_char;

my\_char = 'A'; // NOTE – use single quotes for a single character

cout << "The character is " << my\_char << endl;

A char can be treated just as a number (because it is a number!), and manipulated using math operations: addition, subtraction, etc. It can also be assigned to variables of other types.

The ASCII code for capital A happens to be 65, so this code:

char my\_char;

int my\_int;

my\_char = 'A';

my\_int = my\_char; // int = ASCII value of A, which is 65;

cout << "The character is " << my\_char << endl; // outputs A

cout << "The character ASCII code " << my\_int; // outputs 65

will output

The character is A

The character ASCII code is 65

The conversion from char to int happens behind the scenes in this case, and it’s an important part of the C/C++ language. We’ll get into that in more detail later.

### Boolean

A **boolean** **variable** is declared using the keyword bool. A bool simply stores a value of 1 or 0, representing true or false respectively. You can set a boolean using 0 and 1 literals, or the true and false keywords, e.g.

bool my\_bool;

my\_bool = true; // equivalent to **my\_bool = 1**;

cout << "The value of my\_bool is " << my\_bool << endl;

my\_bool = false; // equivalent to **my\_bool = 0;**

cout << "The value of my\_bool is now " << my\_bool);

will output:

The value of my\_bool is 1

The value of my\_bool is now 0

Notice that the output from cout is 1 or 0, instead of true or false as you might expect. This is because a boolean is generally used in a **logical** way, not as text (true and false are just keywords that represent 1 and 0. You will see the real value of a Boolean later in the unit. But if you want to output the value of a boolean in text format, cout does allow you to do so – but you first need to change its output format for Booleans. Compare the code below to the previous code:

bool my\_bool;

my\_bool = true; // equivalent to **my\_bool = 1**;

cout << "The value of my\_bool is " << std::boolalpha << my\_bool << endl;

my\_bool = false; // equivalent to **my\_bool = 0;**

cout << "The value of my\_bool is now " << std::noboolalpha << my\_bool);

The value of my\_bool is true

The value of my\_bool is now 0

As you can see std::boolalpha is used to switch output for Booleans to text, and std::noboolalpha switches it back to 1 and 0;

### Signed and Unsigned Data Types

By default, int and char are **signed data types**, which means that the type can store both positive and negative values. If chars are 1 byte (as they are on most computers, but it can vary) then the range of values they can contain are -128 to +127. For ints, which are 4 bytes in length, the range will be -2147483648 to +2147483647. (In case you’re wondering, the highest bit is set to 1 for a negative number, and 0 for a positive number. This means that one bit in a signed data type isn’t usable for the number itself).

You can explicitly declare a signed data type, but it isn't required:

signed int my\_signed\_int; // same as int **my\_signed\_int;**

Sometimes you might know that you don't need negative numbers, and therefore you can declare ints and chars as unsigned. This means that the highest bit isn’t needed for the sign, and the range of values that can be represented for chars becomes 0 to 255, and for ints it becomes 0 to 4294967295. For instance:

unsigned int my\_unsigned\_int;

my\_unsigned\_int = 4294967295;

cout << "The value of my\_unsigned\_int is " << my\_unsigned\_int;

Be careful to use a data type that has sufficient range for the values you need. For example, if an **integer** (signed or unsigned) is set to its maximum value and you try to add one then **there is no error**. What happens is the number **wraps around back to zero**, much like a car odometer:

unsigned int my\_unsigned\_int;

my\_unsigned\_int = 4294967295;

cout << "The value of my\_unsigned\_int is " << my\_unsigned\_int << endl; // outputs 4294967295

my\_unsigned\_int = my\_unsigned\_int + 1; // wraps around to 0

cout << "The value of my\_unsigned\_int plus 1 is " << my\_unsigned\_int << endl; // outputs 0

Also note that unsigned integers or just unsigned are actually the preferred type for counting the elements in a list, and other situations when you can't have negative values. We'll get into this later.

### Strings

C++ strings are the first examples we’ll look at which are a bit special and why C++ is so much more powerful than C. The string data type is actually what’s called a *class.* That doesn’t mean much now, but it will soon enough. What’s important is that you can treat strings like any other data type most of the time, but also do more with them.

A string in the most general sense refers simply to "a sequence of characters", and the string data type is designed so you can assign values to which are anything from single letters to entire words and sentences.

Our Hello World example could easily have been written like this:

#include <iostream>

#include <string> // need to include the string header file to use it

using namespace std;

int main()

{

string message = "Hello World!"; // declare a string variable and give it a value

cout << message << endl;

}

Just like with other data types, you can add strings together:

string part1 = "Hello ";

string part2 = "World";

string result = part1 + part2;

result += '!'; // Oops forgot the ! The += operator we’ll get to in a little while

string variables also have a bunch of addition functions you can use, to get information about the data inside, find specific character within the string, change the individual characters, etc. This tutorial just scratches the surface of strings - you’ll use strings a lot.

### Naming Variables

It's very important to give variables human readable and memorable names. The names in these examples are mostly terrible because they don't really do anything. When you are declaring a variable to really use, choose a good name that describes exactly what the variable is and what it's for. Using "my\_" in the name is just stupid – although the examples do exactly that!

Some examples from real student projects:

|  |  |  |  |
| --- | --- | --- | --- |
| **Purpose** | **Great** | **Ok** | **Bad** |
| Player's health points | player\_health\_points | health | h |
| Has player moved yet? | has\_player\_moved | player\_moved | move |
| Games won by player | games\_won\_by\_player | num\_player\_wins | pw |
| Damage done to enemy | damage\_to\_enemy | damage | dmg |

You'll notice two things:

* Good names describe the data they contain
* Good names are longer, bad names are often very short

Marking criteria for your projects includes readability and code quality, which considers how you name your variables and functions!

## Operators

You have already used the most basic arithmetic operator, addition (+) and the multiplication operator (\*). There are three other arithmetic operators, **subtraction** (-), **division** (/) and **modulus** (%). Here is some code that uses all of these operators on integers, which is probably what you would expect:

int my\_int\_1 = 13;

int my\_int\_2 = 4;

cout << "The value of my\_int\_1 + my\_int\_2 is " << my\_int\_1 + my\_int\_2 << endl;

cout << "The value of my\_int\_1 - my\_int\_2 is " << my\_int\_1 - my\_int\_2 << endl;

cout << "The value of my\_int\_1 \* my\_int\_2 is " << my\_int\_1 \* my\_int\_2 << endl;

cout << "The value of my\_int\_1 / my\_int\_2 is " << my\_int\_1 / my\_int\_2 << endl;

cout << "The value of my\_int\_1 % my\_int\_2 is " << my\_int\_1 % my\_int\_2 << endl;

which would output:

The value of my\_int\_1 + my\_int\_2 is 17

The value of my\_int\_1 - my\_int\_2 is 9

The value of my\_int\_1 \* my\_int\_2 is 52

The value of my\_int\_1 / my\_int\_2 is 3

The value of my\_int\_1 % my\_int\_2 is 1

The first 3 are pretty obvious, but the last two may not be so much. Remember from the earlier section on Integers - when using division on integers there can be no fractional part, so the remainder is discarded and just the whole number portion remains, so though 13 divided by 4 equals 3.25, only the 3 portion is kept in the operation.

The **Modulus** operation may be new to you, but all it means is the **remainder of integer division**. You've seen that for integers 13 divided by 4 equals 3, so if you calculate 4 times 3 (12) you can find the remainder of the division (13 minus 12) which equals 1.

If you were to use floating point numbers the only differences would be that the division would return 3.25, and that modulus isn't a valid operation on floating point values:

float my\_float\_1 = 13.0f;

float my\_float\_2 = 4.0f;

cout << "The value of my\_float\_1 + my\_float\_2 is " << my\_float\_1 + my\_float\_2 << endl; // 17

cout << "The value of my\_float\_1 - my\_float\_2 is " << my\_float\_1 - my\_float\_2 << endl; // 9

cout << "The value of my\_float\_1 \* my\_float\_2 is " << my\_float\_1 \* my\_float\_2 << endl; // 52

cout << "The value of my\_float\_1 / my\_float\_2 is " << my\_float\_1 / my\_float\_2 << endl; // 3.25

Note that in the above examples the arithmetic operations aren't assigned to a variable, but are **evaluated in place** in with the cout operations. What this means is the result is calculated and displayed on the screen, but it isn't stored anywhere. This can be used if you don't need the result later on, otherwise you would need store the result in another variable:

float result = my\_float\_1 + my\_float\_2

cout << "The value of my\_float\_1 + my\_float\_2 is " << result;

### Compound Assignment

All of the operators have a shortcut notation called the **compound assignment**. This means that instead of using:

my\_int = my\_int + 25;

my\_float\_1 = my\_float\_1 / my\_float\_2;

you can use the shortcuts

my\_int += 25; // same as my\_int = my\_int + 25;

my\_float\_1 /= my\_float\_2; // same as my\_float\_1 = my\_float\_1 / my\_float\_2;

Some of these operators work with string as well as the fundamental data types, but not all of them. We used the addition operator (+) for strings earlier to add characters in multiple strings together. That’s pretty straightforward and makes sense, but what would the division operator (/) or modulus operator (%) do if used with a string? It would not be clear and obvious at all, so those operators are not usable on string variables. Even subtraction would behave unpredictably in some cases, so it’s not used either.

As used above, the compound addition operator (+=) is very useful for strings if you want to append characters or strings to the end of string variable:

result += '!'; // forgot the !

### Increment and Decrement Operators

Another shortcut that is often used are the **increment** (++) and **decrement** (--) operators. The increment operator adds one to a variable, and the decrement operator subtracts 1. A feature of these operators is they can be both **prefix** and **postfix** operators, that is they can go before or after the variable they are changing. Here are examples to show how they work:

my\_int = my\_int + 1; // normal addition

my\_int += 1; // compound addition

++my\_int; // increment with prefix operator

my\_int++; // increment with postfix operator

All of these statements do exactly the same thing, which is add 1 to a variable. The difference between the prefix and postfix versions only becomes apparent in more complex arithmetic expressions, in that the prefix version updates the variable **before** the rest of the expression, and the postfix **after**. The following code uses both types to illustrate this:

int my\_int1=5;

int my\_int2=5;

int my\_int\_result;

my\_int\_result = ++my\_int1;

cout << "The value of my\_int\_result = ++my\_int1 is " << my\_int\_result << endl; // output 6

my\_int\_result = my\_int2++;

cout << "The value of my\_int\_result = my\_int2++ is " << my\_int\_result << endl; // output 5

As you can see, using ++my\_int1 causes my\_int1 to be incremented from 5 to 6 **before** the assignment to my\_int\_result so it is assigned the value 6, whereas using my\_int2 ++ causes my\_int2 to be incremented **after** the assignment, so my\_int\_result is assigned the value 5. Be careful when you use them, make sure you understand when these operators change the variable if used in a complex arithmetic operation.

### Comma operator ( , )

So the comma operator is one of the legal C/C++ operators, and is used to separate legal expressions. It is useful when declaring multiple variables of the same type at the same time:

int a = 3, b = 10;

Using the comma in this way allows you to keep code more compact. We'll get into other uses as well, primarily when writing and calling functions.

Note that the comma operator is legal to use in a lot of situations, but **it won't actually do anything!** Or worse yet, **it won't do what you expect!** Pay attention to how you see it used in future tutorials and for now just use it between variable declarations.

### More operators

There are quite a few more operators, some of which we'll get to next week. Another group of operators work on binary values and specific bits, but are not used as commonly so we'll get to them later on.

For a complete reference, check here:

<https://www.tutorialspoint.com/cplusplus/cpp_operators.htm>

### Precedence and Brackets

**Precedence** is the order of operations that happen in an arithmetic expression. The most basic rule is that operators are evaluated from **left to right**, but certain operators have a **higher precedence** than others, **so get evaluated first**. Any C/C++ text book will list the precedence for all operators, but for now all you need to know is that multiplication, division and modulus have the same precedence, and they have a higher precedence than addition or subtraction. The upshot of this is that multiplication, division and modulus operations will be evaluated before addition and subtraction. So for instance

int my\_int = 3 + 4 \* 5;

will evaluate to 23, not 35 as you might expect by evaluating from left to right, as precedence means that 4 \* 5 is done first, and not 3 + 4. In order to get round this you need to use **parentheses** (or **brackets**). Brackets allow you to determine exactly how operations should be ordered as operations inside brackets are done first. So if you changed the above line to

int my\_int = (3 + 4) \* 5;

then 3 + 4 would be done first. You are probably best off **always** using brackets to show the order you want, even if it would work correctly without. It just means you can see what is intended without having to look up the precedence rules.

## Input and Output (I/O)

We've used cin and cout for getting input and display text to the screen without looking at them closely, so let's do that now. The simplest way to handle I/O in C++ is to use cin and **cout**, but you have some other options as well depending on what you need.

### Text Output with **cout**

**Hello World** is *the* classic first program because it's so simple.

#include <iostream>

using namespace std;

int main()

{

cout << "Hello World!" << endl;

getchar(); // wait for enter to be pressed

}

By default the place where program output goes is the screen, and cout is the mechanism used in a program for output to the **default output**, so the above code displays "Hello World!" on screen.

Looking at the statements in the main() function:

* cout directs data to the **c**onsole **out**put, using the "<<" operator to identify the data to output.
* endl at the end of the text tells cout to print a new line – it's the **end** of **l**ine command. This makes any subsequent output go to the next line on screen. Just like pressing the enter key would.
* The last line calls the function getchar()which is used here to pause program execution. getchar() waits for the Enter key to read reads single characters from the keyboard. We'll look at it further in a bit.

The above example uses one block of text and one endl, but you can chain together expressions as many as you want, separated by <<. So:

cout << "Hello World" << endl << "Dave is here" << endl;

outputs the same as:

cout << "Hello World" << endl;

cout << "Dave is here" << endl;

and:

cout << "Hello Wolrd");

cout << endl;

cout << "Dave is here";

cout << endl;

Also note that sometimes you will see the old C-language approach to ending a line, using what is called a newline character: "\n". This is also perfectly legitimate, but it’s preferred in C++ that you use endl;

The text that is sent to the output in the above examples, such as "Hello World!" is called a **string literal**, which is a fixed string value embedded in your program. However, you are not restricted to outputting plain text to the screen, using cout - you can output numbers and variables, e.g.:

int my\_age;

my\_age = 25;

cout << "I am " << **my\_age** << " years old"**;**

cout works with any Fundamental Data Type (char, bool, int, float, etc) as well as string. cout won't understand how to deal with new data types, which we'll get to later on.

In order to use cout you need to inform the compiler that you want to use it in your source file. This is done using the following:

#include <iostream>

#include <iostream> tells the compiler to include the **Input/Output Stream** library where cout is defined, along with many other I/O operations. Notice that you **don't** need a semi-colon after the #include directive.

### Input using cin

You are now able to display the output of the program on the screen, but it's often useful to get **input** from whoever is using the program. This is where cin comes in. cin receives input from the **default input,** which is usually the keyboard. It’s the counterpart to displaying output using cout and also part of **iostream**.

int my\_age;

my\_age = 25;

cout << "How old are you?" << endl;

cin >> my\_age;

cout << "You are " << my\_age << " years old";

When the program gets to the cin statement it waits for input from the keyboard. To input a value for this, type in an integer number and press **Enter**. The value that was input is then assigned to the variable, in this case my\_age. Note that any value in the variable being used in the cin statement is overwritten in the same way as a normal assignment statement, so my\_age will no longer be 25 (unless 25 is entered of course).

Just like cout, multiple variables can be read in a single line from the input stream by chaining them together. Let’s say you declare three variables to store information from the user:

string my\_name;

int my\_age;

int my\_height;

These three lines

cin >> my\_name;

cin >> my\_age;

cin >> my\_height;

could be replaced with this line of code:

cin >> my\_name >> my\_age >> my\_height;

assuming the user enters the same values when the program is running.

One point about using cin are important because it can cause confusion. cin expects to read data based on the type of variable the data will go into. If you try to read data into the wrong kind into a variable, it will trigger an internal error in cin, and cin will fail: it will leave the input unread and try to put it into the next variable.

So for example, if you try to read a string into an integer variable, it won't put anything in the integer and the string will remain waiting. For now, make sure that your data matches your variable. We will go over the solution in a few weeks. It is just to use a small piece of code that checks for the failure, and clears it all up - including removing any potentially bad input that you haven't tried reading yet.

### Using other Input functions

cout and cin are the preferred approaches to I/O in C++, but there are alternatives that are just easier to use sometimes, especially instead of cin. You may notice that all of these functions return **integers** and not **chars** – that's one of the reasons why they are useful for some things, but not for everything because extra work can be required.

A few of these older C functions are listed below:

### **getchar()**

The final I/O operation in the Hello World program is getchar(). This function can be used to get the **value of a single character** from the keyboard, which is read after the Enter key is pressed. However, all it is being used for in this case is to **pause** the program so that you can see what is being displayed before the program exits. The value that’s read in is ignored in the example code. If you wanted to store that character, it's easy to do:

#include <iostream>

using namespace std;

int main()

{

cout << "Hello World!" << endl;

char input\_char = getchar(); // wait for enter to be pressed

}

There are alternatives to getchar() that would also work in this case, but this is an easy first example to use. getchar**()** is also defined in **iostream** so you don’t need to include another file..

### \_**getch()**

From here on, we’ll also use \_getch(). It’s very similar to gethchar(), but waits only for the very next key pressed, not just for Enter. To use \_getch() you will also need to include the header file conio.h. \_getch() does not display the character/key that was pressed – it just reads it in and lets you decide what to do.

#include <iostream>

#include <conio.h>

using namespace std;

int main()

{

cout << "Hello World!" << endl;

char input\_char = \_getch(); // wait for any key press

}

Note that there is an old version of this function called getch()which you may come across. It works the same way, but was written for old PC and might not work reliably so the compiler will warn you about using it.

### \_**getche()**

\_getche() works exactly the same as \_getch(), except that it "echoes" the key that was pressed, meaning that you will see something on screen that corresponds to the key press.

## Constants

You will frequently find that you need to use a **literal** value when programming, such as “Hello World”, or 3.14159 for PI or 8 as the width of a chessboard. In these and indeed most cases, you want to use what is called a **constant** instead of just placing the literal directly in the code.

Constant values must be assigned when they are declared and can never be changed, but otherwise act a lot like variables.

* They have a specific data type
* Can be used in operations that don’t attempt to assign them a new value
* Should be named clearly. Often they are in ALL\_CAPITALS to differentiate from variables.

Constants are declared much like variables, but the data type is preceded with the keyword const.

const float PI = 3.1416f;

const int MAX\_NUMBER\_OF\_PLAYERS = 16;

const string WELCOME\_MESSAGE = "Hello World!";

### Why use constants?

1. The name of the constant clearly describes what its purpose is. You don’t have to guess why the number 8 is being used, for example.
2. The value is set in one place and referred to everywhere else, so if you need to change that value, it’s really easy.
3. Code is much clearer and easier to read and understand by others. “Magic numbers” are a very bad practice so don’t fall into the habit of using them.
4. Code is smaller and more efficient, because the compiler can optimize more efficiently.

You will need to use constants in your assessments to get the highest marks. Constants are expanded upon and used even more in advanced C++.

## Variable Scope

Scope simply means where a variable can be seen or accessed by the program. We will go into this in depth later, but it's useful to know a bit now.

There are two types of scope:

* **global** scope means the variable is usable anywhere in the source file, and can made usable to other files in the project as well. Global variables will be explained below, but as a general rule *global variables should not be used*.
* **local** scope means the variable is limited to the specific statement block where it was declared. Remember that a statement block is defined by a pair of braces.

Here is a program that will show how scope works:

#include <iostream>

#include <conio.h>

using namespace std;

**int global\_integer = 10; // Global: usable throughout main.cpp**

int main()

{

**int local\_function\_integer1 = 3; // Local: usable inside main() function statement block**

cout << global\_integer << endl;

cout << local\_function\_integer1 << endl;

**int local\_function\_integer2 = 3; // Local: usable inside main() function statement block**

cout << global\_integer << endl;

cout << local\_function\_integer2 << endl;

\_getch();

}

**// any other functions to follow - global\_integer still in scope**

Here you can see three different variables defined and used in different parts of the program, with each different scope. First is the global integer variable global\_integer, which is accessible inside the main() function and if there were other functions it would also be visible there.

Next is local\_function\_integer1 which is visible throughout the main() function, but not elsewhere in the source file.

Finally there is local\_function\_integer2 which is also visible in all the lines of main() after it is declared. It is declared later because it's not used until later in the function.

### Best Practices – Summary

- Use the data type best suited for the data it will hold

- Give your variables descriptive and consistent names

- Initialize variables when declared or ASAP

- Declare new variables close to where they will be used

- Avoid global variables!

- Use constants instead of literals

- Don’t be afraid to use more variables! You can reduce the number later when everything works.