COMPUTER PROGRAMMING USING C++

(PART 1) ISYS 101

Dr.Nahla AL_Zerkany

Table of Contents

Algorithm Design

An algorithm is a set of steps designed to solve a problem or accomplish a task. Algorithms are usually written in pseudocode, or a combination of your speaking language and one or more programming languages, in advance of writing a program.

Steps for Writing an Algorithm

Steps1

Determine the outcome of your code. What is the specific problem you want to solve or the task you want it to accomplish? Once you have a solid idea of what you're aiming to accomplish, you can determine the steps it will take to get there.

Steps2

Decide on a starting point. Finding your starting and ending point are crucial to listing the steps of the process. To determine a starting point, determine the answers to these questions:

- What data/inputs are available?
- Where is that data located?
- What formulas are applicable to the issue at hand?
- What are the rules to working with the available data?
- How do the data values relate to each other?

Steps3

Find the ending point of the algorithm. As with the starting point, you can find the end point of your algorithm by focusing on these questions:

- What facts will we learn from the process?
- What changes from the start to the end?
- What will be added or no longer exist?

Steps4

List the steps from start to finish. Start with broad steps. To use a real-world example, let's say your goal is to have lasagna for dinner. You've determined that the starting point is to find a recipe, and that the end result is that you'll have a lasagna fully cooked and ready to eat by 7 PM. Your steps may look something like this:

- Search for a recipe online.
- Look for the ingredients you already have in the kitchen.
- Make a list of ingredients you'll need from the store.
- Buy the missing ingredients.
- Return home.
- Prepare the lasagna.
- Remove the lasagna from the oven.

Steps 5

Determine how you will accomplish each step. Now that you have a step-by-step outline, it's time to think about how you might code each step. Which language will you use? What resources are available? What's the most efficient way to accomplish each step in that language? Incorporate some of that code into your algorithm. Expand each step until you've detailed the entire process.

For example, the first step in our lasagna algorithm is Search for a recipe online. But what is involved in this search? Be specific. For example:

Turn on your computer.

Check to make sure you're connected to the internet. Connect to the internet if you aren't already.

Open a web browser.

Enter your search terms.

Click a recipe link.

Determine whether the recipe meets your needs.

Filter out recipes that aren't vegetarian.

Make sure the recipe makes at least 5 servings.

Repeat some of these steps until you find the right recipe.

Consider the resources at your disposal, such as the capabilities of the system you're developing a program for. In the case of lasagna, we assume the person making the lasagna knows how to search the internet, operate an oven, etc.

Steps 6

Review the algorithm. Now that you've written your algorithm, it's time to evaluate the process. Your algorithm is designed to accomplish something specific, and you'll need it to start writing your program. Ask yourself the following questions, and address each as necessary:

Does the algorithm solve the problem/accomplish the task?

Does it have clearly defined inputs and outputs?

Should the end goal be redefined to be more general? More specific?

Can any of the steps be simplified?

Is the algorithm guaranteed to end with the correct result?

Examples of Algorithms in Programming

Algorithm to add two numbers entered by the user

```
Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
        sum←num1+num2
Step 5: Display sum
Step 6: Stop
```

Find the largest number among three different numbers

```
Step 1: Start
Step 2: Declare variables a,b and c.
Step 3: Read variables a,b and c.
Step 4: If a > b
           If a > c
              Display a is the largest number.
           Else
              Display c is the largest number.
        Else
           If b > c
              Display b is the largest number.
           Else
              Display c is the greatest number.
Step 5: Stop
```

Qualities of a good algorithm

- Input and output should be defined precisely.
- Each step in the algorithm should be clear and unambiguous.
- Algorithms should be most effective among many different ways to solve a problem.
- An algorithm shouldn't include computer code. Instead, the algorithm should be written in such a way that it can be used in different programming languages.

Definition of 'Pseudocode'

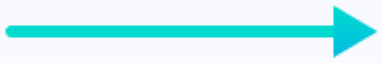
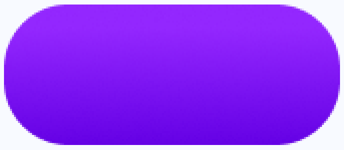
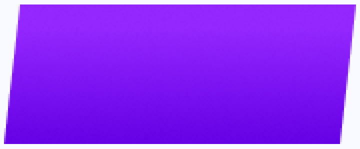
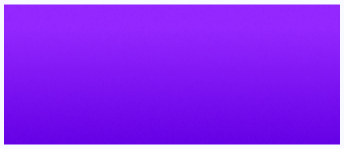
Definition: Pseudocode is an informal way of programming description that does not require any strict programming language syntax or underlying technology considerations. It is used for creating an outline or a rough draft of a program. Pseudocode summarizes a program's flow, but excludes underlying details. System designers write pseudocode to ensure that programmers understand a software project's requirements and align code accordingly.

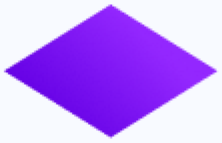
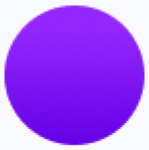
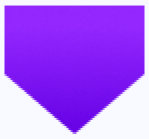
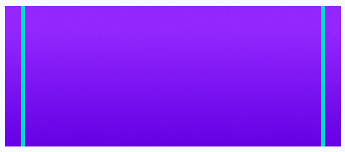
Description: Pseudocode is not an actual programming language. So it cannot be compiled into an executable program. It uses short terms or simple English language syntaxes to write code for programs before it is actually converted into a specific programming language. This is done to identify top level flow errors, and understand the programming data flows that the final program is going to use. This definitely helps save time during actual programming as conceptual errors have been already corrected. Firstly, program description and functionality is gathered and then pseudocode is used to create statements to achieve the required results for a program. Detailed pseudocode is inspected and verified by the designer's team or programmers to match design specifications. Catching errors or wrong program flow at the pseudocode stage is beneficial for development as it is less costly than catching them later. Once the pseudocode is accepted by the team, it is rewritten using the vocabulary and syntax of a programming language. The purpose of using pseudocode is an efficient key principle of an algorithm. It is used in planning an algorithm with sketching out the structure of the program before the actual coding takes place.

Flowcharts

A flowchart is a diagrammatic representation of an algorithm. A flowchart can be helpful for both writing programs and explaining the program to others.

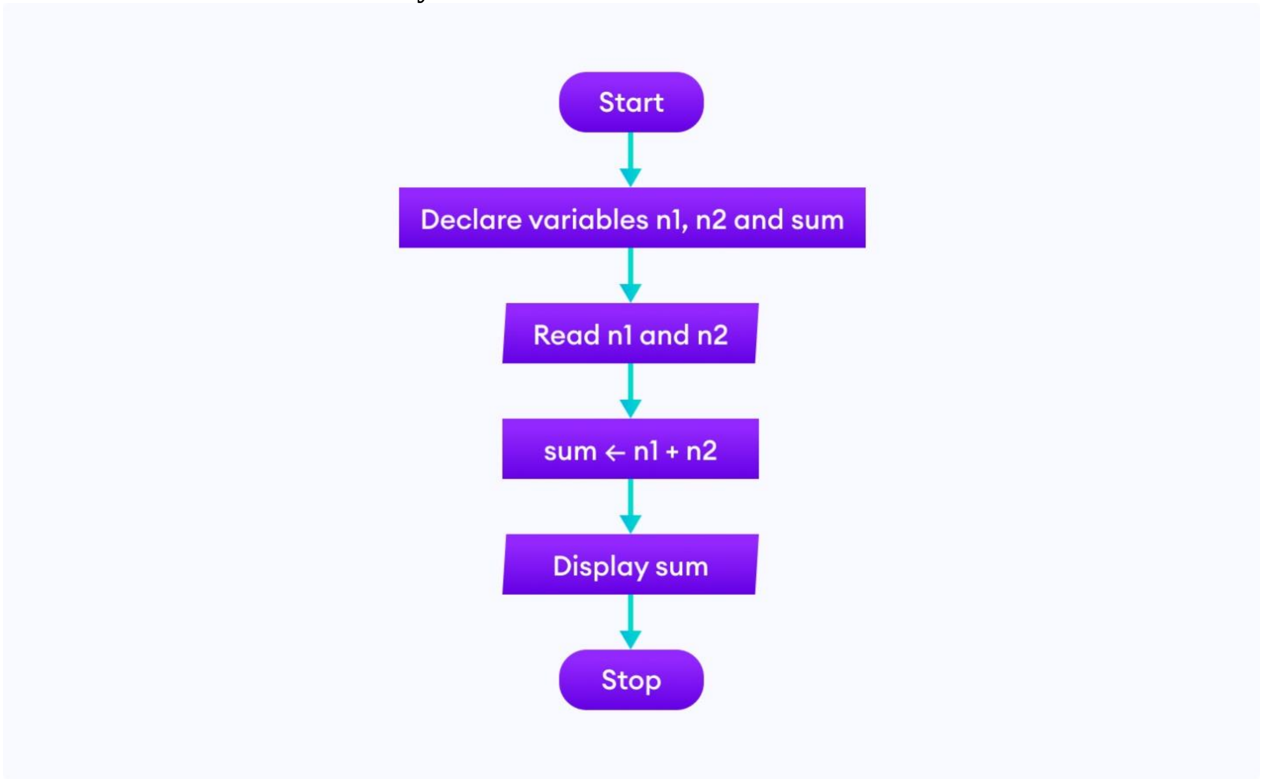
Symbols Used in Flowchart

Symbol	Purpose	Description
	Flow line	Indicates the flow of logic by connecting symbols.
	Terminal(Stop/Start)	Represents the start and the end of a flowchart.
	Input/Output	Used for input and output operation.
	Processing	Used for arithmetic operations and data-manipulations.

	Decision	Used for decision making between two or more alternatives.
	On-page Connector	Used to join different flowline
	Off-page Connector	Used to connect the flowchart portion on a different page.
	Predefined Process/Function	Represents a group of statements performing one processing task.

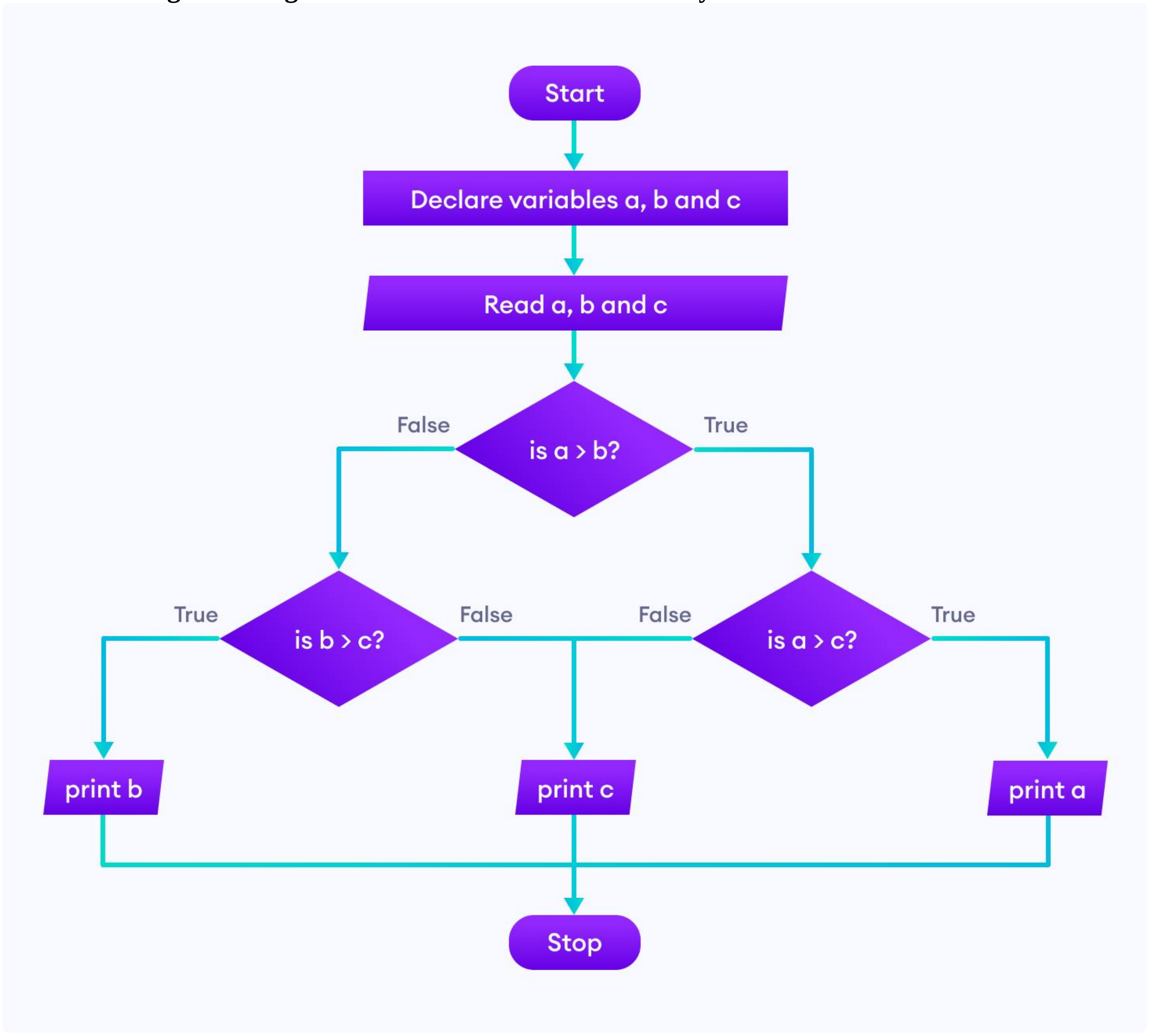
Examples of flowcharts in programming

1. Add two numbers entered by the user.



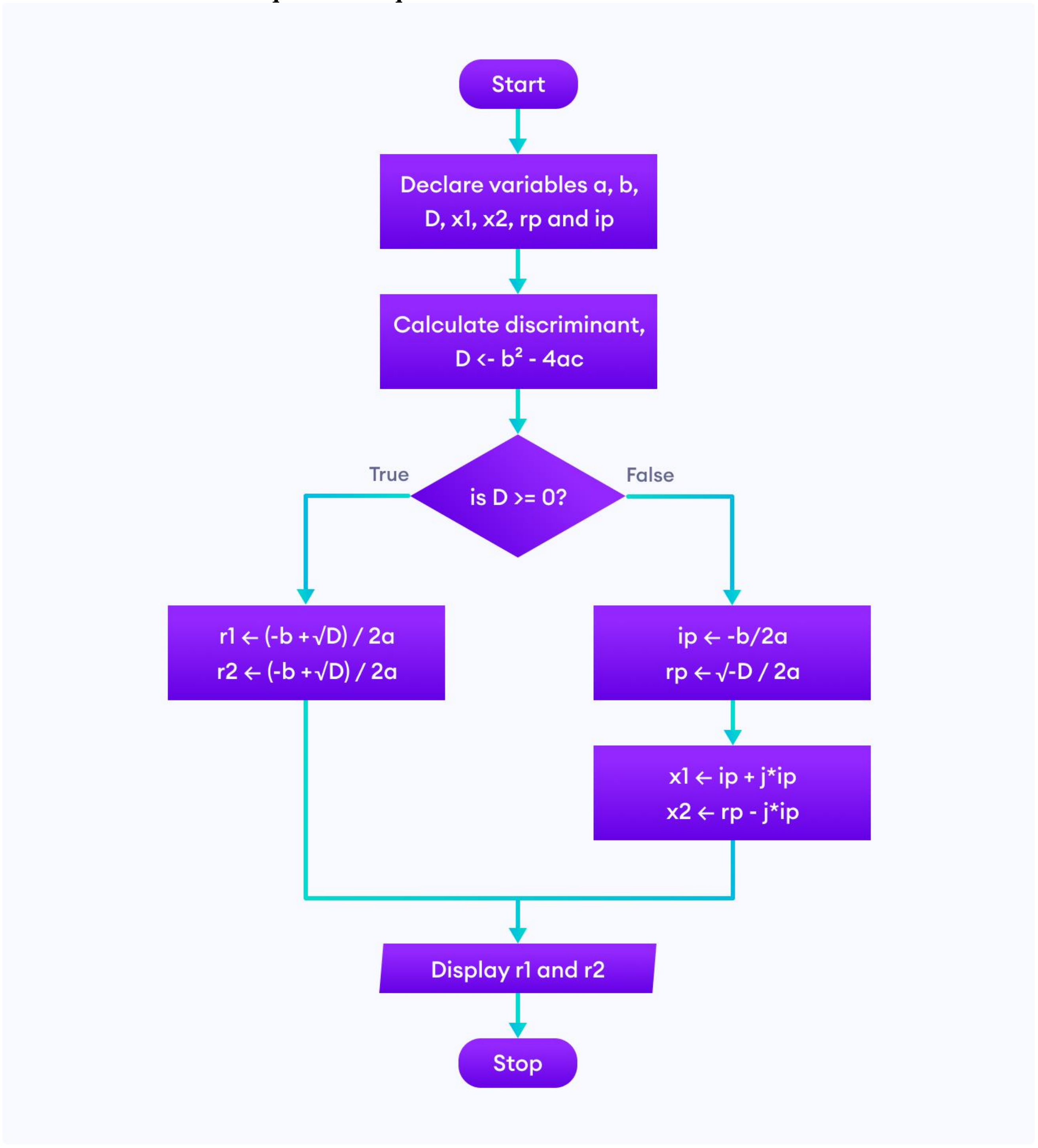
Flowchart to add two numbers

2. Find the largest among three different numbers entered by the user.



Flowchart to find the largest among three numbers.

3. Find all the roots of a quadratic equation $ax^2+bx+c=0$



Introduction to C++

C++ is a middle-level programming language developed by Bjarne Stroustrup starting in 1979 at Bell Labs. C++ runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX. This lectures adopts a simple and practical approach to describe the concepts of C++.

When we consider a C++ program, it can be defined as a collection of objects that communicate via invoking each other's methods

C++ Program Structure:

Let us look at a simple code that would print the words Hello World.

```
#include <iostream>

using namespace std;

// main() is where program execution begins.

int main()
 {
     cout << "Hello World"; // prints Hello World
     return 0;
}
```

Let us look at the various parts of the above program:

1. The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header is needed.

2. The line using namespace std; tells the compiler to use the std namespace. Namespaces are a relatively recent addition to C++.

3. The next line '// main() is where program execution begins.' is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line.

4. The line int main() is the main function where program execution begins.

5. The next line cout << "This is my first C++ program."; causes the message "This is my first C++ program" to be displayed on the screen.

6. The next line return 0; terminates main() function and causes it to return the value 0 to the calling process.

-- In C++, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity. For example, following are three different statements:

```
x = y;

 y = y+1;

add(x, y);
```

A block is a set of logically connected s

```
{

    cout << "Hello World"; // prints Hello World

    return 0;

 }
```

C++ does not recognize the end of the line as a terminator. For this reason, it does not matter where you put a statement in a line. For example:

```
x = y;

y = y+1;

add(x, y);
```

6is the same as:

```
x = y; y = y+1; add(x, y);
```

C++ Identifiers

A C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9). C++ does not allow punctuation characters such as @, $, and % within identifiers. C++ is a case-sensitive programming language. Thus, Manpower and manpower are two different identifiers in C++. Here are some examples of acceptable identifiers:

```
mohd      zara    abc    move_name   a_123   myname50    _temp j     a23b9        retVal
```

C++ Keywords :

The following list shows the reserved words in C++. These reserved words may not be used as constant or variable or any other identifier names.

Asm	else	new	this	auto
bool	explicit	throw	true	enum
case	extern	private	typedef	operator
char	float	try	typename	break
class	for	public	union	export
const	friend	typeid	unsigned	protected
const_cast	goto	reinterpret_cast	using	catch
continue	if	return	virtual	false
default	inline	short	void	register
delete	int	signed	volatile	struct
do	long	sizeof	wchar_t	switch
double	mutable	static	dynamic_cast	template
while	namespace	static_cast		

Whitespace in C++

A line containing only whitespace, possibly with a comment, is known as a blank line, and C++ compiler totally ignores it. Whitespace is the term used in C++ to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins. Statement 1:

```
int age;
```

In the above statement there must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them. Statement 2:

```
fruit = apples + oranges;      //     Get the total fruit
```

In the above statement 2, no whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish for readability purpose.

COMMENTS IN C++

Program comments are explanatory statements that you can include in the C++ code. These comments help anyone reading the source code. All programming languages allow for some form of comments. C++ supports single-line and multi-line comments. All characters available inside any comment are ignored by C++ compiler. C++ comments start with /* and end with */. For example:

```
/* This is a comment */

/* C++ comments can also

 * span multiple lines

*/
```

A comment can also start with //, extending to the end of the line. For example:

```
 #include <iostream>

using namespace std;

int main()

  {

     cout << "Hello World"; // prints Hello World

     return 0;

    }
```

When the above code is compiled, it will ignore **// prints Hello World** and final executable will produce the following result:

```
Hello World
```

Within a /* and */ comment, // characters have no special meaning. Within a // comment, /* and */ have no special meaning. Thus, you can "nest" one kind of comment within the other kind. For example:

```
/* Comment out printing of Hello World:

   cout << "Hello World"; // prints Hello World

 */
```

DATA TYPES

While writing program in any language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory. You may like to store information of various data types like character, wide character, integer, floating point, double floating point, boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Primitive Built-in Types C++ offers the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types:

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_

Several of the basic types can be modified using one or more of these type modifiers:

• signed • unsigned • short • long

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables

Type	Typical Bit Width	Typical Range
Char	1byte	-127 to 127 or 0
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	Range	0 to 65,535
signed short int	Range	-32768 to 32767
long int	4bytes	-2,147,483,647 to 2,147,483,647
signed long int	4bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	2 or 4 bytes	+/- 1.7e +/- 308 (~15 digits)

The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

Following is the example, which will produce correct size of various data types on your computer.

```
#include<iostream>

using namespace std;

int main()

{

   cout << "Size of char : " << sizeof(char) << endl;

    cout << "Size of int : " << sizeof(int) << endl;

   cout << "Size of short int : " << sizeof(short int) << endl;

    cout << "Size of long int : " << sizeof(long int) << endl;

    cout << "Size of float : " << sizeof(float) << endl;

    cout << "Size of double : " << sizeof(double) << endl;

   cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;

    return 0;

}
```

This example uses endl, which inserts a new-line character after every line and << operator is being used to pass multiple values out to the screen. We are also using sizeof() function to get size of various data types.

When the above code is compiled and executed, it produces the following result which can vary from machine to machine:

```
 Size of char : 1

Size of int : 4

Size of short int : 2

Size of long int : 4

Size of float : 4

Size of double : 8

Size of wchar_t : 4
```

VARIABLE TYPES

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable. The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C++ is case-sensitive: There are following basic types of variable in C++ :

Type	Description
bool	Stores either value true or false.
char	Typically a single octet (one byte). This is an teger type.
int	The most natural size of integer for the machine.
Float	A single-precision floating point value.
double	A double-precision floating point value.
void	Represents the absence of type.

C++ also allows to define various other types of variables like Enumeration, Pointer, Array, Reference, Data structures, and Classes. Following section will cover how to define, declare and use various types of variables.

Variable Definition in C++

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type, and contains a list of one or more variables of that type as follows:

```
type variable_list;
```

Here, type must be a valid C++ data type including char, w_char, int, float, double, bool or any user-defined object, etc., and variable_list may consist of one or more identifier names separated by commas. Some valid declarations are shown here:

```
int i, j, k;

char c, ch;

float f, salary;

double d;
```

The line int i, j, k; both declares and defines the variables i, j and k; which instructs the compiler to create variables named i, j and k of type int. Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows:

```
type variable_name = value;
```

Some examples are:

```
 int d = 3, f = 5; // declaration of d and f.

int d = 3, f = 5; // definition and initializing d and f.

byte z = 22; // definition and initializes z.

 char x = 'x'; // the variable x has the value 'x'.
```

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables is undefined. Variable Declaration in C++ A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable declaration has its meaning at the time of compilation only, compiler needs actual variable declaration at the time of linking of the program.

A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program. You will use extern keyword to declare a variable at any place. Though you can declare a variable multiple times in your C++ program, but it can be defined only once in a file, a function or a block of code.

Example:

Try the following example where a variable has been declared at the top, but it has been defined inside the main function:

```
 #include <iostream>
 using namespace std;
int main ()
 {
       // Variable definition:
      int a, b;
      int c;
      float f;
```

```
           // actual initialization
          a = 10;
           b = 20;
           c = a + b;
          cout << c << endl ;
          f = 70.0/3.0;
         cout << f << endl ;
         return 0;
    }
```

When the above code is compiled and executed, it produces the following result:

```
  30
 23.3333
```

CONSTANTS/LITERALS

Constants refer to fixed values that the program may not alter and they are called literals. Constants can be of any of the basic data types and can be divided into Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values. Again, constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal. An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively.

The suffix can be uppercase or lowercase and can be in any order. Here are some examples of integer literals:

```
212           // Legal

215u          // Legal

0xFeeL        // Legal

078           // Illegal: 8 is not an octal digit

032UU         // Illegal: cannot repeat a suffix
```

Following are other examples of various types of Integer literals:

```
85              // decimal
0213          // octal
0x4b           // hexadecimal
30            //int
30u          // unsigned int
30l         // long
30ul          // unsigned long
```

__Floating-point Literals__

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals:

```
 3.14159              // Legal
314159E-5L           // Legal
510E                 // Illegal: incomplete exponent
 210f               // Illegal: no decimal or exponent
.e55               // Illegal: missing integer or fraction
```

Boolean Literals There are two Boolean literals and they are part of standard C++ keywords:

- A value of true representing true.
- A value of false representing false.

You should not consider the value of true equal to 1 and value of false equal to 0.

Character Literals

Character literals are enclosed in single quotes. If the literal begins with L (uppercase only), it is a wide character literal (e.g., L'x') and should be stored in wchar_t type of variable. Otherwise, it is a narrow character literal (e.g., 'x') and can be stored in a simple variable of char type.

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C++ when they are preceded by a backslash they will have special meaning and they are used to represent like newline (\n) or tab (\t). Here, you have a list of some of such escape sequence codes:

Escape sequence	**Meaning**
\\\\	\\ character
\\'	' character
\\"	" character
\\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh . . .	Hexadecimal number of one or more digits

Following is the example to show a few escape sequence characte

```
 #include <iostream>

using namespace std;

int main()

{

        cout << "Hello\tWorld\n\n";

        return 0;

}
```

When the above code is compiled and executed, it produces the following result:

```
Hello   World
```

String Literals

String literals are enclosed in double quotes. A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separate them using whitespaces. Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"
```

Defining Constants

There are two simple ways in C++ to define constants:

- Using #define preprocessor.
- Using const keyword.

The #define Preprocessor Following is the form to use #define preprocessor to define a constant:

```
#define identifier value
```

Following example explains it in detail:

```
#include <iostream>

 using namespace std;

 #define LENGTH 10

 #define WIDTH 5

#define NEWLINE '\n'

int main()

 {

     int area;

     area = LENGTH * WIDTH;

     cout << area;

     cout << NEWLINE;

     return 0;

}
```

When the above code is compiled and executed, it produces the following result:

```
50
```

The const Keyword

You can use const prefix to declare constants with a specific type as follows:

```
const type variable = value;
```

Following example explains it in detail:

```
#include <iostream>

 using namespace std;

int main()

 {

      const int LENGTH = 10;

      const int WIDTH = 5;

      const char NEWLINE = '\n';

      int area;

      area = LENGTH * WIDTH;

      cout << area;

      cout << NEWLINE;

      return 0;

}
```

When the above code is compiled and executed, it produces the following result:

```
50
```

Note that it is a good programming practice to define constants in CAPITALS.

BASIC INPUT/OUTPUT

The C++ standard libraries provide an extensive set of input/output capabilities.C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called input operation and if bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc., this is called output operation.

I/O Library Header Files

There are following header files important to C++ programs:

Header File	Function and Description
<iostream>	This file defines the cin, cout, cerr and clog objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.
<iomanip>	This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as setw and setprecision.
<fstream>	This file declares services for user-controlled file processing.

The Standard Output Stream (cout)

The predefined object cout is an instance of ostream class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The cout is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

```
#include<iostream>

using namespace std;

 int main( )

  {

      char str[] = "Hello C++";

      cout << "Value of str is : " << str << endl;

       return 0;

   }
```

When the above code is compiled and executed, it produces the following result:

```
Value of str is : Hello C++
```

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values. The insertion operator << may be used more than once in a single statement as shown above and endl is used to add a new-line at the end of the line.

The Standard Input Stream (cin)

The predefined object cin is an instance of istream class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The cin is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

```
#include<iostream>

 using namespace std;

int main( )

 {

    char name[50];

    cout << "Please enter your name: ";

    cin >> name;

    cout << "Your name is: " << name << endl;

    return 0;

 }
```

When the above code is compiled and executed, it will prompt you to enter a name. You enter a value and then hit enter to see the following result:

```
Please enter your name: cplusplus

Your name is: cplusplus
```

The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables. The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following:

```
cin >> name >> age;
```

This will be equivalent to the following two statements:

```
cin >> name;

cin >> age;
```

OPERATORS

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide the following types of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators

There are following arithmetic operators supported by C++ language: Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by denumerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

Try the following example to understand all the arithmetic operators available in C++.

```
#include  <iostream>

using namespace std;

int main()

{

 int a = 21;

int b = 10;

int c ;

c = a + b;

cout << "Line 1 - Value of c is :" << c << endl ;

 c = a - b;

cout << "Line 2 - Value of c is :" << c << endl ;

c = a * b;

cout << "Line 3 - Value of c is :" << c << endl ;

c = a / b;

cout << "Line 4 - Value of c is :" << c << endl ;

c = a % b; cout << "Line 5 - Value of c is :" << c << endl ;

c = a++; cout << "Line 6 - Value of c is :" << c << endl ;

c = a--;

cout << "Line 7 - Value of c is :" << c << endl ;

return 0;

}
```

When the above code is compiled and executed, it produces the following result:

```
Line 1 - Value of c is :31

Line 2 - Value of c is :11

Line 3 - Value of c is :210

Line 4 - Value of c is :2

Line 5 - Value of c is :1

Line 6 - Value of c is :21

Line 7 - Value of c is :22
```

<u>Relational Operators</u>

There are following relational operators supported by C++ language Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not,if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true	(A >= B) is not true
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then the condition becomes true	(A <= B) is true.

Try the following example to understand all the relational operators available in C++.

```
#include<iostream>
 using namespace std;
int main()
{
      int a = 21;
     int b = 10;
     int c ;
     if( a == b )
      {
      cout << "Line 1 - a is equal to b" << endl ;
     }
     else
     {
      cout << "Line 1 - a is not equal to b" << endl ;
      }
     if ( a < b )
      {
     cout << "Line 2 - a is less than b" << endl ;
     }
     else
     {
      cout << "Line 2 - a is not less than b" << endl ;
      }
     if ( a > b )
     {
     cout << "Line 3 - a is greater than b" << endl ;
     }
    else
    {
    cout << "Line 3 - a is not greater than b" << endl ;
     }
     /* Let's change the values of a and b */
     a = 5;
    b = 20;
    if ( a <= b )
     {
    cout << "Line 4 - a is either less than \ or equal to b" << endl ;
    }
     if ( b >= a )
     {
    cout << "Line 5 - b is either greater than \ or equal to b" << endl ;
     }
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Line 1 - a is not equal to b

Line 2 - a is not less than b

Line 3 - a is greater than b

Line 4 - a is either less than or equal to b

Line 5 - b is either greater than or equal to b
```

Logical Operators

There are following logical operators supported by C++ language. Assume variable A holds 1 and variable B holds 0, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is nonzero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A && B) is true.

Try the following example to understand all the logical operators available in C++.

```
#include<iostream>

 using namespace std;

main()

{

int a = 5;

int b = 20;

int c ;

if ( a && b )

 {

cout << "Line 1 - Condition is true"<< endl ;

}

 if ( a || b )

 {

cout << "Line 2 - Condition is true"<< endl ;

 }

 /* Let's change the values of a and b */

a = 0;

b = 10;

if ( a && b )

 {

 cout << "Line 3 - Condition is true"<< endl ;

 }

else
{
cout << "Line 4 - Condition is not true"<< endl ;
 }
  if ( !(a && b) )
  { cout << "Line 5 - Condition is true"<< endl ;
  }
 return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
 Line 1 - Condition is true
Line 2 - Condition is true
Line 4 - Condition is not true
Line 5 - Condition is true
```

Assignment Operators

There are following assignment operators supported by C++ language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C +
-=	Subtract AND assignment operator , It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C *
A /=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C /
A %=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A

Try the following example to understand all the assignment operators available in C++.

```
#include <iostream>

using namespace std;

int  main()

{

        int a = 21;

        int c ;

        c = a;

         cout << "Line 1 - = Operator, Value of c = : " <<c<< endl ;

        c += a;

        cout << "Line 2 - += Operator, Value of c = : " <<c<< endl ;

        c -= a;

        cout << "Line 3 - -= Operator, Value of c = : " <<c<< endl ;

         c *= a;

         cout << "Line 4 - *= Operator, Value of c = : " <<c<< endl ;

         c /= a;

        cout << "Line 5 - /= Operator, Value of c = : " <<c<< endl ;

         c = 200;

        c %= a; cout << "Line 6 - %= Operator, Value of c = : " <<c<< endl ;

        return 0;

 }
```

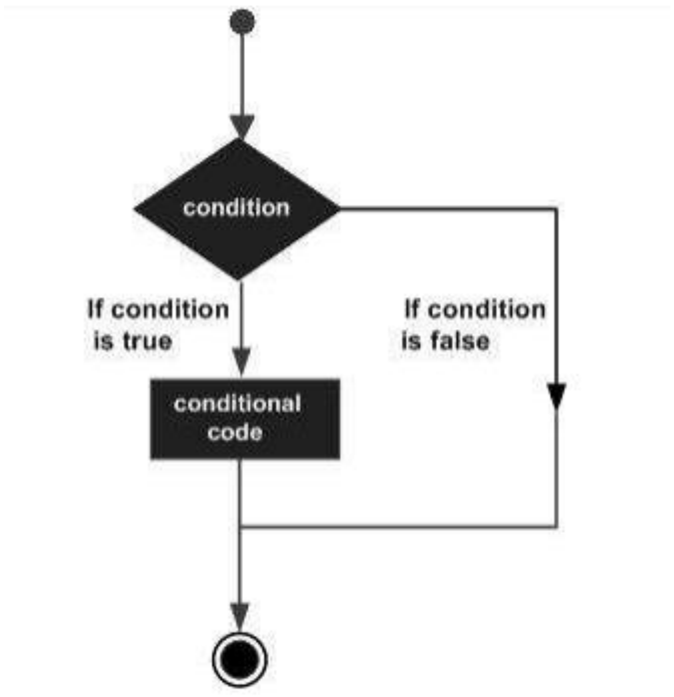
When the above code is compiled and executed, it produces the following result:

```
 Line 1 - = Operator, Value of c = : 21

Line 2 - += Operator, Value of c = : 42

Line 3 - -= Operator, Value of c = : 21

Line 4 - *= Operator, Value of c = : 441

Line 5 - /= Operator, Value of c = : 21

Line 6 - %= Operator, Value of c = : 11
```

DECISION-MAKING STATEMENTS

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general from of a typical decision making structure found in most of the programming languages:



C++ programming language provides following types of decision making statements.

Statement	Desecription
if statement	An 'if' statement consists of a boolean expression followed by one or more statements.
if...else statement	An 'if' statement can be followed by an optional 'else' statement, which executes when the boolean expression is false.
switch statement	A 'switch' statement allows a variable to be tested for equality against a list of values.
nested if statements	You can use one 'if' or 'else if' statement inside another 'if' or 'else if' statement(s).

If Statement

An if statement consists of a boolean expression followed by one or more statements.

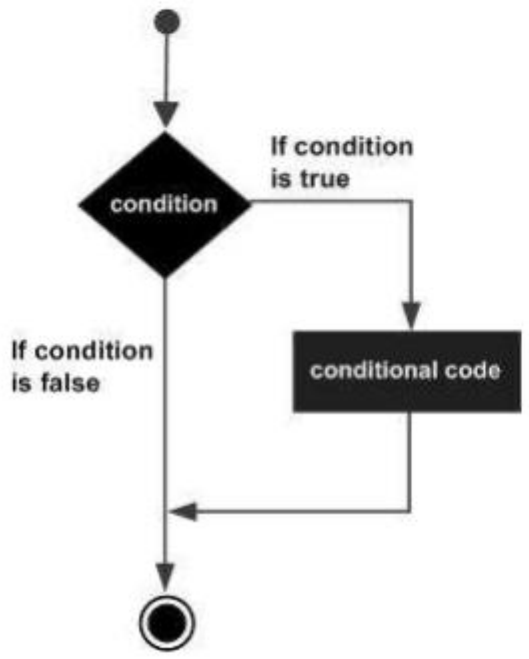
Syntax

The syntax of an if statement in C++ is:

```
if(boolean_expression)
{
 // statement(s) will execute if the boolean expression is true
}
```

If the boolean expression evaluates to true, then the block of code inside the if statement will be executed. If boolean expression evaluates to false, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Flow Diagram



Example

```
 #include<iostream>
 using namespace std;
int main ()
 {
      // local variable declaration:
      int a = 10;
       // check the boolean condition
      if( a < 20 )
      {
       // if condition is true then print the following
      cout << "a is less than 20;" << endl;
       }
      cout << "value of a is : " << a << endl;
```

```
        return 0;

    }
```

When the above code is compiled and executed, it produces the following result:

```
a is less than 20;

value of a is : 10
```

if…else Statement

An if statement can be followed by an optional else statement, which executes when the boolean expression is false.

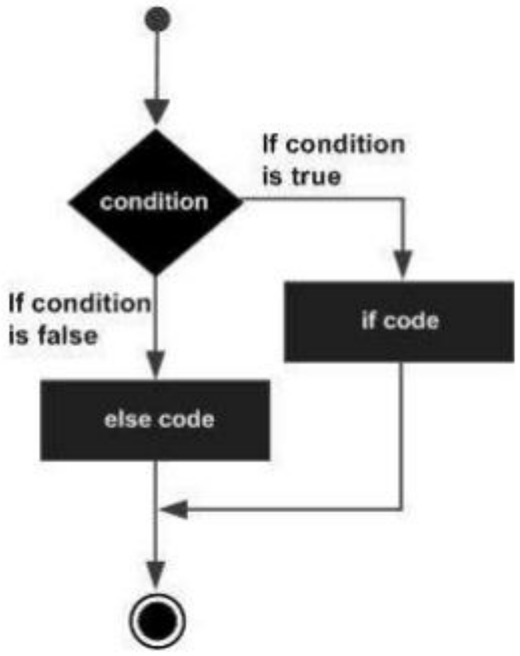
Syntax

The syntax of an if...else statement in C++ is:

```
if(boolean_expression)

{

 // statement(s) will execute if the boolean expression is true

 }

else

{

 // statement(s) will execute if the boolean expression is false

}
```

If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

Flow Diagram



Example

```
 #include<iostream>
 using namespace std;
int main ()
{
       // local variable declaration:
       int a = 100;
       // check the boolean condition
       if( a < 20 )
{
```

```
          // if condition is true then print the following
        cout << "a is less than 20;" << endl;
         }
         else
        { // if condition is false then print the following
        cout << "a is not less than 20;" << endl;
         }
        cout << "value of a is : " << a << endl;
        return 0;
     }
```

When the above code is compiled and executed, it produces the following result:

```
a is not less than 20;
value of a is : 100
```

if...else if...else Statement

An if statement can be followed by an optional else if...else statement, which is very usefull to test various conditions using single if...else if statement. When using if , else if , else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of he remaining else if's or else's will be tested.

Syntax

The syntax of an if...else if...else statement in C++ is:

```
if(boolean_expression 1)

 {

// Executes when the boolean expression 1 is true

 }

 else if( boolean_expression 2)

 {

// Executes when the boolean expression 2 is true

}

 else if( boolean_expression 3)

 {

// Executes when the boolean expression 3 is true

 }

 else

{

 // executes when the none of the above condition is true.

}
```

Example

```
#include<iostream>
 using namespace std;
int main ()
{
 // local variable declaration:
int a = 100;
// check the boolean condition if( a == 10 )
{
// if condition is true then print the following
cout << "Value of a is 10" << endl;
 }
 else if( a == 20 )
{
 // if else if condition is true
cout << "Value of a is 20" << endl;
}
else if( a == 30 )
 {
// if else if condition is true
 cout << "Value of a is 30" << endl;
}
 else {
 // if none of the conditions is true
cout << "Value of a is not matching" << endl;
 }
 cout << "Exact value of a is : " << a << endl;
  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of a is not matching
 Exact value of a is : 100
```

<u>Switch Statement</u>

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

<u>Syntax</u>

The syntax for a switch statement in C++ is as follows:

```
switch(expression)
{
 case constant-expression :
statement(s);
 break; //optional
case constant-expression :
 statement(s);
break; //optional
// you can have any number of case statements.
default : //Optional
statement(s);
}
```

The following rules apply to a switch statement:

• The expression used in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.

• You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

• The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.

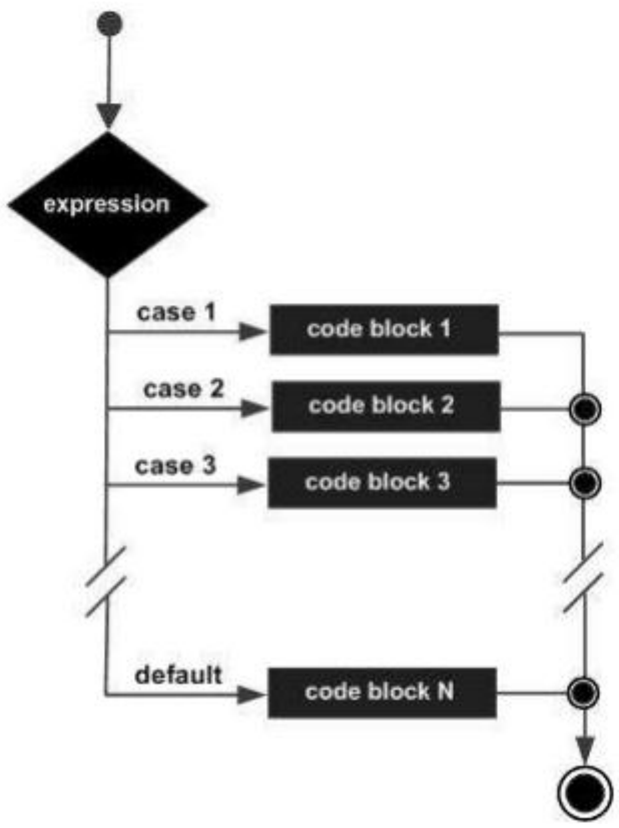
• When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.

• When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

• Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.

• A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Flow Diagram



Example

```
#include<iostream>

 using namespace std;

int main ()

{
```

```
// local variable declaration:
 char grade = 'D';
switch(grade)
 {
       case 'A' :
       cout << "Excellent!" << endl;
      break;
       case 'B' :
      case 'C' :
      cout << "Well done" << endl;
       break;
      case 'D' :
      cout << "You passed" << endl;
       break;
      case 'F' :
       cout << "Better try again" << endl;
       break;
      default :
       cout << "Invalid grade" << endl;
      }
      cout << "Your grade is " << grade << endl;
       return 0;
       }
```

This would produce the following result:

```
You passed

Your  grade is D
```

Nested if Statement

It is always legal to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

Syntax

The syntax for a nested if statement is as follows:

```
 if( boolean_expression 1)

 {

// Executes when the boolean expression 1 is true

if(boolean_expression 2)

{

 // Executes when the boolean expression 2 is true

 }

 }
```

You can nest else if...else in the similar way as you have nested if statement.

Example

```
     #include<iostream>
     using namespace std;
    int main ()
     {
    // local variable declaration:
    int a = 100;
     int b = 200;
    // check the boolean condition
    if( a == 100 )
     {
    // if condition is true then check the following
    if( b == 200 )
     {
// if condition is true then print the following
 cout << "Value of a is 100 and b is 200" << endl;
 }
}
 cout << "Exact value of a is : " << a << endl;
cout << "Exact value of b is : " << b << endl;
 return 0;
    }
```

When the above code is compiled and executed, it produces the following result:

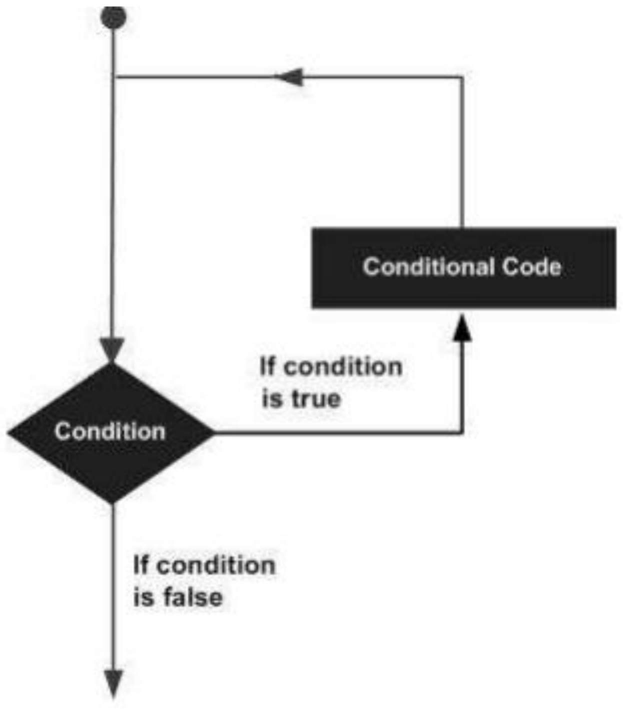
```
Value of a is 100 and b is 200

Exact value of a is : 100

Exact value of b is : 200
```

LOOP TYPES

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:



C++ programming language provides the following type of loops to handle looping requirements.

Loop Type	Description

while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while	loop Like a 'while' statement, except that it tests the condition at the end of the loop body.
nested loops	You can use one or more loop inside any another 'while', 'for' or 'do..while' loop.

for Loop

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The syntax of a for loop in C++ is:

```
for ( init; condition; increment )
 {
 statement(s);
 }
```

Here is the flow of control in a for loop:

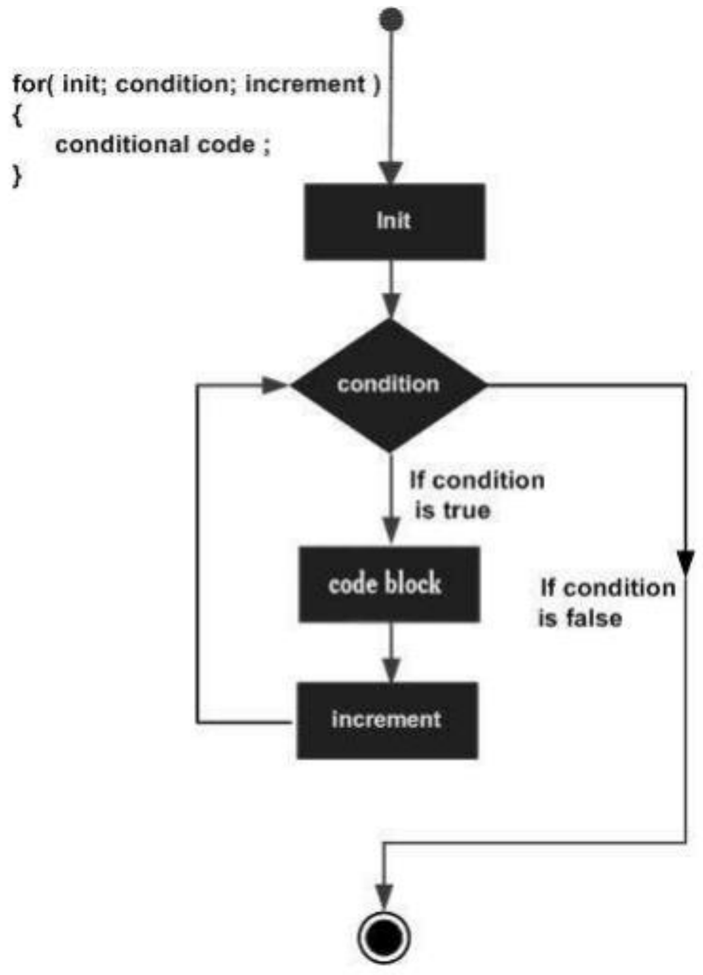
1. The init step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

2. Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.

3. After the body of the for loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram



Example

```
 #include <iostream>

 using namespace std;

int main ()

 {

 // for loop execution

for( int a = 10; a < 20; a = a + 1 )

{

 cout << "value of a: " << a << endl;

 }

return 0;

 }
```

When the above code is compiled and executed, it produces the following result:

```
 value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

 value of a: 16

value of a: 17

value of a: 18

value of a: 19
```

FUNCTIONS

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is main(), and all the most trivial programs can define additional functions. You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function. The C++ standard library provides numerous built-in functions that your program can call. For example, function strcat() to concatenate two strings, function memcpy() to copy one memory location to another location, and many more functions.

A function is known with various names like a method or a sub-routine or a procedure etc.

Defining a Function

The general form of a C++ function definition is as follows:

```
return_type function_name( parameter list )
  {
     body of the function
  }
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function:

• Return Type: A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

• Function Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.

• Parameters: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

• Function Body: The function body contains a collection of statements that define what the function does.

Example:

Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two:

```
// function returning the max between two numbers
int max(int num1, int num2)
  {
    // local variable declaration
    int result;
    if (num1 > num2)
    result = num1;
    else result = num2;
    return result;
  }
```

Function Declarations

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately. A function declaration has the following parts:

```
return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration:

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

```
#include <iostream>
using namespace std;
// function declaration
int max(int num1, int num2);
int main ()
```

```
  {
      // local variable declaration:
      int a = 100;
      int b = 200;
      int ret;
      // calling a function to get max value.
      ret = max(a, b);
      cout << "Max value is : " << ret << endl;
      return 0;
  }
// function returning the max between two numbers
int max(int num1, int num2)
   {
       // local variable declaration
       int result;
       if (num1 > num2)
            result = num1;
       else
            result = num2;
       return result;
   }
```

I kept max() function along with main() function and compiled the source code.

While running final executable, it would produce the following result:

```
Max value is : 200
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit. While calling a function, there are two ways that arguments can be passed to a function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by reference	This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument usedin the call. This means that changes made to the parameter affect the argument.

Call by Value

The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function swap() definition as follows.

```
// function definition to swap the values.
void swap(int x, int y)
 {
   int temp;
   temp = x;
   /* save the value of x */
   x = y;
   /* put y into x */
   y = temp;
   /* put x into y */
   return;
  }
```

Now, let us call the function swap() by passing actual values as in the following example

```
#include <iostream>
using namespace std;
 // function declaration
void swap(int x, int y);
int main ()
  {
   // local variable declaration:
     int a = 100;
     int b = 200;
     cout << "Before swap, value of a :" << a << endl;
     cout << "Before swap, value of b :" << b << endl;
     // calling a function to swap the values.
     swap(a, b);
     cout << "After swap, value of a :" << a << endl;
```

```
    cout << "After swap, value of b :" << b << endl;
    return 0;
  }
```

When the above code is put together in a file, compiled and executed, it produces the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

Which shows that there is no change in the values though they had been changed inside the function.

<u>**Call by Reference**</u>

The call by reference method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

```
// function definition to swap the values.
void swap(int &x, int &y)
 {
   Int temp;
   temp = x;
   /* save the value at address x */
   x = y;
   /* put y into x */
   y = temp;
   /* put x into y */
   return;
  }
```

For now, let us call the function swap() by passing values by reference as in the following example:

```
#include<iostream>
 using namespace std;
 // function declaration
 void swap(int &x, int &y);
int main ()
  {
     // local variable declaration:
    int a = 100;
    int b = 200;
    cout << "Before swap, value of a :" << a << endl;
    cout << "Before swap, value of b :" << b << endl;
    /* calling a function to swap the values using variable reference.*/
    swap(a, b);
    cout << "After swap, value of a :" << a << endl;
    cout << "After swap, value of b :" << b << endl;
    return 0;
  }
```

When the above code is put together in a file, compiled and executed, it produces the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

Default Values for Parameters

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example:

```
#include <iostream>
using namespace std;
 int sum(int a, int b=20)
   {
      int result;
      result = a + b;
      return (result);
    }
int main ()
   {
```

```
   // local variable declaration:
   int a = 100;
   int b = 200;
   int result;
   // calling a function to add the values.
   result = sum(a, b);
   cout << "Total value is :" << result << endl;
   // calling a function again as follows.
   result = sum(a);
   cout << "Total value is :" << result << endl;
   return 0;
 }
```

When the above code is compiled and executed, it produces the following result:

```
Total value is :300
 Total value is :120
```

Reference

1. Fundamentals of Programming C++, Richard L. Halterman, school of Computing Southern Adventist University, December 2, 2018.
2. A first book of c++ by Gary Bronson, 4th edition, 2012 by Gary Bronson
3. Problem solving with c++ by Walter Savitch, 7th edition,2009.
4. C++: The Complete Reference by Herbert Schildt, 4th edition, 2003.