Section B: Classes and Data Abstraction

Data Abstraction

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.

Now, if we talk in terms of C++ Programming, C++ classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this:

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

int main( ) {
   cout << "Hello C++" <<endl;
   return 0;
}
```

Here, you don't need to understand how **cout** displays the text on the user's screen. You need to only know the public interface and the underlying implementation of cout is free to change.

Access Labels Enforce Abstraction

In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels:

- Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
- Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Benefits of Data Abstraction

Data abstraction provides two important advantages:

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data are public, then any function that directly accesses the data members of the old representation might be broken.

Data Abstraction Example:

Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example:

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

class Adder {
  public:
    // constructor
    Adder(int i = 0) {
      total = i;
    }

    // interface to outside world
    void addNum(int number) {
      total += number;
    }

    // interface to outside world
    int getTotal() {
      return total;
    };

  private:
    // hidden data from outside world
    int total;
};

int main( ) {
  Adder a;

  a.addNum(10);
  a.addNum(20);
  a.addNum(30);

  cout << "Total " << a.getTotal() <<endl;
  return 0;
}
```

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

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that the user doesn't need to know about, but is needed for the class to operate properly.

Structure

A structure is a collection of dissimilar data types.

```
struct book
{
char name ;
float price ;
int pages ;
} ;
struct book b1, b2, b3 ;
```

Here b1, b2 and b3 are structure variable of type book. And name, price and pages are called structure members. To access a structure member we need dot(.) operator.

b1.name - name of book b1.
b1.price – price of book b1.
b1.pages – price of book b1.

Dynamic Memory Management

There are two ways to allocate memory:

(i) Static memory allocation

(ii) Dynamic memory allocation

(i) Static memory allocation

To allocate memory at the time of program compilation is known as static memory allocation.
i.e. int a[10];

it allocates 20 bytes at the time of compilation of the program. Its main disadvantage is wastage or shortage of memory space can takes place.

(ii) Dynamic memory allocation

To allocate memory at the time of program execution is known as dynamic memory allocation.C++ provides two dynamic allocation operators: **new** and **delete**. These operators are used to allocate and free memory at run time. Dynamic allocation is an important part of almost all real-world programs. These are included for the sake of compatibility with C. However, for C++ code, you should use the new and delete operators because they have several advantages. The new operator allocates memory and returns a pointer to the start of it. The delete operator frees memory previously allocated using new.

The general forms of new and delete are shown here:

```
p_var = new type;

delete p_var;
```

Here, p_var is a pointer variable that receives a pointer to memory that is large enough to hold an item of type type.

Memory Allocation to an integer.

Solution:

```
#include <iostream.h>
void main()

{

        int
        *p;
        try

         {

                p = new int;
```

```
          }

          catch (bad_alloc xa)

          {

                    cout << "Allocation Failure\n";

                    return 1;

          }

          *p = 100;

          cout << "At " << p << " ";

          cout << "is the value " << *p <<
          "\n"; delete p;

}
```

This program assigns to **p** an address in the heap that is large enough to hold an integer. It then assigns that memory the value 100 and displays the contents of the memory on the screen. Finally, it frees the dynamically allocated memory. Remember, if your compiler implements **new** such that it returns null on failure, you must change the preceding program appropriately.The **delete** operator must be used only with a valid pointer previously allocated by using **new**. Using any other type of pointer with **delete** is undefined and will almost certainly cause serious problems, such as a system crash.

this Pointer

Every object in C++ has access to its own address through an important pointer called this pointer. The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a **this** pointer, because friends are not members of a class. Only non static member functions have a **this** pointer.

Write a program to demonstrate this pointer.

Solution:

```
#include <iostream.h>
class Box

{

    private: double
    length; double
    breadth; double
    height;

    public:

    Box(double l=2.0, double b=2.0, double h=2.0)

    {

    cout <<"Constructor called." <<
    endl; length = l;

    breadth = b;
    height = h;

    }
```

```
            double Volume()
            {
            return (length * breadth * height);
            }
            int compare(Box box)
             {
            return (this->Volume() > box.Volume());
             }
};
void main(void)
{
             Box Box1(3.3, 1.2, 1.5);
            Box Box2(8.5, 6.0, 2.0);
             if(Box1.compare(Box2))
```

```
        {
        cout << "Box2 is smaller than Box1" <<endl;
        }
        else
        {
        cout << "Box2 is equal to or larger than Box1" <<endl;
        }
 }
```

Output:

```
Constructor called.
Constructor called.
Box2 is equal to or larger than Box1
```

Function Overloading

It is the process by which a single function can perform different task, depending upon no of parameters and types of parameters.

Write a program to overload function area () to calculate area of circle and area of a rectangle.

Solution:

```
#include <iostream.h>
float area(int);


int area(int, int);
void main( )


{


      int r, l, b;


      cout << “Enter the Value of r, l & b:
      ”; cin>>r>>l>>b;


      cout<< “Area of circle is
      ”<<area(r)<<endl; cout<< “Area of
      rectangle is ”<<area(l,b);


}


float area(int a)


      return (3.14*a*a);


}
```

```
int area(int a, int b)

{

        return (a*b);

}
```

Output:

```
Enter the Value of r, l &
b: 7 8 6

Area of circle is
153.86 Area of circle
is 48
```

Ambiguity in Function Overloading

Suppose we have two functions:

```
void area(int,int);

void area(float,int);
void main()

{

        area(10,10);            // Unambiguous function call, calls area(int, int){ }

        area(10.0,10);          // Ambiguous function call, error!
}
```

Note: Here, the second area() function will not compile and will generate error ambiguity between area(int,int) and area(float, int) . It's because 10.0 is treated as a double, not a float. Either of our functions could accept a double, but our compiler doesn't know which one you want to use.

Following functions cannot be overloaded:

```
void f(int x);
void f(int
&x);
```

Above two functions cannot be overloaded when the only difference is that one takes a reference parameter and the other takes a normal, call-by-value parameter.

```
typedef int integer; enum
days{mon,tue,wed} void
f(int);

void f(mon);
```

Friend Classes

It is possible for one class to be a **friend** of another class. When this is the case, the **friend** class and all of its member functions have access to the private members defined within the other class.

```
#include <iostream.h>
class TwoValues

{
        int a;
        int b;
        public
        :


        TwoValues(int i, int j)


        {


        a =
        i; b
        = j;


        }


        friend class Min;


};


class Min


{


        public:


        int min(TwoValues x);


};


int Min::min(TwoValues x)
```

```
{

        return x.a < x.b ? x.a : x.b;

}

int main()

{

        TwoValues ob(10,
        20); Min m;

        cout <<
        m.min(ob); return
        0;
}
```

Output:

```
10
```

Note: In this example, class Min has access to the private variables a and b declared within the TwoValues class.

Static Data Members

The data member of a class preceded by the keyword static is known as static member.

When we precede a member variable's declaration with **static**, we are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Hence static variables are called class variables.

Unlike regular data members, individual copies of a static member variable are not made for each object. No matter how many objects of a class are created, only one copy of a **static** data member exists. Thus, all objects of that class use that same variable.

All **static** variables are initialized to zero before the first object is created.

Normal data members are called object variable but static data members are called class variables.

Demonstration of static data members.

Solution:

```
#include<iostream.h>
class A


{


        int p; static
        int q;
        public:
        A();


        void incr(void);
        void
        display(void);


};
```

```
A :: A()

{

            p=5;

}

int A:: q=10;
void A::
incr()

{

            p++;

            q++;

}

void A:: display()

{

            cout<<p<<"\t"<<q<<endl;

}
```

```
void main()

{


        A a1, a2,
        a3;
        a1.incr();
        a1.display()
        ; a2.incr();
        a2.display()
        ; a3.incr();
        a3.display()
        ;


}
```

Output:

```
6       11

6       12

6       13
```

Note: Here p is a normal variable, whose value is 5 for all 3 objects a1, a2 and a3 (For each object, separate copy of p exists). But q is static variable or member, whose initial value is 10 and a single copy of q exists for all the objects.

Constructor

A constructor is a special member function whose task is to initialize the object of a class. Its name is same as the class name.

A constructor does not have a return type.

A constructor is called or invoked when the object of its associated class is created. It is called constructor because it constructs the values of data members of the class. A constructor cannot be virtual (shall be discussed later on).

A constructor can be overloaded.

There three types of constructor:

(i) Default Constructor

(ii) Parameterized Constructor

(iii) Copy constructor

Default Constructor

The constructor which has no arguments is known as default constructor.

Demonstration of default Constructor.

Solution:

```
#include<iostream.h> class Add

{
        int x, y, z;

        public:

        Add(); // Default Constructor void calculate(void);
        void display(void);
};
Add::Add()
{
        x=6;
        y=5;
}

void Add :: calculate()
{
        z=x+y;

}

void Add :: display()

{
        cout<<z;
}

void main()

{

        Add a; a.calculate();

        a.display();
```

```
}
```

Output:

```
11
```

Note: Here in the above program when the statement Add a; will execute (i.e. object is created), the default constructor Add () will be called automatically and value of x and y will be set to 6 and 5 respectively.

Parameterized constructor

The constructor which takes some argument is known as parameterized constructor.

Write a program to initialize two integer variables using parameterized constructor and add them.

Solution:

```
#include<iostream.h>
class Add

{

        int x, y, z;
        public:
        Add(int,
        int);
```

```
            void
            calculate(void);
            void display(void);

};

Add :: Add(int a, int b)

{

            x=a;

            y=b;

}

void Add :: calculate()

{

            z=x+y;

}

void Add :: display()

{
        cout<<z;

}

void main()

{

        Add a(5, 6);
        a.calculate();
        a.display();

}
```

Output:

```
11
```

Note: Here in the above program when the statement Add a(5, 6); will be executed (i.e. object creation), the parameterized constructor Add (int, int) will be called automatically and value of x and y will be set to 5 and 6respectively.

A parameterized constructor can be called:

(i) **Implicitly:** Add a(5, 6);

(ii) **Explicitly :**Add a=Add(5, 6);

If the constructor has one argument, then we can also use object-name=value-of-argument; instead of object-name (value-of-argument); to initialize an object.

What is Constructor Overloading?

If a program contains more than one constructor, then constructor is said to be overloaded.

Destructor

It is a special member function which is executed automatically when an object is destroyed. Its name is same as class name but it should be preceded by the symbol ~.

It cannot be overloaded as it takes no argument.

It is used to delete the memory space occupied by an object. It has no return type.

It should be declared in the public section of the class.

Demonstration of Destructor.

Solution:

```
#include<iostream.h>
class XYZ

{

        int x;
        public:
        XYZ( );
        ~XYZ( );

        void display(void);
```

```
}; XYZ::XYZ(
)

{

        x=9;

}

XYZ:: ~XYZ( )

{

        cout<<"Object is destroyed"<<endl;

}

void XYZ::display()

{

        cout<<x;

}

void main()

{
        XYZ xyz;
        xyz.display();


}
```

Output:

```
9

Object is destroyed.
```

Class scope and Accessing Class Memebers

A class's data members (variables declared in the class definition) and member functions (functions declared in the class definition) belong to that class's scope. Nonmember functions are defined at file scope.

Within a class's scope, class members are immediately accessible by all of that class's member functions and can be referenced by name. Outside a class's scope, public class members are referenced through one of the handles on an objectan object name, a reference to an object or a pointer to an object. The type of the object, reference or pointer specifies the interface (i.e., the member functions) accessible to the client.

Member functions of a class can be overloaded, but only by other member functions of that class. To overload a member function, simply provide in the class definition a prototype for each version of the overloaded function, and provide a separate function definition for each version of the function.

Variables declared in a member function have block scope and are known only to that function. If a member function defines a variable with the same name as a variable with class scope, the class-scope variable is hidden by the block-scope variable in the block scope. Such a hidden variable can be accessed by preceding the variable name with the class name followed by the scope resolution operator (`::`). Hidden global variables can be accessed with the unary scope resolution operator.

The dot member selection operator (`.`) is preceded by an object's name or with a reference to an object to access the object's members. The arrow member selection operator (`->`) is preceded by a pointer to an object to access the object's members.

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Let us take previously defined class to access the members of the class using a member function instead of directly accessing them:

```
class Box {
   public:
      double length;         // Length of a box
      double breadth;        // Breadth of a box
      double height;         // Height of a box
      double getVolume(void);// Returns box volume
};
```

Member functions can be defined within the class definition or separately using **scope resolution operator, ::**. Defining a member function within the class definition declares the function **inline**, even if you do not use the inline specifier. So either you can define **getVolume()** function as below:

```
class Box {
   public:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box

      double getVolume(void) {
         return length * breadth * height;
      }
};
```

If you like you can define same function outside the class using **scope resolution operator, ::** as follows:

```
double Box::getVolume(void) {
   return length * breadth * height;
}
```

Here, only important point is that you would have to use class name just before :: operator. A member function will be called using a dot operator (**.**) on a object where it will manipulate data related to that object only as follows:

```
Box myBox;          // Create an object

myBox.getVolume();  // Call member function for the object
```

Let us put above concepts to set and get the value of different class members in a class:

```
#include <iostream>

using namespace std;

class Box {
   public:
      double length;         // Length of a box
      double breadth;        // Breadth of a box
      double height;         // Height of a box

      // Member functions declaration
      double getVolume(void);
      void setLength( double len );
      void setBreadth( double bre );
      void setHeight( double hei );
};

// Member functions definitions
double Box::getVolume(void) {
   return length * breadth * height;
}
```

```
void Box::setLength( double len ) {
   length = len;
}

void Box::setBreadth( double bre ) {
   breadth = bre;
}

void Box::setHeight( double hei ) {
   height = hei;
}

// Main function for the program
int main( ) {
   Box Box1;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;
   return 0;
}
```

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

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

Separating Interface from Implementation

There have been various proposals for separating an interface from an implementation. For example the language Java supports this separation [An *Interface* defines a set of method signatures, while a *Class* defines the structure and behavior of its instances. A class may match several interfaces, if it implements the methods defined by each interface. Classes are instantiable, while interfaces are not. In contrast, we propose a different separation of interface and implementation, based on the properties supported by the context relation. A base class defines the structure and interface (method signatures) of its instances, and may provide implementations for the interface as well. If an implementation of a method is not provided directly in the base class, the implementation may be separately given in one or more classes that are context-related to the base class. In contrast to the notion of *Interface* in Java, a base class that defines only interfaces *may still be instantiated*. To prevent a method resolution error at run-time, we require an implementation of the method to either exist directly in the base class or to exist in some context-related class that is declared to be the default implementation

In C++, dynamic binding is required when a *virtual* method is invoked through a pointer. In a language such as Java all methods are assumed virtual (unless declared final) and all objects are referenced through pointers. The context relation allows method update, not just for subclasses but for the base class itself. Dynamic binding occurs regardless of whether a method is invoked through a pointer or variable. We propose the use of context tables, a dynamic version of static virtual function tables, to avoid pointer chasing through lists of context objects. The use of the tables will support fast method lookup. For simplicity, we assume all methods to be context-updatable, although it would be possible to add a keyword to limit dynamic lookup such as the *final* keyword in Java.

A metaobject protocol (MOP) is an interface that allows a programmer to customize properties of the programming language, such as adding persistence and concurrency . A *reflective* programming language is one which supports such customizations, allowing the program to reason about its own execution state and alter behavior accordingly . The context relation and context objects can be implemented using reflection.

Access Functions and Utility Functions

Access Functions

Access functions can read or display data. Another common use for access functions is to test the truth or falsity of conditions—such functions are often called predicate functions. An example of a predicate function would be an `isEmpty` function for any container class—a class capable of holding many objects, like a `vector`. A program might test `isEmpty` before attempting to read another item from the container object. An `isFull` predicate function might test a container-class object to determine whether it has no additional room. Useful predicate functions for our `Time` class might be `isAM` and `isPM`.

Utility Functions

A utility function (also called a helper function) is a `private` member.

The standard function has several **utility functions** for number of conversions, variable-length, argument processing, sorting and searching, and random number generation. To use these function the header <cstdlib> is required.

This header defines the types "div_t", "ldiv_t", "size_t". Following table lists the macros supported by <cstdlib> header.

Proxy Class

A **proxy class** in C++ is used to implement the **Proxy** Pattern in which an object is an interface or a mediator for some other object. A typical use of a **proxy class** in C++ is implementing the [] operator since the [] operator may be used to get data or to set data within an object.

Container Classes

A container stores many entities and provide sequential or direct access to them. List, vector and strings are such containers in standard template library. The string class is a container that holds chars. All container classes access the contained elements safely and efficiently by using iterators. Container class is a class that hold group of same or mixed objects in memory. It can be heterogeneous and homogeneous. Heterogeneous container class can hold mixed objects in memory whereas when it is holding same objects, it is called as homogeneous container class.

A class is said to be a container class which is utilized for the purpose of holding objects in memory or persistent media. A generic class plays a role of generic holder. A container class is a good blend of predefined behavior and an interface that is well known. The purpose of container class is to hide the topology for the purpose of objects list maintenance in memory. A container class is known as heterogeneous container, when it contains a set of different objects. A container class is known as homogeneous container when it contains a set of similar objects.

The following are the standardized container classes :
1. std::map :
Used for handle sparse array or a sparse matrix.
2. std::vector :
Like an array, this standard container class offers additional features such as bunds checking through the at () member function, inserting or removing elements, automatic memory management and throwing exceptions.
std::string :
A better supplement for arrays of chars.

Dynamic Memory Allocation

C++ supports three basic types of memory allocation, of which you've already seen two.

- **Static memory allocation** happens for static and global variables. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.
- **Automatic memory allocation** happens for function parameters and local variables. Memory for these types of variables is allocated when the relevant block is entered, and freed when the block is exited, as many times as necessary.
- How does dynamic memory allocation work?
- Your computer has memory (probably lots of it) that is available for applications to use. When you run an application, your operating system loads the application into some of that memory. This memory used by your application is divided into different areas, each of which serves a different purpose. One area contains your code. Another area is used for normal operations (keeping track of which functions were called, creating and destroying global and local variables, etc…). We'll talk more about those later. However, much of the memory available just sits there, waiting to be handed out to programs that request it.

- When you dynamically allocate memory, you're asking the operating system to reserve some of that memory for your program's use. If it can fulfill this request, it will return the address of that memory to your application. From that point forward, your application can use this memory as it wishes. When your application is done with the memory, it can return the memory back to the operating system to be given to another program.
- Unlike static or automatic memory, the program itself is responsible for requesting and disposing of dynamically allocated memory.
- **Initializing a dynamically allocated variable**
- When you dynamically allocate a variable, you can also initialize it via direct initialization or uniform initialization (in C++11):

```
int *ptr1 = new int (5); // use direct initialization
int *ptr2 = new int { 6 }; // use uniform initialization
```

- **Deleting single variables**
- When we are done with a dynamically allocated variable, we need to explicitly tell C++ to free the memory for reuse. For single variables, this is done via the scalar (non-array) form of the **delete** operator:

```
// assume ptr has previously been allocated with operator new
delete ptr; // return the memory pointed to by ptr to the operating system
ptr = 0; // set ptr to be a null pointer (use nullptr instead of 0 in C++11)
```

What does it mean to delete memory?

The delete operator does not *actually* delete anything. It simply returns the memory being pointed to back to the operating system. The operating system is then free to reassign that memory to another application (or to this application again later).

Although it looks like we're deleting a *variable*, this is not the case! The pointer variable still has the same scope as before, and can be assigned a new value just like any other variable.

Note that deleting a pointer that is not pointing to dynamically allocated memory may cause bad things to happen.