**Chapter Two**

1. **Basic Concepts of C++ Programming**

***2.1 Structure of C++ program***

Structure of a C++ program includes:

* ***Documentation section***
* ***Linking section***
* ***Definition section***
* ***Global declaration section***
* ***Class declaration***
* ***Function***
* ***main() functions***

***{***

 ***Initializations;***

 ***Executable parts;***

***}***

* ***Function1()***

***{***

 ***Initializations;***

 ***Executable parts;***

***}…n***

* ***Documentation Section:*** include comment parts, which are ignored by compiler. The comments are used to describe the functionality of each statement in the program, its copyright, author, and date of compilation and purpose of the program to the user. C++ supports single line (// comment) and multiline (/\* comments\*/).

***Syntax:*** //This statement is a single line comment

 /\*This is multiline comment\*/

* ***Link section:*** here we can link the necessary files and libraries to current files. Using #include directive we can link the necessary libraries to current files.

***Syntax: #include<file or library name>***

***Example: #include<iostream.h>***

* ***Definition Section:*** It is possible to define symbolic constants. Symbolic constants are normal identifiers which value cannot be altered.

***Syntax: #define identifier constant***

* ***Global declaration Section:*** variables declared under this section are available throughout the program.
* ***Class declaration Section:*** defines the class declaration.
* ***Functions:*** The function *main*, which has been written in the program, is an example of function. A function is defined as group of instructions, which are assigned a name and accessed by the name.

***Syntax:***

***return type function name ()***

 ***{***

***Statement of function;***

 ***}***

A function is identified by the compiler with the help of parenthesis ( ) after the function name. Without the parenthesis the compiler thinks it as a variable or complaints that it is undefined symbol. So to define a function, parenthesis is must. The *statement inside the brace* {} forms the body of the function and specifies the task of the function. *Return type specifies* the type of data the function has to send to the calling function after the execution. It may be the result of the task performed in the function.

* ***main ( ) function:*** the execution of the program starts from main function. Every program must have only one main function. All other functions are called either directly or indirectly from main. The initialization or other executable statements are included with in the main function.
* ***Initialization part:*** the variables that are used in the program should be initialized here. These variables are not available to all the functions. In C++ we can initialize the variables at any part in the function.
* ***Executable part:*** A statement declared under executable part performs some tasks. Every statement under global declaration, initialization and executable parts should be terminated with the *semi colon ‘;’. A semi colon acts as a statement terminator like full stop in English.*

#include<iostream.h>

int main()

{

Statements;

Other functions;

}

***Syntax:***

***2.2 The C++ Compilation Process***

C++ is a 3rd-generation language. These types of languages must be translated into a machine language in order to be executed by a CPU. The process of translating high-level language into machine language is called the compilation process.

The compilation process consists of the following steps.

 edit source code -> compile -> link -> execute

 (editor) (compiler) (linker) (loader)

***Program source code*** is entered into a file using a text editor. After the code has been entered, a compiler program is started that translates the source into an object code file. The object code file is linked with other object code files that come with the compiler and an executable file (or program) is created. In order to execute the program, a program called the loader copies the executable file into the memory of the computer and sends an execute command to the CPU. It should be noted that errors can occur during each step.

 A ***source file*** ending with ".c" contains C source code; whereas, a file ending with ".cpp" is a C++ file. A file ending with ".h" can be both a C and C++ header file. Sometimes the suffix ".hpp" (or ".H") is used to indicate a C++ only header file.

The ***compiler*** is a program that usually consists of many phases. The first phase of compilation is called ***preprocessing***. The [***preprocessor***](http://deru.com/~gdt/c/handouts/preprocessor0.shtml) does many things, but two features that must be learned immediately are file inclusion and macro (manifest constant) definitions. After preprocessing, the compiler executes two primary steps: lexical analysis and parsing. During lexical analysis, the source code is broken up into tokens and the tokens are passed to the parser. The parser does syntax and semantic analysis, which includes the generation of object code (i.e. machine language).

The ***linker*** "combines" all object code files into an executable file. Typically, the object files created by your source files are linked with object files that are packaged into libraries.

Most implementations allow each step of the compilation process to be executed as a stand-alone procedure. For example, compile a source file but do not invoke the linker; execute the preprocessor only; or, invoke the linker only.

Some older compilers translate C source code into assembly language, and then execute an assembler program to translate the assembly language into machine language.

Early C++ compilers (those prior to 1992) translated C++ code into C code and then executed the C compiler.

The ***loader*** reads a program (i.e. executable file) into memory. Once this is completed, it becomes a process and the CPU executes it.

#  *2.3 Syntax and Semantics*

The **Syntax** of a programming language consists of the rules for the correct use of the language. This involves the correct grammatical construction and arrangement of the language, correct spelling, hyphenation, inflection and so on.

 The **semantics** of a programming language deal with the meanings given to syntactically correct constructs of the language. Usually, semantics is defined in terms of the program’s run-time behavior: What happens when the program is executed with a certain set of inputs, what statements are executed, what values are assigned to the variables, and what output is produced.

 Thus syntax has nothing to do with “meaning” or run-time behavior of a program. A program could be syntactically correct yet meaningless. The program below (code fragment) is syntactically correct but does not have any meaning at runtime (never terminates).

  sum=0;

while (sum!=-1)

sum=sum+10;

 Yet syntax is a prerequisite to meaningful expression. Thus, a programming language must have a good syntactic definition before it can properly support the development of meaningful programs.

# *2.4 The parts of a simple C++ Program*

* + - To understand the basic parts of a simple program in C++, let’s have a look at the following code:

#include<iostream>

using namespace std;

void main()

 {

 cout<<”\n Hello World!”;

 }

* + - Any C++ program file should be saved with file name extension “ ***.CPP*** ”
		- Type the program directly into the editor, and save the file as hello.cpp, compile it and then run it. It will print the words Hello World! On the computer screen.
		- The first character is the #. This character is a signal to the preprocessor. Each time you start your compiler, the preprocessor runs through the program and looks for the hush or sharp (#) symbols and act on those lines before the compiler runs.
		- The ***include*** instruction is a preprocessor instruction that directs the compiler to include a copy of the file specified in the angle brackets in the source code.
		- If the path of the file is not specified, the preprocessor looks for the file under ***c:\tc\include\*** folder or in ***include*** folder of the location where the editor is stored.
		- The effects of line 1, i.e. include<iostream> is to include the file iostream into the program as if the programmer had actually typed it.
		- When the program starts, main() is called automatically.
		- Every C++ program has a main() function.
		- The return value type for main() here is void, which means main function will not return a value to the caller (which is the operating system).
		- The main function can be made to return a value to the operating system.
		- The Left French brace “{“signals the beginning of the main function body and the corresponding Right French Brace “}” signals the end of the main function body. Every Left French Brace needs to have a corresponding Right French Brace.
		- The lines we find between the braces are statements or said to be the body of the function.
		- A statement is a computation step which may produce a value or interact with input and output streams.
		- ***The end of a single statement ends with semicolon (;).***
		- The statement in the above example causes the string “Hello World!” to be sent to the “cout” (VDU) stream which will display it on the computer screen.

**A brief look at cout and cin**

* + - ***cout*** is an object used for printing data to the screen.
		- To print a value to the screen, write the word ***cout***, followed by the insertion operator also called output redirection operator (<<) and the object to be printed on the screen.
		- Syntax: ***cout***<<Object;
		- The object at the right hand side of **<<** operator can be variables, strings or expressions.

E.g.

cout<<”Hello There “; //prints Hello There on the screen

cout<<x; // prints the content of variable x on the screen

cout<<”x”; // prints the character x on the screen

cout<<100; // prints number 100 on the screen

cout<< x + y; // prints the result of the sum of the content of variables x and y

* + - **cin** is an object used for taking input from the keyboard.
		- To take input from the keyboard, write the word ***cin***, followed by the input redirection operator (>>) also called extraction operator and the object name to hold the input value.
		- Syntax: cin>>Object
		- **cin** will take value from the keyboard and store it in the memory. Thus the cin statement needs a variable which is a reserved memory place holder.
		- The user of the program should supply correct form of data when it is taken as input from the keyboard. If the input value is different from the variable’s data type or capacity, then there is a possibility of erroneous result and erratic behavior of the program.
		- Both << and >> return their right operand as their result, enabling multiple input or multiple output operations to be combined into one statement. The following example will illustrate how multiple input and output can be performed:

E.g.:

* ***cin>>var1>>var2>>var3;***

Here three different values will be entered for the three variables. The input should be separated by a space, tab or newline for each variable.

It is equivalent to

**cin>>var1;**

**cin>>var2;**

**cin>>var3;**

* ***cout<<var1<<”, “<<var2<<” and “<<var3;***

Here the values of the three variables will be printed where there is a “,” (comma) between the first and the second variables and the “and” word between the second and the third.

**Putting Comments on C++ programs**

* + - A comment is a piece of descriptive text which explains some aspect of a program.
		- Program comments are text totally ignored by the compiler and are only intended to inform the reader how the source code is working at any particular point in the program.
		- C++ provides two types of comment delimiters:
1. ***Single Line Comment***: Anything after // {double forward slash} (until the end of the line on which it appears) is considered a comment. ***Eg***:

cout<<var1; //this line prints the value of var1

1. ***Multiple Line Comment***: Anything enclosed by the pair /\* and \*/ is considered a comment. ***Eg:***

***/\*this is a kind of comment where***

***Multiple lines can be enclosed in***

***one C++ program \*/***

* + - Comments should be used to enhance (not to hinder) the readability of a program. The following two points, in particular, should be noted:
1. A comment should be easier to read and understand than the code which it tries to explain. A confusing or unnecessarily-complex comment is worse than no comment at all.
2. Over-use of comments can lead to even less readability. A program which contains so much comment that you can hardly see the code can by no means be considered readable.
3. Use of descriptive names for variables and other entities in a program, and proper indentation of the code can reduce the need for using comments.

#  *Variables and Constants*

**2.5.1 Variables**

* + - A variable can simply assumed as a name given to a certain portion of the computer’s memory space by the programmer. Or we can say a variable is a name that can be given a variety of values. This memory space will be used to store values or the information which is stored at that location is known as value of the variable. And these values can change during the execution of a program. We can define variables (we name variables) anywhere in the program, but it is a must to do so before using them. A variable can hold only one value at a time.
		- We could have called the variables any names we wanted to invent, as long as they were valid identifiers. A valid identifier is a sequence of one or more letters, digits or underline symbols (\_). The length of an identifier is not limited, although for some compilers only the 32 first characters of an identifier are significant (the rest are not limited).
		- Variables in C++ are given names by the programmer, in the way he/she thinks it is meaningful and that it reflects clearly what it represents in the program. Variable names may consist of alphabets (both upper and lower cases), the digits from 0 to 9, and the underscore ( \_ ).
		- All variables have three important properties:
			* ***Data Type***: a type which is established when the variable is defined. (e.g. integer, real, character etc). Data type describes the property of the data and the size of the reserved memory. A variable can hold data in either of these types: a number or a character. The number may be an integer or a number with a decimal point (floating point), and the data size may be small or large. Data type is all about this.
			* ***Name***: a name which will be used to refer to the value in the variable. A unique identifier for the reserved memory location
			* ***Value***: a value which can be changed by assigning a new value to the variable.

**Fundamental Variable types**

* + - Several other variable types are built into C++. They can be conveniently classified as ***integer***, ***floating-point*** or ***character*** variables.
		- Floating-point variable types can be expressed as fraction i.e. they are “real numbers”.
		- Character variables hold a single byte. They are used to hold 256 different characters and symbols of the ASCII and extended ASCII character sets.
		- The types of variables used in C++ program are described in the next table, which lists the variable type, how much room.

 ***Data Types***

|  |  |  |
| --- | --- | --- |
| ***Type*** | ***Length*** | ***Range*** |
| unsigned char | 8 bits9 | 0 to 255 |
| Char | 8 bits | -128 to 127 |
| Enum | 16 bits | -32,768 to 32,767 |
| unsigned int | 16 bits | 0 to 65,535 |
| short int | 16 bits | -32,768 to 32,767 |
| Int | 16 bits | -32,768 to 32,767 |
| unsigned long | 32 bits | 0 to 4,294,967,295 |
| Long | 32 bits | -2,147,483,648 to 2,147,483,647 |
| Float | 32 bits | -3.4x10-38 to 3.4x10+38 |
| Double | 64 bits | -1.7x10-308 to 1.7x10+308 |
| long double | 80 bits | -3.4x10-4932 to 1.1x10+4932 |
| Bool | 8 bits  | true or false (top 7 bits are ignored) |

**Signed and Unsigned**.

* + - Signed integers are either negative or positive. Unsigned integers are always positive.
		- Because both signed and unsigned integers require the same number of bytes, the largest number (the magnitude) that can be stored in an unsigned integer is twice as the largest positive number that can be stored in a signed integer.
		- E.g.: Lets us have only 4 bits to represent numbers

|  |  |  |
| --- | --- | --- |
| ***Unsigned*** |  | ***Signed*** |
| Binary | Decimal |  | Binary | Decimal |
| 0 | 0 | 0 | 0 | →0 |  | 0 | 0 | 0 | 0 | →0 |
| 0 | 0 | 0 | 1 | →1 |  | 0 | 0 | 0 | 1 | →1 |
| 0 | 0 | 1 | 0 | →2 |  | 0 | 0 | 1 | 0 | →2 |
| 0 | 0 | 1 | 1 | →3 |  | 0 | 0 | 1 | 1 | →3 |
| 0 | 1 | 0 | 0 | →4 |  | 0 | 1 | 0 | 0 | →4 |
| 0 | 1 | 0 | 1 | →5 |  | 0 | 1 | 0 | 1 | →5 |
| 0 | 1 | 1 | 0 | →6 |  | 0 | 1 | 1 | 0 | →6 |
| 0 | 1 | 1 | 1 | →7 |  | 0 | 1 | 1 | 1 | →7 |
| 1 | 0 | 0 | 0 | →8 |  | 1 | 0 | 0 | 0 | →0 |
| 1 | 0 | 0 | 1 | →9 |  | 1 | 0 | 0 | 1 | → -1 |
| 1 | 0 | 1 | 0 | →10 |  | 1 | 0 | 1 | 0 | → -2 |
| 1 | 0 | 1 | 1 | →11 |  | 1 | 0 | 1 | 1 | → -3 |
| 1 | 1 | 0 | 0 | →12 |  | 1 | 1 | 0 | 0 | → -4 |
| 1 | 1 | 0 | 1 | →13 |  | 1 | 1 | 0 | 1 | → -5 |
| 1 | 1 | 1 | 0 | →14 |  | 1 | 1 | 1 | 0 | → -6 |
| 1 | 1 | 1 | 1 | →15 |  | 1 | 1 | 1 | 1 | → -7 |

* + - In the above example, in case of unsigned, since all the 4 bits can be used to represent the magnitude of the number the maximum magnitude that can be represented will be 15 as shown in the example.
		- If we use signed, we can use the first bit to represent the sign where if the value of the first bit is 0 the number is positive if the value is 1 the number is negative. In this case we will be left with only three bits to represent the magnitude of the number. Where the maximum magnitude will be 7.

**Declaring Variables**

* + - Variables can be created in a process known as ***declaration***.
		- Syntax: ***Datatype Variable\_Name;***
		- The declaration will instruct the computer to reserve a memory location with the name and size specified during the declaration.
		- Good variable names indicate the purpose of the variable or they should be self descriptive.

E.g. int myAge; //variable used to store my age

* + - If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas. For example:

int a, b, c ;

This declares three variables (a, b, c), all of them of type int, and has exactly the same as:

int a;

int b;

int c;

**Keywords** are essential parts of a programming language, having a predefined meaning in the language and we cannot use them as variable names or any other purpose other than their predefined usage.

* + - **Reserved words** or **keywords** and are summarized in the following table.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Asm | continue | float | new | signed | try |
| Auto | Default | For | operator | sizeof | typedef |
| Break | Delete | friend | private | static | union |
| Case | Do | Goto | protected | struct | unsigned |
| Catch | Double | If | public | switch | virtual |
| Char | Else | inline | register | template | void |
| Class | Enum | Int | return | this | volatile |
| Const | Extern | Long | short | throw | while |

* **Identifiers**

**Identifiers: -** name given to any programming element (like variables, functions, arrays, structures, arguments, classes, pointers etc) is known as identifiers. Identifiers are the fundamental requirement of any languages. They are named by the programmer (but by keeping the rules for giving names). A name should consist of one or more characters, each of which may be a letter (i.e., 'A'-'Z' and 'a' - 'z'), a digit i.e., '0'-'9'), or an underscore character ('\_').

* + - A valid identifier is a sequence of one or more letters, digits or underscores symbols. The length of an identifier is not limited.
		- Neither space nor marked letters can be part of an identifier.
		- Only letters, digits and underscore characters are valid.
		- The first character must be a letter or underscore (They can never begin with a digit).
		- It should not be a **keyword**.
		- It should not have **special characters**  (except the underscore)

**Special characters:** include:

* Punctuation marks like **.,;,:,?,,** etc
* Arithmetic operators like **+,-,\*,/,%**
* Others like **=,(,{,[,$,’,”,# ,&** etc

**Note**: the C ++ language is “***case sensitive*** “, that means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. E.g. the variable ***Age*** is not identical with variable ***age.***

**Initializing Variables**

* When declaring a variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable.
	+ - When a variable is assigned a value at the time of declaration, it is called variable initialization.
		- This is identical with declaring a variable and then assigning a value to the variable immediately after declaration.
		- Initializing a variable is done by appending an equal sign followed by the value to which the variable will be initialized. The syntax is:

*DataType variable name = initial value;*

e.g. int a = 0;

or: int a;

 a=0;

 **Scope of Variables**

* + - Scope of a variable is the boundary or block in a program where a variable can be accessed. The boundary or block is identified by the left and right French brackets.
		- In C++, we can declare variables anywhere in the source code. But we should declare a variable before using it no matter where it is written.
		- There are 2 types of variable visibilities. These are:-
1. ***Global variables***: are variables that can be referred/ accessed anywhere in the code, within any function, as long as it is declared first. A variable declared before any function immediately after the include statements are global variables.
2. ***Local Variables***: the scope of the local variable is limited to the code level or block within which they are declared.
	* + In the following example, the integer data type num1 is accessible everywhere whereas z and is only accessible in the add function and num2 is accessible in main function. This means cout<<z; or any statement involving z is only valid in add function. E.g:

#include<iostream>

Using namespace std;

int num1;

int add( int x, int y)

{ int z;

 ….

}

int main()

{ unsigned short age;

 float num2;

 cout<<”\n Enter your age:”;

 …

}

* + - In C++ the scope of a local variable is given by the block in which it is declared.
		- If it is declared within a function, it will be a variable with a function scope. If it is declared in a loop, its scope will be only in the loop, etc.

**Characters**

* + - Characters variables (type char) are typically one byte in size, enough to hold 256 different values. A char can be represented as a small number (0 - 255).
		- Char in C++ are represented as any value inside a ***single quote***.

E.g.: ‘x’, ‘A’, ‘5’, ‘a’, etc.

* + - When the compiler finds such values (characters), it translates back the value to the ASCII values. E.g. ‘a’ has a value 97 in ASCII.

**Special Printing characters**

* + - In C++, there are some special characters used for formatting. These are:

\n new line

\t tab

\b backspace

\” double quote

\’ single quote

\? Question mark

\\ backslash

**2.5.2 Constants**

* + - A constant is any expression that has a ***fixed value***.
		- Like variables, constants are data storage locations in the computer memory. But, constants, unlike variables their content cannot be changed after the declaration.
		- Constants must be initialized when they are created by the program, and the programmer can’t assign a new value to a constant later.
		- C++ provides two types of constants: literal and symbolic constants.
1. ***Literal constant***: is a value typed directly into the program wherever it is needed.

E.g.: int num = 43;

1. s a literal constant in this statement:
2. ***Symbolic constant***: is a constant that is represented by a name, similar to that of a variable. But unlike a variable, its value can’t be changed after initialization.

E.g.:

Int studentPerClass =15;

students = classes \* studentPerClass;

studentPerClass is a symbolic constant having a value of 15.

And 15 is a literal constant directly typed in the program.

* + - In C++, we have two ways to declare a symbolic constant. These are using the ***#define*** and the ***const*** key word.

**Defining constants with #define**

* + - The ***#define*** directive makes a simple text substitution.
		- The define directive can define only integer constants

E.g.: #define studentPerClass 15

* + - In our example, each time the preprocessor sees the word studentPerClass, it inserts 15 into the text.

 **Defining constants with the const key word**

* + - Here, the constant has a type, and the compiler can ensure that the constant is used according to the rules for that type.

E.g.: const unsigned short int studentPerClass = 15;

 **Enumerated constants**

* + - Used to declare multiple integer constants using a single line with different features.
		- Enables programmers to define variables and restrict the value of that variable to a set of possible values which are integer.
		- The enum type cannot take any other data type than integer
		- enum types can be used to set up collections of named integer constants. (The keyword enum is short for “enumerated''.)
		- The traditional way of doing this was something like this:

#define SPRING 0

#define SUMMER 1

#define FALL 2

* + - An alternate approach using enum would be

enum SEASON{ SPRING, SUMMER, FALL, WINTER };

* + - You can declare COLOR to be an enumeration, and then you can define five possible values for COLOR: RED, BLUE, GREEN, WHITE and BLACK.

E.g.: enum COLOR {RED,BLUE,GREEN,WHITE,BLACK};

* + - Every enumerated constant has an integer value. If the programmer does not specify otherwise, the first constant will have the value 0, and the values for the remaining constants will count up from the initial value by 1. thus in our previous example RED=0, BLUE=1, GREEN=3, WHITE=4 and BLACK=5
		- But one can also assign different numbers for each.

E.g.: enum COLOR{RED=100,BLUE,GREEN=500,WHITE,BLACK};

Where RED will have 100 and Blue will have 101 while Green will have 500, White 501 and Black 502.

* 1. **Expressions and Statements**
		+ In C++, a statement controls the sequence of execution, evaluates an expression, or does nothing (the null statement).
		+ All C++ statements end with a semicolon.

E.g.: x = a + b;

The meaning is: assign the value of the sum of a and b to x.

* + - ***White******spaces***: white spaces characters (spaces, tabs, new lines) can’t be seen and generally ignored in statements. White spaces should be used to make programs more readable and easier to maintain.
		- ***Blocks***: a block begins with an opening French brace ({) and ends with a closing French brace (}).
		- ***Expressions***: an expression is a computation which yields a value. It can also be viewed as any statement that evaluates to a value (returns a value).

E.g.: the statement 3+2; returns the value 5 and thus is an expression.

* + - Some examples of an expression:

E.g.1:

3.2 returns the value 3.2

PI float constant that returns the value 3.14 if the constant is defined.

secondsPerMinute integer constant that returns 60 if the constant is declared

 E.g.2: complicated expressions:

 x = a + b;

 y = x = a + b;

The second line is evaluated in the following order:

1. add a to b.
2. assign the result of the expression a + b to x.
3. assign the result of the assignment expression x = a + b to y.
	1. **Operators**
		* An operator is a symbol that makes the machine to take an action.
		* Different Operators act on one or more operands and can also have different kinds of operators.
		* C++ provides several categories of operators, including the following:
* Assignment operator
* Arithmetic operator
* Relational operator
* Logical operator
* Increment/decrement operator
* Conditional operator
* Comma operator
* The size of operator
* Explicit type casting operators, etc

**Assignment operator (=)**

* + - The assignment operator causes the operand on the left side of the assignment statement to have its value changed to the value on the right side of the statement.
		- Syntax: Operand1=Operand2;
		- Operand1 is always a variable
		- Operand2 can be one or combination of:
			* A ***literal constan***t: Eg: x=12;
			* A ***variable***: Eg: x=y;
			* An ***expression***: Eg: x=y+2;

**Compound assignment operators (+=, -=, \*=, /=, %=, >>=, <<=, &=, ^=)**

* + - Compound assignment operator is the combination of the assignment operator with other operators like arithmetic and bit wise operators.
		- The assignment operator has a number of variants, obtained by combining it with other operators.

E.g.:

*value += increase*; is equivalent to *value = value + increase*;

*a -= 5*; is equivalent to *a = a – 5*;

*a /= b*; is equivalent to *a = a / b*;

*price \*= units + 1* is equivalent to *price = price \* (units + 1)*;

* + - And the same is true for the rest.

**Arithmetic operators (+, -, \*, /, %)**

* + - Except for remainder or modulo (%), all other arithmetic operators can accept a mix of integers and real operands. Generally, if both operands are integers then, the result will be an integer. However, if one or both operands are real then the result will be real.
		- When both operands of the division operator (/) are integers, then the division is performed as an integer division and not the normal division we are used to.
		- ***Integer division always results in an integer outcome.***
		- ***Division of integer by integer will not round off to the next integer***

E.g.:

9/2 gives 4 not 4.5

 -9/2 gives -4 not -4.5

* + - To obtain a real division when both operands are integers, you should cast one of the operands to be real.

E.g.:

 int cost = 100;

 Int volume = 80;

 Double unitPrice = cost/ (double) volume;

* + - The ***module(%)*** is an operator that gives the remainder of a division of two integer values. For instance, 13 % 3 is calculated by integer dividing 13 by 3 to give an outcome of 4 and a remainder of 1; the result is therefore 1.

E.g.:

a = 11 % 3

 a is 2

**Relational operator (==, !=, > , <, >=, <=)**

* + - In order to evaluate a comparison between two expressions, we can use the relational operator.
		- The result of a relational operator is a **bool** value that can only be true or false according to the result of the comparison.

E.g.:

 (7 = = 5) would return false or returns 0

(5 > 4) would return true or returns 1

* + - The operands of a relational operator must evaluate to a number. Characters are valid operands since they are represented by numeric values. For E.g.:

‘A’ < ‘F’ would return true or 1. it is like (65 < 70)

**Logical Operators (!, &&, ||)**

* + - **Logical negation (!)** is a unary operator, which negates the logical value of its operand. If its operand is non zero, it produces 0, and if it is 0 it produce 1.
		- **Logical AND (&&)** produces 0 if one or both of its operands evaluate to 0 otherwise it produces 1.
		- **Logical OR (||)** produces 0 if both of its operands evaluate to 0 otherwise, it produces 1.

E.g.:

 !20 //gives 0

 10 && 5 //gives 1

 10 || 5.5 //gives 1

 10 && 0 // gives 0

**N.B**. In general, any non-zero value can be used to represent the logical true, whereas only zero represents the logical false.

**Increment/Decrement Operators: (++) and (--)**

* + - The auto increment (++) and auto decrement (--) operators provide a convenient way of, respectively, adding and subtracting 1 from a numeric variable.

E.g.:

if a was 10 and if a++ is executed then a will automatically changed to 11.

**Prefix and Postfix**:

* + - The prefix type is written before the variable. Eg (++ myAge), whereas the postfix type appears after the variable name (myAge ++).
		- Prefix and postfix operators can not be used at once on a single variable: Eg: ++age-- or --age++ or ++age++ or - - age - - is invalid
		- In a simple statement, either type may be used. But in complex statements, there will be a difference.
		- The prefix operator is evaluated before the assignment, and the postfix operator is evaluated after the assignment.

E.g.

int k = 5;

(auto increment prefix) y= ++k + 10; //gives 16 for y

(auto increment postfix) y= k++ + 10; //gives 15 for y

(auto decrement prefix) y= --k + 10; //gives 14 for y

(auto decrement postfix) y= k-- + 10; //gives 15 for y

**Conditional Operator (?:)**

* + - The conditional operator takes three operands. It has the general form:

Syntax:

*operand1 ? operand2 : operand3*

* + - First operand1 is a relational expression and will be evaluated. If the result of the evaluation is non zero (which means TRUE), then operand2 will be the final result. Otherwise, operand3 is the final result.

*E.g.: General Example*

*Z=(X<Y? X : Y)*

*This expression means that if X is less than Y the value of X will be assigned to Z otherwise (if X>=Y) the value of Y will be assigned to Z.*

E.g.:

 int m=1,n=2,min;

min = (m < n ? m : n);

The value stored in min is 1.

E.g.:

 (7 = = 5 ? 4: 3) returns 3 since 7 is not equal to 5

**Comma Operator (,)**

* + - Multiple expressions can be combined into one expression using the comma operator.
		- The comma operator takes two operands. Operand1,Operand2
		- The comma operator can be used during multiple declaration, for the condition operator and for function declaration, etc
		- It first evaluates the left operand and then the right operand, and returns the value of the latter as the final outcome.

E.g.

int m,n,min;

int mCount = 0, nCount = 0;

min = (m < n ? (mCount++ , m) : (nCount++ , n));

* + - Here, when m is less than n, mCount++ is evaluated and the value of m is stored in min. otherwise, nCount++ is evaluated and the value of n is stored in min.

**The sizeof() Operator**

* + - This operator is used for calculating the size of any data item or type.
		- It takes a single operand (e.g. 100) and returns the size of the specified entity in bytes. The outcome is totally machine dependent.

E.g.:

 a = sizeof(char)

b = sizeof(int)

c = sizeof(1.55) etc

**Explicit type casting operators**

* + - Type casting operators allows you to convert a datum of a given type to another data type.

E.g.

 int i;

float f = 3.14;

i = (int)f; 🡪 equivalent to i = int(f);

Then variable i will have a value of 3 ignoring the decimal point

**Operator Precedence**

* + - The order in which operators are evaluated in an expression is significant and is determined by precedence rules. Operators in higher levels take precedence over operators in lower levels.

***Precedence Table:***

|  |  |  |
| --- | --- | --- |
| Level | Operator | Order |
| Highest | sizeof()++ -- (pre fix) | Right to left |
|  | \* / % | Left to right |
|  | + - | Left to right |
|  | < <= > >= | Left to right |
|  | == != | Left to right |
|  | && | Left to right |
|  | || | Left to right |
|  | ? : | Left to right |
|  | = ,+=, -=, \*=, /=,^= ,%=, &= ,|= ,<<= ,>>= | Right to left |
|  | ++ -- (postfix) | Right to left |
| Lowest | , | Left to right |

E.g.

a = = b + c \* d

c \* d is evaluated first because \* has a higher precedence than + and = =.

The result is then added to b because + has a higher precedence than = =

And then == is evaluated.

* + - Precedence rules can be overridden by using brackets.

E.g. rewriting the above expression as:

a = = (b + c) \* d causes + to be evaluated before \*.

* + - Operators with the same precedence level are evaluated in the order specified by the column on the table of precedence rule.

E.g. a = b += c the evaluation order is right to left, so the first b += c is evaluated followed by a = b.

### ***2.8 Debugging and programming errors***

When we attempt to produce an efficient program we should take sufficient care to maintain clarity and readability of a program. The program errors are called ***bugs.*** The art of locating and eliminating bugs or errors is called ***debugging.***

There are actually three different kind of errors:

* compile-time errors: identified by the compiler, includes
	+ missing brackets
	+ undeclared variables
	(e.g. misspelling variable name in one place)
	+ incorrect use of single or double quotes
	+ invalid identifier names
* run-time errors: attempts by the program to do something illegal while executing, can include
	+ trying to divide by zero (will crash)
	+ using or printing a variable value before initializing it (usually will not crash, just give very weird results)
* logic errors: the implementation doesn't correctly solve the problem, includes any form of miscalculation or incorrect ordering of instructions

***Worksheet 2***

For each of the problems write a C++ code to perform the required task. Your program should be based on the flow chart you drawn in the first worksheet.

1. Receive a number and determine whether it is odd or even.
2. Obtain two numbers from the keyboard, and determine and display which (if either) is the larger of the two numbers.
3. Receive 3 numbers and display them in ascending order from smallest to largest
4. Add the numbers from 1 to 100 and display the sum
5. Add the even numbers between 0 and any positive integer number given by the user.
6. Find the average of two numbers given by the user.
7. Find the average, maximum, minimum, and sum of three numbers given by the user.
8. Find the area of a circle where the radius is provided by the user.
9. Swap the contents of two variables using a third variable.
10. Swap the content of two variables without using a third variable.
11. Read an integer value from the keyboard and display a message indicating if this number is odd or even.
12. read 10 integers from the keyboard in the range 0 - 100, and count how many of them are larger than 50, and display this result
13. Take an integer from the user and display the factorial of that number

### Spotting errors

For each of the following programs, spot the flaws:

**Program 1**

/ this program will calculate the area of a square

#include <iostream>

Using namespace std;

 void main()

{

 int area; / store calculated area of square

 int side; / input length of side of square

  cout << "How wide is the square?" << endl;

 cin >> side;

  area = side \* side;

  cout << "The square has area area" << endl;

}

**Program 2**

 // this program prints information about the author

 void main()

 {

 cout << 'This program was written by Dave Wessels';

 cout << 'He didn't do a great job of it, ' << endl;

 cout << 'It won't even compile';

}

 // this program calculates user age

#include <iostream.h>

 int main()

 int currentyear;

 int birthyear;

  cout << "Enter the year you were born" << endl;

 cin >> birthyear;

  cout << "Enter the current year" << endl;

 cin >> currentyear;

  age = currentyear - birthyear;

  cout << "This year you will be" << age << endl;

}

**Program 3**

 // this program gets the user to enter three

 // integers and prints them out lined up on

 // three separate lines, e.g.

 // 17

 // 1024

 // 206

#include <iostream>

Using namespace std;

 void main()

{

 // variables for the three integers

 int firstint; int secondint; int thirdint;

  cout << "Please enter three integers" << endl;

 cin << firstint

 cin << secondint

 cin << thirdint

  cout << "The three integers are:;

 cout << setw(5) << firstint;

 cout << setw(4) << secondint;

 cout << setw(3) << thirdint;

  return 0;

}

**Program 4**

// given the radius of a circle,

// this program prints its circumference,

// and prints its area to three decimal places

#include <iostream>

#include <iomanip.h>

 Using namespace std;

const int Pi = 3.1415

void mian()

(

 int radius; // input circle radius

 int circumference; // calculated circumference

 float area; // calculated area

  cout << "Enter the integer radius of the circle";

 cout << endl;

 cin >> radius;

  circumference = 2 \* radius \* Pi;

 area = Pi \* raduis \* radius;

 cout << "The circumference is " << circumference;

 cout << ", the area is ";

 cout << setprecision(2) << area << endl;

 )

For each of the problems write a C++ code to perform the required task. Your program should be based on the flow chart you drawn in the first worksheet.

1. Receive a number and determine whether it is odd or even.
2. Obtain two numbers from the keyboard, and determine and display which (if either) is the larger of the two numbers.
3. Receive 3 numbers and display them in ascending order from smallest to largest
4. Add the numbers from 1 to 100 and display the sum
5. Add the even numbers between 0 and any positive integer number given by the user.
6. Find the average of two numbers given by the user.
7. Find the average, maximum, minimum, and sum of three numbers given by the user.
8. Find the area of a circle where the radius is provided by the user.
9. Swap the contents of two variables using a third variable.
10. Swap the content of two variables without using a third variable.
11. Read an integer value from the keyboard and display a message indicating if this number is odd or even.
12. read 10 integers from the keyboard in the range 0 - 100, and count how
many of them are larger than 50, and display this result
13. Take an integer from the user and display the factorial of that number
14. Take an integer from the user and display the factorial of that number
15. Read any natural number from the user and count the number of digit.
16. Read any natural number from the user and determine whether it is Armstrong number or not. (153 = 13 + 23 + 53 = 153)
17. Read two number from the user and display the result of ab?
18. Read N- Number of integer from the keyboard, and display prime number only.