Getting Started

To write a C++ program, one only need a plain text editor and a compiler. One can use the web compiler such as repl.it, or the web linux subsystem such as console.cloud.google.com (it provides a text editor and the g++ compiler), or download visual studios. There a numerous free C++ compiler available for both Windows and Mac; and each operating system provides a built-in plain text editor; one just have to look.

Minimum Coding

Now, let us get started by writing the simplest C++ program one can write which is:

```
int main()
{
 return 0;
}
```

This program does nothing; however, it is the minimum code necessary to compile a C++ program. To compile the code, one needs to save the file with the extension `.cpp`. The above code consists only of a *function definition*. A function defintion starts with a *function header*, which is the first line and the remainder of the definition is the *body*. This function is called the *main function*, and it must be present in every compilable C++ program. Furthermore, within the body of the main function, you must have the return statement (`return 0;`). This line tells the compiler that the program executed successfully and it should always be the last line in the body of the main function. Anyway for now, remember to have this code in all programs. Later on, we will discuss how to define functions and use them in more details.

Scopes

In the above code, the area before and after the main function is called *global scope*. It is any area outside of any curly braces. Items defined in global scope are accessible everywhere. The area enclosed between the curly braces of the body of the main function is called a *local scope*. Local scopes are area enclosed in curly braces. They are usually preceded by a header line. Items defined in local scopes are accessible in the scope and local scopes within it (known as nested scopes).

No two items within the same scope can have the same *identifier* that is a name that must starts with either a letter or underscore that is followed by any number of letters, underscores or digits; whereas, items within different scopes can have the same identifer. Whenever there are multiple items with the same identifier, the compiler determines which item you are referring to by starting with the scope that is calling (or accessing) the identifer. If an item with the identifier is not defined in that scope, the compiler checks the immediate parent scope of the current scope. It continues this process until it finds an item with the identifer in a parent scope with the global scope being the final scope it checks. If it cannot find the identifier, it produces a syntax error. Furthermore, it is important to note that the compiler will never check any of the nested scopes within the scope where the call is made; and neither will it check any code within the scope that follows the call. The latter condition is due that the fact that programs are sequential.

Streaming: Outputs

The original program works, but it is not interesting. By adding a couple of lines to the code as below, the program can do significantly more.

```
#include <iostream>


int main()
{
 std::cout << "Hello World";
 return 0;
}
```

This program will print the message `"Hello World"` (without the quotations) on the terminal. This program is utilizing the *cout* function (actually an object) from the *iostream library* which is a collection of objects and functions. Libraries are added to programs by using the *preprocessor directive* `#include`. To include the system libraries, one type `#include` followed by the library name in angular braces (`<>`). One can get a list of libraries available in C++ by accessing a C++ API online.

Meanwhile, the cout object is used to display content on the terminal. Its syntax is

$$\texttt{cout} \texttt{ << } \langle \textit{argument} \rangle \texttt{;}$$

where an $\langle \textbf{\textit{argument}} \rangle$ is either a literal, a variable, an expression, or a return-value function call. Each form of an argument will be seen later in the lecture or in future lectures. In the above code, the argument of the cout statement is a *string literal* where a *literal* is an immutable form of a data type. And the symbol `<<` is called the *ostream operator* (or insertion operator).

Additionally, a single cout statement can be used to display multiple content with the following syntax:

$$\texttt{cout } \left[\texttt{<< } \langle \textit{argument} \rangle\right]^{+} \texttt{;}$$

where $\left[\langle \textbf{\textit{content}} \rangle\right]^{+}$ means that $\langle \textbf{\textit{content}} \rangle$ can be repeated 1 or more times. There is also the regular expressions $\left[\langle \textbf{\textit{content}} \rangle\right]^{*}$ and $\left[\langle \textbf{\textit{content}} \rangle\right]^{?}$ which means $\langle \textbf{\textit{content}} \rangle$ can be repeated 0 or more times, and $\langle \textbf{\textit{content}} \rangle$ is repeated 0 or 1 times respectively. An example of this is in the following code

```
#include <iostream>


int main()
{
 std::cout << "Hello"<< ""<< "World";
 return 0;
}
```

This code and the one before it will produce the same output; however, the new code does so by printing three string literal arguments. It is important to note that cout statements work like text editors such as word. That is, whenever one takes a break from typing, and then, continue typing, it starts from the last location of the cursor. For instance, the code below will produce the same output as the previous two codes but in a slightly difference way

```
#include <iostream>

int main()
{
 std::cout << "World";
 std::cout << "";
 std::cout << "Hello";
 return 0;
}
```

Comments

When code gets more complex, it becomes harder to read and understand. One way to alleviate the issue is by adding notes and documentation to the code. This is achieved by adding *comments* which is code that does not compile. There are two forms of comments: single line comments and multi-line comments. The syntax for single line comments is

$$//\langle string \rangle$$

Any code that follows the double forward slashes (`//`) on the same line will become a comment. The syntax for multi-line comments is

$$/*\langle string \rangle */$$

Any code enclosed in the forward slash asterisk (`/*`) and asterisk forward slash (`*/`) will become a comment. It can span to multiple lines. It is important to remember to close the comment block; otherwise, one will receive unexpected errors.

Data Types, Literals & Operations

A computer can process data in different formats. The primary formats of data are numerical, character, string and boolean. Numerical data can be either an integer or a decimal also referred to as a *floating point*. Unlike in mathematics where there is such thing as infinity, there is no such thing with computers since computers have finite memory space. Therefore, there are multiple forms of integer and floating point data types to accommodate various interval ranges. To elaborate, a computer consists of transitors which contain what you can say are switches called *bits* (binary digits) that can either have a charge or no charge (be on or off). If one have a sequence of 8 bits, one has a *byte*. With a single byte, one can represent 256 different outcomes. To represent numbers, computers use the binary number system and 2s-complement for negative numbers. In C++, you have the following integer and floating point data types

Name	Size
Integer Types	

short	2 bytes
unsigned short	2 bytes
int	4 bytes
unsigned int	4 bytes
long	8 bytes
unsigned long	8 bytes
Floating Point Types	
float	4 bytes
double	8 bytes
long double	12 bytes

For the integer types, the unsigned versions only deal with nonnegative numbers; whereas, the standard integer types (or signed types) deal with negative through positive numbers. However, the signed integers represents a smaller range of positive numbers than their unsigned versions. For the floating point types, the larger the size of the type, the more decimal places it can hold. The literal for numerical data types match the way they are written in mathematics. For instances, examples of integer literals are 2, -3, 289, -1293 and so on. And examples of floating point literals are 8.022, -3.1415, 0.75, 129.2 and so on. Like with mathematics, one can perform arithmetic operations of the numerical types. The list of operations are as follows:

Operation	**Symbol**	**Example**	**Result**
Addition	`+`	`6 + 4`	10
		`8.0 + 4`	12.0
		`1.0 + 8.0`	9.0
Subtraction	`-`	`6- 4`	2
		`8.0 - 4.0`	4.0
		`1.0 - 8.0`	-7.0
Multiplication	`*`	`6 * 4`	24
		`8.0 * 4`	32.0
		`1.0 * 8.0`	8.0
Division	`/`	`6 / 4`	1
		`8.0 / 4`	2.0
		`1.0 / 8.0`	0.125
Modulus	`%`	`6 % 4`	2
		`8.0 % 4`	ERR
		`1.0 % 8.0`	ERR

The last operation modulus only work with integer operands. It returns the integer remainder of the numerator divided by the denominator. It is important to note that when any operand of the modulus operation is negative, the result may not be what one expect. Anyway, like in mathematics, the operations have an order of precedence which is

1. Parentheses - `()`
2. Multiplication, Division, Modulus - `*, /, %`
3. Addition, Subtraction - `+ , -`

For operations that have the same precedence, they are evaluated from left to right in an *expression*. For instances, the expression

```
3 + 4 * (8 - 2) / 3 - 89 % 5
```

will evaluate to `7`.

Now, the character data type is called *char*. It is 1 byte in size. Each character is associated with a decimal value from 0 to 255 based on the ASCII table which stands for American Standard Code for Information Interchange. The first 128 characters are

Dec	Chr	Dec	Chr	Dec	Chr	Dec	Chr
0	NUL	32	SPC	64	@	96	\
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	TAB	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	′	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x

25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	}
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	{
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL