

## Enumeration Data type

An enumeration is a user-defined data type that consists of integral constants. To define an enumeration, keyword **enum** is used.

### Enumerated Type Declaration

When you create an enumerated type, only blueprint for the variable is created. Here's how you can create variables of enum type.

```
enumboolean { false, true };  
  
// inside function  
  
enumboolean check;
```

### Example 1: Enumeration Type

```
#include<iostream>  
usingnamespacestd;  
  
enum week {Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday};  
  
int main()  
{  
    week today;  
    today=Wednesday;  
    cout<<"Day "<< today+1;  
    return0;  
}
```

#### **Output**

Day 4

## **Example2: Changing Default Value of Enums**

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

enum seasons { spring =34, summer =4, autumn =9, winter =32};

int main(){

    seasons s;

    s = summer;
    cout<<"Summer = "<< s <<endl;

    return 0;
}
```

### **Output**

Summer = 4

## **Why enums are used in C++ programming?**

An enum variable takes only one value out of many possible values.

## Symbolic Constant

An identifier that represents a constant value throughout the life of the program is known as Symbolic Constants. It allows programmers to attach meaningful names to data values and hence enhances the readability of the programs.

steps and rules:

- 1) Const must be initialized.
- 2) Const is local to the function where it is declared.
- 3) By the qualifier extern along with const, it can be made global.
- 4) If data type is not given it is treated as integer.

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

constint a = 100;  // Const Variable

classTestConst
{
    public:
        void display() const// Const Function
        {
            cout<< "Value of a in the const function is " << a;
        }
};

int main ()
{
    Test int1;
    int1.display();
    cout<< a;
    return 0;
}
```

## Type Casting In C++

Typecasting is the concept of converting the value of one type into another type. For example, you might have a float that you need to use in a function that requires an integer.

### Implicit conversion

Almost every compiler makes use of what is called automatic typecasting. It automatically converts one type into another type. If the compiler converts a type it will normally give a warning. For example this warning: conversion from 'double' to 'int', possible loss of data.

The problem with this is, that you get a warning (normally you want to compile without warnings and errors) and you are not in control. With control we mean, you did not decide to convert to another type, the compiler did. Also the possible loss of data could be unwanted.

### Explicit conversion

The C and C++ languages have ways to give you back control. This can be done with what is called an explicit conversion. Sure you may still lose data, but you decide when to convert to another type and you don't get any compiler warnings.

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

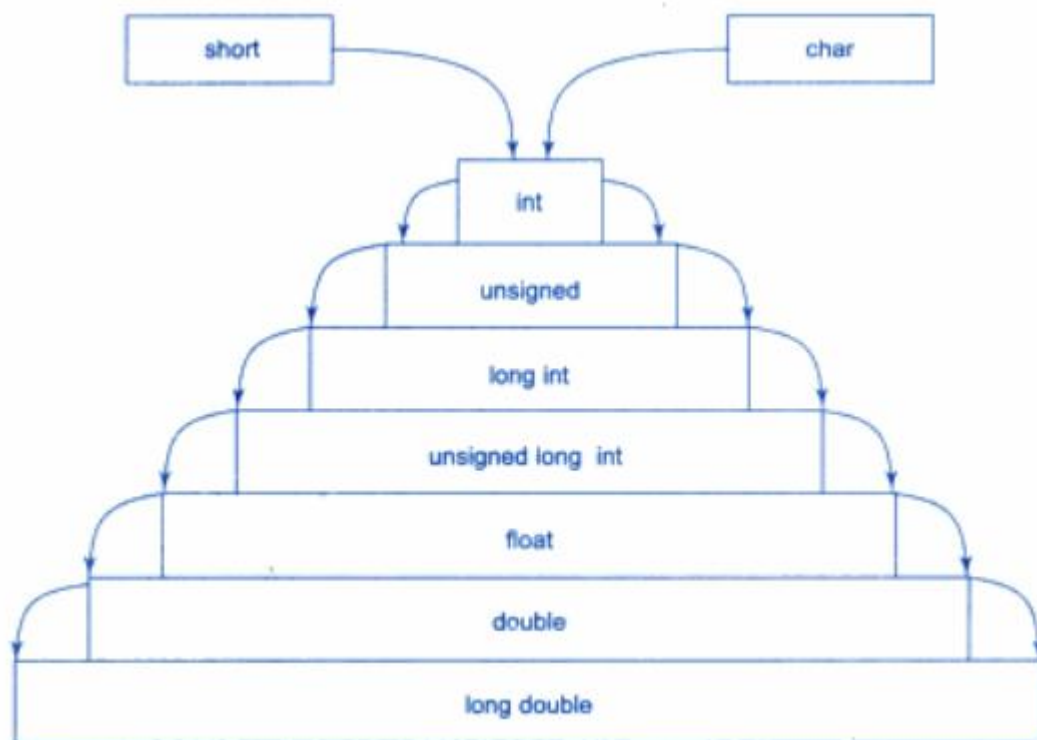
int main()
{
    int a;
    double b=2.55;

    a = b;
    cout<< a <<endl;

    a = (int)b;
    cout<< a <<endl;

    a = int(b);
    cout<< a <<endl;
}
```

**Note:** the output of all cout statements is 2.



## Friend Function in C++

If a function is defined as a friend function then, the private and protected data of a class can be accessed using the function.

The compiler knows a given function is a friend function by the use of the keyword **friend**.

For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword friend.

### Declaration of friend function in C++

```
class class_name
{
    ... ..
    friend return_type function_name(argument/s);
    ... ..
}
```

## Example 1: Working of friend Function

```

/* C++ program to demonstrate the working of friend function.*/
#include<iostream>
using namespace std;

class Distance {
private:
int meter;
public:
Distance():
    meter(0)
    {

    }

friend int func(Distance); //friend function
};

int func(Distance d){
    //function definition
    d.meter=10; //accessing private data from non-member function
    return d.meter;
}

int main()
{
    Distance D;
    cout<<"Distance: "<<func(D);
    return 0;
}

```

Here, friend function `addFive()` is declared inside `Distance` class. So, the private data `meter` can be accessed from this function.

Though this example gives you an idea about the concept of a friend function, it doesn't give show you any meaningful use.

A more meaningful use would be when you need to operate on objects of two different classes. That's when the friend function can be very helpful.

You can definitely operate on two objects of different classes without using the friend function but the program will be long, complex and hard to understand.

## Example 2: Addition of members of two different classes using friend Function

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

// forward declaration
class B;
class A {
private:
int numA;
public:
A():numA(12){}
// friend function declaration
friend int add(A, B);
};

class B {
private:
int numB;
public:
B():numB(1){}
// friend function declaration
friend int add(A, B);
};

// Function add() is the friend function of classes A and B
// that accesses the member variables numA and numB
int add(A objectA, B objectB)
{
return(objectA.numA+objectB.numB);
}

int main()
{
A objectA;
B objectB;
cout<<"Sum: "<< add(objectA,objectB);
return 0;
}
```



**Output**

Sum: 13

In this program, classes A and B have declared `add()` as a friend function. Thus, this function can access private data of both class.

Here, `add()` function adds the private data `numA` and `numB` of two objects `objectA` and `objectB`, and returns it to the main function.

To make this program work properly, a forward declaration of a class `class B` should be made as shown in the above example.

This is because `class B` is referenced within the `class A` using code: `friend int add(A, B);`.

## C++ Math Function

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

int main()
{
    double number, dsqrt,dpow,dlog;

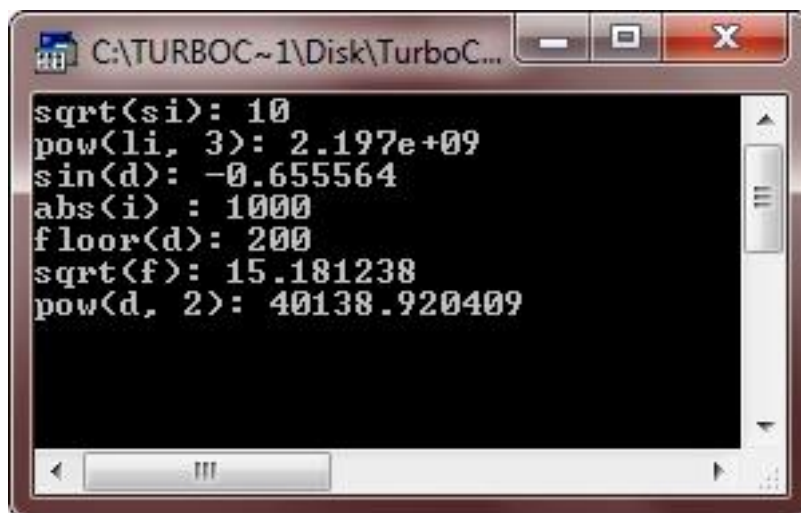
    cout<< "Please enter a number \n";
    cin>> number;

    dsqrt =sqrt(number);
    dpow =pow(number,5);
    dlog =log(number);

    cout<< " Math Example\n";
    cout<< " Square Root is " <<dsqrt<< "\n";
    cout<< " Raised to the fifth power is " <<dpow<< "\n";
    cout<< " Log is " <<dlog<< "\n";

    return 0;
}
```

```
/* C++ Mathematical Functions */  
  
#include<iostream.h>  
#include<conio.h>  
#include<math.h>  
void main()  
{  
    clrscr();  
  
    shortint si = 100;  
    int i = -1000;  
    longint li = 1300;  
    float f = 230.47;  
    double d = 200.347;  
  
    cout<<"sqrt(si): "<<sqrt(si)<<endl;  
    cout<<"pow(li, 3): "<<pow(li, 3)<<endl;  
    cout<<"sin(d): "<<sin(d)<<endl;  
    cout<<"abs(i) : "<<abs(i)<<endl;  
    cout<<"floor(d): "<<floor(d)<<endl;  
    cout<<"sqrt(f): "<<sqrt(f)<<endl;  
    cout<<"pow(d, 2): "<<pow(d, 2)<<endl;  
  
    getch();  
}
```



The screenshot shows a Turbo C++ console window with the following output:

```
sqrt(si): 10  
pow(li, 3): 2.197e+09  
sin(d): -0.655564  
abs(i) : 1000  
floor(d): 200  
sqrt(f): 15.181238  
pow(d, 2): 40138.920409
```

## Scope Rules of C++

The program part(s) in which a particular piece of code or a data value (e.g., variable) can be accessed is known as the piece-of-code's or variable's scope.

The scope rules of a language are the rules that decide, in which part(s) of the program a particular piece of code or data item would be known and can be accessed therein.

There are four kinds of scopes in C++:

- local scope
- function scope
- file scope
- class scope

### **C++ Local Scope**

A name declared in a block (i.e., {...}) is local to that block and can be used only in it and the other blocks contained under it. The names of formal arguments are treated as if they were declared in the outermost block of that function.

### **C++ Function Scope**

The variables declared in the outermost block of a function have function scope i.e., they can be accessed only in the function that declares them. Also labels (of goto) have function scope i.e., they cannot be used outside the function.

### **C++ File Scope**

A name declared outside all blocks and functions has file scope i.e., it can be used in all the blocks and functions written inside the file in which the name declaration appears.

### **C++ Class Scope**

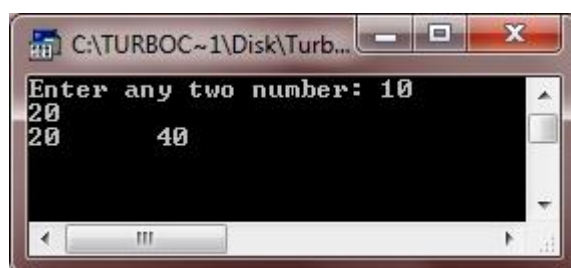
A name of a class member has class scope and is local to its class.

## C++ Scope Rules Example

Let's Look at the following example :

```
#include<.....>
int x, y;      // file scope (Global Variables)
void main()
{
    clrscr();
    int a, b;   // function scope for main()
    float amt;  // local variables of main()

    void check(int);
    :
    {
        char grade; // block scope (local variable of block)
        :           // block 2
    }
}
void check(int i) // function scope for check()
{
    long temp;    // local variables of check()
}
```



Here is an example program, demonstrating scope rules in C++

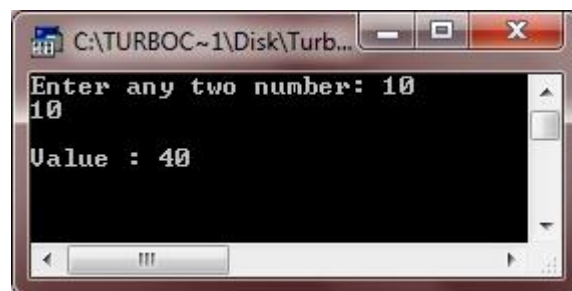
```
/* C++ Scope Rules */

#include<iostream.h>
#include<conio.h>

int a, b; // global variables
int fun(int);

void main()
{
    clrscr();
    int num1, num2; // local variables of main()
    cout<<"Enter any two number: ";
    cin>>num1>>num2;
    if(num1==num2)
    {
        int temp; // block scope (local variable of this if block)
        temp = num1 + num2;
        cout<<"\nValue : "<<fun(temp);
    }
    else
    {
        cout<<fun(num1)<<"\t"<<fun(num2);
    }
    getch();
}

int fun(int x) // function scope for fun()
{
    int res; // local variable of fun()
    res = x * 2;
    return res;
}
```



## Typedef Keyword in C++

C++ allows you to define explicitly new data type names by using the keyword typedef. Using typedef does not actually create a new data class, rather it defines a new name for an existing type. This can increase the portability of a program as only the typedef statements would have to be changed.

Using typedef can also aid in self-documenting your code by allowing descriptive names for the standard data types.

### C++ typedef Syntax

The syntax of the typedef statement is :

```
typedef type name;
```

where type is any C++ data type and name is the new name for this type. This defines another name for the standard type of C++. For example, you would create a new name for float values by using the following statement :

```
typedef float amount;
```

New name for the type float has been created through typedef.

This statement tells the compiler to recognize amount as an alternative name for float. Now you could create float variables using amount.

```
amount loan, saving, instalment;
```

See, variable for type amount (a typedef name for float) being created.

Here loan, saving, instalment are variables of type amount which is another word for float.

Thus, loan, saving, instalment are all float variables, internally.

**Note** - Remember that typedef does not create any new data types rather provides an alternative name for standard types. Reference provides an alias name for a variable and typedef provides an alias name for a data type. Let's take an example, demonstrating typedef in C++

## C++ typedef Example

Here is an example program, illustrating typedef in C++

```
/* C++ typedef - Example Program of typedef in C++ */  
  
#include<iostream.h>  
#include<conio.h>  
void main()  
{  
    clrscr();  
  
    typedef int integer;  
    /* now you can easily use integer to create  
    * variables of type int like this */  
    integer num1, num2, sum;  
    cout<<"Enter two number: ";  
    cin>>num1>>num2;  
    sum=num1+num2;  
    cout<<"Sum = "<<sum;  
  
    getch();  
}
```





```
/* C++ typedef - Example Program of typedef in C++ */
```

```
#include<iostream.h>
#include<conio.h>
void main()
{
    clrscr();

    typedef int integer;
    /* now you can easily use integer to create
    * variables of type int like this */
    integer num1;

    /* Remember that, you are always free to use
    * the original keyword to define variable
    * of int/other data type at any time like
    * this */
    int num2, sum;
    cout<<"Enter two number: ";
    cin>>num1>>num2;
    sum=num1+num2;
    cout<<"Sum = "<<sum;

    getch();
}
```



```
/* C++ typedef - Example Program of typedef in C++ */
```

```
#include<iostream.h>
#include<conio.h>
void main()
{
    clrscr();

    typedef int integer;
    integer num1;
    typedef integer integer_type;
    integer_type num2;
    typedef integer_type integer_data_type;
    integer_data_type sum;

    cout<<"num1 = ";
    cin>>num1;
    cout<<"num2 = ";
    cin>>num2;
    sum=num1+num2;
    cout<<"Sum = "<<sum;

    getch();
}
```



## C++ Storage Classes

Storage classes in C++, basically defines the scope and life time of the variable, functions in your program. Let's discuss about these C++ storage classes in detail one by one.

Storage class of a variable defines the lifetime and visibility of a variable. Lifetime means the duration till which the variable remains active and visibility defines in which module of the program the variable is accessible.

1. Automatic
2. Static
3. Register
4. Mutable

| Storage Class | Keyword  | Lifetime       | Visibility | Initial Value |
|---------------|----------|----------------|------------|---------------|
| Automatic     | auto     | Function Block | Local      | Garbage       |
| Static        | static   | Whole Program  | Local      | Zero          |
| Register      | register | Function Block | Local      | Garbage       |
| Mutable       | mutable  | Class          | Local      | Garbage       |

## 1. Automatic Storage Class

Automatic storage class assigns a variable to its default storage type. *auto* keyword is used to declare automatic variables. However, if a variable is declared without any keyword inside a function, it is automatic by default. This variable is **visible** only within the function it is declared and its **lifetime** is same as the lifetime of the function as well. Once the execution of function is finished, the variable is destroyed.

### Syntax of Automatic Storage Class Declaration

```
auto datatype var_name1 [= value];
```

### Example of Automatic Storage Class

```
auto int x;  
  
float y = 5.67;
```

## 2. Static Storage Class

Static storage class ensures a variable has the **visibility** mode of a local variable but **lifetime** of an external variable. It can be used only within the function where it is declared but destroyed only after the program execution has finished. When a function is called, the variable defined as static inside the function retains its previous value and operates on it. This is mostly used to save values in a recursive function.

### Syntax of Static Storage Class Declaration

```
static datatype var_name1 [= value];
```

### For example,

```
static int x = 101;  
  
static float sum;
```

### **3. Register Storage Class**

Register storage assigns a variable's storage in the CPU registers rather than primary memory. It has its lifetime and visibility same as automatic variable. The purpose of creating register variable is to increase access speed and makes program run faster. If there is no space available in register, these variables are stored in main memory and act similar to variables of automatic storage class. So only those variables which requires fast access should be made register.

### **Syntax of Register Storage Class Declaration**

```
register datatype var_name1 [= value];
```

**For example,**

```
register int id;  
  
register char a;
```

## Example of Storage Class

**Example 2: C++ program to create automatic, global, static and register variables.**

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

int g;  //global variable, initially holds 0

void test_function()
{
    static int s;  //static variable, initially holds 0
    register int r;  //register variable
    r=5;
    s=s+r*2;
    cout<<"Inside test_function"<<endl;
    cout<<"g = "<<g<<endl;
    cout<<"s = "<<s<<endl;
    cout<<"r = "<<r<<endl;
}

int main()
{
    int a;  //automatic variable
    g=25;
    a=17;
    cout<<"Inside main"<<endl;
    cout<<"a = "<<a<<endl;
    cout<<"g = "<<g<<endl;
    test_function();
    return 0;
}
```

Inside test\_function

g = 25

s = 10

r = 5

Inside main

a = 17

g = 25

Inside test\_function

g = 25

s = 20

r = 5



### 3. Mutable Storage Class

In C++, a class object can be kept constant using keyword *const*. This doesn't allow the data members of the class object to be modified during program execution. But, there are cases when some data members of this constant object must be changed. **For example**, during a bank transfer, a money transaction has to be locked such that no information could be changed but even then, its state has to be changed from - *started to processing to completed*. In those cases, we can make these variables modifiable using a **mutable** storage class.

#### Syntax for Mutable Storage Class Declaration

```
mutable datatype var_name1;
```

#### For example,

```
mutable int x;  
  
mutable char y;
```

## Example of Mutable Storage Class

### Example 3: C++ program to create mutable variable.

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

class test
{
    mutable int a;
    int b;
public:
    test(int x,int y)
    {
        a=x;
        b=y;
    }
    void square_a() const
    {
        a=a*a;
    }
    void display() const
    {
        cout<<"a = "<<a<<endl;
        cout<<"b = "<<b<<endl;
    }
};

int main()
{
    const test x(2,3);
    cout<<"Initial value"<<endl;
    x.display();
    x.square_a();
    cout<<"Final value"<<endl;
    x.display();
    return 0;
}
```

A class *test* is defined in the program. It consists of a mutable data member *a*. A constant object of class *test* is created and the value of data members are initialized using user-defined constructor. Since, *b* is a normal data member, its value can't be changed after initialization. However *a* being mutable, its value can be changed which is done by invoking *square\_a()* method. *display()* method is used to display the value the data members.

**Output**

Initial value

a = 2

b = 3

Final value

a = 4

b = 3

## **C++ Break, Continue, Goto Statement**

The jump (break, continue, goto, and return) statements unconditionally transfer program control within a function. C++ has four statements that perform an unconditional branch :

- return
- goto
- break
- continue

Of these, you may use return and goto anywhere in the program whereas break and continue are used inside smallest enclosings like loops etc. In addition to the above four, C++ provides a standard library function `exit()` that helps you break out of a program.

The return statement is used to return from a function.

## C++ goto Statement

The goto statement can transfer the program control anywhere in the program. The target destination of a goto statement is marked by a label. the target label and goto must appear in the same function.

Here is the syntax of the goto statement in C++:

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

where label is a user supplied identifier and can appear either before or after goto. For example, the following code fragment :

```
a = 0 ;  
start :  
cout<< "\n" << ++a ;  
if(a < 50)  
    goto start ;
```

prints number from 1 to 50. The cout prints the value of ++a and then if checks if a is less than 50, the control is transferred to the label start; otherwise the control is transferred to the statement following if.

## C++ break Statement

In C++, there are two statements `break;` and `continue;` specifically to alter the normal flow of a program.

Sometimes, it is desirable to skip the execution of a loop for a certain test condition or terminate it immediately without checking the condition.


The `break;` statement terminates a loop ([for](#), [while](#) and [do..while loop](#)) and a [switch statement](#) immediately when it appears.

### Syntax of break


```
break;
```

In real practice, `break` statement is almost always used inside the body of conditional statement (`if...else`) inside the loop.


```
while (test expression) {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
}
```



```
do {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
}
while (test expression);
```



```
for (initial expression; test expression; update expression) {
    statement/s
    if (test expression) {
        break;
    }
    statements/
}
```



NOTE: The `break` statement may also be used inside body of `else` statement.

```
#include <iostream>
using namespace std;
int main()
{
    int number,sum;

    // test expression is always true
    while (true)
    {
        cout << "Enter a number: ";
        cin >> number;

        if (number!=0)
        {
            sum += number;
        }
        else
        {
            // terminates the loop if number equals 0.0
            break;
        }

    }
    cout << "Sum = " << sum;

    return 0;
}
```

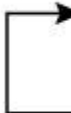
## C++ continue Statement

It is sometimes necessary to skip a certain test condition within a loop. In such case, `continue;` statement is used in C++ programming.

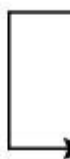
## Syntax of continue

```
continue;
```


In practice, `continue;` statement is almost always used inside a conditional statement.



```
while (test expression) {
    statement/s
    if (test expression) {
        continue;
    }
    statement/s
}
```



```
do {
    statement/s
    if (test expression) {
        continue;
    }
    statement/s
} while (test expression);
```



```
for (initial expression; test expression; update expression) {
    statement/s
    if (test expression) {
        continue;
    }
    statements/
}
```

NOTE: The `continue` statement may also be used inside the body of an `else` statement.



**C++ program to display integer from 1 to 10 except 6 and 9.**

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

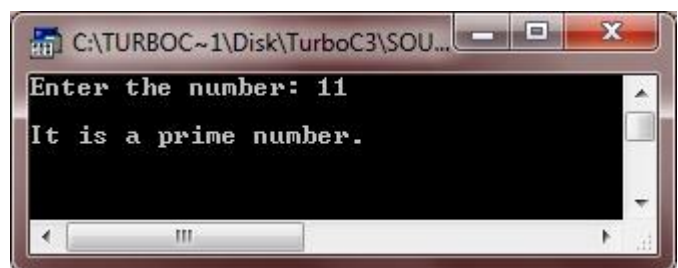
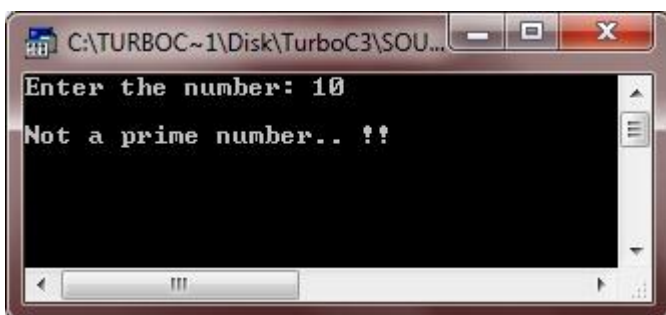
int main()
{
    for (int i = 1; i <= 10; ++i)
    {
        if ( i == 6 || i == 9)
        {
            continue;
        }
        cout << i << "\t";
    }
    return 0;
}
```

## C++ exit() function

Like you can break out of loops using a break statement, you can break out of a program using library function of C++, the exit() function. This function causes the program to terminate as soon as it is encountered, no matter where it appears in the program listing. Following program illustrates the use of exit() function :

/\* C++ Jump Statements - C++ exit() Function \*/

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
void main()
{
    clrscr();
    int num, i;
    cout<<"Enter the number: ";
    cin>>num;
    for(i=2; i<=num/2; i++)
    {
        if(num%i==0)
        {
            cout<<"\nNot a prime number.. !!";
            getch();
            exit(0);
        }
    }
    cout<<"\nIt is a prime number.";
    getch();
}
```



## C++ #define

Preprocessor commands are called DIRECTIVES, and begin with a pound or hash symbol (#). No white space should appear before the #, and semi colon is NOT required at the end.

Many things that can be done during preprocessing phase include :

- Inclusion of other files through #include directive
- Definition of symbolic constants and macros through #define directive

You have already learnt to work with #include preprocessor directive that lets you include desired header files in your program. This discussion or ours is dedicated to #define preprocessor directive.

The #define preprocessor allows us to define symbolic names and constants. For example,

```
#define PI 3.14159
```

## C++ #define Example

Now consider the complete example wherein every occurrence of PI will replace with the value 3.14159, the value defined with #define directive.

/\* C++ #define - Example Program of #define \*/

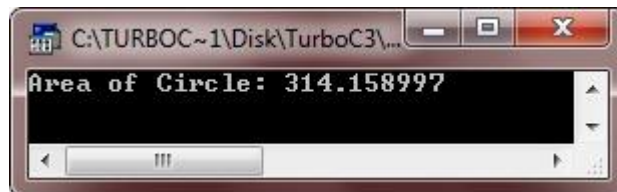
```
#include<iostream.h>
#include<conio.h>

#define PI 3.14159

void main()
{
    clrscr();

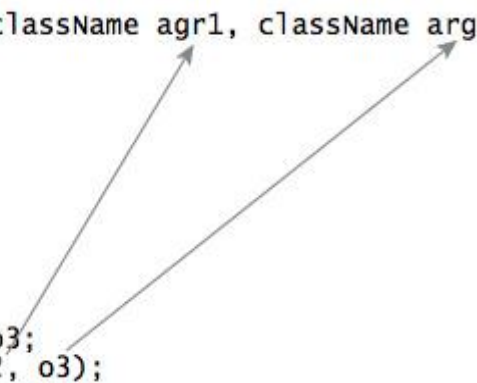
    int r = 10;
    float cir;
    cir = PI * (r * r);
    cout<<"Area of Circle: "<<cir<<endl;

    getch();
}
```



## Object as Function Argument in C++

```
class className {  
    ... ..  
public:  
    void functionName(className agr1, className arg2)  
    {  
        ... ..  
    }  
    ... ..  
};  
int main() {  
    className o1, o2, o3;  
    o1.functionName (o2, o3);  
}
```



The diagram illustrates the argument passing mechanism in the provided C++ code. Two arrows originate from the object variables `o2` and `o3` in the `main` function's call to `o1.functionName (o2, o3);`. One arrow points to the parameter `agr1` in the `functionName` definition, and the other points to the parameter `arg2`. This indicates that the objects `o2` and `o3` are passed by value to the function.

## Example 1: Pass Objects to Function

C++ program to add two complex numbers by passing objects to a function.

```
#include <iostream>
using namespace std;
class Complex
{
private:
    int real;
    int imag;
public:
    Complex(): real(0), imag(0) { }
    void readData()
    {
        cout << "Enter real and imaginary number respectively:"<<endl;
        cin >> real >> imag;
    }
    void addComplexNumbers(Complex comp1, Complex comp2)
    {
        // real represents the real data of object c3 because this function is called using
        code c3.add(c1,c2);
        real=comp1.real+comp2.real;

        // imag represents the imag data of object c3 because this function is called using
        code c3.add(c1,c2);
        imag=comp1.imag+comp2.imag;
    }

    void displaySum()
    {
        cout << "Sum = " << real<< "+" << imag << "i";
    }
};
int main()
{
    Complex c1,c2,c3;

    c1.readData();
    c2.readData();

    c3.addComplexNumbers(c1, c2);
    c3.displaySum();

    return 0;
```

```
}
```

Enter real and imaginary number respectively:

2

4

Enter real and imaginary number respectively:

-3

4

Sum = -1+8i

**File Create, Write, Open, Read in C++**

|          |                          |
|----------|--------------------------|
| ios::in  | Open a file for reading. |
| ios::out | Open a file for writing. |

**C++ program to create a file.**

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    fstream file; //object of fstream class

    //opening file "sample.txt" in out(write) mode
    file.open("sample.txt",ios::out);

    if(!file)
    {
        cout<<"Error in creating file!!!";
        return 0;
    }

    cout<<"File created successfully.";

    //closing the file
    file.close();

    return 0;
}
```

**C++ program to create or open, read, write and close a file.**

```
#include <iostream>
#include <fstream>
```



```
using namespace std;

int main()
{
    fstream file; //object of fstream class

    //opening file "sample.txt" in out(write) mode
    file.open("sample.txt",ios::out);

    if(!file)
    {
        cout<<"Error in creating file!!!"<<endl;
        return 0;
    }

    cout<<"File created successfully."<<endl;
    //write text into file
    file<<"ABCD.";
    //closing the file
    file.close();

    //again open file in read mode
    file.open("sample.txt",ios::in);

    if(!file)
    {
        cout<<"Error in opening file!!!"<<endl;
        return 0;
    }

    //read untill end of file is not found.
    char ch; //to read single character
    cout<<"File content: ";

    while(!file.eof())
    {
        file>>ch; //read single character from file
        cout<<ch;
    }

    file.close(); //close file

    return 0;
}
```

## C++ program to read a content of file and close a file

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

int main()
{
    char ch;
    const char *fileName="sample.txt";

    //declare object
    ifstream file;

    //open file
    file.open(fileName,ios::in);
    if(!file)
    {
        cout<<"Error in opening file!!!"<<endl;
        return -1; //return from main
    }

    //read and print file content
    while (!file.eof())
    {
        file >> noskipws >> ch;    //reading from file
        cout << ch;    //printing
    }
    //close the file
    file.close();

    return 0;
}
```

## Remove/Delete a file in C++

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
void main()
{
    clrscr();
    int status;
    char fname[20];
    cout<<"Enter name of file, you want to delete : ";
    gets(fname);
    status=remove(fname);
    if(status==0)
    {
        cout<<"file "<<fname<<" deleted successfully..!!\n";
    }
    else
    {
        cout<<"Unable to delete file "<<fname<<"\n";
        perror("Error Message ");
    }
    getch();
}
```

## Pre-increment and post-increment in C++

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

main() {
    int a = 21;
    int c ;

    // Value of a will not be increased before assignment.
    c = a++;
    cout << "Line 1 - Value of a++ is :" << c << endl ;

    // After expression value of a is increased
    cout << "Line 2 - Value of a is :" << a << endl ;

    // Value of a will be increased before assignment.
    c = ++a;
    cout << "Line 3 - Value of ++a is :" << c << endl ;
    return 0;
}
```

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

```
Line 1 - Value of a++ is :21
Line 2 - Value of a is :22
Line 3 - Value of ++a is :23
```

## Exception Handling in C++

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try**, **catch**.

- **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
#include<iostream>
using namespace std;
int main()
{
    int a,b,c;
    float d;

    cout<<"Enter the value of a:";
    cin>>a;
    cout<<"Enter the value of b:";
    cin>>b;
    cout<<"Enter the value of c:";
    cin>>c;

    try
    {
        if((a-b)!=0)
        {
            d=c/ (a-b);
            cout<<"Result is:"<<d;
        }
        else
        {
            throw(a-b);
        }
    }

    catch(int i)
    {
        cout<<"Answer is infinite because a-b is:"<<i;
    }

}
```