Chapter 0

Introduction :

C++, the language designed by Stroustrup is a multiparadigm language. It supports

- Procedural Decomposition
  - Programmer can divide the problem into smaller parts and conquer using functions. C++ can be used as a “better C “.
- Object Based Programming
  - Programmer can map the entities of the problem domain to the user defined types and can express the structure and behaviour of these entities in the most normal way.
- Object Oriented Programming
  - Programmer can create a hierarchy of types matching the hierarchy of the entities of the problem domain. Client programmer can choose the right type while developing programs where Server programmer can develop utilities addressing the most common type.
- Generic Programming
  - Programmer can develop routines, which can cater to a variety of types.

It is possible for a programmer to choose a combination of these paradigms while solving a given problem.

Features of C++ :.

- C++ is a superset of ‘C’. Barring a few minor differences, every program in ‘C’ is also a legal program in C++.

- C++ provides for efficient implementations. Most of the features of C++ are resolved at compile time to make the programs efficient at runtime.
- C++ is a strongly typed language. Types are enforced at compile time and there is no extra overhead at runtime. Type of a variable cannot be changed during runtime.
- C++ supports dynamic memory management. It has operators new and delete for dynamic allocation and de-allocation. Programmer has flexibility to provide his own implementation of these operators.
- Functions in C++ have the following additional features.
    - Type checking of arguments and parameters through prototype.
    - More than one function can have the same name. Names of the functions are suitably changed by the compiler.
    - Client programmer need not provide all the arguments in a function call. A few parameters can have default values.
    - Functions can be expanded at the place of call so that efficiency of the program is enhanced.
    - Parameter passing can be by value or by reference.
    - Functions can be generic so that, at compile time, it is possible to generate the function code.
- C++ provides a mechanism for user defined type, called class.
- C++ allows the programmer to provide different semantics for the operators of C++.
- C++, being an object oriented programming language, supports the following concepts.
    - Abstraction
    - Encapsulation
    - Hierarchy
    - Polymorphism
- C++ provides a better way for handling unusual cases and errors.

- C++ has standardized libraries, which will help the client programmer. A lot of time and effort can be saved by using the libraries provided.

Object Oriented Concepts :

It is difficult to map the entities of the problem domain to the entities of the solution domain. An entity of the problem domain will have a few attributes, each of these attributes will have different values at different instances, each of these can be modified by some operation.

For example, a fan has a number of blades, make, colour, cost, a number of speeds and so on. A particular fan may have 3 blades where the other may have 4 blades. A fan may be switched on or off and it may be made to rotate at a particular speed.

To represent the fan, a number of variables of various types are required. Relation between these variables can be enforced by putting them together in a structure. To make the entity complete, it should be possible to support operations on these entities.

A Class consists of variables and functions. These variables are called data members or attributes. These functions are referred to as member functions or methods. A variable of a class type is called an object. An object is an instance of the class. Invoking a member function using an object is referred to as passing a message to the object.

“Type” in a programming language stands for a range of values and set of operations. Class concept allows the programmer to make his own type.

- **Abstraction**
    - Is an external view of the object.

    - Obtained by selecting the essential features and ignoring the non-essential features.
    - Depends on the observer.

- **<u>Encapsulation</u>**
    - Putting related items together.
    - Is the implementation of abstraction
    - Distinguishes between interface and implementation
    - Exposes interface
    - Hides implementation

- **<u>Hierarchy</u>**
    - Arrange in order
    - There are two types of hierarchy
        - Inheritance or “is a”
        - Composition or “has a”
    - **<u>Inheritance</u>**
        - Used for creating a hierarchy of types.
        - Is also a re-use machansim.
        - Allows the client programmer to choose the type best suited for his problem
        - Allows the server programmer to address the most general type in his routines.
    - **<u>Composition</u>**
        - Attributes themselves are objects.
        - Is a re-use mechanism.
        - Flexible compared to inheritance.
        - Decision as regards to the type of the object can be delayed until run time.

- **Polymorphism**
    - Behaviour of the object for a given message is context sensitive.
    - Generally implies resolution of the method at runtime.
    - Provides flexibility in development of routines addressing the most general type.
    - Is a runtime overhead.

Chapter 1

Simple Input / Output Operations :

We will have a look at our first program in C++.

```
/* program ex1 */
/* Demonstration of output operation */
#include <iostream.h>

int main(void)
{
     cout << "Greetings from Kumar " << endl;
     return 0;
}
```

We include a file called iostream.h. This file has some information about the identifiers cout and endl as well as the operators <<.

As with C, C++

a) is case sensitive
b) supports free format source code
c) program consists of one or more functions
d) has a function called main where the execution starts.

C++ supports free format output operation with the help of streams. cout is variable associated with the output stream - stdout and the operator <<, called the insertion operator places the string on to the stream. This will cause the output appear on the standard output device. The identifier endl causes the output device to go to the next line as well as flush the output buffer.

Observe

a) that the operator << can be used to place any simple type values (int, double, …) on to the variable of output stream
b) that the operator << has different semantics based on the context – this is also left shift operator on integers
c) that the result of the operation is a reference to an output stream variable – this allows cascading of the operator <<.

```
/* program ex2 */
/* Demonstration of simple input / output operations */
#include <iostream.h>
int main(void)
{
     char name[20];
     int age;
     cout << "Greetings from Kumar " << endl;
     cout << "enter your name : ";
     cin >> name;
     cout << "enter your age : ";
     cin >> age;
     cout << "hello " << name << " you are " << age <<
     "year young! " << endl;
     return 0;
}
```

This program uses the extraction operator >> to take the input from the variable cin., which is a variable of input stream and places the value in a variable.

Observe

a) Operator >> requires l-value or reference to a variable on the right

b) There is no necessity of finding the address of the variable (as is done with scanf( )

c) Operaror >> is also context sensitive – has different semantics based on the context

d) that the result of the operation is a reference to an output stream variable – this allows cascading of the operator <<.

<u>Namespaces : Avoiding pollution of global namespace :</u>

Any program which solves a non-trivial problem will be generally consist of a at least a few thousand lines. We place this program in C++ in a number of files. This is required for the following reasons.

a) Declaration of variables, functions as well as definitions of type like structures, classes are placed in .h files and definition of functions and implementation of classes are placed in .cpp files.

b) Maintenance becomes better with a number of modules

c) Each module generally addresses only one concept / sub-problem – can be useful in reuse

d) Generally, program is developed by a group of programmers, each having there own set of files.

When a program is spilt across files, it is possible that the variables introduced in the global scope may clash resulting in linker time errors. To avoid the class of names in the global space, names are introduced in one or more named space by the programmers. There is a namespace called std to which the names of all standard library functions and other declarations will be introduced. These header files do not have any extension. To refer to the names introduced by these namespaces, either we can use “using declaration” or specify explicitly the namespace to which the name belongs by using scope resolution operator.

```
/* program ex3 */
/* Demonstration of using namespace  */
#include <iostream>  // observe no .h exension
using namespace std;

int main(void)
{
     char name[20];
     int age;
     std::cout << "Greetings from Kumar " << std::endl;
     cout << "enter your name : ";
     cin >> name;
     cout << "enter your age : ";
     cin >> age;
     cout << "hello " << name << " you are " << age <<
     "year young! " << endl;
     return 0;
}
```

<u>Constants and Variables :</u>

A variable has a value which can be changed where as a constant has a value which can not be changed during the program execution.

Constants in C++ have a name, type as well as a value. Definition of constants can be placed in .h files. Constant by default has internal linkage – is available in the same translation unit in which it is defined and not in other files. Named constants are used in the program for the following reasons.

a) improves readability and therefore decreases the effort for maintenance – clearly indicates what it stands for.

b) Can be modified at only one place – avoids the pitfalls of search and replace.

Let us list where all constants could be used.

a) constants in program body
b) constant parameter in functions
c) constant members in objects / structures
d) constant object through which a method is invoked.

We try use constants whenever we can. A program is said to be const-correct if constants are used meaningfully and correctly.

Variable generally has a name, a location, a value and type where type signifies the following

a) Typename provides an abstraction
b) is an encapsulation of range of values and set of operators

Let us examine a few constant definitions.

```
int x;
const int  a = 10;  // a is a constant integer
const int b;    // this is a syntax error – no value provided
int * const p = &x; // p is a const pointer to integer;
                    // should be initialized with definition
                    // value of p cannot be changed
                    // value of *p can be changed
const int * q = a;  // q is a pointer to an int which is a
                    // constant
                    // value of q can  be changed
                    // value of *q cannot be changed
```

In this example, both name and age are made constant. It is a convention to use upper case letters for names of constants.

```
/* program ex4 */
/* Demonstration of named Constants  */
#include <iostream>
using namespace std;

int main(void)
{
     const char *NAME = "INDIA";
     const int AGE = 53;
     cout  <<  "hello  "  <<  NAME  <<  "  you  are  "  <<  AGE  <<
     "year young! " << endl;
     return 0;
}
```

<u>User Defined Function :</u>

To solve complicated problems, we adopt the age-old proven method of divide and conquer. We develop the program modularly by writing a number of functions, where each function caters to one and only one aspect of the problem.

In this example, the main function is invoking a function called disp to display the name and the age of the person. The calling and the called functions need not be in the same file. To check whether the arguments match the parameters, C++ introduced the concept of prototype, which is also now available in ANSI C.

The prototype has

a) the return type
b) the name of the function
c) the list of parameters types (names do not matter).

- The compiler uses the prototype to check whether the arguments match the parameters in **number, order and type**.
- Prototype is the declaration of the function
- Should be placed in a .h file where as the implementation / definition of the function should be placed in a .cpp file.

```
/* program ex5 */
/* Demonstration of user defined function  */
#include <iostream>
using namespace std;

void disp(char *, int); // Prototytpe

int main(void)
{
     char name[20];
     int age;
     cout << ‶enter your name : ‶;
     cin >> name;
     cout << ‶enter your age : ‶;
     cin >> age;
     disp(name, age);
     return 0;
}
```

```
void disp(char *name, int age)
{
     cout << "hello " << name << " you are " << age <<
     "year young! " << endl;
}
```

Default parameter passing mechanism is “by value” where arguments to copied to the corresponding parameters created in the activation record of the called function, on the stack.

Also, observe that there is no clash between the variable names of the called and the calling functions as they are in different scope.

Function call Mechanism :

Even though C++ does not specify a particular way of implementing the function call mechanism, most of the implementations follow the convention listed below.

a) Should be the most efficient
b) Generally left to right stacking – variable number of parameters and explicit declaration are exceptions to this rule
c) Called function to cleanup the stack - code to be inserted at only one place
d) Names are decorated – function names are modified based on the number and type of parameters – also called name mangling.

Function Overloading – Static Polymorphism :

In development of programs, we generally have a hierarchy of programmers. One person may develop all utility functions and other may write programs using these utility functions. We refer to the former as the server and the latter as the client. Please note these are only relative terms.

Client using the function disp may sometime want to display only the name. It is easy for the server to provide another function for displaying the name. Then the client will have to remember which function to invoke based on his needs. If there are too many function names to remember, it is very likely that the client may go wrong.

In C++, it is possible to have more than one function with the same name where the calls are resolved at compile time based on matching the arguments to the parameters. There is no extra overhead at run time as the resolution is a compile time phenomenon. In this example, the function name disp is overloaded. One instance of display requires a character pointer whereas the other requires two parameters a character pointer and an integer.

```
/* program ex6 */
/* Demonstration of overloading of functions  */
#include <iostream>
using namespace std;

void disp(char *, int); // observe the difference in
                         // prototypes
void disp(char *);
int main(void)
{
     char name[20];
     int age;
     cout << "enter your name : ";
     cin >> name;
     cout << "enter your age : ";
     cin >> age;
     cout << ‟I want name and age : ‟ << endl;
     disp(name, age);
     cout << ‟ I want the name alone ‟ << endl;
     disp(name);
     return 0;
}

void disp(char *name, int age)
{
     cout << ‟hello ‟ << name << ‟ you are ‟ << age <<
     ‟year young! ‟ << endl;
}

void disp(char *name)
{
     cout << ‟hello ‟ << name  << endl;
}
```

A few points to note about overloading of functions in C++.

a) More than one function with the same name
b) Function call resolved at compile time
c) Functions should be declared in the same scope
d) Resolution based on the #, type & order of arguments & parameters
e) Resolution does not depend on the result type
f) Used when it is intuitive to the client
g) Used when the algorithms are different

Function Call Resolution :

Arguments are matched to the corresponding parameters based on the following rules. After each step, the compiler checks whether there is a match. If there is a match, then the function call is resolved. If there is more than one match, then there is an ambiguity and the compiler throws up an error. If there is no match, then the compiler moves to the next step.

a) Exact match / trivial convertion
b) Generic Function (templates)
c) Promotion
    1) Convertion of integral types like char, short .. . to int
    2) Convertion to float .. to double.
d) Standard Conversion
e) User defined Conversion
f) Type unsafe Conversion

Default Parameters :

There are many functions which have a large number of parameters. The client may not be interested in providing all these arguments in the call and he might be happy if the system can choose some default appropriate values for some of these parameters.

For example, a function which creates a window will have parameters indicating the size of the window (top left and bottom right corners), cursor type, font, background and foreground colours and so on. The client may not always want to provide all these information. In C++, it is possible to provide default values for parameters. This has no overhead at runtime as it is a compile time mechanism.

```
/* program ex7 */
/* Demonstration of dafault parameters  */
#include <iostream>
using namespace std;

void disp(char * = “INDIA“, int = 53 );
// observe the default values of parameters



int main(void)
{
     cout << “calling with two arguments “ << endl;
     disp(“Bangladesh“, 29);
     cout << “calling with one argument “ << endl;
     disp(“India“);
     cout << “ calling with no arguments “ << endl;
     disp();
     return 0;
}

void disp(char *name, int age)
{
     cout  <<  “hello  “  <<  name  <<  “  you  are  “  <<  age  <<
     “year young!“ << endl;
}
```

A few points to note about default parameters.

a) The parameter should have a reasonable default
b) It is a way of extending the function.

c) Specified as part of the declaration.
d) Can be specified for functions whose source codes are not available – function writer may know nothing about these defaults.
e) Only rightmost parameters can be default – all parameters can also be default
f) Leftmost arguments should be specified.
g) It is also possible to provide default values for additional parameters by building on the previous declarations.

<u>Reference Parameters :</u>

‘C’ provides only one mechanism of parameter passing – by value. Arguments are copied to the corresponding to the parameter and changes to the arguments are not reflected back in the arguments of the calling function.

In case the arguments should be changed by the called function, then a pointer to the argument is passed, thus simulating parameter passing by reference. In the called function, pointer has to be dereferenced to access the argument.

C++ supports parameter passing by value as well as parameter passing by reference. There is no necessity of dereferencing the parameter as the reference parameter is automatically dereferenced. It is not possible to make out the parameter passing mechanism by looking at the call and we have to examine the prototype of the function to make out the parameter passing mechanism.

Here the classical example of swapping two variables using reference parameters.

```
/* program ex8 */
/* Demonstration of reference parameters  */
```

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

void swap(int &, int &); // & indicates that the parameter
                         // is passed by reference

int main(void)
{
     int a = 10, b = 20;
     swap(a, b); // cannot make out the parameter passing
     cout << "a : " << a << " b : " << b << endl;
     return 0;
}

void swap(int &x, int &y)
{
     int temp;
     temp = x; // x in this case is same as a
     x = y;    // y is same as b ; no * operator
     y = temp;
}
```

A few points about parameter passing by reference :

- Required when an argument should be changed using the corresponding parameter in the called function
- Less overhead in terms of time & space at run time compared to parameter passing by value – variables of user defined types are always passed by reference
- Can avoid problems resulting in shallow copy of parameter passing by value – this happens when a structure or an object has a pointer as a member

- Can be passed a const reference to avoid changing the argument in the function
- It is also possible to return by reference when an lvalue is required in the calling function.

Pointers and Dynamic Allocation :

C++ supports dynamic allocation through two operators : new and delete. The operator new is used to allocate space where as the operator delete is used to de-allocate space.

The operator new requires a type as the operand. It allocates as much memory as required for the particular type and returns a pointer to the type specified. The operator new also works on user defined types.

```
int *p;
p = new int;
*p = 20;
double *q = new double;
*q = 2.15;
```

To remove the space allocated, delete can be used.
delete p;
delete q;

Operator new can be given an additional operand within square brackets indicating how many components are required. Then operator new allocates memory for the number of components indicated and returns a point to the block of memory. This works like a dynamic array.

int *p = new int[5];

```
for(int i = 0; i < 5; i++)
    p[i] = i;
```

To remove the block of components allocated by new [] operator, delete [] should be used.

```
delete [ ] p;
```

Alias, Garbage and Dangling Reference :

Let us examine the code down below.

```
Int *p, *q;
p = new int;
*p = 20;
q = p; // q and p refer to the same location; they are now aliases
*q = 30; // even *p would have changed; *p and *q are one and the same
           // location.
```

Both p and q point to the same location. Changing one will affect the other too.

Let us look at the following code.

```
int *p;
p = new int; // an int is created
*p = 20;
p = new int; // AHA!  One more int is created
```

When the new operator is called second time, p gets the address of the new location and its earlier value is lost. There is no way we can refer to the location created by the first call to the new operator. We refer to this as garbage. Garbage is a location without access.

Another interesting piece of code :

```
Int *p, *q;
```

```
p = new int;
*p = 20;
q = p;
delete p;
*q = 111; // Where does q point now ?
```

We make p point to the location dynamically allocated and q is made an alias of p. When delete p, whatever p points to is removed from our memory space. Now we are trying to access though q. Here we are accessing a location which is not even existing. This is called Dangling Reference.

Garbage and Dangling Reference are generally referred to as memory leak. C++ gives the programmer to play with memory; onus of managing the memory is now on the programmer.

Reference Variable :

Let us look at the code below.

```
int a = 10;
int &b = a; // b is a reference to a; b is same as a
```

At this point, no new integer location is allocated to b. Wherever b is used, it automatically refers to a.

```
cout << b << endl; // outputs 10
int c  = b; // c becomes 10
c = 20;
b = c;  // same as a = c
```

Here, a does not refer to c, but a is assigned the value of c.

A few points to note about reference variables.

a) Reference variables are always associated with another variable
    1) int &b; // syntax error
b) Reference are always tied to the same variable throughout its life.
c) References can never be undefined.

Pointers & reference :

Here, there are a few points which distinguish pointers and references.

a) Pointers may be undefined or NULL ; references should always be associated with a variable
b) Pointers may be made to point to different variables at different time; reference is always associated with the same variable throughout its life
c) Pointers should be explicitly dereferenced; References are automatically deferenced

Efficiency and Flexibility :

In C, macros are used for the following benefits.

a) The code of the macro is expanded at the place of call. Thus overhead of function call is avoided. There in no necessity of creation and destruction of activation records.
b) The parameters in macros have no type of information. They are generic. Macros provide flexibility. The same macro can cater to a number of types of arguments.
c)

In spite of these plus points, macros have a very great disadvantage. They give unusual side effects.

Macros are not recommended in C++. In stead, we use

a) inline functions to gain in efficiency

b) template functions to make the function flexible and generic.

<u>Inline function :</u>

In this program, the swap function is expanded inline. Inline functions take care of any side effects that may result due to the expansion of the code at the point of call. If the functions are recursive or has too may loops and/or selection, the compiler may decide to ignore the inline directive.

One of the serious drawbacks of this feature is that the source code has to be made part of the client program. The concept of information hiding and encapsulation suffers a serious setback here.

```
/* program ex9 */
/* Demonstration of inline parameters  */
#include <iostream>
using namespace std;

inline void swap(int &x, int &y)
{
     int temp;
     temp = x; // x in this case is same as a
     x = y;    // y is same as b ; no * operator
     y = temp;
}

int main(void)
{
     int a = 10, b = 20;
     swap(a, b); // cannot make out the function is inlined
```

```
    cout << "a : " << a << " b : " << b << endl;
    return 0;
}
```

A few points about inline function :

a) Does not change the semantics of the function
b) Is a request to the compiler
c) Has internal linkage
d) Source code gets exposed to the client
e) May result in bloating of source code
f) Unlike macros, has no side effects
g) Used for gaining efficiency in time

<u>Template function :</u>

There are many instances where the structure of the code does not change with the type. All sorting routines implement someway of comparing and rearranging elements. Function to add n integers will be similar to function adding n doubles.

In such cases, it is possible to write generic routines where type information can be passed as parameter in the calling code either explicitly or implicitly. This is used by the compiler to generate the function at compile time.

Here is an example of a swapping function which swaps variables of any type.

```
/* program ex10 */
/* Demonstration of template functions  */
#include <iostream>
```

```
using namespace std;
// keyword class or typename can be used
template <typename T> // T is a type parameter;
                       //is an identifier;
void swap(T &x, T &y)
{
     T temp;
     temp = x; // x in this case is same as a
     x = y;    // y is same as b ; no * operator
     y = temp;
}
int main(void)
{
     int a = 10, b = 20;
     swap(a, b); // causes instantiation of the template
                 // function ; Because a and b are int,
                 // T becomes int; can also call
                 // swap<int>(a, b);
     cout << "a : " << a << " b : " << b << endl;
     return 0;
}
```

A few points about template functions :

a) Mechanism to generate functions at compile time
b) No extra overhead at run time
c) Used for making functions generic
d) Used when the behaviour / algorithms of the functions are same
e) Gets implicitly instantiated by the call
f) Can also be explicitly instantiated

g) Can be specialized to take care of cases the algorithm may not be appropriate
h) Instantiated for each type only once.

Chapter 2

Structure and Class :

In ‘C’, it is possible put related items, even if they are heterogeneous, in a structure. Structures thus encapsulate related data, but it does not provide a mechanism by which we can also specify how the data can be acted upon.

Here is an example which illustrates use of a structure as it is in ‘C’. There is no direct relationship between the functions and the structures. Even the attributes of the structure can be directly manipulated in the client program.

Observe that the program is split into three files. A header file consisting of the definition of the structure Person. An implementation file where all the functions on the structure Person are defined. The client program which makes use of the structure Person and its implementation.

```
/* ex1 : Example on Structure */
/* ex1.h */
struct Person
{
     char name[20];
     int age;
};

/* ex1.cpp */
/* implementation file */
#include “ex1.h”
#include <iostream>
using namespace std;
```

```
void read(Person &z)
{
     cout << “enter name and age : “;
     cin >> z.name >> z.age;
}

void write(const Person &y) const
{
     cout << “name : “ << y.name << “ age : “ << y.age <<
endl;
}



/* main.cpp : client program */
#include “ex1.h”
#include <iostream>
using namespace std;

int main()
{
     Person p;
     read(p);
     // p.age -= 10; // is possible!
     write(p);
     return 0;
}
```

We would like to put not only the data, but also the operations on it in a single component. C++ structures support this concept. This being a new concept, we would prefer to call it “class” rather than structure. But technically there is only one difference between them. In a structure, by default, all members are public and therefore can be accessed by the client. In a class, by default, all members are private and therefore cannot be accessed by the client.

A class is mechanism for making our own type – type indicates a range of values and a set of operations. A class has variables referred to as data members or attributes. A class also has functions. These are referred to as member functions or methods.

A variable of class type is said to be an object. We say that it is an instance of the class.

Members of the class can be placed in private or public sections. Members which are public are accessible to the client. Members which are private can be accessed by member functions of the class.

```
/* ex2.h */
/* class definition file */
class Person
{
     public:
     void read();
     void write() const;
     private:
     char name[20];
     int age;
};
```

While defining the member function of the class, class name and the scope resolution operator preceeds the name of the function. This clearly binds this function to the class. Observe that the members are directly used in the function body. The functions are accessed using an object of the class by using the dot (.) operator. This is similar to the way a data member of the object is accessed. The object through which the call is made, is passed implicitly by reference to the function. This implicit pointer is “this”. To refer to the member “name” in the function, we can also say “this->name”. “this” is a keyword in C++.

```
/* ex2.cpp */
/* class implementation file */
#include <iostream>
using namespace std;
#include “ex2.h”

void Person::read()
{
     cout << “enter name and age : “;
     cin >> name >> age; // same as cin >> this->name
                         //              >> this->age;
}

/*
     object through which the call is made is a constant –
     name and age of the object pointed by this can not be
     changed in this function
*/
void Person::write() const
{
     cout << “name : “ << name << “ age : “ << age << endl;
}
```

```
/* main.cpp */
/* client program using class Person */
main()
{
     Person p; // p is an object of the class Person
     p.read(); // p implicitly passed by reference
     p.write(); // P is  passed as a constant object
     return 0;
}
```

A few points to note :

<u>Class :</u>

a) Is a mechanism for defining user defined type
b) Encapsulates data and functions
c) Provides a way of mapping from the problem domain to the solution domain
d) Same as structure, but all the members are private by default
e) An Object is an instance of the class
f) Size of the object depends only on the data members of the class and their layout and does not depend on the member functions.
g) Invoking member functions of the class is resolved at compile time and therefore no extra overhead at runtime

<u>Data member :</u>

a) Represents the attributes of the class

<u>Member Function :</u>

a) Provides behaviour for the class

Access Specifier :

- Private : default specifier for the class; can be accessed only by member functions of the class or friend functions
- Public : can be accessed from the cleint function also

  This point will be revisited while discussing inheritance.
- Protected : can be accessed from member functions of the same class or any of its publicly derived classes – cannot be accessed from the client function -

Constructors and Destructors :

It is always a good idea to initialize variables. One extreme view is that the variable be defined at a point where it can be initialized.

When we operate on files, we open the file in the beginning and we close them, at the end of the program.

An object of the class can be initialized by special function called the constructor. Any cleanup operation can be done by another special function called the destructor.

Here are a few points about constructors and destructors.

Constructor:

a) Is a member function
b) Has the same as the class
c) Is invoked when an object is created
   1) Is not invoked when a pointer is defined
   2) Is invoked when an object is created dynamically
   3) Is invoked as many times as the number of elements in an array of objects

d) Is used to initialize the object / acquire resources
e) Has no return type
f) Can have parameters
g) Can be overloaded
h) Can have default parameters
i) Once a constructor is provided, there should be constructor for all possible definitions
j) A constructor which takes no arguments is called a default constructor
k) A constructor which takes one argument is called a single argument constructor. This also acts as a conversion function.

<u>Destructor :</u>

a) Is a member function
b) Has the same as the class, preceeded by ~
c) Is invoked when an object is removed
    1) Not invoked when a pointer goes out of scope
    2) Invoked when a dynamically allocated object is deleted
    3) Invoked as many times as the number of elements in an array when array goes out of scope
d) Is used to release resources
e) Has no return type
f) Can have no parameters
g) Can not be overloaded

It is possible to have classes with only constructors and no destructors.

Here is the class definition.

```
/* ex3.h */
class Person
{
     public:
     Person(char *, int); // constructor with 2 parameters
     Person(); // default constructor
     ~Person();
     void read();
     void write() const;
     private:
     char name[20];
     int age;
};

// portion of the implementation file
// initialization list used to initialize the data members
#include “ex3.h”
Person::Person(char *s, int n) : age(n)
{
     strcpy(name, s);
}

// implementation of the default constructor
Person::Person() : age (0)
{
     name[0] = ‘\0’;
}
```

```
// destructor does not do anything meaningful in this
example
Person::~Person()
{
     cout << “object being removed “ << endl;
}
/* implementation of read and write functions not shown here
*/

/*
       This example shows the use of constructors and
destructors
   In this example, constructor is used to initialize the
   object.
*/
#include <iostream>
using namespace std;
#include “ex3.h”
int main()
{
     Person p; // default constructor called
     p.read();
     p.write();
     Person q(“India“, 53); // constructor with two
arguments called
     q.write();
     return 0;
}
```

Initialization list :

Initialization lists are used to initialize the data members of the class. They can be used only in the constructor and no other member function. It is always better to use initialization list instead of assignment in the body of the constructor. Initialization list is executed before the body of the constructor is executed.

A few points about the initialization list.

a) Used in constructor to initialize data members of the class
b) Invoked in the order of declaration in the class and not in the order of occurrence in the initialization list
c) Efficient compared to making assignment
d) Necessary to initialize const member or reference of the class

These points will be revisited while discussing inheritance.

e) Necessary to invoke the constructor of the base class
f) Base class constructors are invoked in the order of derivation and not in the order of occurrence in the initialization list
g) If a base class constructor is not specified, default base class constructor will be invoked
h) Base class constructors are invoked before the members of the class are initialized

Dynamic Memory Management using Constructors and Destructors :

In the last example, we used an array in the class to hold the name of the person. Array has the limitation that the size of the array be fixed at compile time. Instead, by using a pointer and dynamic memory allocation and de-allocation, we can manage the size of the name efficiently.

In this example, constructor allocates space and the destructor removes the space so allocated.

The class definition file is as follows.

```
/* ex4.h */
Class Person
{
     public:
     Person(char *, int);
     Person();
     ~Person();
     void read();
     void write() const;
     private:
     char *name;
     int age;
};

/* part of the class implementation file */
#include “ex4.h”
#include <cstring> // same as string.h of C

// constructor allocating space in the initialization list
Person::Person(char *s, int n) : name(new char[ strlen(s) +
1]), age(n)
{
     strcpy(name, s);
}
```

```
Person::Person() : age(0), name(0)
{
}

// destructor deallocating space
Person::~Person()
{
     delete [ ] name;
}

void Person::read()
{
     char x[80];
     cout << “enter name and age : “;
     cin >> x >> age;
     delete [] name;
     name = new char[ strlen(x) + 1];
     strcpy(name, x);
}

void Person::write() const
{
     cout << “name : “ << name << “ age : “ << age << endl;
}
```

```
/* client program */
#include “ex4.h”
int main()
{
     Person p;
     p.read();
     p.write();
     Person q(“India“, 53);
     q.write();
     return 0;
}
```

Copy Constructor :

There are instances where an object is initialized with an existing object. In such cases, a special constructor called the copy constructor gets invoked. A default copy constructor provided by the compiler does a memberwise copy. This will not work if the object has pointers.

We have shown here the portion of the code which depicts the copy constructor.

```
Class Person
{
     public:
     Person(char *, int);
     Person();
     Person(const Person &); // copy constructor
     // note the object is passed by reference
     ~Person();
```

```
    void read();
    void write() const;
    private:
    char *name;
    int age;
};
```

This is the implementation of the copy constructor. In the initialization list, memory is allocated for the name of the new object being created and age of the existing object is copied to the age of the new object. In the body of the constructor, name of the existing object is copied to the name of the new object.

```
Person::Person(const Person &rhs) : name(new
char[strlen(rhs.name) + 1]), age(rhs.age)
{
    strcpy(name, rhs.name);
}
```

Here is the client code to invoke the copy constructor.

```
int main()
{
    Person p(“India“, 53);
    p.write();
    Person q(p);  // same as Person q = p; -- no assignment
    q.write();
    return 0;
}
```

A few points about the copy constructor :

- Is a member function

- Is invoked when an object is created and
  - Is initialized with an existing object or
  - Parameter passing is by value or
  - Result is returned by value
- System generates a copy constructor by default which does memberwise copy
- Programmer is required to provide a copy constructor if the object has pointers – provide deep copy instead of shallow copy

Assignment Operator :

C++ provides the facility by which operators can be given a new semantics based on the context. One very important operator that is generally overloaded for any class is the assignment operator.

Compiler generates, by default, an assignment operator for each class which will memberwise copy. This will give problems if the object has pointers and results in garbage and dangling reference.

Whenver the class contains pointers and memory is managed dynamically, the programmer is required to provide a new definition for the assignment operator.

Here is the class definition

```
/* ex6.h */
Class Person
{
     public:
     Person(char *, int);
     Person();
     Person(const Person &);
     ~Person();
     // note that the assignment operator returns an object
of class person by r     // reference
     Person & operator=(const Person &);
     void read();
     void write() const;
     private:
     char *name;
     int age;
};
```

/* This is the implementation of the assignment function */

```
Person& Person::operator=(const Person &rhs)
{
     if(this == &rhs)
          return *this;
     delete [] name; // remove whatever existed before
     name = new char[strlen(rhs.name) + 1]; // make space
for the new name
           strcpy(name, rhs.name); // copy the name
     n = rhs.age;
     return *this;
}
```

Self assignment /* a = a; */ is first checked. If this is not taken care, delete [ ] name would result in the loss of the name of the object and the next step rhs.name would be a dangling reference.

The old value of the object (name) is first removed and new space is created for the name in the object rhs. Then rhs.name is copied to name of the object through which the call is made.

The function returns a reference to the object so that the operator= can be cascaded.

Here is the client program.

```
int main()
{
        Person p(“India”, 53);
        p.write();
        Person q(“Bangla”, 30);
        p = q;  // this is assignment and not copy constructor
        p.write();
        return 0;
}
```

A few points about assignment operator :

a) System defines an assignment operator by default, which does memberwise copy
b) Has to be a member function
c) Required to be defined when the object has pointers
d) Self assignment and storage to be taken care by the programmer in case the function is redefined

Generally, any new class developed will have the following :

a) one or more constructors
b) destructor
c) copy constructor
d) assignment operator

Such a class is said to be in “canonical form”.

Friend function :

There are instances where a function might be required to access the data members of a class even though they are private and this function cannot be made a member of this class. In such cases, we declare that this function is a friend of the class. A friend function can access the private members of the class to which it is a friend.

In this example there is a function called doctor which is a friend of the class. This function accesses the data member age even though it is not a member.

```
Class Person
{
     public:
     Person(char *, int);
     Person();
     Person(const Person &);
     ~Person();
     Person & operator=(const Person &);
     friend void doctor(Person &); // friend declaration;
               // generally placed in the public section
     void read();
     void write() const;
     private:
     char *name;
     int age;
};
```

```
// implementation of the function
void doctor(Person & p)
{
     cout << “administering viagra! “ << endl;
     p.age -= 10; // accessing private member
}

int main()
{
     Person p(“unknown”, 50);
     cout << “displaying before medicine “ << endl;
     p.write();
     doctor(p); // invoking the friend function – not
               // invoked using an object
     cout << “displaying after medicine “ << endl;
     p.write();
     return 0;
}
```

A few points about the friend function :

a) Is declared within a class , but is not part of the class
b) Is part of the interface of the class
c) Declaration can be placed anywhere within the class, but putting in public section is preferred
d) Can access the members of the class to which it is a friend
e) Required when private members of more than one class have to be accessed
f) Required when the first operand of an operator function is not an object of the class and the private members of the second object have to be accessed

g) Required when the first operand of an operator function is not an object of the same class as the second operand and the private members of the second object have to be accessed
h) Will not have this pointer unless it is also member of a class
i) Can be a member of one class and friend of another
j) Commutative operator functions are generally implemented as friend functions

Friend Class :

There are instances where a class might require to use another class. The former class may want access private members of the latter class. In this example, we are implementing a class called queue. This class has functions which would like to access the members of the class node. class node is not directly available to the client. Note the constructor of class node is private and therefore the client cannot instantiate it. The class queue is made a friend of the class node, allowing all the functions of the class queue to access the members of the class node, even if they are private to class node. The member functions of the class queue can instantiate class node as they can invoke the private constructor of class node. class node is a helper class and is an implementation detail of class queue.

Here, is the definition of the two classes.

```
/* ex7.h */
class node; // declaration
class queue
{
     public:
     queue();
     ~queue();
     void add(int); // add at the rear
     int remove(); // remove from the front
     bool empty();
     private:
     node *f, *r;  // front and rear pointers
};
class node
{
     private:
     int info;
     node *link;
     node(int); // private constructor!
     friend class queue;
};

node::node(int x) : info(x), link(0)
 // pointer grounded – always safe
{
}

queue::queue() : f(0), r(0)
// both front and rear are null when queue is empty
{
}
```

```
queue::~queue() // walk throrugh the queue to delete the
nodes
{
     node *temp = f;
     while(temp)
     {
          f = f->link;
          delete temp;
          temp = f;
     }
}

void queue::add(int x)
{
     node *temp = new node(x);
 // constructor of the class node invoked
     if(r == 0) // empty queue
          r = f  = temp;
     else // non-empty queue
          r = r->link = temp;
}

int queue::remove()
{
     node *temp = f;
     int x = f->info;
     f = f->link;
     delete temp;
     if(f == 0) r = 0;
     return x;
}
```

```
bool queue::empty()
{
     return f == 0;
}

main()
{
     queue q;
     // node z(10);   -- will be syntax error
     q.add(10);
     q.add(20);
     cout << q.remove() << endl;
     cout << q.remove() << endl;
     return 0;
}
```

A few points about the friend class :

a) Is part of the implementation
b) If class A is a friend of class B, then all the functions of class A can access members of class B
c) The relation is not symmetric
d) The relation is not transitive
e) Generally declared in the private section of the class

Operator Functions :

C++ provides flexibility to the developer so that he can make his classes reflect the way the operations are done in the domain of the application.

C++ allows the programmer to provide new semantics for the operators in C++. These are called operator functions. Already we have seen the assignment operator.

These are a few points about operator functions :

- should be intuitive to the client
- should make sense in the domain of application
- should reflect the semantics of built-in operators
- only operators of C++ with a few exceptions
- Precedence as in C++
- Association as in C++
- Rank as in C++
- At least one operand should be of user defined type
- Can be a member function, free function or friend function
- A few operators have to be member functions only

Let us experiment with a few operator functions. We implement a class called String which can hold a string. This String class is in canonical form with constructors, copy constructor, destructor and assignment operator.

Binary Operator :

We will first introduce the operator function + for concatenation of two strings. We will assume that neither of the two strings will change.

```
class String
{
     public:
     String(const char* = "" );
     String(const String &);
     ~String();
     String& operator=(const String&);
     void read()
     void write() const;

     String operator+(const String &);
     private:
     char *p;

}

String::String (const char *s) : p(new char[strlen(s) + 1])
{
     strcpy(p, s);
}

String::String    (const    String&    rhs)    :    p(new
char[strlen(rhs.p) + 1])
{
     strcpy(p, rhs.p);
}

String::~String()
{
     delete [ ] p;
}
```

```
String String::String (const String& rhs)
{
     if(this == &rhs)
          return *this;
     p = new char[strlen(rhs.p) + 1];
     strcpy(p, rhs.p);
     return *this;
}

void String::read()
{
     delete [ ] p;
     char x[20];
     cout << “enter the string : “;
     cin >> x;
     p = new char[strlen(x) + 1];
     strcpy(p, x);
}

void String::write() const
{
     cout << “value : “ << p << endl;
}
/*
     create a temporary object of class String
     find the total length of the strings combined and add 1
to it
     allocate so much of space
     copy the first string into the new object temp
     concatenate the second string
     return temp by value
*/
```

```
String String::operator+(const String &rhs)
{
     String temp;
     temp.p = new char [ strlen(p) + strlen(rhs.p) + 1];
     strcpy(temp.p, p);
     strcat(temp.p, rhs.p);
     return temp;
}
```

Let us look at the client program.

```
int main()
{
     String s1("India");
     String s2("great");
     String s3;
     s3 = s1 + s2; // same as s1.operator+(s2);
     s3.write();
}
```

Can we use the following expression in the client ?
s3 = s1 + "great";

This works fine. "great " is converted to an object of class String by the single argument constructor. Then the operator+ is invoked.

How about this ?
s3 = "India " + s2;
Since the first argument in not an object of the class, no conversion function can be applied. This will be a syntax error.

Can we make this function a friend ?

Arguments are matched to the corresponding parameters and conversion functions are applied if necessary. This works for all the three cases mentioned above.

More often than not, binary operators are implemented as friend functions.

```
// declaration friend to be included in the class
// and the member function
// operator+ to be removed
String operator+(const String &lhs, const String &rhs)
{
     String temp;
     temp.p = new char [ strlen(lhs.p) + strlen(rhs.p) + 1];
     strcpy(temp.p, lhs.p);
     strcat(temp.p, rhs.p);
     return temp;
}
```

Unary Operator ++ :

C++ provides different semantics for pre-increment and post-increment operators on simple types. In case of post increment operation, the old value is the value of the expression and then the variable is changed. In case of prefix operation, value of the variable is changed and the new value is the value of the expression.

In operator function ++, compiler plays a trick to provide a way of providing two different semantics.

A function call ++s1; will translate to s1.operator++() and
A function call s1++; will translate to s1.operator++(1);

The programmer can now overload the two functions and provide two different semantics.

```
// each character is incremented by 1 and the changed
// string is passed back as result
String& String::operator++() // pre increment
{
     l = strlen(p);
     for(int i = 0; i < l; i++)
          ++p[i];
     return *this;
}
```

Note that in preincrement operator function, no temporary object is required. The object can be passed back by reference. No necessity to invoke the copy constructor.

```
// copy of the string is made; each character is incremented by 1 and the old
// string is passed back as result
```

```
String String::operator++(int) // post increment operator
{
     String temp(*this);
     l = strlen(p);
     for(int i = 0; i < l; i++)
          ++p[i];
     return temp;
}
```

Note that a temporary object is required – constructor should be called on this object – we can not pass the result by reference – copy constructor to be invoked to return the value.

Always prefer preincrement over postincrement.

Index Operator :

The client may look upon the String as an array and may want to access each of the elements of the string in the program.

He may want to do the following.
String s(“cat”);
s[0] = “r”; // cat should become now rat
write(s);

To support this operation, we require the index operator [ ].
Here is the implementation of the function.

```
char& String::operator[ ] (int i)
{
     return p[i];
}
```

Please note the function returns a reference to char as it may appear to the left or right of assignment.

Conversion Function :

It is possible to convert from one type to another provided the conversion makes sense. Look at the following example.

```
String s("India");
cout << s << endl;
```

The client may want to display the length of the string, by converting the occurrence of s to the length of the string stored.

Here is an example of conversion of String to its length.
Observe the following peculiarity in the conversion function.

a) name of the function should be one of the types – could be user defined
b) no return type specified – return type defaults to the type, the name of the function stands for.

```
String::operator int()
{
     return strlen(p);
}
```

<u>Insertion and Extraction Operators :</u>

The client may want to treat the objects of class Person on par with any other simple type. He may want to write the code as follows.

```
Person x(“India”, 53);
cin >> x ;
cout << x << endl;
```

We are required to overload the operators of insertion and extraction. These functions have to access the members of the class Person and the first operand of the function call is not an object of the class Person. We have to make these functions friends of the class Person.

Here is the code.

```
#include <iostream>
using namespace std;
class Person
{
        public:
        :
        .
        // declarations of friend functions >> and <<
        friend istream& operator>>(istream& , Person &);
        friend ostream& operator<<(ostream& , const Person &);

        private:
        char *name;
        int age;
};

// implementation of the functions >> and <<
#include “…”
istream& operator>>(istream&  is, Person &rhs)
{
        char x[80];
        cin >> x >> rhs.age;
        delete [ ] rhs.name;
        rhs.name = new char [ strlen(x) + 1 ];
        strcpy(rhs.name, x);
        return is;
}

ostream& operator<<(ostream& os, const Person &rhs)
{
        os << “name : “ << rhs.name << “ age : “ << rhs.age << endl;
        return os;
}
```

<u>Static Members</u> :

A class may have static data members and static member functions.

A static data member belongs to the class and not to each object. All of the objects of the same class will share the same static data member. Any information related to the class can be maintained in the static data member. This can be used to control the number of objects of the class.

A static member is declared in the class definition like any other non-static data member, but it has to be defined in one of the implementation files explicitly.

A static member function can only access the static data members. A non-static data member can also access the static data member of the class.

```
class Ex
{
     public:
     static void disp();
     Ex() { ++count; } // constructor accesses the static
data member count
     ~Ex() { --count; } // destructor accesses the static
data member count

     private:
     static int count;
};

int Ex::count = 0; // definition of the static data member

void Ex::disp() // definition of the static function
{
     cout << “number of objects : “ << count << endl;
}

/*
      observe that the static function can be invoked in two
ways
   c) using classname::functionname()
   d) using objectname.functionname()
*/
```

```
/*
     This program displays the number of objects of type Ex
     at any particular instance
*/
int main()
{
     Ex::disp();
     Ex a;
     a.disp();
     Ex::disp();
     {
          Ex b;
          b.disp();
          Ex::disp();
     }
     a.disp();
     Ex::disp();
     Return 0;
}
```

Chapter 3

Inheritance:

It is possible that we may find classes which have things in common. They may share the structure or the behaviour or both. Structure refers to the data members or attributes. Behaviour is characterized by the methods. In such cases, we factor out the commonality between classes as a new class. This new class represents another higher layer of abstraction.

Let us have classes like chair and table. Both are furniture. We may make a class called furniture, which can be suitably used by the classes chair and table. This mechanism is called inheritance (subclassing, specialization). Furniture is called the base class (super class, parent class) where as the classes chair and table are called derived class (sub class, child class). Observe that the chair “is a” furniture, table “is a “ furniture.

Inheritance is used to setup a class hierarchy. As we will see later in this chapter, this allows the programmer to write the code addressing the base class. This allows the client to choose the class with right granularity.

Inheritance is also a method of reuse. But this is not its main purpose.
When do we use inheritance?

a) **The derived class should pass the litmus test of “is a”.**
2 wheeler “is an” automobile. ok
4 wheeler “is an” automobile. ok
car “is a” 4 wheeler. ok
steering wheel “is a” car . not ok
Event “is a” Date. not ok
The last two are examples of composition or “has a” hierarchy.
Car “has a” steering wheel.
Event “has a” Date – may even have more than one Date.

In such cases, using inheritance mechanism of the language is a misuse of the language construct.

b) Every derived class has to support each and every public method of its base class. A object of the derived class should be able substitute an object of the base class at all times. The base class provides a set of public methods for the client to use. It is the contract or the protocol of the class. This interface should be honoured by the derived class as well, for the concept of substitution to hold good.

   Let us have a class Bird. Let this class provide a public interface called fly(). Let the abstraction of fly be that the altitude of the bird is greater than zero when the bird flies. Can we derive Emu from Bird? The answer is no as Emu can not fly in such way that the altitude is greater than zero. This would violate the protocol of the base class.

Let us have a look at an example of inheritance.

```
Class Person
{
     public:
     Person(char *, int);
     Person();
     Person(const Person &);
     ~Person();
     Person & operator=(const Person &);
     void read();
     void write() const;
```

```
    private:
    char *name;
    int age;
};


class Employee : public Person // public derivation
{
    public:
    Employee(…);
    ~Employee();
    Employee(const Employee &);
    Employee& operator=(const Employee &);
    void readEmp();
    void writeEmp() const;
    void findsal();
    private:
    int empid;
    double sal;

};
```

Person class is called the base class and Employee class is called the derived class. Employee class inherits all the methods of the base class Person, but for constructors, destructors, copy constructor and assignment. Any of the inherited functions can be directly invoked using an object of class Employee. An object of class Employee will also have the attributes of the Person class like the name and the age. It has additional attributes like empid and sal. It also has an additional method called findsal(). This class also should implement the constructors, copy constructor, destructor and assignment operator. Generally a derived class will take care of its attributes and then forward the call to the base class.

The mechanism of inheritance discussed so far does not have any runtime overheads either in terms of space or time and is a compile time mechanism.

Constructor & Destructor :

Constructor and Destructor are not inherited. The derived class will have to provide the implementation.

When an object of the derived class is created, the control is transferred to the constructor of the derived class. The base class sub-objects are initialized in the initialization list by invoking the constructors of the base classes. These base class constructors are invoked in the order of inheritance and not in the order in which they are specified in the constructor. If the base class constructors are not explicitly invoked in the initialization list, then the default constructor of the base class will be invoked. After the base class subobjects are initialized by executing the constructors of the corresponding base classes, the data members of the derived class will be initialized in the order in which they are declared in the class. Then the body of the derived class constructor will be executed.

Destructors are executed in the reverse order of constructors. The derived class destructors will be executed before that of the base class destructors. The base class destructors will be executed in the reverse order of inheritance.

```
class Base
{
     public:
     Base() { cout << “in Constructor of the base class “
          << endl; }
     ~Base() { cout << “in Destructor of the base class “ <<
     endl; }
};

class Derived : public Base
{
     public:
     Derived()  {  cout  <<  “in  Constructor  of  the  derived
class “ << endl; }
     ~Derived()  {  cout  <<  “in  Destructor  of  the  derived
class “ << endl; }
};

int main()
{
     Derived d;
}
```

The output of this program :

```
     in Constructor of the base class
     in Constructor of the derived class
     in Destructor of the derived class
     in Destructor of the base class
```

Let us have a look another program where the base class subobject is initialized by explicitly calling the constructor of the base class in the initialization list.

```
class Base
{
     public:
     Base(int z)  : a(z) { cout << “in Constructor of the
     base class “ << endl; }
     void disp() { cout << “a : “ << a << endl; }
     private:
     int a;
};

class Derived : public Base
{
     public:
     Derived(int x, int y)  : b(y), Base(x)
     { cout << “in Constructor of the derived class “ <<
endl; }
     void disp() { Base::disp(); cout <<  “d : “ << d <<
endl; }
     private:
     int b;
};

int main()
{
     Derived o(1, 2);
     o.disp();
     return 0;
}
```

Note that the base class subobject has to be initialized in the initialization list. Putting the code like Base(x) in the body of the constructor would result in an anonymous base class object being created and will not result in the initialization of the base class subobject.

a) Base class constructors may be explicitly invoked using initialization list.
b) If the constructor of the Base class is not explicitly invoked, then the default of the Base class will be executed – is a compile time error if there is no default constructor in the Base class.
c) Order of invocation is based on the order of derivation and not based on the order of listing in the initialization list.

Copy Constructor :

Copy Constructor is not inherited. The derived class will have to provide the implementation.

```
class Base
{
     public:
     Base(int z)  : a(z) { cout << “in Constructor of the
base class “ << endl; }
     Base(const Base &rhs) : a(rhs.a) { }
     void disp() { cout << “a : “ << a << endl; }
     private:
     int a;
};
```

```
class Derived : public Base
{
    public:
    Derived(int x, int y)  : b(y), Base(x)
        { cout << “in Constructor of the derived class “
<< endl; }
    Derived(const Derived& rhs) : b(rhs.b), Base(rhs) { }
    // note how the base class constructor is invoked
    // we are passing derived class object to a base class
reference !
    void disp() { Base::disp(); cout <<  “b : “ << b <<
endl; }
    private:
    int b;
};

int main()
{
    Derived o1(1, 2);
    o1.disp();
    Derived o2(o1);
    o2.disp();
    return 0;
}
```

The object of the derived class is passed to the base class reference. This reference can access only the base class subobject of the derived class object.

Assignment :

Assignment operator is not inherited. The derived class will have to provide the implementation. Assignment operator of the derived class will copy the additional members present only in the derived class and then invoke the assignment of the base class on the base class subobject.

Since the object does not have pointers, self assignment is not checked.

```
class Base
{
	public:
	Base(int z)   : a(z) { cout << “in Constructor of the
base class “ << endl; }
	Base(const Base &rhs) : a(rhs.a) { }
	Base& operator=(const Base &rhs)
	{
		a = rhs.a;
		return *this;
	}
	void disp() { cout << “a : “ << a << endl; }
	private:
	int a;
};

class Derived : public Base
{
	public:
	Derived(int x, int y)  : b(y), Base(x)
```

```
          { cout << “in Constructor of the derived class “
<< endl; }
     Derived(const Derived& rhs) : b(rhs.b), Base(rhs) { }

     Derived & operator=(const Derived &rhs)
     {
          b = rhs.b;
          Base::operator= (rhs); // calling the base class
                               // assignment operator
          return *this;
     }
     void disp() { Base::disp(); cout <<  “b : “ << b <<
endl; }
     private:
     int b;
};
int main()
{
     Derived o1(1, 2);
     o1.disp();
     Derived o2(3, 4);
     o2.disp();
     o1 = o2; // one derived object is assigned to another
     o1.disp();
     return 0;
}
```

<u>Access Specifiers :</u>

C++ supports three access specifiers in the class – private, public and protected.

a) Private :
   1) default specifier for the class.
   2) can be accessed only by member functions of the class or friend functions.
   3) Has implementation details and are hidden from the client.
   4) Changing the structure or the behaviour of members should have very little effect on the client code.

b) Public :
   1) can be accessed from the client function
   2) part of the interface of the class
   3) should seldom be modified
   4) client code breaks down when the structure or the behaviour of members is changed

c) Protected :
   1) can be accessed from member functions of the same class or any of its publicly derived classes
   2) cannot be accessed from the client function
   3) is like public to the derived class and private to the client
   4) any modification to the members will have an impact on its derived classes and not on the client

<u>Virtual Functions and Polymorphism :</u>

Let us look at the following example. Class B inherits from class A publicly. Both have a function called f.

```
class A
{
  public:
  void f() { cout << “in base class” << endl; }
};

class B : public A
{
  public :
  void f() { cout << “in derived class” << endl; }
};
```

Let us access the functions using pointers from the client program.

```
int main()
{
  A oA, *pA;
  B oB, *pB;

  pA = &oA;
  pA->f();   // calls the function of the base class A

  pB = &oB;
  pB->f();   // calls the function of the derived class B

//pB = &oA; // will not compile
//pB->f();

  pA = &oB;
  pA->f();    // calls the function of the base class A

  return 0;
}
```

Base class pointer or reference can point or refer to a derived class object. Derived class pointer can not automatically be made to point to the base class object. Generally derived class may have additional members compared to the base class. If the derived class pointer were allowed to point to the base class object, and if the client were to try to access the additional members using this pointer, the compiler will not be able to give an error. If this is allowed, this may cause dangling reference at run time. Therefore derived class pointer cannot be made to point to base class object automatically,

In the last case, even though the pointer points to an object of the derived class, the base class function is invoked as the pointer by type is a pointer to the base class. This happens as the function calls are generally resolved at compile time.

Many a times, it is desirable to postpone the resolution of the call until run time and the function should be invoked based on the object to which the pointer points to rather than the type of the pointer. This is achieved by declaring the function to be virtual.

```
class A
{
  public:
  virtual void f() { cout << "in base class" << endl; }
};

class B : public A
{
  public :
  void f() { cout << "in derived class" << endl; }
};
```

Let us access the functions using pointers from the client program.

```
int main()
{
  A oA, *pA;
  B oB, *pB;

  pA = &oA;
  pA->f();  // calls the function of the base class A

  pB = &oB;
```

```
    pB->f();  // calls the function of the derived class B

    pA = &oB;
    pA->f();  // calls the function of the base class B

    return 0;
}
```

This concept, where the function calls are resolved at runtime based on the object to which they point to, is called dynamic polymorphism. Dynamic polymorphism is implemented in C++ using virtual functions.

Function Overriding :

When a Base class is a virtual function, the derived class can use the same without modification. If the derived class so desires provide a different implementation for the virtual function, it can do so. This mechanism of providing a definition in the derived class for a virtual function defined in the base class is called function overriding.

A few points related to overriding :

a) Should be member functions of the class
b) Only in inheritance
c) Function should be virtual
d) Signatures of the functions should be same
e) Can extend the functionality of the base class function
f) extra overhead at runtime of a per-class table and a per-object pointer
g) extra overhead at runtime of a dereferencing of pointer

The last two points are discussed in the next section.

VTBL and VPTR :

To implement polymorphic behaviour, the compiler introduces a pointer in the class definition which has one or more virtual function. The size of an object of a class with virtual function goes up by the size of a pointer. This pointer, hidden from the programmer, is generally referred to as the vptr. During the construction of the object, this pointer is made to point to a table of virtual functions called vtbl. This table, which is one for each class with virtual functions, is created at compile time. When we use virtual functions, there is an additional per object pointer and additional per class table.

Let us look at a class hierarchy with a few virtual functions to understand how vtbl and vptr support dynamic dispatch.

```
class A
{
     public:
     virtual void f1();
     virtual void f2();
     virtual void f3();
};

class B : public A
{
     public:
     void f1();
     virtual void f5();
     virtual void f6();
};

class C : public A
```

```
{
     public:
     void f1();
     void f2();
     virtual  void f7();
};

class D : public B
{
     public:
     virtual void f2();
     virtual void f6();
     virtual void f8();
}
```

Let us look at the tables for each class

class A supports three functions.

Function name	Address
f1()	A::f1
f2()	A::f2
f3()	A::f3

class B supports five functions :

Function name	Address
F1()	B::f1
F2()	A::f2
F3()	A::f3
F5()	B::f5
F6()	B::f6

class C supports four functions :

Function name	Address
F1()	C::f1
F2()	C::f2
F3()	A::f3
F7()	C::f5

class D supports four functions :

Function name	Address
F1()	B::f1
F2()	D::f2
F3()	A::f3
F5()	B::f5
F6()	D:f6
F8()	D:f8

Let us look at an example as to how function calls at resolved at runtime .

```
A *p;
p = new D;
p->f1();  // this will be changed to p->vptr->f1()
```

At the time of construction, vptr of the object of class D will point to the vtbl of class D. Using this table, at run time, the function f1 of class will be invoked.

There is an additional dereference at run time to resolve the call.

Pure Virtual Functions and Abstract Base Class :

If the base class provides only the interface of the methods, without implementation, then the virtual functions of the base class are made pure virtual

– no definition is provided for the function and the pointer to the function is grounded.. Such a class is called an Abstract Class. Abstract class can not be instantiated. A derived class becomes a concrete class only if all the pure virtual functions of the base class are overridden by this class or any of its ancestors.

A class with only pure virtual function provides interface only. Each of the derived classes will inherit the signature of the methods and override them in their classes.

Let us look at an example of geometric shapes.

The class Shape does not know which shape we might consider. Unless the geometric figure is specified as a rectangle or a circle, there is no way of reading the attributes or finding the area of the closed figure. The class Shape can only specify the signature of these functions, but cannot provide a default implementation.

```
class Shape
{
     public:
     virtual void read() = 0;  // pure virtual function
     virtual double area() = 0; // pure virtual function
};

class Rect : public Shape
{
     public:
     void read()
      {
          cout << “enter length and breadth : “;
          cin >> l >> b;
     }
     virtual double area()
     {
          return l * b;
     }
     private :
     double l, b;
};
```

This code addresses only the base class ; works fine even if new classes are derived from the class Shape as long as all of them a meaningful implementation of area.

```
double area(Shape *p[ ], int n)
{
     double sum = 0.0;
     for(int i = 0; i < n; i++)
          sum += p[i]->area(); // behaves polymorphically
     // calls the function area of the object of the
     class // to which p[i] points to.
     return area;
}

// this code requires modification each time a new
derived // class is introduced
Shape* make()
{
     return new Rect;
}

void addshapes(Shape *p[ ], int n)
{
     for(int i = 0; i < n; i++)
     {
          p[i] = make();  // makes an object of a derived //
class of class Shape and returns a pointer to the Shape //
class
          p[i] -> read(); // polymorphic
      }
 }

int main()
{
     Shape *p[10];
     addshapes(p, 10);
     cout << area(p, 10) << endl;
     return 0;
}
```

Virtual functions supporting dynamic dispatch provides flexibility to the programmer, the client code becomes more robust and does not break down when new classes are derived.

Virtual Destructors :

Whenever the constructor of the derived class is non trivial, the base class destructor should be virtual to force the execution of the derived class destructor before that of the base class.

Let us look at an example.

```
class A
{
  public:
  A() { }
  ~A() { cout << “Destructor of the base class \n” ; } //
incorrect
};

class B : public A
{
  public :
  B() : p(new int) { }
  ~B( ) { cout << “deallocating memory “ << endl; delete
p; }
  private:
  int *p;
};

int main()
{
  A *z = new B;
  delete z;
  return 0;
}
```

When the object of the class B is created, default base class constructor is executed and then the constructor of B is executed when memory is allocated for an integer and the address is put in p. When the pointer z is deleted, the destructor of the base class A will be invoked as the pointer z by type is a pointer to the base class A. As the destructor of the derived class is not executed, there will be memory leak and what the pointer p points to is not released.

.

If the destructor of the base class A is made virtual, then when delete is applied on the pointer z, polymorphic behaviour will be exhibited. The destructor of the derived class B will be invoked which in turn invoke the destructor of the base class A. Destructor of the derived class B will release memory pointed to by p and thus there will be no memory leak.

Following statement should replace the destructor code in the class A.

```
   virtual ~A() { cout << “Destructor of the base class”
} // correct
```

Multiple Inheritance :

```
Class B
{
     public:
     void f() { cout << “in B “ << endl; }
};

class C
{
     public:
     void f() { cout << “in C” << endl;
};

class D : public B, public C
{
}

D oD;
oD.f(); // error
oD.B::f(); // f of class B
```

When a class inherits from more than one class, then we say that there is multiple inheritance. Multiple inheritance can lead to ambiguity as the base classes may have the members with the same name. In this example, class D is derived from both class B and class C. Both class C and class B have a function called f. If the function is invoked using an object of class D, there is an ambiguity and the compiler flags this as an error. The reference to function can be resolved based upon the class name.

Virtual Base Classes :

Consider this example.
class B as well as class C derive from class A. class D derives from both class C and class B. Now, the base class subobject A will occur twice in D. This may not be desirable.

```
class A
{
     A(int w) : a(w) { }
     private :
     int a;
};

class B :public A
{
     public:
     B(int w, int x) : A(w), b(x) { }
     private :
     int b;
};

class C : public A
{
     public:
     C(int w, int y) : A(w), c(y) { }
     private :
     int c;
};

class D : public B, public C
```

```
{
     public:
     D(int w, int x, int y, int z) : B(w, x), C(w, y), d(z)
{ }
     private :
     int d;
};
```

An object of class D will have the following structure.

D oD;

oD

Base class subobject of class B	a
Also has base class sub object of class A	b
Base class subobject of class C	a
Also has base class sub object of class A	c
Member of class D	d

Observe that the base class subobject of class A appears twice in an object of class D. In case the client is interested in only one subobject of class A, he should use virtual derivation of class A and then class A is called a virtual class.

Inheritance, by default, follows the value semantics, where the base class subobject is embedded into the object of the derived class. When virtual inheritance is used, reference semantics is followed and each derived class object will have a pointer to the base class subobject.

Let us look at the change in code as well as the structure of object of class D.

```
class A
{
     A(int w) : a(w) { }
     private :
     int a;
};

class B : virtual public A
{
     public:
     B(int w, int x) : A(w), b(x) { }
     private :
     int b;
};

class C : public virtual A
{
     public:
     C(int w, int y) : A(w), c(y) { }
     private :
     int c;
};

class D : public B, public C
{
     public:
     D(int w, int x, int y, int z) : B(w, x), C(w, y), A(w),
d(z) { }
     private :
     int d;
};
```

An object of class D will have the following structure.

D oD;

oD

Base class subobject of class A	a
Base class subobject of class B	&a
With a reference to base class sub object of class A	b
Base class subobject of class C	&a
With a reference to base class sub object of class A	c
Member of class D	d

Now, we can observe that there is only one base class object.

Generally, the constructor of a class will invoke the constructors of its immediate predecessor classes. Because there is only one instance of class A, who should call the constructor of class A? Most derived class, in this example, class D, will have to explicitly invoke the constructor of class A. In this case, Class B and Class C will not invoke the constructor of the class A.

<u>Type Casting :</u>

Type casting is used to convert the value of an expression from one type to another. C++ supports four types of casting or coercion operators.

a) const_cast

This is used to remove constness or volatileness of a data item. If there is a function which requires a pointer to an int and our program has a constant int, then we can write the code as follows.

```
void  f(int *p);

const int a = 10;
// f(&a);  // error : pointer to a constant cannot be passed to a pointer to a
```

```
                    // variable
f( const_cast<int *>(&a)); // fine
```

b) static_cast

This is same as the type casting of ‘C’.

```
int n = static_cast <int> (3.14);
```

The above statement explicitly converts the double 3.14 to an int.

c) dynamic_cast

This is used in converting the base class pointer to a derived class pointer. Conversion from derived class pointer to the base class pointer is automatic, but not the other way. This is also referred to as down casting.

d) reinterpret_cast :

This is a type unsafe cast. This is seldom used.

Run Time Type Identification (RTTI):

How do we query a base class pointer at runtime to determine to object of which class does it point to ? Is it possible to invoke a method of the derived class not found in the base class after ascertaining as to where the pointer points to.

It is possible with RTTI. There are two approaches.

a) typeid operator :

   typeid is an operator of C++. When applied on an expression or a typename, it instantiates an object of the class type_info. This class is implementation dependent. It is guaranteed to provide the following .

   i) a method called name() which gives the name of the type
   ii) operator == to compare two typeid objects
   iii) operator != to compare two typeid objects

   This type_info class can not be instantiated by the client.

   Let us see how to use typeid to find the type of an expression at runtime.

```
# include <typeinfo>
class A
{
     public:
     :
     .
     virtual ~A();
};

class B : public A
{
     public:
     void f() { } // only in B
};
```

```
void func(A *p)
{
     if(typeid (*p) == typeid(B)) // if *p is an object
of class B …
          p->f();
}
```

Observe that the calling is safe, as we have checked whether p is a pointer to an object of class B before invoking the function f.

b) Dynamic cast :

Dynamic cast is used to convert a pointer to the base class to a pointer to the derived class in the class hierarchy. The pointer will be 0 if the operation fails. If the operation is successful, the pointer will pointer to an object of the derived class.

Here is an example of how to use dynamic cast.

```
class A
{
     public:
     :
     .
     virtual ~A();
};

class B : public A
{
     public:
     void f() { } // only in B
```

```
    };

    void func(A *p)
    {
```
/* convert base class pointer to a pointer to the derived class; pB will be a pointer to an object of the derived class, if casting succeeds; otherwise it will be 0.*/
```
        B *pB = dynamic_cast<B *>(p);
        If  (pB)    //  check  whether  the  casting  was
    successful
            pB->f();
  }
```

<u>Composition</u> :

Composition is a reuse mechanism. A class contains objects as attributes. There are 3 ways of embedding the objects in a class.

a) embed by value
   1) Life of the attribute synchronized with the object
   2) Any change to the embedded object also affects the class of the embedding object

b) embed by reference
   1) Life of the attribute not synchronized with the object
   2) Should always exist
   3) Can not be changed during the life of the object

c) embed by pointer

   1) Life of the attribute not synchronized with the object
   2) Can be null
   3) Can point to number of objects
   4) Can point to different objects at different times.

```
class Person
{
        private:
        Name name;
        Address &addr;
        Phone *phone;

};
```

a) Every person has a name. Life of the attribute name is same as that of the object of class Person. Name is a good candidate for embedding by value.
b) Every person will have an address. But, it is possible that the address is shared by all persons living under the same roof. Addr is a good candidate for embedding by reference.
c) Every person may or may not have a phone. A person may also have more than one phone. Phone is a good candidate for embedding by pointer.

Chapter 4

Class Templates :

There are many data structures which have certain properties independent of the components they contain.

A queue has two ends referred to as the front and the rear. We add from the rear and remove from the front. It does not matter what type of component we put in the queue.

A Stack is a data structure with only one end called the stack top. It supports two operations : push and pop. The operation push is used to add an element to the top of the stack and the operation pop is used to remove an element from the top of the stack.

We may have stacks of different component types at different times in the same application or in different applications. For example, a stack of operators is required to convert an infix expression to postfix expression and a stack of operands is required to evaluate a postfix expression. Can we parameterize the class definition and implementation so that the type of the component is specified at the compile time by the client ?

We can do that by using template classes.

Let us first implement of a stack of integer. All the member functions have been defined within the body of the class. All these functions become inline by default.

The stack is simulated using a dynamic array created at the time of construction. The maximum number of elements is specified by the object.

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

class Stack
{
     public:
     Stack(int m =  10) : p(new int[n]),  n(m)
     {
          sp = p;
     }
     ~Stack()
     {
          delete [] p;
     }
     void push(int x)
     {
          *sp++ = x;
     }
     int pop()
     {
          return *--sp;
     }
     bool full()
     {
          return sp == p + n;
     }
     bool empty()
     {
          return p == sp;
     }
     private:
     int *p, *sp; // sp is the stack pointer
     // p does not change after initialization
     int n;
};

/* this is the client program */
int main()
{
     int n;
     cout << “enter size of the stack : “;
     cin >> n;
     stack s(n);
     for(int i = 0; i < n; i++)
          s.push(i);
     while (! s.empty())
          cout << s.pop() << endl;
     return 0;
}
```

This client program will push n elements to the stack and pop them out.

Let us now convert this class into a template class.
We instantiate the class explicitly at the time of definition. Here are some examples.

Stack<int> s(10); // stack of integer
Stack<double> s1(10); // stack of double
Stack < Person> p(5); // stack of user defined class Person

The type specified should match the parameter specified in the template declaration. Once the class is a template, all the functions in the class also become template functions.

The class as well as the functions are generated at compile time based on the definition of the object in the client code. To carry out this process, the code has to be exposed to the client.

This feature has no extra overhead at runtime.

// T is a parameter which stands for a typename

```
template <class T> // can use the keyword class or typename
class Stack
{
     public:
     Stack(int m =  10) : p(new T[n]),  n(m)
     // component type is T
     {
          sp = p;
     }
     ~Stack()
     {
          delete [] p;
     }
     void push(T x)
     {
          *sp++ = x;
     }
```

```
    T pop()
    {
        return *--sp;
    }
    bool full()
    {
        return sp == p + n;
    }
    bool empty()
    {
        return p == sp;
    }
    private:
    T *p, *sp;
    int n;
};


int main()
{
    stack<int> s(4);
    for(int i = 0; i < 4; i++)
        s.push(i);
    while (! s.empty())
        cout << s.pop() << endl;
    return 0;
}
```

Observe that the following are required for this code work on user defined types.

a) default constructor : required for the constructor of the class stack to create an array of objects on the heap.
b) Copy constructor : required for push to receive an object by value as well as pop to return by value. We can modify both the functions to always use references.
c) Assignment Operator : required for carrying out the assignment operation in the push function.

Let us implement the functions outside the body of the class definition.

```
template <class T>
class Stack
{
     public:
     Stack(int =  10);
     ~Stack();
     void push(T x);
     T pop();
     bool full();
     bool empty();
     private:
     T *p, *sp;
     int n;
};
```

```
template <class T>
Stack<T>::Stack(int m =  10) : p(new T[n]),  n(m)
{
     sp = p;
}

template <class T>
Stack<T>::~Stack()
     {
          delete [] p;
     }

template <class T>
void Stack<T>::push(T x)
{
     *sp++ = x;
}
template <class T>
T Stack<T>::pop()
{
     return *--sp;
}

template <class X>
bool Stack<X>::full()
     {
          return sp == p + n;
     }
```

```
template <class T>
bool Stack<T>::empty()
{
     return p == sp;
}

int main()
{
     stack<int> s(4);
     for(int i = 0; i < 4; i++)
          s.push(i);
     while (! s.empty())
          cout << s.pop() << endl;
     return 0;
}
```

Observe that whenever a class name has to be specified, it has to be qualified by the parameterized type name within angle brackets.

Exception Handling :

A program may encounter unusual conditions during execution. This unusual condition could be an error. There are many ways of taking action when such a situation arises

.

a) Abort the program

   Not always a good idea – the program should follow the concept of graceful degradation.

b) set a flag

   All the code to be executed after this step should check the flag.

c) set an error number

    A new error will cause the earlier value to be removed.

d) exception handling

    Separate the normal and error handling codes.

Basic concepts of the exception handling are :

a) separate the normal code and exception handling code
b) improves readability of the program
c) concentrate on the normal flow for optimization – generally executed 95 % of time where as the exception handler is executed 5 % time.
d) Exception handler could be dynamically far away from the place where the exception is generated – it is not necessary for each routine to handle all the exceptions generated.

In C++, try block should enclose the code where an exception is likely to occur. The scope of the try block is dynamic – has scope over all the functions invoked from within the try block. Within the dynamic scope of the try block, there could be exceptions thrown. The keyword throw is followed by an expression. Try block is followed by one or more catch blocks. Catch blocks are like function definitions. The object thrown by the throw statement is matched against the parameter of the catch blocks in the order of occurrence. The catch block whose parameter matches the type of expression thrown by throw command will be executed and the control is transferred to the end of the try block. Please note the control is not transferred back to the point where the exception arose. If none of the catch blocks matches the expression thrown, then the program is exited.

<u>Simple Exception Objects</u> :

In this example , simple expressions of type int and double are thrown when the marks read is not in the valid range.

```
int main()
{

     int marks;
     try
     {
          cin >> marks; // check for marks
          if(marks < 0)
               throw 1; // throw an int
          if (marks > 100)
               throw 1.0; // throw a double
          cout << “marks : “ << marks << endl;
     }
     catch(int)
     {
          cout << “marks too low “ << endl;
     }
     catch(double)
     {
          cout << “marks too high “ << endl;
     }
     return 0;
}
```

<u>Exception Objects with value :</u>

In this example, both the throws are of the same type, but have different int value. In the catch block, by using selection, we can make out the different unusual cases.

```
int main()
{

     int marks;
     try
     {
          cin >> marks;
          if(marks < 0)
               throw 1;
          if(marks > 100)
               throw 2
          cout << “marks : “ << marks << endl;
     }
     catch(int z)
     {
          if(z == 1)
               cout << “marks too low “ << endl;
          else if(z == 2)
               cout << “marks too high “ << endl;
     }
     return 0;
}
```

Dynamic scope of Try Block :

This illustrates the concept of dynamic scope of the try block. When an exception is thrown in the function readmarks, invoked from within the try block in main, the control is transferred to the catch blocks following this try block.

```
void readmarks(int &marks);

int main()
{

     int marks;
     try
     {
          readmarks(marks); // dynamic scope
          cout << “marks : “ << marks << endl;
     }
     catch(int z)
     {
          if(z == 1)
               cout << “marks too low “ << endl;
          else if(z == 2)
               cout << “marks too high “ << endl;
     }
     return 0;
}
```

```
void readmarks(int &marks)
{
          cin >> marks;

          if(marks < 0)
               throw 1;
          if(marks > 100)
               throw 2;

}
```

Exception Objects from classes :

Exception objects need not be simple expressions. They can as well be objects of some class as this example shows.

```
class low{};
class high{};

void readmarks(int &marks);

int main()
{

     int marks;
     try
     {
          readmarks(marks);
          cout << “marks : “ << marks << endl;
     }
     catch(low &)
     {

               cout << “marks too low “ << endl;
     }
     class(high &)
     {

               cout << “marks too high “ << endl;
     }
     return 0;
}

void readmarks(int &marks)
{
          cin >> marks;

          if(marks < 0)
               throw low();
           // invoking the constructor of class low
          if(marks > 100)
               throw high();
          // invoking the constructor of class high

}
```

Exception classes in a class hierarchy :

These classes can also be in a class hierarchy.
Class low and class high are derived from the class except. This class has a pure virtual function called disp which are overridden by the classes low and high. Objects of class low and high are captured by a reference to class except.

Remember base class reference can refer to any derived class object. Virtual function disp with dynamic dispatch will invoke the function of the class to which the base class reference refers.

```
class except
{
     public:
     virtual void disp() = 0;
};

class low
{
     public:
     void disp() { cout << “too low “ << endl; }
};
class high
{
     public:
     void disp() { cout << “too high “ << endl; }

};

void readmarks(int &marks);

int main()
{
     int marks;
     try
     {
          readmarks(marks);
          cout << “marks : “ << marks << endl;
     }
     catch(except &e) // base class reference
     {
          e.disp();
     }
     return 0;
}

void readmarks(int &marks)
{
          cin >> marks;

          if(marks < 0)
               throw low();
          if(marks > 100)
               throw high();

}
```

Propagation of Exception – managing memory :

When an exception is thrown not physically contained in the try block, stack is unwound as the functions are exited until the catch blocks following the try block are reached. When the stack is unwound, all objects on the stack are removed and the corresponding destructors are invoked.

But, if the function has local pointers using which space has been allocated, then the pointer is removed without releasing the space resulting in memory leak.

This can be avoided by having a “catch all catch block” which catches the exception, releases the memory and then throws back the same exception.

```
void readmarks(int &marks);

int main()
{
     int marks;
     try
     {
          readmarks(marks);
          cout << “marks : “ << marks << endl;
     }
     catch(int z)
     {
          if(z == 1)
               cout << “marks too low “ << endl;
          else if(z == 2)
               cout << “marks too high “ << endl;
     }
     return 0;
}

void readmarks(int &marks)
{
          cin >> marks;
          int *p = new int[10];
          try
          {
               if(marks < 0)
                    throw 1;
               if(marks > 100)
                    throw 2;
          }
          catch(…)// catch all
          {
               delete [] p;
                // exception flow :release memory
               throw ; // rethrow the exception
          }
          delete [ ] p; // normal flow : release memory

}
```

Nested Try Blocks :

Here is an example of nested try blocks. When an exception thrown in the inner block is not handled in that block – no catch block found for the type of exception thown – then the exception is propogated to the outer block. If the exception is not handled even in the outer most block, then the program is exited.

```
int main()
{

     int marks;
     try
     {
          cin >> marks;
          if(marks == 50 )
          throw (“fifty:fifty“); // not handled at all
          try
          {
               if(marks == 0)
                    throw(“AHA!”); //inner block
               if(marks == 100)
                    throw 1; // outer block
          }
          catch(char *)
          {
               cout << “great performance! “ << endl;
          };
          if(marks < 0)
               throw 1;
          if marks > 100)
               throw 1.0;
          cout << “marks : “ << marks << endl;
     }
     catch(int)
     {
          cout << “marks too low “ << endl;
     }
     catch(double)
     {
          cout << “marks too high “ << endl;
     }
     return 0;
}
```

<u>Books for Reference :</u>

1. The Practice of Programming – Brian W. Kernighan & Bob Pike
2. Effective C++ - Scott Meyers
3. More Effective C++ - Scott Meyers
4. Large Scale C++ Software Design – John Lakos
5. STL Tutorials & Reference Guide – David R Musser & Atul Saini
6. Design Patterns – Gamma et al
7. C++ Strategies & Tactics – Robert B Murray
8. C++ Programming Style -- Tom Cargill
9. Inside the C++ object Model – Lippman
10. Essential C++ -- Lippman
11. Advanced C++ - James Coplien