Module 1: The C++ Programming Language and C++ Program Control Structures

Lesson 1: Introduction to C++, Program Development Lifecycle, and C++ Programming Elements

Learning Outcomes

- ➢ LO 1.1.1 Identify the different programming language at explain its importance
- ➢ LO 1.1.2 Define the elements in a C++ program.
- ➢ LO 1.1.3 Assert the correct primitive datatype for a specified value.

Introduction:

Computers are created to solve intricate mathematical problems. Though they may appear as remarkably intelligent machines, they are not yet self-aware or cannot program themselves. These computers need the direction of human beings in order to perform a specified task. These tasks are known as programs and is accomplished through a step-by-step procedure known as algorithms. We have all carried out an instruction using English or our own dialect as a medium of communication. Well, computers tend to operate that way, except that it functions on a very strict grammatical rules with no tolerance for ambiguity. Moreover, the step-by-step procedure must be in tedious detail so that a given task may be accomplished as desired.

Machine Language:

Inside a computer all data are represented by the states of tiny electronic switches. These switches which are basically microscopic transistors has 1's and 0's state. On the early days of the electronic computer, programs had to be written in a combination of 1's and 0's. However, there was no standardized format for storing bits (1's and 0's) of information, thus the machine language of one machine differs from the machine language of the other. This paved way to a standardized data storage of modern computers which are binary codes. Despite this, machine language is still enormously far on how humans communicate, making it very prone to error and tedious even for trained programmers. Consider the equation

$$z = x + y$$

Suppose 10010100 is the binary instruction for load, and 11001100 is the instruction for store, and 10101010 is the instruction for add, you would need the following instruction to accomplish this task

```
10010100 10000000   //load the content of address of x to accumulator

10101010 10000001   //multiply the content of address of y to
accumulator

11001100 10000011   //store the address of accumulator to z address
```

Assembly Language

Mnemonics were made in order to make the machine language easy-to-remember Mnemonics also known as memory aids are alphabetic abbreviations for machine instruction and paved way for assembly languages that made the programmer's job easier. Take for example the machine code and assembly mnemonics of the following:

```
10010100 – LOAD

11001100 – STORE

10101010 – ADD
```

Using these assembly mnemonics above would yield to the following instructions to compute the value of z

```
LOAD x
ADD y
STORE z
```

However since the computer would not directly understand these mnemonics, an assembler is required to translate these mnemonics to its machine instruction counterpart.

High Level Language

The move from machine language to assembly language significantly made programming much more convenient. However programmers were still directed to think in terms of specific machine instructions. The big leap forward is making programming easier by making it closer to natural languages (i.e. English). This idea made way for high level languages such as Pascal, BASIC, FORTRAN, COBOL, C/C++ and Java. For example the C++ code for solving z would be as follows:

```
z= x + y;
```

It would now be self-explanatory to understand this code given a little background in arithmetic. But, similar to assembly language the computer would not execute this directly, as a compiler is needed to convert a high level language instruction into machine language.

Program Development Lifecycle

Programming is problem solving, thus a systematic approach is necessary in order to achieve the specified objectives. This enables the programmer not only to solve the problem but also outline a documentation on how the solution came about. In a way making the code much easier to update and maintain by either the same or a different programmer. A usual problem solving process are as follows:

1. Analyze the problem then identifies the input requirements, solution requirements, and output requirements.
2. Design an algorithm based on the requirements.
3. Convert the algorithm into a C++ code then check and debug the errors.
4. Modify the program if necessary

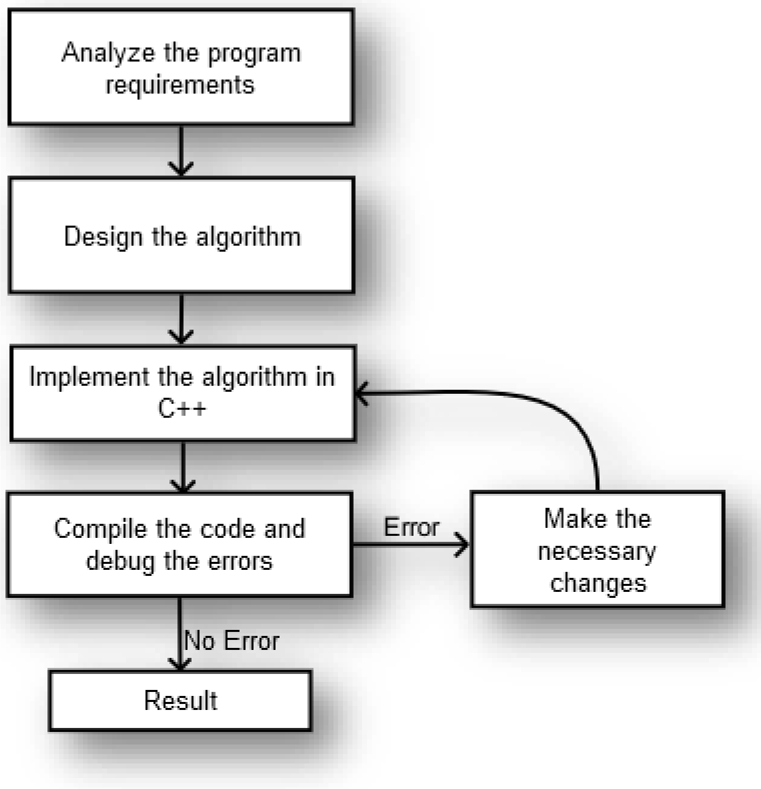
Analyze the program requirements
Design the algorithm
Implement the algorithm in C++
Compile the code and debug the errors
Error
Make the necessary changes
No Error
Result

Figure 1: Graphical view of the program development life cycle

C++ Programming and its Basic Components

The C++ programming language is regarded as C programming language with classes together with other modern features. Dennis Ritchie developed the C programming language at AT&T Bell Laboratories in the 1970's. This language was then used to create and maintain the UNIX operating system up until now. As a general purpose programming language it can be employed to write a variety of program but its popularity came to prominence since it is closely linked to UNIX operating system. Therefore if you want to create an application in the UNIX environment, it is imperative that you would know how to use C. Through the years it became so popular, that other operating systems also adopted C language. Thus, its application is not only limited to computer that used UNIX.

Even with its popularity, C wasn't free from any inadequacy. Since C by itself is a contradiction that is it can be regarded as a high-level language with low level language features. Most experts agree that C is a middle level language and inherited the strengths and weakness of both programming paradigm. For instance C has high-level language features that enable the programmer to easily read and write assembly language programs. Yet, it can also directly manipulate the computer's memory similar to assembly language. Though this may be useful for systems programming, but application programmers would find the learning curve quite steep. Moreover automatic checks that is available to most high-level language is absent in C.

To address these issues, Bjarne Stroustrup still from AT&T Bell laboratories develop C++ in the 1980's. He designed C++ as a subset of C and made C++ better than C. Since C++ is a subset of C most C++ programmers aren't considered as C programmers but C programmers can be considered as C++programmers. The lacking features on C such as a facility for classes of object-oriented programming and automatic checks were addressed in C++.

Object-Oriented Programming in C++

Since C++ integrates a new programming paradigm into C, it is fair to examine the old paradigm that C follows. Generally, computer programs deals with two components, the *data* and the *algorithm*. The data constitutes of what information to process while the algorithm entails on how to process the information. Conceptually C follow the procedural approach to programming, in which it emphasizes on algorithms. This approach forced the programmers to implement the actions that needed to be done in order to accomplish a certain task. Early programming languages based on this paradigm ran into an organizational problem as the code base grew in size mainly due to branching statements. Consequently, the developers of C respond to this by making a more disciplines structured approach. This includes branching features is limited to `while`, `for` and `do-while` loop. Thus, making C a top down approach, for it would break down a large task into small manageable ones.

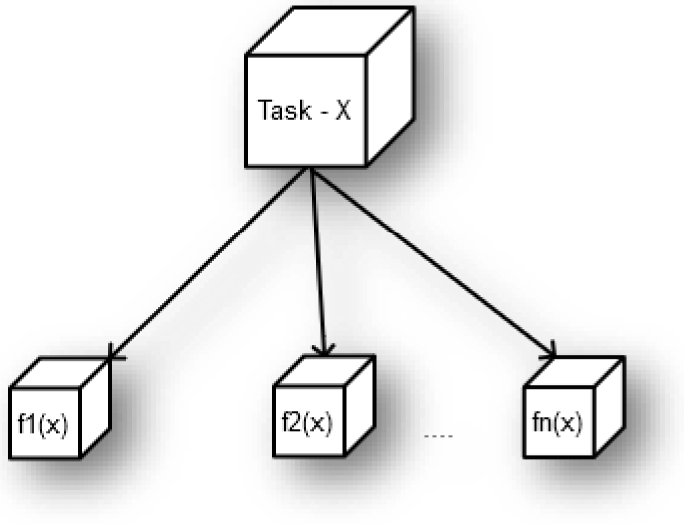

Figure 2: Top down approach programming

As far as structured programming is concerned it brings clarity, straightforwardness of maintenance and reliability to a small-scale program. But large scale programming still remains a challenge. Object-oriented programming (OOP) brings a fresh approach to this challenge, due to the fact that instead on focusing on the algorithm it shifts its attention to the data. The idea here is to design certain data forms that would fit to the indispensable attributes of a problem. For example in C++, a class specifies a new form of data, while an object is the data structure based on the class. A class could be a car with properties such as model, and manufacturer while the object would place values on the model such as Land Cruiser™ and manufacturer such as Toyota®. In general, a class would define the data and operation that best represent an object. So, on this approach one would proceed to design a program from defining lower level organization of classes then incorporate these classes to create the final program (also called bottom-up approach).

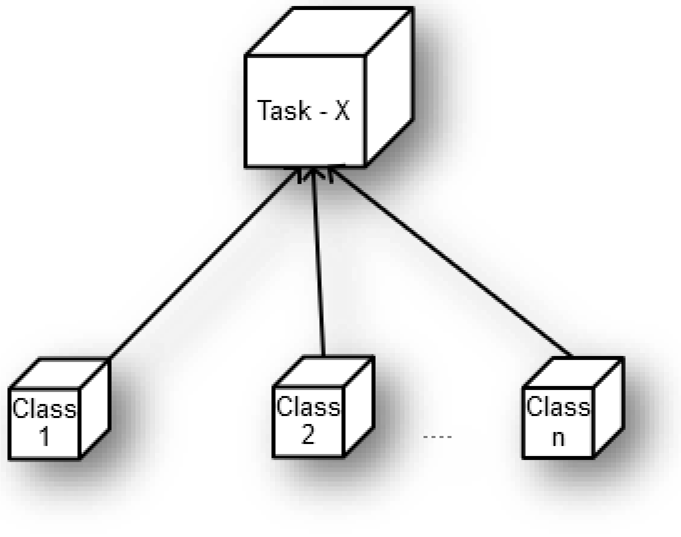

Figure 3: Bottom up approach in programming

In addition to binding data and methods together in a class, OOP is capable of

- safeguarding data from improper access
- create multiple definition for operator and function (polymorphism)
- derive new classes from old ones (inheritance)

It may be a daunting task to create a reliable and reusable class. However, some vendors would provide useful class libraries that are already field tested then would easily be incorporated to your program. This would simplify the program development process lets you adapt and reuse a robust code base.

Basic Elements of a C++ program

```
//your first C++ code of z = x + y equation
#include <iostream>
using namespace std;
int main()
{
    int x = 0, y = 0, z = 0;
    cout<<"Enter the value of x: ";
    cin>>x;
    cout<<"Enter the value of y: ";
    cin>>y;
    z = x + y;
    cout<<"x + y = "<<z<<endl;
    cout<<"Thank you!";
}
```

Listings 1: Sample C++ code based on the z = x + y equation

The ***C++ comment*** is a remark that is given by the programmer to identify or explain a section of the code. The comments are ignored by the compiler. In C++ double slash (//) indicates a single line comment while a slash-asterisk asterisk-slash (/* */) is used for multiple line comments. During the early stages of your programming journey it is recommended to write comments on critical parts of your code. For example in Listings 1 the placed on the topmost part of the program to describe what it does.

```
//your first C++ code of z = x + y equation
```

The ***preprocessor*** is a part of the program that provides the source file with the capabilities provided from the included header file. On Listings 1 the invoke directive and header file is:

```
#include <iostream>
```

This header would provide the features for your program to communicate from the outside world (input) or to the outside world (output). Moreover the input/output in C++ involves several definition from the `iostream` file, for instance `cin` is for input and `cout` for output. There are many header files supported by C++ each supporting its own set of facility. The traditional C heads must end with `h` extension, but modern C++ headers reserved the `h` extension to old C headers. Instead header files on C++ removed the headers for example `<math.h>` in C is now `<cmath>` in C++.

.

The ***namespace*** support is a new feature in C++ that is designed to simplify coding by combining preexisting code from different vendors. For example Vendor1 and Vendor2 both have the function called `abc()`, since the compiler would not understand which version the programmer mean, the namespace facility would guide the programmer to which vendor s/he mean. So, `Vendor1::abc()` is the `abc()` function of Vendor1 consequently `Vendor2::abc()` is the `abc()` function of Vendor2.

The ***main() function*** is the entry point of your program. This constitute to a function definition, which basically has two parts, the `int main()` is known as the *function header* while the portion enclosed in curly braces ({ and })is the *function body*. The function header summarizes the function's interface with entire program while the function body denote the instructions the must be executed by the computer. Each complete instruction is called a *statement*, and each statement must end with a *semicolon*.

Lesson 2: Data types, Arithmetic operators, I/O Statements and Programming Form and Style

Learning Outcomes

- LO 1.2.1 Write a sequential C++ code using I/O statements, assignment operators and basic mathematical function.
- LO 1.2.2 Implement proper programming practices to make a C++ code more readable.

Declaring Variables and Primitive Data Type

Identifiers

An identifier is the name that a variable would have. In C++ it must begin with either a letter or an underscore and the rest of the characters may be a letter, underscore, or digit. The syntax diagram is shown below

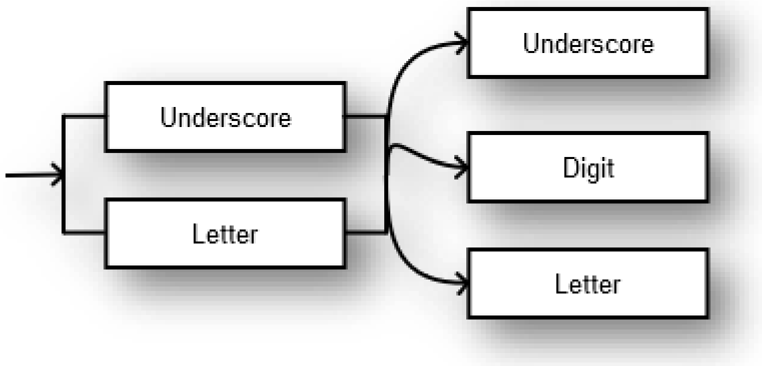

Figure 4: Syntax Diagram for Identifiers

C++ differentiates uppercase and lowercase characters, in other words it is *case sensitive*. So, the identifiers `sample`, `SAMPLE`, and `Sample` may have similar meaning but the compiler considers these as three distinct identifiers. That is why it would create confusion if you will try to create two variants of the same variable. A common practice would mix the upper case with the lowercase by using the uppercase character to indicate a word boundary (e.g. `mySample, yourSample`), this is otherwise known as ***camel case***.

Reserved words are predefined names in C++ and the programmer is not allowed to use it as names for variables (e.g. `if() else`). On most Integrated Development Environment (IDE) the reserved words would appear with a different color. ***Keywords*** on the other hand are not part of the core C++ language and are defined in headers (e.g. `cin, cout`). The programmer may redefine its usage but confusion would arise since it would deviate from its standard meaning. So to be safe, it is better use these keywords for their standard purpose only.

Table 1: Some examples of correct and incorrect identifiers

Correct Identifiers	Incorrect Identifiers
`Interest`	`4thDigit`
`_max`	`θ`
`final_`	`if`
`x9991`	`id number`
`a_very_long_identifier`	`(age)`
`computeInterestRate`	`pre-paid`

C++ datatypes

Before a variable would be used it must be declared first. When a variable is declared, you are stating to the compiler, eventually the computer to allocate memory for storing the variable. Any legal identifier excluding reserved words may act as variable name but it must first be preceded with datatype. A data type is the way C++ categorizes data according to the operation fit to it. For example, arithmetic operation are not suitable to manipulate a group of character. C++ has three major datatype categories:

1. Simple datatype
2. Structured datatype
3. Pointers

For this chapter, only simple data types would be covered, for they are the building blocks of structured data type on Chapter 5. Furthermore there are thee classification of simple datatypes namely

1. ***Integral*** – these are the data types without a decimal place. So if a number with a decimal place would be assigned to it,

only the whole number part would be considered. Except for bool datatype, they can be signed by default or be unsigned if an unsigned identifier would be placed prior to the data type (e.g. unsigned int x).

2. ***Floating point*** – are data types that deal with real numbers (i.e. numbers with decimal places)
3. ***Enumeration*** – are user defined datatypes

For this Chapter only the integral data type (Table 2) and floating point data type from Table 3 would be included.

Table 2: Integral data type

Name	**Description**	**Size (bytes)**	**Range**
`bool`	Boolean	1	True, false
`char`	Character	1	signed: -128 to 127 unsigned: 0 to 255
`short int`	Integer.	2	signed: -32768 to 32767 unsigned: 0 to 65535
`long int` or `int`	Long integer.	4	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`long long int`	Long long integer	8	signed: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 unsigned: 0 to 18,446,744,073,709,551,615

Table 3: Floating point data type

Name	Description	Size (bytes)	Range	Precision (Digits)
`float`	Single Precision	4	3.4e +/- 38	7
`double`	Double precision	8	1.7e +/- 308	15
`long double`	Long double precision.	10	1.7e +/- 4932	19

Assignment operator

Changing or initializing the value of a variable is straightforward, you would just use the assignment operator symbolized by an equal sign (=). An assignment operator must always have the left side operand (variable) accept whatever the evaluated value of the right hand operand (expression). Here are some examples:

```
int x = 10;
int sum = 5 + x;
```

Note: that the assignment operator is not an equality operator, it does not evaluate if the left hand operand is equal with the right hand operand.

Arithmetic Operators, Operator Precedence and Associativity

Computers are highly valued for their ability to quickly calculate numerical data. So, C++ has standard arithmetic operators as shown on Table 4: Standard arithmetic operators

Table 4: Standard arithmetic operators

Operator	Operation	Precedence Number	Associativity
`()`	Override default precedence rules	1	left-to-right
`-`	Negation	2	right-to-left
`* / %`	Multiplication, division, and modulo operator	3	left-to-right
`+ -`	Addition and subtraction	4	left-to-right

All the operators can be applied to manipulate both integral and floating point data, except modulo operator (i.e. modulo requires integers for it would get the remainder from a division process). Moreover C++ uses precedence rules to address which operator is to be processed first. Normally, algebraic precedence is followed, but when operators have the same precedence (e.g. multiplication and division) C++ would look at the associativity. For operators acting on the same operand with left-to-right associativity, the left operator would be evaluated first. Consequently for right-to-left associativity, the right operator must be evaluated first. For example:

```
5 * 2 - 6 + 6 * 3 / 5 + 8;
```

If an integer would be expected as the result this means that:

```
(((5 * 2) - 6) + ((6 * 3) / 5)) + 8;

((10 - 6) + (18 / 5)) + 8; //evaluate *

((10 - 6) + 3) + 8;  //evaluate / (note: this is integer division)

(4 + 3) + 8;   //evaluate -

7 + 8;         //evaluate first +

15;            //evaluate +
```

Evaluating Mixed Expression

It is obvious that integer expressions would yield to integer results while floating-point results would produce floating-point results. But, a mixed expression are operands that has different data types. To evaluate the mixed statement follow the rules below.

1. Before evaluation, an integer operand paired with a floating-point operand would be changed to floating-point.
2. The entire expression are then evaluated corresponding to precedence and associative rules.

For example:

```
3.4 * 3 - 5.6 + 4 / 2;
10.2 - 5.6 + 4 / 2;     //evaluate 3.4 * 3.0 (rule 1)
10.2 - 5.6 + 2;         //evaluate 4 / 2  (rule 2)
4.6 + 2;                //evaluate  10.2  -  5.6  (same  data
type)
6.6                     //evaluate 4.6 + 2.0 (rule 1)
```

Type Casting

As you can see, implicit (automatic) type conversion would occur on mixed operands. If you are not be careful about this mechanism, it can generate unintended results. To avoid this coerced conversion, a cast operator is used. The format is as follows:

```
static_cast<dataType>(expression);
```

Initially the expression is evaluated, then converted to a values specified by the data type. So, if a floating-point data is casted as an integer, the number from right side of the decimal place will be dropped. Conversely, a decimal place will be added if an integer would be casted as a floating-point data.

```
float e = 2.71828;
static_cast<int>(e); // the output is 2
```

Finally, if larger-sized (e.g. `long int`) integral data type is converted to a smaller-sized (e.g. `short int`) data type possible loss of data will occur since the content of the larger will not be accommodated to the smaller. This principle is also true with the floating-point data type.

Input/output Statements and Basic Mathematical Functions

Programming Form and Style

Using `cout`

The `cout` (pronounced as see-out) is a predefined object of the `ostream` (from `iostream` header) class definition. It would bear the properties of the `ostream` class such as inserting individual floating-point data type or a character string (i.e. series of characters enclosed in double quotation) into an output system. The `<<` indicates that there is a stream of data being inserted from its right to the output system. This stream is then terminated when the statement would end. Consider the example in Listings 1

```
cout<<"x + y = "<<z<<endl;
```

This output stream would insert character string `"x + y = "` and the numerical value of integer variable z. Using objects to display output is much easier compared to functions since it would not require the you to understand the internal components in order to use that object. The only requirement is you must know how to interface with it.

Using `cin`

The `cin` (pronounced as see-in) is a predefined object of the `istream` (from `iostream` header) class definition. It has the properties of the istream class so, it would insert data from the input device into a variable on a program. The >> operator extract characters from the input stream then converts it with respect to the data type that would receive it from the right side of this symbol. Take for example Listings 1

```
cin>>y;
```

This statement would send an input stream to integer `y`.

The `endl` manipulator

When a C++ a `cout` statement would end, it does not move automatically to the next line. Distinct manipulators such as the endl is used since it would cause the cursor to move one line down to the left. Similar with `cout`, `endl` is included in the `iostream` header file and belongs to the `std` namespace. Example in Listings 1

```
cout<<"x + y = "<<z<<endl;

cout<<"Thank you!";
```

The output to this is the “x + y” character string and the numerical value of z, followed by the character string “Thank you”. However if the endl is removed the output would be “x + y” character string, numerical value of z and the “Thank you!” character string would be on the same line.

Basic math functions

Built-in functions in C++ are basic modules that are essential building blocks of a program and vital for OOP definition in C++. There a two varieties of function: the value returning function and those that don't. Math functions in C++ are value returning function. Table 5 shows some basic math functions.

Table 5: Some math functions in C++

Name	Description of Computation	Datatype of Argument	Result
`abs (argument)`	The absolute value of the argument	double/integer	same as argument
`log10 (argument)`	The common logarithm (base 10) of the argument	double/integer	double
`sqrt (argument)`	The positive square root of the argument	Positive double/integer	double
`pow (argument_x, argument_y)`	Raises argument_x to argument_y	double	double
`cos (radians_argument)`	The cosine of the argument in radians	double/integer	double
`sin (radians_argument)`	The sine of the argument in radians	double/integer	double

You should include the `<cmath>` header in order to make use of these functions. When the function is invoked the *argument* or *parameter* is the information that is passed to the function and the result will be the *return value*. For example:

```
double root = sqrt (4);
```

This statement will extract the square root of the argument which is 4. Then the return value is stored on the variable named `root`. Although a specific chapter (Module 2: Modular Programming) is intended for this topic, an overview of function will provide an outlook on how functions operate.

Programming Form and Style

Use of blanks, semicolon, curly braces and commas

Blanks one or more of them are used to separate identifier, reserved words and other form of symbols. But they must never appear inside an identifier or reserved word.

Semicolons as previously mentioned are used to terminate statements. All statements in C++ must end with a semicolon.

Curly braces { and } are not statements but rather are delimiting brackets. Since they are responsible for enclosing a body of a function. Some other uses will be discussed on the latter chapters.

Commas are used to separate items in a list. Take for instance declaring more than one variable with the same data type, a comma is used to separate the variables that you declared.

Syntax vs Semantics

Syntax are rules set by the C++ compiler to determine if a statement is legal or illegal. Syntax errors are detected during compilation time. For example

```
int x = 5
```

When this line is compiled an error message would occur because of a missing semicolon. These errors are almost unavoidable and most likely happen when you are beginning your programming journey. In some cases many error messages would stem from one error. But, if the root syntax error is removed then the program is recompiled, the subsequent syntax error would vanish. Quickly correcting syntax errors need familiarity and experience to C++ programming.

Semantics on the other hand are the rules that give meaning to statements (e.g. order of precedence). Unlike syntax error, there will be no warning message that is provided by the compiler. So, you may eliminate all syntax error but you will still get the wrong result. Take for example:

```
2 + 2 * 5;  `      //result is 12

(2 + 2) * 5;       //result is 9
```

Both lines are syntactically correct but are semantically different. So these statement are not interchangeable.

Naming identifiers and documentation

Identifiers that you declared must be self-documenting. Self-documenting identifiers will make less comments. As you have recalled when two words are necessary to describe an identifier it is recommended that you would use camel case. However, it is recommended that a named constant will be all in uppercase with an underscore as separator for different words. For example,

```
double sumOfSquares = 0.0;

const double YARDS_PER_METER = 1.093613;
```

Moreover, if necessary you program must be well documented so that it would be easier to understand and modify. Comments must appear to explain the purpose of the program and some specific statement. But modern practices in coding would require the programmer to factor out a very large code to its fundamental components so that each component would be self-documenting.

Code formatting for ease of readability

C++ provides you the freedom of formatting your program. But to make it more readable you must have a structured style formatting. Most programmers observe these three (3) simple rules

- `Only one statement per line.`
- `An opening brace and a closing brace must have its own line.`

All statements under a brace must be indented

Lesson 3: Introduction to Program Control Structures

Learning Outcomes

- LO 1.3.1 Analyze the logical flow of a given problem.
- LO 1.3.2 Apply the proper relational and logical operator to properly control a selection or repetition structure.

Introduction to Program Control Structures

So far the program that was introduced in the previous chapter would only deal with sequential approach, no branching decision nor repetition. However, program control structures would enable a program not only to do sequential execution but also selection and repetition.

In ***selection*** structure, the program would execute a decision or branch to alter the flow of the program depending on some condition(s)

In ***repetition*** structure, the program repeats a statement or a group of statements if a certain condition(s) is still true.

Although, this may be a new concept most of us perform these statement in our daily lives. For example,

```
If(tired)
       Rest
Else
       Play
```

```
While (jug is not full)
       Put water on jug
```

The portion enclosed on the parenthesis is the requirement that would direct a decision. In C++ this requirement must be packaged in such a way that it must only be answerable by yes or no.

Relational and Logical Operators

Since a control structure needs to be answerable by yes or no, relational operators are necessary to accomplish this task. These operators are also known as comparison operators since they would make a Boolean (yes or no) decision by comparing the left hand operand with a right hand operand. Moreover the precedence of these operators would come *after arithmetic operators*.

Table 6: List of Relational Operators

Operator	Operation	Precedence number
<	Less than	1
<=	Less than or equal to	1
>	Greater than	1
>=	Greater than or equal to	1
==	Equal to	2

`!=`	Not equal to	3

Relational operators can be used on any simple data type. But when comparing characters one has to refer on an ASCII character table for the numerical values of the characters.
For example

Table 7: Some usage of relational operators

Expression	Meaning	Value
`3 < 20;`	3 lesser than 20	true
`int a = 5, b =10; a>=b;`	(a is 5) greater than or equal to (b is 10)	false
`float x = 3; x!=4;`	(x is 3) not equal to 4	true

Logical operators on the other hand would enable you to combine Boolean statements together. They would accept logical values (result of a Boolean operation) as operand and produce logical values as result. The table below shows these operators.

Table 8: Logical operators

Operator	Description	Precedence
`!`	Not	1
`&&`	And	2
`||`	Or	3

The `!` operator is a unary (prefix) operator and has only one operand, while the `&&` and `||` are binary(infix) operators. When a `!` operator is placed before a Boolean expression it would reverse its result. Moreover this operator has the *same precedence as the unary operator* from the previous chapter.

Table 9: Truth table for not operator

Expression	!(Expression)
true (nonzero)	false (0)
false (0)	true (1)

The `&&` (and) operator would be evaluated as true if both of the right and the left expression are true. Conversely, an `||` operator is evaluated as true if only one or all of its statements are true. Both this operator has a precedence that would come *after the relational operators.*

Table 10: Truth table for and & or operator

Expression 1	Expression 2	Expression 1 `&&` Expression 2	Expression 1 `||` Expression 2
false (0)	false (0)	false (0)	false (0)
false (0)	true (non-zero)	false (0)	true (1)
true (non-zero)	false (0)	false (0)	true (1)
true (non-zero)	true (non-zero)	true (1)	true (1)

Take for example

Table 11: Some examples of using not, and & or operators

Expression	Value	Explanation
`int x=0; !(x==0);`	false	Since x is equal to zero (true) then inverting the true statement would produce false.
`int y=10;` `(y>0) && (y<20);`	true	Since the first expression is true (y is greater than 0) and the second expression is true (y is lesser than 20) so the expression is evaluated as true.
`int z = 12;` `(z>50) || (3<10);`	true	Since the first expression is false (y greater than 50) and the second expression is true (3 is lesser than 10) so the expression is evaluated as true.

Lesson 4: Selection Structure

Learning Outcomes

- LO 1.4.1 Implement an appropriate selection structure for a given analyzed problem.

There may only be two logical values (true and false) that would result from a relational and logical operator. But, these values would turn out to be essential in performing decision making that alters the flow of a program. There are two selection structures in C++, the `if` statement and the `switch` structure. The `if` statement is generally used for one, two and multiple path selection, while `switch` is inherently a multiple path structure.

One and Two way selection **`(if-else)`**

A one way selection structure in C++ begins with a reserved word `if`, then it is followed by a parenthesis that would accept a logical expression. The succeeding statement is the one that would be executed if the logical expression is true. If the statement is false this statement would be skipped. Refer to Figure 5

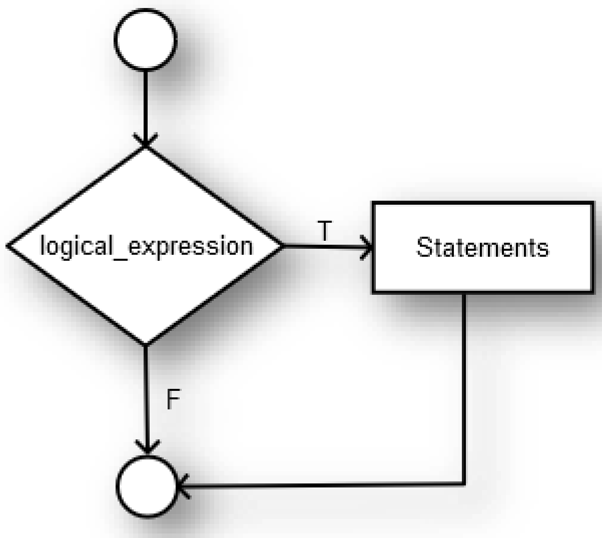

Figure 5: One way selection structure flowchart

The syntax and some example with explanation is shown below.

Table 12: Syntax and example of an if statement

Syntax	**Example**	**Explanation**

`if (logical_expression) statement_if_true;`	`int x =5,y=10;`
	`if (y<11) cout<<y;`
	
	`if (x>=3 && x<10) cout<<x;`
		The first if statement would evaluate if y is lesser than 11, then display the numerical value of y if true.
		
		The second would evaluate if x is greater than or equal to 3 and if x is lesser than or equal to 10. Then display the numerical value of x if the statement is true.

Take note that when a semicolon would be placed immediately after an `if` statement this would mean that you preemptively terminating the condition without any statement to be executed. Example:

```
if (7>5);//no output since there is no statement to execute
    cout<<"7";
```

There are numerous instances however that two options needs to be done. C++ provides a mechanism on this using the two way `if… else` statement. It starts with an `if` statement followed by a reserved word `else`. The first statement would execute if the logical expression is true, otherwise the statement after the else would execute. Figure 6 illustrates this operation.

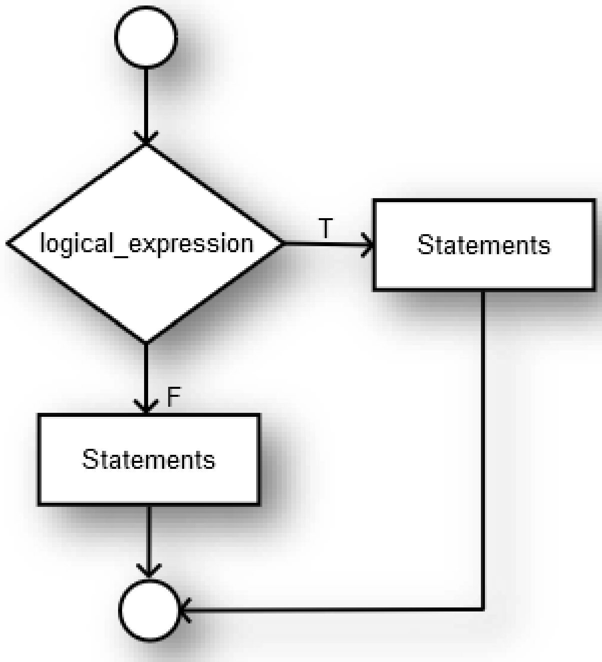

Figure 6: Two way selection structure flowchart

The syntax format together with an example with explanation is shown below.

Table 13: Syntax and example of an if-else statement

Syntax	**Example**	**Explanation**
if (logical_expression) statement_if_true; else statement_if_false	`int x = 5;` `if (x%2==0)` `cout<<"even";` `else` `cout<<"odd";`	The remainder of variable x divided by two would be evaluated. If the result is zero (i.e. x is divisible by zero) it is even. Otherwise the number is odd.

Block of statements

Block statement is a group of code that is composed of more than one line. Since an `if` statement would only execute one statement at a time, C++ provides a feature of blocking statements by enclosing them within curly braces. The form and an example is shown below.

Table 14: Form and example of compound statements

Form	**Example**
`{`	`int x = 3, y = 4;`
`Statement 1;`	`if (x%y == 0)`
`Statement 2;`	`{`
`Statement 3;`	`cout<<"x is divisible by y";`
`…`	`cout<<"y is a factor of x";`
`Statement n;`	`}`
`}`	`else`
	`{`
	`cout<<"x is not divisible by y";`
	`cout<<"y is not a factor of x";`
	`}`

The blocking of statements (also known as compound statements) is very essential when properly structuring your code and will be commonly used for this chapter.

Precaution for Relational Operator Associativity

Suppose you would only extract the square root of the numbers from 7 to 12. Typically you are to write the code below.

```
int var = 3;
if (7<=var<=12)
  cout<<sqrt(var);
```

When this code would be executed the integer var (although not within the 7 – 10 range) would satisfy the condition since the associativity of the <= operator is from left to right. So it would execute on this fashion:

```
(7<=3)<=20    //evaluate <=
(0)<=20       //evaluate <=
(1)
```

The proper way to write the logical expression is:

`7<=var && var<=12` or `var>=7 && var<=12`

Precaution for equality and assignment operator

To review, the assignment operator would place a value to a variable while an equality operator would evaluate if the left side operand is equal to the right side operand. Yet, C++ allows you to use any expression that can either be evaluated as true or false. Take for example,

```
int z = 7;
if (z = 6)
  cout<<"Equal";
```

The integer z is assigned with a value of 7 and the `if` statement assigned 6 to z. Meaning z is not evaluated if it is equal to 7 but rather assigned with a value of 6 thus resulting to true after the evaluation. Even an experienced programmer would make the mistake of replacing = with the == operation. This can be a very serious problem in a program that is why it is extremely necessary to check the logical operator of a logical expression.

Multiple Path selection `(if-else if)`

The previous `if` statement would only allow one or two options, but above this, multiple selection path would be employed. This is done by nesting an if to an else statement. Since no else can stand alone by itself, it must pair with the recent incomplete if.

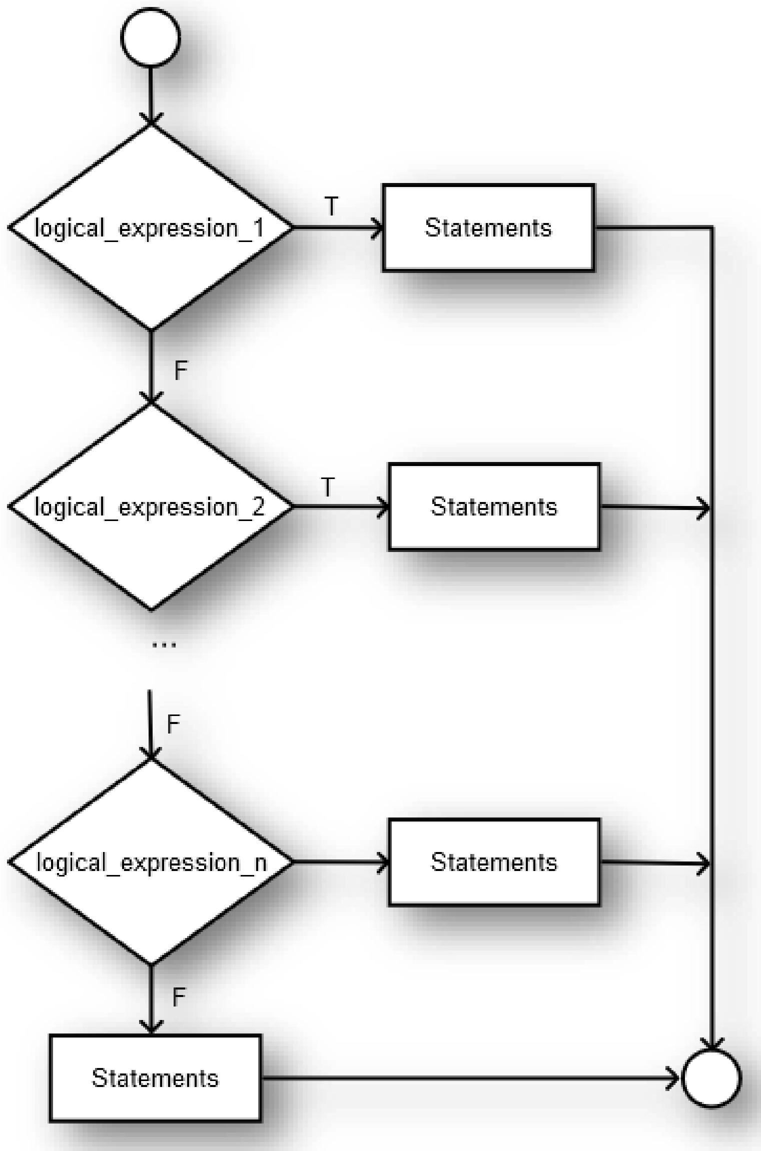

Figure 7: Multiple Path if -else statement flowchart

`Refer to the syntax and example below.`

Table 15: Syntax and example of a multiple path if-else if statement

Syntax	**Example**
if (logical_expression)	int grade = 96;
statement_if_true;	if (grade>100)
else	cout<<"Above
if (logical_expression)	limit";
statement_if_true (under	else if
previous else)	(grade>90)
else	cout<<"A";
if (logical_expression)	else if (grade>80)
…	cout<<"B";
else	else if (grade>70)
	cout<<"C";
	else if (grade>60)
	cout<<"D";
	else if (grade>50)
	cout<<"E";
	else
	cout<<"F";

The example above would evaluate the value of grade for the first if statement, since the result evaluates as false it would proceed else statement then evaluate if statement under it. This would evaluate as true so the A character would be displayed and the else below it would be terminated. In order that there would be no excessive indention the example is formatted differently from the syntax.

`switch` statement

A `switch` statement in C++ is an expression that does not require the evaluation of a logical expression. Instead, the expression would be evaluated if it would match a particular case value. Since a perfect match is necessary the expression must only be an integral datatype. The `switch, case, break` and `default` are reserved words typically used on switch structure.

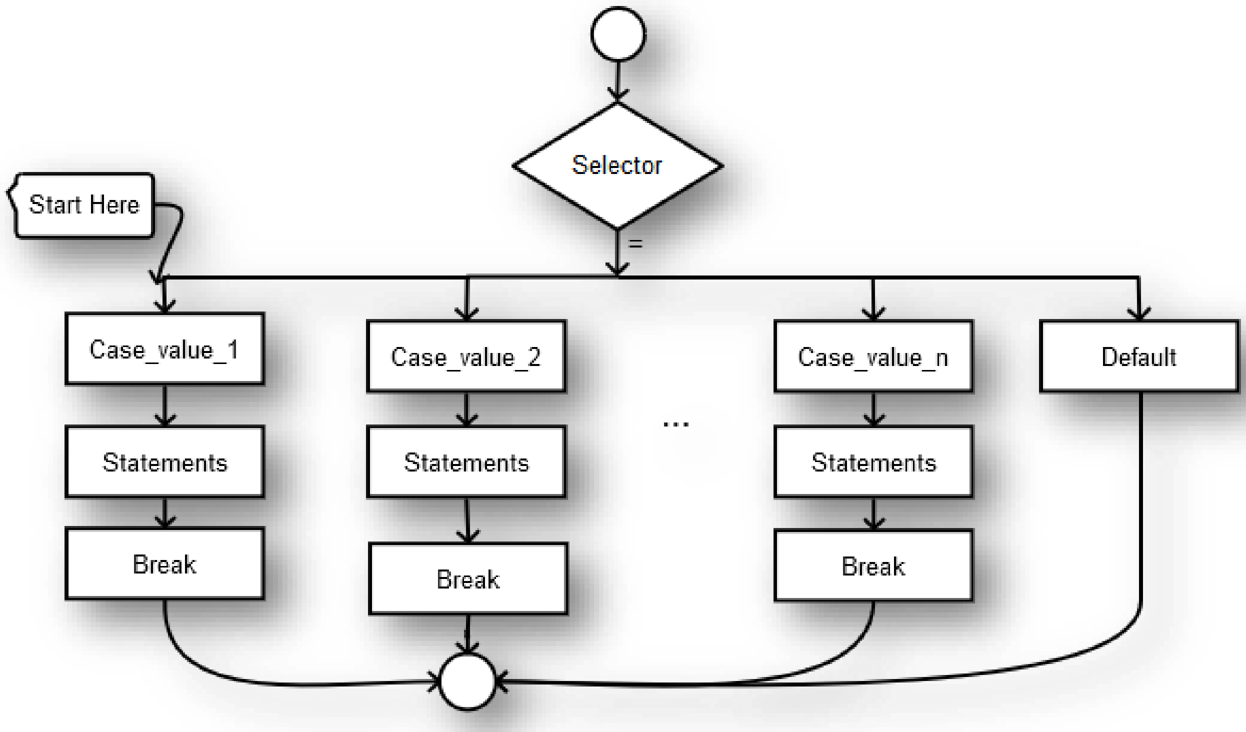

Figure 8: Switch statement flowchart

Here are the rules on how the `switch` statement operates.

1. When the value of the expression is a match against a case value, it would execute the statements under that case value until a `break` statement or the end of the `switch` statement is found.
2. If there is no match, the `default` label would execute. If there is no `default` label, the entire `switch` statement would be skipped.
3. The `break` statement is used to immediately exit from the `switch` statement.

The syntax and an example is shown below.

Table 16: Syntax and example of case statement

Syntax	**Example**
`switch (expression)`	`char gradeEquiv = 'A';`
`{`	`switch (toupper(gradeEquiv))`
`case value1:`	`{`
`statements1`	`case 'A': cout<<"90-100";`
`break;`	`break;`
`case value2:`	`case 'B': cout<<"80-89";`
`statements2`	`break;`
`break;`	`case 'C': cout<<"70-79";`
`.`	`break;`
`.`	`case 'D': cout<<"60-69";`
`.`	`break;`
`case valuen:`	`case 'E': cout<<"50-59";`
`statementsn`	`break;`
`break;`	`case 'F': cout<<"Below 50";`
`default:`	`break;`
`statements`	`default: cout<<"No such grade`
`}`	`exist";`
	`}`

The `toupper (expression`) statement on the `switch` expression is used to convert the `gradeEquiv` character to upper case. So by evaluating the expression it would match the case “A” then display “90-100” character string.

Lesson 5: Repetition Structure

Learning Outcomes

- LO 1.5.1 Write a repetition structure to address iterative problems.
- LO 1.5.2 Nest any combination of selection and repetition structure to address a complex problem

There are instances that some statements in a program would be repeated for a number of times. One solution may be to copy-paste the same statement on the particular number of times, but this is very impractical and often times impossible. That is why, C++ provides repetition structure (i.e. loop) repeatedly execute a statement or a group of statements until the specified condition is met. There are generally two types of loop, the pretest and posttest loop.

- The ***pretest loop*** would evaluate a logical expression before the statement(s) within the loop is/are executed.
- The ***posttest loop*** would evaluate the logical expression after the statement(s) within the loop is/are executed. Thus, a posttest loop will execute at least one regardless of the condition.

`while` loop

In C++ a while loop is type of pretest loop. The while is a reserved word in C++, and is type of pretest loop. The logical expression would act as the controller if the loop is to continue or to stop. The statements that are within the loop body can be a compound statement or a simple statement.

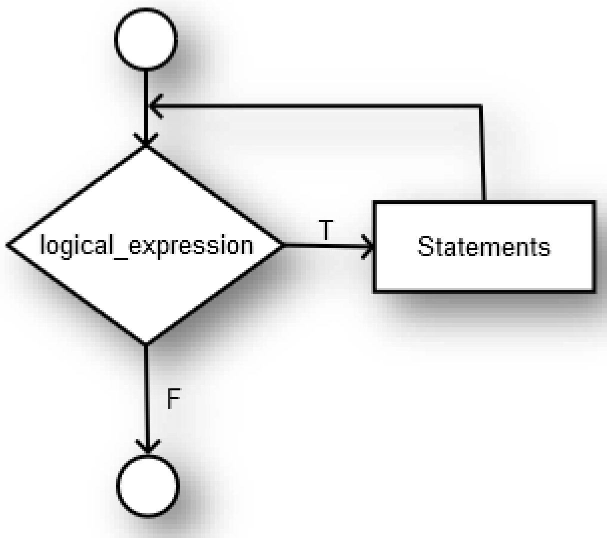

Figure 9: While loop flowchart

The syntax and an example is shown below

Table 17: Syntax and example of while loop

Syntax	Example
`while (logical_expression) { statement(s)_while_true }`	`int ctr=0; while (ctr<10) { cout<<ctr<<","; ctr = ctr + 1; } //output: 0,1,2,3,4,5,6,7,8,9`

The `while` loop on the example first initialize the `ctr` variable by 0. This would act as the loop control variable (LCV). The `ctr` value is first displayed then updated within the loop body. The loop would continue while the LCV is lesser than 10, once it would reach 10 then the loop will terminate.

Counters and Accumulator

The LCV for most loops is ***counter***. That is a numeric variable, usually an integral type that is initialized by a value then updated by a constant value. On the other hand, and ***accumulator*** is a numeric variable that is used to add collect (add together) some value. So an accumulator would be updated by different values. Take for example,

```
int ctr = 1;
int acc = 0;
while (ctr<=5)
{
    acc = acc + ctr;
    cout<<acc<<",";
    ctr++;
}
```

```
//output:
1,3,6,10,15
```

The example above would display the accumulated value in `acc` from `ctr`. The loop will terminate if the counter variable `ctr` would be greater than 5. Since the accumulator and counter would require an arithmetic process and an assignment operator, C++ provides simplified operators to these two operations.

Table 18: Some assignment operators

Operator	Operation	Precedence
`++,--`	Postfix increment	1
`--, ++`	Prefix decrement	2
`+=`	Addition with assignment	3
`-=`	Subtraction with assignment	3
`*=`	Multiplication with assignment	3
`/=`	Division with assignment	3

```
For example,
   int x = 3;
   x +=2; //x=x+2 output: 5
   x++; // x = x+1 output: 6
```

Sentinel Values and Flags

A ***sentinel value*** is a special value that would terminate a loop. This is generally useful when a series of data has to be inputted but the user is given a special value to terminate the series of input. A sentinel-controlled `while` loop would continue to execute as long as the program has not evaluated the sentinel value.

Table 19: Format and example of a sentinel controlled while loop

Format	Example

`while (variable != sentinel)`
`{`
`.`
`.`
`  //update LCV:`
`  cin >> variable;`
`.`
`.`
`      }`
	`const int SENTINEL = -1;`
	`int input = 0;`
	`while (input!=SENTINEL)`
	`{`
	`    cin>>input;`
	`}`
	`/*output: series of input terminated by -1 */`

A ***flag value*** on the other hand is a Boolean value that would terminate a loop. A `while` that is flag-controlled would terminate if the value of the flag would be in opposite to the initial value.

Table 20: Flag-controlled while loop format and example

Format	Example
`flag = false;`	`bool flag = false;`
`while (!flag)`	`int flag_input = 1;`
`{`	`while (!flag)`
`.`	`{`
`.//update LCV here`	`    cin>>flag_input;`
`  if (logical_expression)`	`    if (flag_input<0)`
`    flag = true;`	`        flag = true;`
`.`	`}`
`.`	`/*output:  series  of`
`       }`	`input terminated by any`
	`value lesser than 0`
	`*/`

`for` loop

A `for` loop structure is another type of pretest loop that would simplify the writing of counter-controlled `while` loop. The `for` is a reserved word in C++.

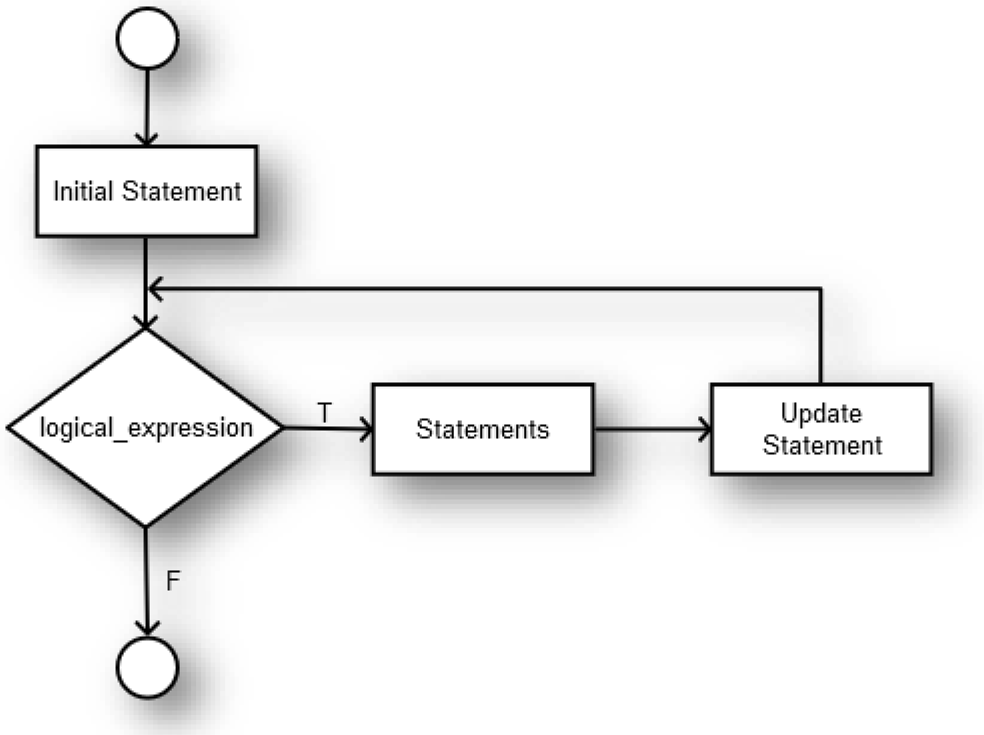

Figure 10: for loop flowchart

The initial statement, loop condition and update statement are the loop control variables. In summary the for loop operates with the following steps

1. The initial statement is read.

2. The loop condition is evaluated. If true the loop statement executes, then update statement is accomplished.
3. Step 2 is repeated until loop condition would be evaluated as false.

Table 21: Syntax and example of a for loop statement

Syntax	**Example**
`for`	`for (int i=0;i<10;i++)`
`(initialization;`	`{`
`loop_condition;`	`    if (i%2==0)`
`update`	`    {`
`statement)`	`        cout<<i<<",";`
`{`	`    }`
`  //loop body`	`}`
`}`	`//output: 0,2,4,8,`

The example would process a series of number from 0-9 and would only display the value if it is divisible by 2 (even numbers).
Some notes when using for loop

- If the loop condition is false the entire for loop statement will be skipped
- The update statement must change the LCV (initialization) so that the condition would eventually be evaluated as false.
- A semicolon before the loop body would immediately terminate the loop.
- Omitting all three statements (i.e. initialization, loop condition and update statement) would implement an infinite loop.

`do while` loop

The do – while loop is the only posttest loop in C++. The statement within the loop is executed before the logical expression is evaluated. The execution of the loop will continue as long as the logical expression is true. So, to avoid an infinite loop it is better to update the statement that would make the logical expression false.

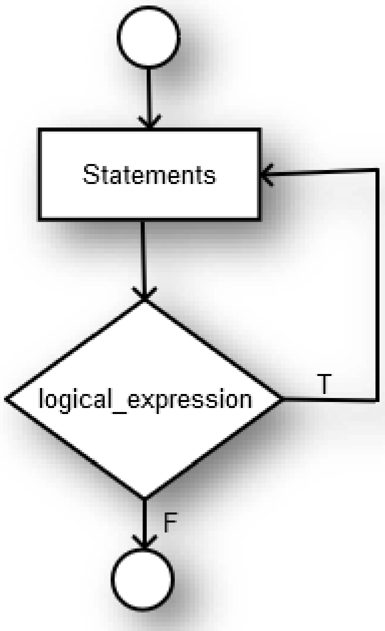

Figure 11: do-while loop flowchart

The syntax and an example is shown below

Table 22: Format and example of do-while loop statement

Syntax	**Example**

`do`
`{`
`   Statements;`
 	
`}while(logical_expression);`
	`int do_ctr = 10;`
	`do`
	`{`
	`cout<<do_ctr<<",";`
	`    do_ctr -=2;`
	`}while (do_ctr>0);`
	`//output:`
	`10,8,6,4,2`

The do_ctr value is initialized to 10. Inside the do-while loop structure the do_ctr is displayed then decremented by 2. The logical expression would then evaluated if do_ctr is greater than 0. Once the logic expression is falsified the loop would terminate.

`break` and `continue` statements

When a `break` statement is executed in a `switch` structure it would exit that particular structure. Similarly, when used in a repetition structure it would make an instant exit from the structure without the need of a LCV. On the other hand, the `continue` statement is used skip the rest of the statements on a repetition structure. In a `for` loop the update statement is executed after the `continue` statement. Conversely, a `while` and `do-while` loop the logical expression is immediately evaluated after the `continue` statement.

Table 23: Example and explanation of using the continue and break statement

Example	Explanation
`int limit = 0; while(1) { limit++; if (limit==10) break; else if (limit>6) continue; else cout<<limit<<","; } //output: 1,2,3,4,5,6`	The while statement is always true since it evaluate the true logic expression. However when the limit variable would reach 10, a break statement would be executed, but before that the loop will display the values of limit before it would exceed 6 due to the continue statement.

Nested control structures

Nesting repetition structures is done by placing an entire loop (inner loop) within another loop (outside). This may fairly be a confusing idea but nesting repetition structures are common occurrence of our daily lives. Take for example, when telling the duration of your laboratory class schedule one has to know that each hour is composed of 60 mins and laboratory session is 3 hours. So, one can conclude that in a laboratory schedule the inner loop is the minutes while the outer loop is the hours. This concept is represented by the C++ code below.

```
for (int hr=0;hr<3;hr++)
   for (int mins=0;mins<60;mins++)
          cout<<hr<<":"<<mins<<" "<<endl;

/*output:  displays  all  mins  and  hours
within a one laboratory session*/
```