C++ Language

Tokens:

The smallest individual units are known as tokens such as keywords, identifiers, strings, operators & special symbols.

1. Keywords are the reserved (special) words and can't be used as variable, functions names.
2. Identifiers refer to the names of the variables, arrays, functions, classes etc.
3. Constants refer to fixed values that we cannot change in a program.
4. String is a group of characters.
5. Operators are special symbols which operate on variable & constants and form expressions.
6. Special symbols are () {} [] etc.

1. **C++ Keywords:** There are 32 keyword in C and 31 new keywords are added in C++, so 63 keywords in C++. Keywords are special words whose meaning has all ready been known to the compiler. All keywords should write in small letters and cannot be used as a identifier or variable names.
2. **Identifiers**: Identifiers are names given to various program elements like variables, array, functions, structure etc.

   Rules for writing Identifiers.

   (a) Identifier name must start with alphabets or underscore (_).
   (b) From second character onwards any combination of digits, alphabets or underscore is allowed.
   (c) No special symbols are allowed in variables or identifiers name.
   (d) Keywords cannot be used as an identifier name.
   (e) For ANSI C++ maximum length of identifier is 31, but many compiler support more than 32.

Difference between C & C++:

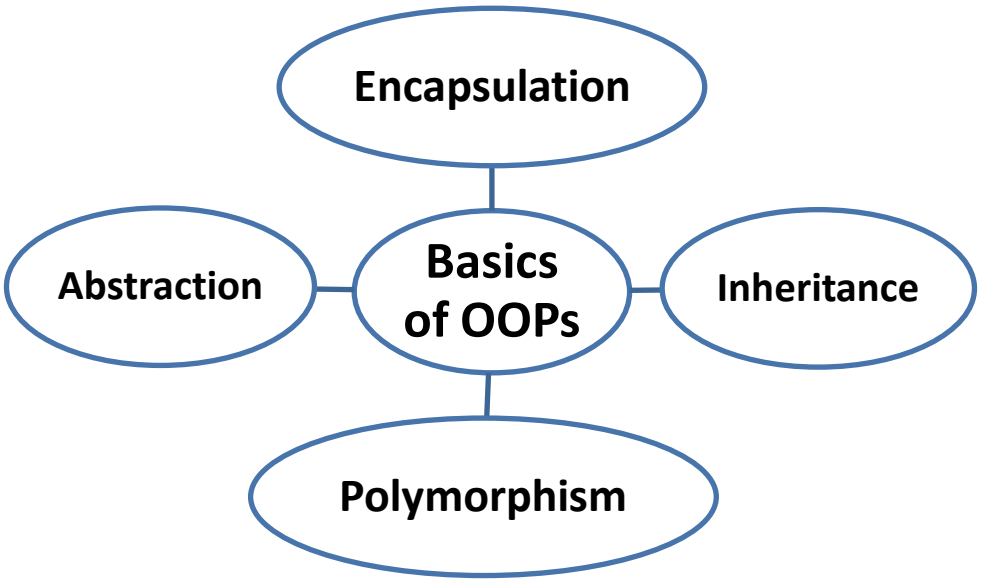
S.No.	C	C++
1.	C is a top down structured programming language. In C main function is defined first and then other sub functions are defined.	C++ is a bottom up object oriented programming language. In C++ all the sub functions are defined first in a class then main function is defined.
2.	C programs are saved with .C extension.	C++ programs are saved with .CPP extension.
3.	"stdio.h" header file is used for input/output in C.	"iostream.h" header file is used for input/output in C++.
4.	Scanf is a library function for input the data, is declared in stdio.h header file.	cin is used in C++ to get the data from key board is declared in iostream.h header file.
5.	printf is a library function for display the data, is declared in stdio.h header file.	cout is used in C++ to print the result is declared in iostream.h header file.
6.	Function prototype (declaration) is not compulsory in C.	In C++ it is compulsory for all functions.
7.	In function we can pass argument by value or by reference.	In function we can pass argument by value, by address and by reference.
8.	In C variable declaration must be at the top of a scope.	Flexible declaration. In C++ we can declare variable anywhere, but before use.

9.	ANSI C recognizes only the first 32 characters of a variable name.	In ANSI C++ there is no such limit on length of variable name, all characters are equally important.
10.	There are 32 keywords in C.	There are 32 + 31 = 63 keywords in C++.

Object Oriented Programming

Object oriented programming is a programming style that is associated with the concept of OBJECTS, having data members and related member functions.

Objects are instances of classes and are used to interact amongst each other to create applications. Instance means, the object of class on which we are currently working. C++ can be said to be as C language with classes. In C++ everything revolves around object of class, which has their methods and data members.



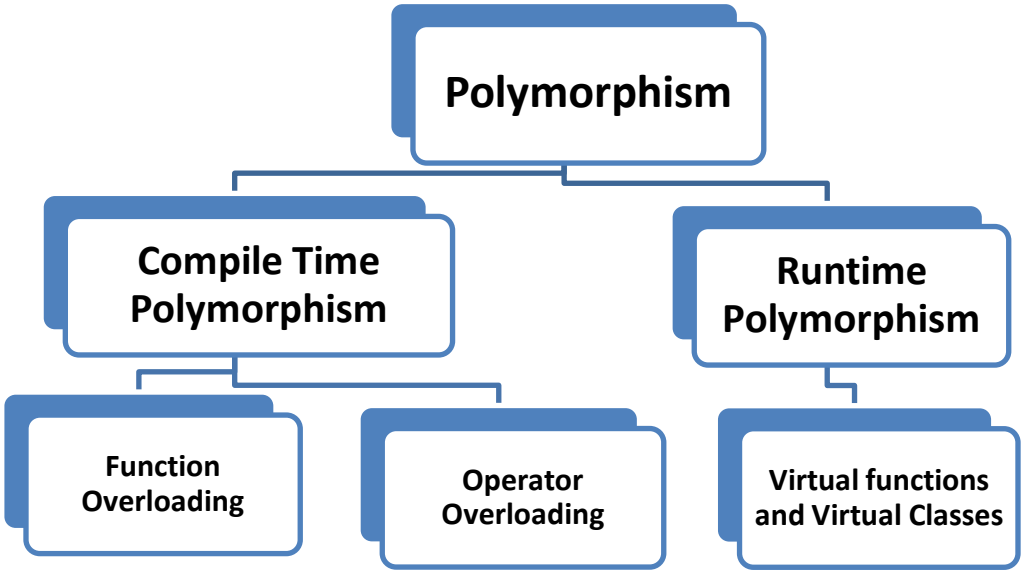
For e.g. - We consider human body as a class, we do have multiple objects of this class, with variable as color, hair etc. and methods as walking, speaking etc.

Now, let us discuss some of the main features of object oriented programming which you will be using in C++.

1. **Objects** – Objects are the basic unit of OOP. They are instances of class, which have data members and uses various member functions to perform tasks.
2. **Classes**- Class is a combination of data members and member functions. Class can also be defined as data type but it also contains functions in it.
3. **Abstraction**- Abstraction means using the data members in the program, without knowing their background.
4. **Encapsulation**- Encapsulation means wrapping up the data members and member functions. In other words, Encapsulation is all about binding the data variables and functions together in class.

5. **Polymorphism**- Polymorphism ia a ability of an object to take many different forms at different instances. Polymorphism is of 2 types-
   A) Compile -Time Polymorphism
   B) Run-time Polymorphism

   Polymorphism classification can be viewed graphically-

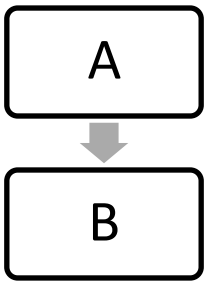


A) **Compile Time Polymorphism**- In this method object is bound to the function call at the compile time itself. Compile time polymorphism is of two types-
   i) **Function Overloading**- Function overloading means different functions have the same name but their arguments or parameters are different and each function works for their specific requirement.
   ii) **Operator overloading**- Operator overloading means same operator can be reused for other purpose.

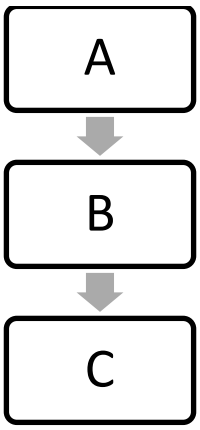
B) **Run-time polymorphism**-In this method, object is bound to the function call only at the run time. Run time polymorphism is created at the time of execution. Run time polymorphism is achieved by virtual functions.
Virtual Function- Virtual function is concept of achieving the runtime polymorphism, where the different functions or classes are bind to the program, whenever they are exactly needed

6. **Inheritance**- Inheritance means using the written code again and again. The class which is inherited is called base class or super class and the class which inherits is called derived class. So, when a derived class inherits a base class, the derived class can use all the functions which are defined in base class, hence making code reusable.
   Types of inheritance-
   a) Single Inheritance
   b) Multilevel inheritance
   c) Multiple Inheritance
   d) Hierarchical Inheritance
   e) Hybrid Inheritance(virtual inheritance)

a) Single inheritance- In single inheritance one derived class inherits the properties of one base class. It is the simplest form of inheritance.

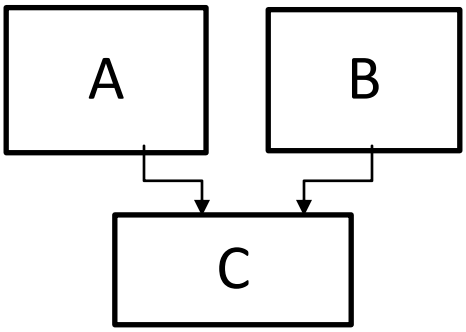


(b) Multilevel Inheritance- In multilevel inheritance, the derived class inherits the property of base class, which in turn inherits the property of some other base class. The base class
for one, is derived class for the other.

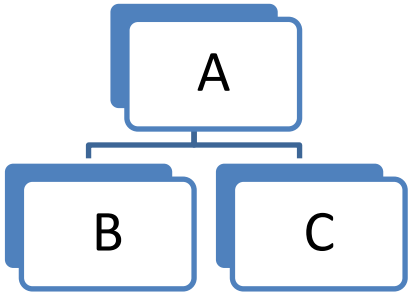


In the above fig. derived class C inherits the property of base class B where derived class B inherits the property of base class A. So we can say that in multilevel inheritance, one derived class will become the base class for some other derived class.

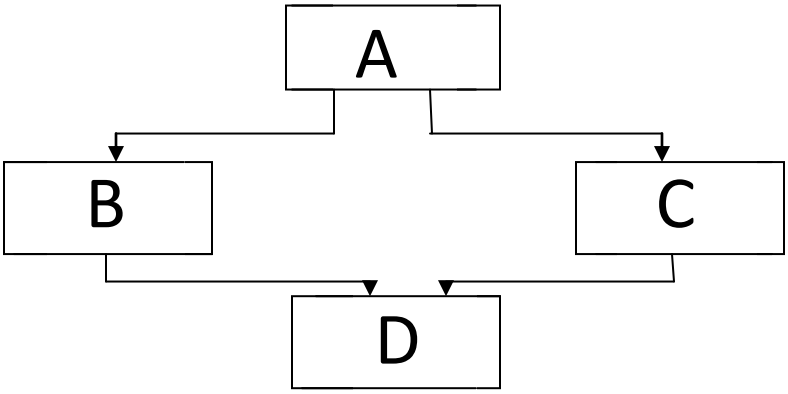
(c) Multiple Inheritance- In multiple inheritance single derived class inherits the property of two or more than two base classes.



(d) Hierarchical Inheritance- In hierarchical inheritance, multiple derived classes inherits the property of single base class.



(e) Hybrid (virtual) Inheritance- Hybrid inheritance is a combination of multiple and hierarchical inheritance.



In the above fig. A is the base class for derived class B and C, where as D is a derived class for base class B and C.

7) Overloading- Overloading is the part of polymorphism where a function or operator is made and defined many times, to perform different functions, they are said to be overloaded.

8) Exception Handling- Exception handling is a feature of OOP, to handle unresolved exceptions or errors produced at runtime.

`

Benefits of Object Oriented Programming:

1. **Modularity-** The source code for the class can be written and maintained independently of the source code for other classes. Once created, an object can be easily passed around inside the system.
2. **Information hiding-** By interacting only with an object method, the details of its internal implementation remain hidden from the outside world.
3. **Code re-use** – If a class already exists, you can use objects from that class in your program. This allows programmers to implement, test, and debug complex, task-specific objects which you can then use in your own code.
4. **Easy debugging-** If a particular object create a problem, then you can simply remove it from your application and use a different object as its replace.

Programming Styles:

a) **Procedural v/s object oriented:** Programs are made up of modules, which are parts of a program that can be coded and tested separately, and then assembled to form a complete program.

The basic difference between procedural language and object-oriented languages are-

Procedural Programming	Object-oriented Programming
1. Procedural Programming is based on Procedures and tasks	1. Object oriented Programming focus on objects rather than tasks.
2. In procedural programming, a task cans Be a smaller part of a problem	2. In object-oriented an object can be fruit, color etc.
3. Procedural Programming follows the Top-down approach.	3. Object-oriented follow the top-down as Well as bottom-up approach.
4. The code for the tasks and code for the data are grouped separately and work independently.	4. In object-oriented code for the tasks and code for the data are grouped Together and work together.
5. Examples of procedural programming are FORTRAN, COBOL and BASIC.	5. Examples of object oriented Programming is C++, VB & JAVA.

b) Programming Methodology- Programming methodology means arrangement of data structures, algorithms, hardware and software interaction and the end user.

Software development is generally carried out in two different ways-

1) Top-Down Approach
2) Bottom-Up Approach

1) Top-Down Approach- In top-down approach, overall system is designed first, without going in details of the system. In top-down approach, software testing is complicated because system is tightly bound with each other units.

2) Bottom-Up Approach- In bottom-up approach, the basic building blocks are first created and then the code is being implemented. In bottom-up approach, large programs are divided in smaller units or subprograms. So, it will become easy for the user to read and modify.

c) Generic Programming- Generic programming is a style of computer programming in which algorithms are written in terms of types-to-be-specified later that are then when needed for specific types provided as parameters.

d) Reusability- Reusability means, a segment of source code can be used again to add new functionalities with slight modification. Reusable modules and classes reduce implementation time.

e) Robustness- Robustness is the ability of a computer system to cope with errors during execution. Robustness can also be defined as the ability of an algorithm to continue operating despite abnormalities in input, calculations etc. Robustness can encompass many areas of computer science, such as Robust Programming, Robust machine learning and robust security network.

Friend Function in C++

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions. A friend can be a function, function template, or member function, or a class or class template.

To declare a friend function of a class, precede the function prototype in the class definition with keyword **friend** as follows:

```
class Box
{
  float width;
public:
  float length;
  friend void print_width ( Box box );
  void set_width(float wid );
};
```

For Example:

// This program illustrate the usages of a friend function

```
#include<iostream.h>

#include<conio.h>

class matrix

{       int i,j,m,n;

        int mat[10][10];

        public:
```

```
        void read_mat( );
        friend matrix add_mat ( matrix x, matrix y);
        void display_mat( );
};
Void matrix:: read_mat( )
{       cout<< "Enter the size of the matrix:";
        Cin>>m>>n;
        Cout<< "Enter the matrix:";
        For(i=0;i<m;i++)
        For (j=0;j<n;j++)
        {       cin>>mat[i][j];  }
        }
Void matrix::display_mat( )
        {       cout<< "The matrix is:";
                for(i=0;i<m;i++)
                {
                for(j=0;j<m;j++)
                cout<<mat[i][j]<< " ";
        }       }
Matrix add_mat(matrix x, matrix y)
        {       matrix   z;
                Int k,l;
                If((x.m==y.m)  && (x.n == y.n))
                {       z.m = x.m;
                        z.n = x.n;
                        for( k=0; k<x.m; k++)
                        {       for(l=0;l<x.n;l++)
                                z.mat[k][l] = x.mat[k][l] + y.mat[k][l];
                        }       }
                        else { cout<< "The matrix cannot be added:";
```

```
                    exit(0);  }
                    return ( z );
                    }
Void main( )
    {       matrix a,b,c;
            a.read_mat( );
            b.read_mat( );
            c= add_mat(a,b);
            c.display_mat( );
            getch( );
    }
```

The Class Constructor:

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class. A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

Rules for Constructor Definition and Usages:

1. The name of the constructor must be same as that of the class.
2. A constructor can have a parameter list.
3. A constructor function can be overloaded.
4. A constructor with no arguments is the default constructor.
5. If there is no constructor in a class, a default constructor is generated by the compiler.
6. The constructor is executed automatically.

Following example explains the concept of constructor:

Parameterized Constructor:

A default constructor does not have any parameter, but if you need, a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation as shown in the following example:

The Class Destructor:

A **destructor** is an inverse of a constructor in the sense that it removes the memory of an object which was allocated by the constructor during the creation of an object.

Rules for Destructor Definition and Usages:

1. The name of the destructor must be same as that of the class prefixed by the tilde character (~).
2. A destructor cannot have a parameter list.
3. It has no return type, not even void type.
4. Destructor cannot be overloaded i.e. there can be only one destructor in a class.
5. If there is no destructor in a class, a default destructor is generated by the compiler.
6. The destructor is executed automatically when the control reaches at the end of class scope.

C++ Templates

What is Virtual base class? Explain its uses.

When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class' name with the word virtual.

Consider following example:

```
class A
{
   public:
      int i;
};

class B : virtual public A
{
    public:
          int j;
};

class C: virtual public A
{
     public:
        int k;
};

class D: public B, public C
{
public:
```

```
    int sum;
};

int main()
{
        D ob;
        ob.i = 10; //unambiguous since only one copy of i is inherited.
        ob.j = 20;
        ob.k = 30;
        ob.sum = ob.i + ob.j + ob.k;
        cout << "Value of i is : "<< ob.i<<"\n";
        cout << "Value of j is : "<< ob.j<<"\n"; cout << "Value of k is :"<< ob.k<<"\n";
        cout << "Sum is : "<< ob.sum <<"\n";

return 0;
}.
```

Exception handling

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,** and **throw**.

- **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **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:

```
try
{
   // protected code
}catch( ExceptionName e1 )
{
   // catch block
}catch( ExceptionName e2 )
{
   // catch block
} catch (ExceptionName eN)
{
   // catch block
}
```

You can list down multiple **catch** statements to catch different type of exceptions in case your **try** block raises more than one exception in different situations.

Throwing Exceptions:

Exceptions can be thrown anywhere within a code block using **throw** statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a, int b)
{
   if( b == 0 )
   {
      throw "Division by zero condition!";
   }
   return (a/b);
}
```

Catching Exceptions:

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try
{
   // protected code
}catch( ExceptionName e )
{
  // code to handle ExceptionName exception
}
```

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows:

```
try
{
   // protected code
}catch(...)
{
  // code to handle any exception
}
```

The following is an example, which throws a division by zero exception and we catch it in catch block.

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

double division(int a, int b)
{
   if( b == 0 )
   {
      throw "Division by zero condition!";
```

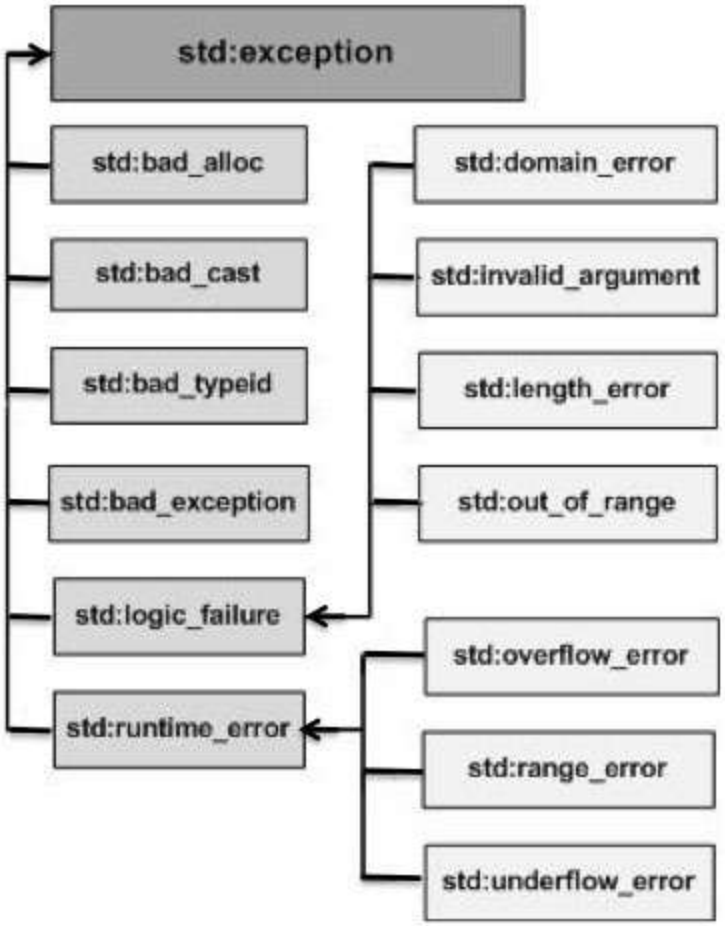
```
   }
   return (a/b);
}

int main ()
{
   int x = 50;
   int y = 0;
   double z = 0;

   try {
     z = division(x, y);
     cout << z << endl;
   }catch (const char* msg) {
     cerr << msg << endl;
   }

   return 0;
}
```

Because we are raising an exception of type **const char***, so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would produce the following result:

```
Division by zero condition!
```

C++ Standard Exceptions:

C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. These are arranged in a parent-child class hierarchy shown below:



Here is the small description of each exception mentioned in the above hierarchy:

Exception	Description
std::exception	An exception and parent class of all the standard C++ exceptions.
std::bad_alloc	This can be thrown by **new**.
std::bad_cast	This can be thrown by **dynamic_cast**.
std::bad_exception	This is useful device to handle unexpected exceptions in a C++ program
std::bad_typeid	This can be thrown by **typeid**.
std::logic_error	An exception that theoretically can be detected by reading the code.
std::domain_error	This is an exception thrown when a mathematically invalid domain is used
std::invalid_argument	This is thrown due to invalid arguments.
std::length_error	This is thrown when a too big std::string is created
std::out_of_range	This can be thrown by the at method from for example a std::vector and std::bitset<>::operator[]().
std::runtime_error	An exception that theoretically can not be detected by reading the code.
std::overflow_error	This is thrown if a mathematical overflow occurs.
std::range_error	This is occured when you try to store a value which is out of range.
std::underflow_error	This is thrown if a mathematical underflow occurs.

Define New Exceptions:

You can define your own exceptions by inheriting and overriding **exception** class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way:

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

struct MyException : public exception
{
  const char * what () const throw ()
  {
    return "C++ Exception";
  }
};

int main()
{
  try
  {
    throw MyException();
  }
  catch(MyException& e)
  {
    std::cout << "MyException caught" << std::endl;
    std::cout << e.what() << std::endl;
  }
  catch(std::exception& e)
  {
    //Other errors
  }
}
```

Pointers

Dynamic Allocation Operators New and Delete:

As we know that in a program when a variable is declared. A storage (memory) location is made available to that variable. This memory location remains available, even if it is not needed, to the program as long as it is being executed. This type of storage memory locations are called static memory location.

On the other hand a dynamic variable does not have a name and can be referenced only through a pointer.

In C++ dynamic memory can be allocated by an operator called **new**.

The general form of usage of this operator is given below.

int *ptr;

ptr= new int

A dynamic variable can be return to the system by an operator **delete.** The general form of usage of this operator is given below.

delete ptr

Once the above statement is obeyed, the dynamic variable is return back to the system and there will be no any data will be left back, who become cause for **memory bleeding.**

For Example.

The below program demonstrate the usage of operators: **new and delete.**

```
#include<iostream.h>
#include<conio.h>
void main()
{
int *ptr;
ptr= new int;
cout<< "Enter an integer:";
cin>>*ptr;
*ptr= *ptr + 5;
cout<< " The incremented value =" << *ptr;
delete ptr;
getch();
}
```