Control Structures:

There are three types of control structures:

1. Sequence
2. Selection also called as decision making statements
3. Iteration

Decision making statements: C++ language supports the decision making-statements as listed below.

- The if statement
- The if-else statement
- The nested if-else statements.
- The else-if ladder
- The switch case statement.
- The break statement
- The default keyword

The decision-making statement checks the given condition and then executes its sub-block. The decision statement decides the statement to be executed after the success or failure of a given condition.

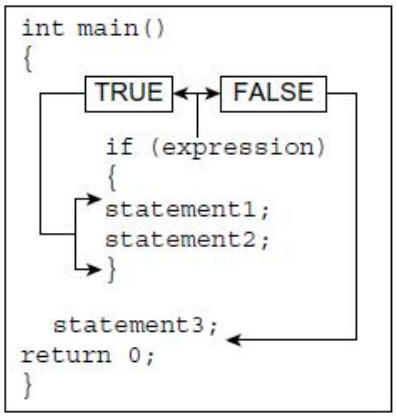
The if Statement: C++ uses the keyword if to execute a set of command lines or a command line when the logical condition is true. It has only one option. The syntax for the simplest if statement is as shown in figure.

Syntax for the simplest *if* statement:

```
if (expression) /* no semi-colon */
                Statement;
```

Syntax for the simplest *if* statement:

```
if (expression) /* no semi-colon */
{
        Statement 1;
        Statement  2;
        ----------
        ----------
}
```



/* Write a program to declare the price of the book with if **statement and check its price. If its price is less than or equal to 600, print the output with some comment, else terminate.*/**

```
#include<iostream>
using namespace std;
int main()
{
        int price;
        cout<<"\nEnter the price of the book:";
        cin>>price;
        if(price<=600)
        {
            cout<<"\n Hurry up buy the book!!!!!";
        }
        return 0;
}
```

Output:

```
Enter the price of the book: 345
Hurry up buy the book!!!!!
```

Multiple Ifs: The syntax of multiple ifs is shown in figure. Programs on multiple ifs are as follows.

Syntax for the multiple ifs:

```
if (expression) /* no semi-colon */
        Statement 1;
if (expression) /* no semi-colon */
        Statement 2;
if (expression) /* no semi-colon */
        Statement 3;
```

/* **Write a program to enter the percentage of marks obtained by a student and display the class obtained.*/**

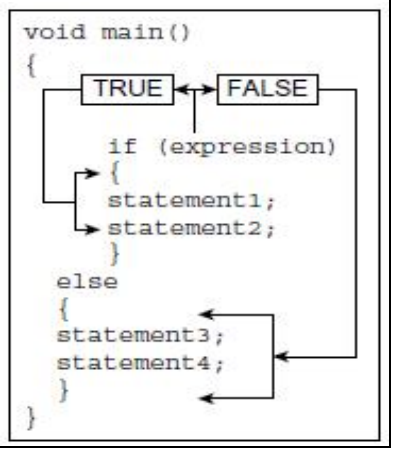
```
#include<iostream>
using namespace std;
int main()
{
        float per;
        cout<<"\nEnter the percentage:";
        cin>>per;
        if(per>=90 && per<=100)
            cout<<"\nDistinction";
```

```
        if(per>=70 && per<90)
            cout<<"\nFirst class";
        if(per>=50 && per<70)
            cout<<"\nSecond class";
        if(per>=40 && per<50)
            cout<<"\nPass";
        if(per<40)
        cout<<"\nFail";
        return 0;
}
```

Output:

```
Enter the percentage: 95
Distinction.
```

The if-else Statement: In the if–else statement, if the expression/condition is true, the body of the if statement is executed; otherwise, the body of the else statement is executed.

Syntax:
 	
if(expression)
execute the statement1;
else
 	
execute the statement2;
	

/*Write a program to enter age and display a message whether the user is eligible for voting or not. Use if-else **statement.*/**

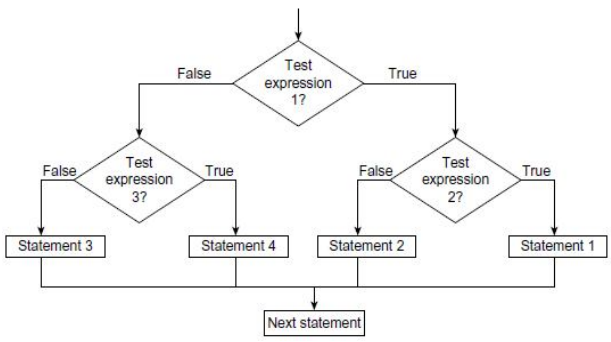
```
#include<iostream>
using namespace std;
int main()
{
        int age;
        cout<<"Enter Your Age:";
        cin>>age;
        if (age>=18)
        {
            cout<<"You are eligible for voting.";
        }
        else
        {
            cout<<"You are not eligible for voting"<<endl;
            cout<< "Wait for"<<18-age<<"year(s).";
        }
        return 0;
}
```

Output:

```
Enter Your Age: 17
You are not eligible for voting
Wait for 1 year(s).
```

Nested if-else Statements: In this kind of statement, a number of logical conditions are tested for taking decisions. Here, the if keyword followed by an expression is evaluated. If it is true, the compiler executes the block following the if condition; otherwise, it skips this block and executes the else block. It uses the if statement nested inside an if-else statement, which is nested inside another if-else statement. This kind of nesting can be limitless.

```
if (expression1)
{
    if(expression2)
          statement1;
    else
         statement2;
}
else
{
          if(expression3)
                statement3;
         else
```



statement4; } Next statement;	

/* Program to print the largest of three numbers. */

```
#include<iostream>
using namespace std;
int main()
{
        int a,b,c;
        cout<<"\nEnter three numbers:";
        cin>>a>>b>>c;
        if(a>b)
        {
            if(a>c)
                cout<<"\n a is largest ";
            else
                cout<<"\n c is largest ";
        }
        else
        {
            if(b>c)
                cout<<"\n b is largest ";
            else
                cout<<"\n c is largest ";
        }
        return 0;
}
```

Output:

```
Enter three numbers: 3 6 89
c is largest.
```

The else-if Ladder: A common programming construct is the else-if ladder, sometimes called the if-else-if staircase because of its appearance. In the program one can write a ladder of else-if. The program goes down the ladder of else-if, in anticipation of one of the expressions being true.

```
/*Syntax of else-if statement can be given as follows.*/
if(condition)
{
        statement 1;    /* if block*/
        statement 2;
}
else if(condition)
{
        statement 3;    /* else if block*/
        statement 4;
}
else
{
        statement 5;    /* last else block */
        statement 6;
}
```

/***Write a program to calculate energy bill. Read the starting and ending meter reading. The charges are as follows.**

No. of units Consumed	**Rates in (Rs.)**
200 - 500	3.5
100 - 200	2.50
Less than 100	1.50 */

```
#include<iostream>
using namespace std;
int main()
{
        int previous,current,consumed;
        float total;
        clrscr();
        cout<<"\nInitial & Final Readings:";
        cin>>previous>>current;
        consumed = current-previous;
        if(consumed>=200 && consumed<=500)
        total=consumed *3.50;
        else if(consumed>=100 && consumed<=199)
        total=consumed *2.50;
        else if(consumed<100)
        total=consumed*1.50;
        cout<<"\n Total no of units consumed: "<<consumed;
        cout<<"\n Electric Bill for units "<< consumed<< "is" <<total;
        return 0;
}
```

Output:

Initial & final readings: 100 235
Total no of units consumed: 135
Electric bill for units 135 is 337.5

Unconditional Control Transfer Statements

The goto statement: This statement does not require any condition. This statement passes control anywhere in the program without considering any condition. The general format for this statement is shown in figure.

Here, a label is any valid label either before or after goto. The 'label' must start with any character and can be constructed with rules used for forming identifiers. Avoid using the goto statement.

Syntax:

```
goto label;
-----
-----
-----
label:
```

/*Write a program to demonstrate the use of goto **statement.*/**

```
##include<iostream>
using namespace std;
void main()
{
        int x;

        cout<<"Enter a Number:";
        cin>>x;
        if (x%2==0)
                goto even;
        else
                goto odd;
        even:
                 cout<<x<<" is an Even Number.";
                return;
        odd:
                 cout<<x<<" is an Odd Number.";
}
```

The break Statement: The break statement allows the programmer to terminate the loop. The break skips from the loop or the block in which it is defined.

5.7.3 The continue Statement

The continue statement works somewhat like the break statement. Instead of forcing the control to the end of the loop (as it is in case of break), the continue case causes the control to pass on to the beginning of the block/loop. In the case of for loop, the continue case initiates the testing condition and increment on steps has to be executed (while rest of the statement following the continue are neglected). For while and do-while, the continue case causes control to pass on to conditional tests. It is useful in a programming situation where it is required that particular iterations occur only up to some extent or when some part of the code has to be neglected. The programs on continue are explained in the control loop chapter.

5.8 THE switch STATEMENT

The switch statement is a multi-way branch statement and an alternative to if-else-if ladder in many situations. The expression of switch contains only one argument, which is then checked with a number of switch cases. The switch statement evaluates the expression and then looks for its value among the case constants. If the value is matched with a particular case constant, then those case statements are executed until a break statement is found or until the end of switch block is reached. If not, then simply the default (if present) is executed (if a default is not present, then the control flows out of the switch block). The default is normally present at the bottom of the switch case structure. But we can also define default statement anywhere in the switch structure. The default block must not be empty. Every case statement terminates with a ':' (colon). The break statement is used to stop the execution of succeeding cases and pass the control to the end of the switch block.

```
switch(variable or expression)

{

case constant A:

statement;

break;

case constant B:

statement;

break;
```

```
default:
statement;
}
```

Function: In C++, a function is a group of statements that is given a name, and which can be called from some point of the program.

The most common syntax to define a function is:

```
return_type function_name ( parameter1, parameter2, ...)
{
statements
}
```

Where:

- return_type is the type of the value returned by the function.
- function_name is the identifier by which the function can be called.
- parameters (as many as needed): Each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma
- statements is the function's body. It is a block of statements surrounded by braces { } that specify what the function actually does.

Advantages:

- **Reusability:** A function once written can be invoked again and again, thus helping us to reuse the code and removing data redundancy.
- **Modularity:** Functions can help us in breaking a large, hard to manage problem into smaller manageable sub-problems.
- **Reduced Program Size:** Functions can reduce the size of the program by removing data redundancy.
- **Easy Debugging:** Using functions, debugging of a program becomes very easy, as it is easier to locate and rectify the bug in the program if functions are used.
- **Easy Updating:** If we need to update some code in the program, then it is much more easier in case we have used functions, as the changes need to be made in one place only (in function).

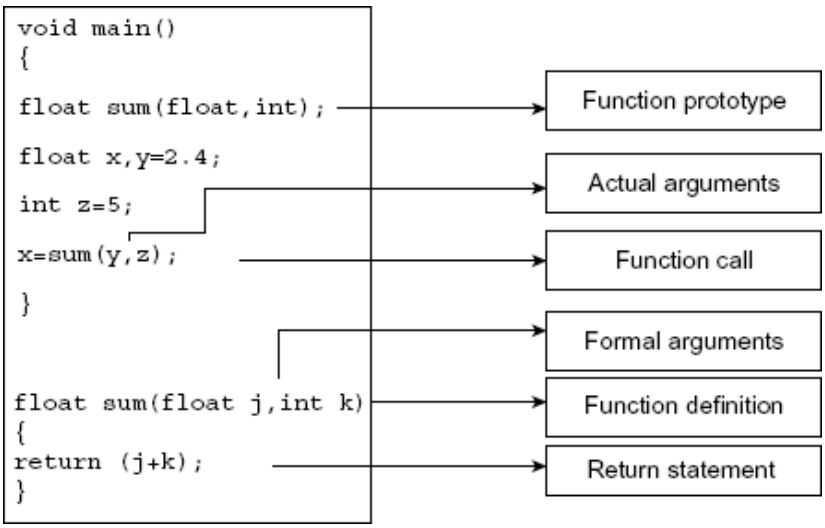
C++ functions are classified in two categories. They are

- Library functions and
- User-defined functions.

The library functions can be used in any program by including respective header files. The header files must be included using #include pre-processor directive. For example, a mathematical function uses math.h header file. The programmer can also define and use his/her own functions for performing some specific tasks. Such functions are called user-defined functions.

Parts of A Function: Parts of a function are as follows.

1. Function prototype declaration
2. Function call
3. Definition of a function
4. Actual and formal arguments
5. Return statement



1. **Function prototype Declaration:** A function prototype declaration consists of function's return type, name, and arguments list. The statements given below are the examples of function prototypes.

   ```
   void show (void);
   float sum (float, int);
   float sum (float x, int y);
   ```

2. **Function Call:** A function must be called by its name followed by argument, or without argument, list enclosed in parenthesis and terminated by semicolon.

   Syntax of function call is as follows:

   ```
   function-name(with/without argument list);
   ```

   In the above statement, function-name is the name of the function, arguments are within the bracket and arguments are separated by comma. If arguments are absent one can write void within the bracket.

3. **Function Definition:** The first line is called function definition and function body follows it. The function definition and function prototype should match with each other. The function body is enclosed within curly braces. The function can be defined anywhere. If the function is defined before its caller, then its prototype declaration is optional.

   Syntax of function call is as follows:

   ```
   return_data_type function-name(argument/parameter list);
   {
       variable declarations
       function statements
   }
   ```

4. **Actual and Formal Argument:** The arguments declared in caller function and given in the function call are called **actual arguments**. The arguments declared in the function definition are known as **formal arguments**.

5. **The return Statement:** The return statement is used to return value to the caller function. The return statement returns only one value at a time. When a return statement is encountered, complier transfers the control of the program to caller function. The syntax of return statement is as follows:

    return (variable name); or

    return variable name;

Passing Arguments:

The main objective of passing argument to function is message passing. The message passing is also known as communication between two functions, that is between caller and called functions. There are three methods:

1. Call by value (pass by value)
2. Call by address (pass by address)
3. Call by reference (pass by reference)

Call by Value: In this type, values of actual arguments are passed to the formal arguments and operation is done on the formal arguments. Any change in the formal arguments does not effect to the actual arguments because formal arguments are photocopy of actual arguments. Changes made in the formal arguments are local to the block of called function. Once control returns back to the calling function, the changes made will vanish.

The following example illustrates the use of call by value.

```
#include<iostream>
using namespace std;
void swap (int, int);

int main()
{
    int x,y;
    cout<<"\n Enter Values of X & Y:";
    cin>>x>>y;
    cout<<"\n\n In function main() before swap()";
    cout<<"\n Values X="<<x <<" and Y= "<<y;
```

```
        swap(x,y);
        cout<<"\n\n In function main() after swap() ";
        cout<<"\n Values X="<<x <<" and Y= "<<y;
        return 0;
}
void swap(int a, int b)
{
        int k;
        k=a;
         a=b;
        b=k;
        cout<<"\n In function swap() ";
        cout<<"\n Values X="<<a <<" and Y= "<<b;
}
```

Output:

```
Enter Values of X & Y :5 4
In function main()  before swap() Values X=5 and Y= 4
In function swap()  Values X=4 and Y= 5
In function main()  after swap() Values X=5 and Y= 4
```

Call by Address: In this type, instead of passing values, addresses of actual parameters are passed to the function by using pointers. Function operates on addresses rather than values. Here the formal arguments are pointers to the actual arguments. Because of this, when the values of formal arguments are changed, the values of actual parameters also change. Hence changes made in the argument are permanent. The following example illustrates passing the arguments to the function using call by address method.

```
 #include<iostream>
using namespace std;
void swap (int *, int *);
int main()
{
        int x,y;
        cout<<"\n Enter Values of X & Y:";
        cin>>x>>y;
        swap(&x,&y);
```

```
        cout<<"\n In main() Values X="<<x <<" and Y="<<y;
        return 0;
}
void swap(int *a, int *b)
{
        int *k;
        *k=*a;
        *a=*b;
        *b=*k;
        cout<<"\n In swap() Values X="<<*a <<" and Y="<<*b;
}
```

OUTPUT

```
Enter Values of X & Y :5 4
In swap()Values X=4 and Y=5
In main()Values X=4 and Y=5
```

Call by Reference: C passes arguments by value and address. In C++ it is possible to pass arguments by reference. C++ reference types, declared with *'&' operator*, they declare aliases for objects variables and allow the programmer to pass arguments by reference to functions. The reference decelerator (&) can be used to declare references outside functions.

For Ex.

```
int k = 0;
int &kk = k; // kk is an alias for k
kk = 2; // same effect as k = 2
```

Example:

```
#include<iostream>
using namespace std;
void swap (int &, int &);

int main()
{
        int x,y;
```

```
        cout<<"\n Enter Values of X & Y:";
        cin>>x>>y;
        swap(x,y);
        cout<<"\n In main()Values X="<<x <<" and Y="<<y;
        return 0;
}
void swap(int &a, int &b)
{
        int k;
        k=a;
        a=b;
        b=k;
        cout<<"\n In swap()Values X="<<a <<" and Y="<<b;
}
```

Output:

```
Enter Values of X & Y :5 4
In swap()Values X=4 and Y=5
In main()Values X=4 and Y=5
```

Default Arguments: A default argument is a value provided in function declaration that is automatically assigned by the compiler if the caller of the function does not provide a value for the argument.

Ex:

```
#include <iostream>
using namespace std;
int sum(int a, int b=10, int c=20, int d=30)
{
        return a+b+c+d;
}
int main() {
        int a=2,b=3,c=4,d=5;
        cout<<"Sum:"<<sum(a,b,c,d)<<endl;
        cout<<"Sum:"<<sum(a,b,c)<<endl;
        cout<<"Sum:"<<sum(a,b)<<endl;
        cout<<"Sum:"<<sum(a)<<endl;
        return 0;
}
```

Output:

Sum:14
Sum:39
Sum:55
Sum:62

***Const* Arguments:** The constant variable can be declared using const keyword. The const keyword makes variable value stable.

Ex. The following program generates an error

```
#include <iostream>
using namespace std;
int increment(const int a);
int main() {
        int a=2;
        cout<<"Incremented val:"<<increment(a);
        return 0;
}
int increment( const int a)
{
        return ++a;

}
```

Output:

error: increment of read-only parameter 'a'

Inline Functions: When a function is declared as inline, the compiler copies the code of the function in the calling function. i.e., function body is inserted in place of function call during compilation.

Syntax:

```
inline function_name(with or without arguments)
{
        statement 1;
        statement  2;
 }
```

Following are few situations where inline functions may not work:

1. The function should not be recursive.
2. Function should not contain static variables.
3. Function containing control structure statements such as `switch`, `if`, `for` loop, etc.
4. The function `main()` cannot be used as inline.

The inline functions are similar to macros of C. The main limitation of macros is that they are not functions, and errors are not checked at the time of compilation. The function offers better type testing and does not contain limitations as present in macros. Consider the following example:

```
#include <iostream>
using namespace std;
inline int Max(int x, int y)
{
  return (x > y)? x : y;
}
int main( )
{
  cout << "Max (20,10): " << Max(20,10) << endl;
  cout << "Max (0,200): " << Max(0,200) << endl;
  cout << "Max (100,1010):"<<Max(100,1010)<<endl;
  return 0;
}
```

Output:

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010):1010
```

Function Overloading

It is possible in C++ to use the same function name for number of times and for different intentions. Defining multiple functions with same name is known as function overloading or function polymorphism. Polymorphism means one function having many forms. The overloaded function must be different in their argument list and with different data types. The following are examples of overloaded functions. All the functions defined should be equivalent to their prototypes.

```
int test() { }
int test(int a){ }
int test(double a){ }
int test(int a, double b){ }
```

Example:

```
#include <iostream>
using namespace std;

void test(int);
void test(float);
void test(int, float);

int main() {
   int a = 5;
   float b = 5.5;

   test(a);
   test(b);
   test(a, b);

   return 0;
}

void test(int var) {
   cout<<"Integer number: "<<var<<endl;
}
void test(float var){
   cout<<"Float number: "<<var<<endl;
}
void test(int var1, float var2) {
   cout<<"Integer number: "<<var1;
   cout<<" and float number:"<<var2;
}
```

Output:

```
Integer number: 5
Float number: 5.5
Integer number: 5 and float number:5.5
```

Principles of Function Overloading:

1. If two functions have the similar type and number of arguments (data type), the function cannot be overloaded. The return type may be similar or void, but argument data type or number of arguments must be different. For example,
   a) sum(int,int,int);

sum(int,int);

Here, the above function can be overloaded. Though the data type of arguments in both the functions are similar, number of arguments are different.

b) sum(int,int,int);

sum(float,float,float);

In the above example, number of arguments in both the functions are same, but data types are different. Hence, the above function can be overloaded.

2. Passing constant values directly instead of variables also results in ambiguity.
3. The compiler attempts to find an accurate function definition that matches in types and number of arguments and invokes that function. The arguments passed are checked with all declared function. If matching function is found then that function gets executed.
4. If there are no accurate match found, compiler makes the implicit conversion of actual argument. For example, `char` is converted to `int` and `float` is converted to `double`.
5. If internal conversion fails, user-defined conversion is carried out with implicit conversion and integral promotion.

Precautions with Function Overloading

1. Only those functions that basically do the same task, on different sets of data, should be overloaded. The overloading of function with identical name but for different purposes should be avoided.
2. In function overloading, more than one function has to be actually defined and each of these occupy memory.
3. Instead of function overloading, using default arguments may make more sense and fewer overheads.
4. Declare function prototypes before *main*() and pass variables instead of passing constant directly. This will avoid ambiguity that frequently occurs while overloading functions.

Recursion:

In many programming languages including C++, it is possible to call a function from a same function. This function is known as recursive function and this programming technique is known as recursion.

In recursion a function calls itself and the control goes to the same function and it executes repeatedly until some condition is satisfied. In this type of recursive calls a function starts with a new value every time.

Rules for Recursive Function

1. In recursion, it is essential to call a function by itself; otherwise recursion would not take place.
2. Only the user-defined function can be involved in the recursion. Library function cannot involve in recursion because their source code cannot be viewed.
3. A recursive function can be invoked by itself or by other function. It saves return address with the intention to return at proper location when return to a calling statement is made. The last-in-first-out nature of recursion indicates that stack data structure can be used to implement it.
4. To stop the recursive function, it is necessary to base the recursion on test condition, and proper terminating statement such as exit() or return() must be written using the if() statement.

Example:

```
#include<iostream>
using namespace std;

int main()
{
        unsigned long int fact(int);
        dint f,x;
        cout<<"\nEnter a Number:";
        cin>>x;
        f=fact(x);
        cout<<"\nFactorial of " <<x <<" is "<<f;
        return 0;
}
unsigned long int fact(int a)
{
        unsigned long factorial;
        if(a==1)
```

```
        return 1;
    else
        factorial=a*fact(a-1);
    return factorial;
}
```

Output:

```
Enter a Number:6
Factorial of 6 is 720
```

Library Functions:

ceil, ceill and floor, floorl: The functions ceil and ceill round up the given float number, whereas the functions floor and floorl round down the float number. They are defined in math.h header file. Their declarations are as given below.

```
double ceil(double n);
double floor(double n);
dlng double ceill(long double (n));
long double floorl(long double (n));
```

modf and modfI: The function `modf` breaks `double` into integer and fraction elements, and the function `modfl` breaks long `double` into integer and fraction elements. These functions return the fractional elements of a given number. They are declared as given below.

```
double modf(double n, double *ip);
long double modfl(long double (n), long double *(ip));
```

Ex.

```
int main()
{
        double f, i;
        double num = 211.57;
        f = modf(num, &i);
        cout<<"\n The Complete Number:
"<<num;
        cout<<"\n The Integer elements: "<<i;
        cout<<"\n Fractional Elements: "<<f;
        return 0;
}
```

Output:

```
The Complete Number :
211.57

The Integer elements : 211

Fractional Elements : 0.57
```

abs, fabs, and labs: The function `abs()` returns the absolute value of an integer. The `fabs()` returns the absolute value of a floating-point number, and `labs()` returns the absolute value of a long number.

```
int abs(int n);
double fabs(double n);
long int labs(long int n);
```