Unit –II

Control Structures

Control structures allows to control the flow of program's execution based on certain conditions C++ supports following basic control structures:

1) Selection Control structure

2) Loop Control structure

1) Selection Control structure:

Selection Control structures allows to control the flow of program's execution depending upon the state of a particular condition being true or false .C++ supports two types of selection statements :if and switch. Condition operator (?:) can also be used as an alternative to the if statement.

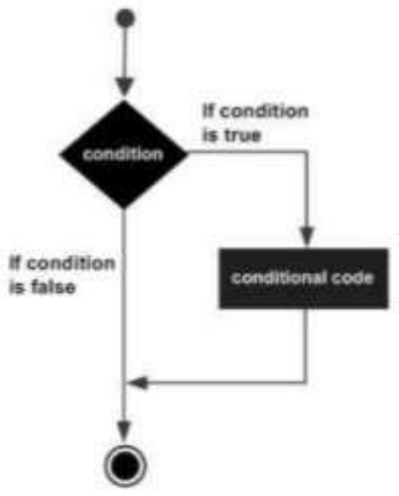
If Statement:

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

```
if(condition)

{

// statement(s) will execute if the condition is true

}
```

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

Flowchart showing working of if statement



```
// A Program to find whether a given number is even or odd using if … else

statement#include<iostream.h>

#include<conio.h>

int main()

{  int n;

cout<<"enter

number";cin>>n;

if(n%2==0)

cout<<"Even number";

else

cout<<"Odd number";

return 0;

}
```

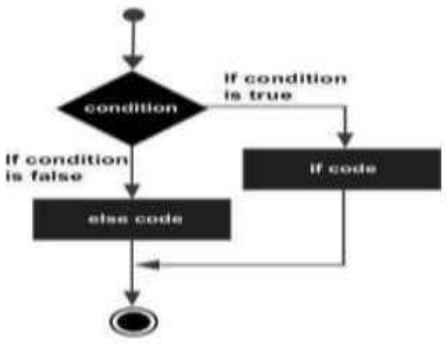
The if...else Statement

The syntax is shown as:

```
if(condition){

// statement(s) will execute if the condition is true

}

else{

// statement(s) will execute if condition is false

}
```

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

Flowchart



if...else if...else Statement

An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.

The Syntax is shown as:

```
if(condition 1){

// Executes when the condition 1 is true

}

else if(condition 2){

// Executes when the condition 2 is true

}

else if(condition 3){

// Executes when the condition 3 is true

}

else {

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

}
```

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).

The syntax for a nested if statement is as follows:

```
if( condition 1){

// Executes when the condition 1 is true

 if(condition 2){

// Executes when the condition 2 is true

          }

}
```

B) Switch

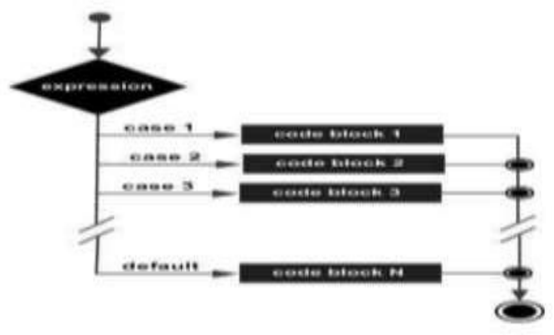
C++ has a built-in multiple-branch selection statement, called switch, which successively tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed. The general form of the switch statement is:

```
switch (expression) {
case constant1:
statement sequence
break;
case constant2:
statement sequence
break;
case constant3:
statement sequence
break;
.
.
default
statement sequence
}
```

The expression must evaluate to a character or integer value. Floating-point expressions, for example, are not allowed. The value of expression is tested, in order, against the values of the constants specified in the case statements. When a match is found, the statement sequence associated with that case is executed until the break statement or the end of the switch statement is reached. The default statement is executed if no matches are found. The default is optional and, if it is not present, no action takes place if all matches fail.

The break statement is one of C++'s jump statements. You can use it in loops as well as in the switch statement. When break is encountered in a switch, program execution "jumps" to the line of code following the switch statement.

Flowchart



2) Loop control structures

A loop statement allows us to execute a statement or group of statements multiple times. Loops or iterative statements tell the program to repeat a fragment of code several times or as long as a certain condition holds. C++ provides three convenient iterative statements: while, for, and do-while.

while loop

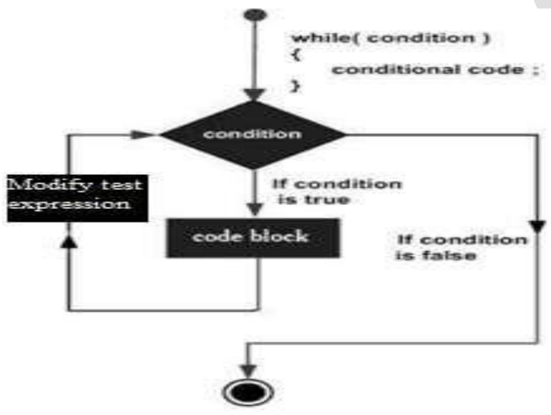
A while loop statement repeatedly executes a target statement as long as a given condition is true. It is an entry-controlled loop.

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

```
while(condition){

statement(s);

}
```

Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any non-zero value. The loop iterates while the condition is true. After each execution of the loop, the value of test expression is changed. When the condition becomes false, program control passes to the line immediately following the loop.

Flowchart



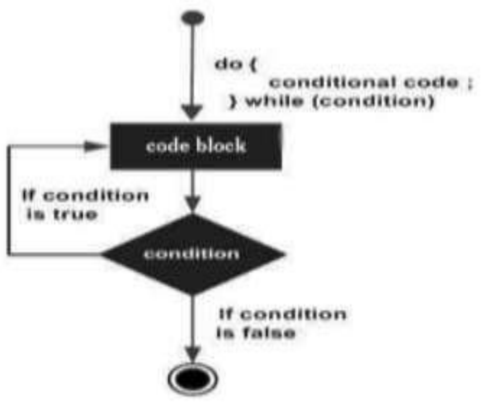
```
// A program to display numbers from 1 to

100#include<iostream.h>


#include<conio.h>

int main(){


  int i=1;

while(i<=100){

cout<<i ;


  i++;

}

return 0;

}
```

The do-while Loop

The do-while loop differs from the while loop in that the condition is tested after the body of the loop. This assures that the program goes through the iteration at least once. It is an exit-controlled loop.

```
do{

statement(s);

}while( condition );
```

The conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested. If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.



```
// A program to display numbers from 1 to

100#include<iostream.h>

#include<conio.h>

int main( ){

 int i=1;

do

{           cout<<i ;

            i++;

} while(i<=100);

return 0;

}
```

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. 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:

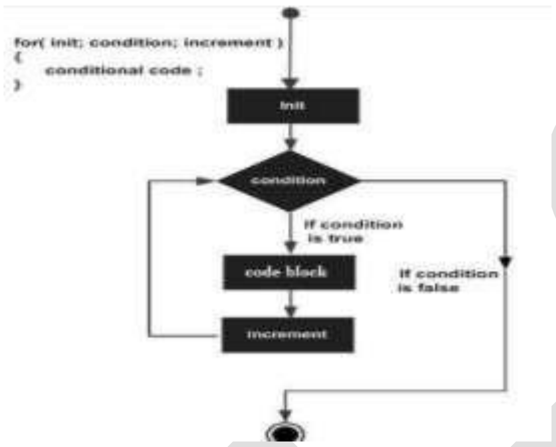
1. The init step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

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 C++ 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



```
// A program to display numbers from 1 to 100

#include<iostream.h>

#include<conio.h>

int main() {

 int i ;

for (i=1;i<=100;i++)

{

cout<<i ;

return 0;

}
```

C++ Functions

A function groups a number of program statements into a unit and gives it a name. This unit can then be invoked from other parts of the program. The function's code is stored in only one place in memory, even though the function is executed many times in the course of the program's execution. Functions help to reduce the program size when same set of instructions are to be executed again and again. A general function consists of three parts, namely, function declaration (or prototype), function definition and function call.

Function declaration — prototype:

A function has to be declared before using it, in a manner similar to variables and constants. A function declaration tells the compiler about a function's name, return type, and parameters and how to call the function. The general form of a C++ function declaration is as follows:

return_type function_name(parameter list);

Function definition

The function definition is the actual body of the function. The function definition consists of two parts namely, function header and function body.

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

return_type function_name(parameter list)

{ body of the function }

Here, **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.

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.

Calling a Function

To use a function, you will have to call or invoke that function. 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.

A c++ program calculating factorial of a number using functions

```
#include<iostream.h>

#include<conio.h>

int factorial(int n);                    //function

declarationint main(){
```

```
int no, f;

cout<<"enter the positive number:-";

cin>>no;

f=factorial(no);                    //function call

cout<<"\nThe factorial of a number"<<no<<"is"<<f;

return 0;

}

int factorial(int n)        //function definition

{           int i , fact=1;

for(i=1;i<=n;i++){

            fact=fact*i;

}

return fact;

}
```

Inline Functions

An inline function is a function that is expanded inline at the point at which it is invoked, instead of actually being called. The reason that inline functions are an important addition to C++ is that they allow you to create very efficient code. Each time a normal function is called, a significant amount of overhead is generated by the calling and return mechanism. Typically, arguments are pushed onto the stack and various registers are saved when a function is called, and then restored when the function returns. The trouble is that these instructions take time. However, when a function is expanded inline, none of those operations occur. Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code. For this reason, it is best to inline only very small functions. inline is actually just a request, not a command, to the compiler. Thecompiler can choose to ignore it. Also, some compilers may not inline all types of functions. If a function cannot be inlined, it will simply be called as a normal function.

A function can be defined as an inline function by prefixing the keyword inline to the function header as given below:

```
inline function header {

function body

}
```

```
// A program illustrating inline function

#include<iostream.h>

#include<conio.h>


inline int max(int x, int y){

if(x>y)


return x;

else

return y;


}

int main( ) {

int a,b;


cout<<"enter two numbers";

cin>>a>>b;


cout << "The max is: " <<max(a,,b) << endl;

return 0;


}
```

Macros Vs inline functions

Preprocessor macros are just substitution patterns applied to your code. They can be used almost anywhere in your code because they are replaced with their expansions before any compilation starts. Inline functions are actual functions whose body is directly injected into their call site. They can only be used where a function call is appropriate.

inline functions are similar to macros (because the function code is expanded at the point of the call at compile time), inline functions are parsed by the compiler, whereas macros are expanded by the preprocessor. As a result, there are several important differences:

- Inline functions follow all the protocols of type safety enforced on normal functions.
- Inline functions are specified using the same syntax as any other function except that they include the inline keyword in the function declaration.
- Expressions passed as arguments to inline functions are evaluated once.
- In some cases, expressions passed as arguments to macros can be evaluated more than once.

- macros are expanded at pre-compile time, you cannot use them for debugging, but you can use inline functions.

Reference variable

A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable. To declare a reference variable or parameter, precede the variable's name with the &.The syntax for declaring a reference variable is:

datatype &Ref = variable name;

Example:

```
int main(){

int var1=10;                    //declaring simple variable

int & var2=var1; //declaring reference variable

cout<<"\n value of var2 =" << var2;


return 0;

}
```

var2 is a reference variable to var1.Hence, var2 is an alternate name to var1.This code prints the value of var2 exactly as that of var1.

Call by reference

Arguments can be passed to functions in one of two ways: using call-by-value or call-by-reference. When using call-by-value, a copy of the argument is passed to the function. Call-by-reference passes the address of the argument to the function. By default, C++ uses call-by-value.

Provision of the reference variables in c++ permits us to pass parameter to the functions by reference. When we pass arguments by reference, the formal arguments in the called function become aliases to the actual arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data.

Example

```
#include <iostream.h>

#include<conio.h>


void swap(int &x, int &y); // function declaration

int main (){


int a = 10, b=20;

cout << "Before swapping"<<endl;
cout<< “value of a :" << a <<” value of b :" << b << endl;

swap(a, b);          //calling a function to swap the values.

cout << "After swapping”<<endl;

cout<<” value of a :" << a<< “value of b :" << b << endl;

return 0;

}

void swap(int &x, int &y) {              //function definition to swap the values.

int temp;

temp = x;x

= y;

y = temp;

}
```

Output:

```
Before swapping             value of a:10 value of b:20

After swapping              value of a:20 value of b:10
```

Function Overloading

Function overloading is the process of using the same name for two or more functions. Each redefinition of the function must use either different types of parameters or a different number of parameters. It is only through these differences that the compiler knows which function to call in any given situation. Overloaded functions can help reduce the complexity of a program by allowing related operations to be referred to by the same name. To overload a function, simply declare and define all required versions. The compiler will automatically select the correct version based upon thenumber and/or type of the arguments used to call the function. Two functions differing only in their return types cannot be overloaded.

Example #include<iostream.h>

```
#include<conio.h>

int sum(int p,int q,int r);

double sum(int l,double m);

float sum(float p,float q)
```

```
int main(){

cout<<"sum="<< sum(11,22,33);          //calls  func1

cout<<"sum="<< sum(10,15.5);           //calls  func2

cout<<"sum="<< sum(13.5,12.5);         //calls  func3

return 0;

}

int sum(int p,int q,int r){            //func1

   return(a+b+c);

  }

 double sum(int l,double m){           //func2

    return(l+m);

  }

   float sum(float p,float q){         //func3

   return(p+q);

  }
```

Default arguments

C++ allows a function to assign a parameter a default value when no argument corresponding to that parameter is specified in a call to that function. The default value is specified in a manner syntactically similar to a variable initialization. All default parameters must be to the right of any parameters that don't have defaults. We cannot provide a default value to a particular argument in the middle of an argument list. When you create a function that has one or more default arguments, those arguments must be specified only once: either in the function's prototype or in the function's definitionif the definition precedes the function's first use.

Default arguments are useful if you don't want to go to the trouble of writing arguments that, for example, almost always have the same value. They are also useful in cases where, after a program is written, the programmer decides to increase the capability of a function by adding another argument. Using default arguments means that the existing function calls can continue to use the old number of arguments, while new function calls can use more.

Example

```
#include <iostream.h>

#include<conio.h>
```

```
int sum(int a, int b=20){

return( a + b);


}

int main (){

int a = 100, b=200, result;

result = sum(a, b);              //here a=100 , b=200

cout << "Total value is :" << result << endl;


result = sum(a);                 //here a=100 , b=20(using default value)

cout << "Total value is :" << result << endl;


return 0;

}
```