UNIT I

DATA ABSTRACTION & OVERLOADING

STRUCTURE OF A PROGRAM

Probably the best way to start learning a programming language is by writing a program. Therefore, here is our first program:

```
// my first program in C++
#include <iostream>
using namespace std;
int main ()
{
cout << "Hello World!";
return 0;
}
```

Output:

```
Hello World!
```

// my first program in C++

This is a comment line. All lines beginning with two slash signs (//) are considered comments and do not have any effect on the behavior of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

#include <iostream>

Lines beginning with a hash sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor. In this case the directive #include<iostream> tells the preprocessor to include the iostream standard file. This specific file (iostream) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

using namespace std;

All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name *std*. So in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library, and in fact it will be included in most of the source codes included in these tutorials.

int main ()

This line corresponds to the beginning of the definition of the main function. The main function is the point by where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it – the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a main function.

The word main is followed in the code by a pair of parentheses (()). That is because it is a function declaration: In C++, what differentiates a function declaration from other types of expressions are these parentheses that follow its name. Optionally, these parentheses may

enclose a list of parameters within them. Right after these parentheses we can find the body of the main function enclosed in braces ({}). What is contained within these braces is what the function does when it is executed.

cout << "Hello World!";

cout represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (which usually is the screen). cout is declared in the iostream standard file within the std namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code. Notice

return 0;

The return statement causes the main function to finish. return may be followed by a return code (in our example is followed by the return code 0). A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.

BASIC INPUT/OUTPUT

C++ uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen or the keyboard. A stream is an object where a program can either insert or extract characters to/from it. We do not really need to care about many specifications about the physical media associated with the stream - we only need to know it will accept or provide characters sequentially.

The standard C++ library includes the header file iostream, where the standard input and output stream objects are declared.

Standard Output (cout)

By default, the standard output of a program is the screen, and the C++ stream object defined to access it is cout. cout is used in conjunction with the *insertion operator*, which is written as << (two "less than" signs).

```
cout << "Output sentence"; // prints Output sentence on screen
cout << 120; // prints number 120 on screen
cout << x; // prints the content of x on screen
```

The << operator inserts the data that follows it into the stream preceding it. In the examples above it inserted the constant string Output sentence, the numerical constant 120 and variable x into the standard output stream cout. Notice that the sentence in the first instruction is enclosed between double quotes (") because it is a constant string of characters. Whenever we want to use constant strings of characters we must enclose them between double quotes (") so that they can be clearly distinguished from variable names. For example, these two sentences have very different results:

```
cout << "Hello"; // prints Hello
cout << Hello; // prints the content of Hello variable
```

The insertion operator (<<) may be used more than once in a single statement:

```
cout << "Hello, " << "I am " << "a C++ statement";
```

This last statement would print the message Hello, I am a C++ statement on the screen. The utility of repeating the insertion operator (<<) is demonstrated when we want to print out a combination of variables and constants or more than one variable:

cout << "Hello, I am " << age << " years old and my zipcode is " << zipcode;

If we assume the age variable to contain the value 24 and the zipcode variable to contain 90064 the output of the previous statement would be:

Hello, I am 24 years old and my zipcode is 90064

It is important to notice that cout does not add a line break after its output unless we explicitly indicate it, therefore, the following statements:

cout << "This is a sentence.";
cout << "This is another sentence.";

will be shown on the screen one following the other without any line break between them:

This is a sentence.This is another sentence.

Even though we had written them in two different insertions into cout. In order to perform a line break on the output we must explicitly insert a new-line character into cout. In C++ a new-line character can be specified as \n (backslash, n):

cout << "First sentence.\n ";
cout << "Second sentence.\nThird sentence.";

This produces the following output:

First sentence.
Second sentence.
Third sentence.

Additionally, to add a new-line, you may also use the endl manipulator. For example:

cout << "First sentence." << endl;
cout << "Second sentence." << endl;

would print out:

First sentence.
Second sentence.

The endl manipulator produces a newline character, exactly as the insertion of '\n' does, but it also has an additional behavior when it is used with buffered streams: the buffer is flushed. Anyway, cout will be an unbuffered stream in most cases, so you can generally use both the \n escape character and the endl manipulator in order to specify a new line without any difference in its behavior.

Standard Input (cin).

The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the cin stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream. For example:

int age;
cin >> age;

The first statement declares a variable of type int called age, and the second one waits for an input from cin (the keyboard) in order to store it in this integer variable. cin can only process the input from the keyboard once the RETURN key has been pressed. Therefore, even if you request a single character, the extraction from cin will not process the input until the user presses RETURN after the character has been introduced. You must always consider the type of the variable that you are using as a container with cin extractions. If you request

an integer you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters.

```
// i/o example
#include <iostream>
using namespace std;
int main ()
{
int i;
cout << "Please enter an integer value: ";
cin >> i;
cout << "The value you entered is " << i;
cout << " and its double is " << i*2 << ".\n";
return 0;
}
```

Output:

Please enter an integer value: 702

The value you entered is 702 and its double is 1404.

The user of a program may be one of the factors that generate errors even in the simplest programs that use cin (like the one we have just seen). Since if you request an integer value and the user introduces a name (which generally is a string of characters), the result may cause your program to misoperate since it is not what we were expecting from the user. So when you use the data input provided by cin extractions you will have to trust that the user of your program will be cooperative and that he/she will not introduce his/her name or something similar when an integer value is requested. A little ahead, when we see the string stream class we will see a possible solution for the errors that can be caused by this type of user input. You can also use cin to request more than one datum input from the user:

```
cin >> a >> b;
```

is equivalent to:

```
cin >> a;
cin >> b;
```

In both cases the user must give two data, one for variable a and another one for variable b that may be separated by any valid blank separator: a space, a tab character or a newline.

C structures Vs C++ Structure:

1. C structures contain only variables inside the structures whereas c++ structures can have both variables as well as functions.
2. C structures don't provide protection for its members whereas c++ structures provide protection through private and protected access specifiers.
3. No need to specify struct keyword while declaring structure variable in c++, whereas it is mandatory in C.

C	C++
Struct Employee { Int age; Char name[12]; }emp1;	Struct Employee { int age; private: Char name[12];

Void getdata()
{
---- 	
---- 	
}
	Void getdata()
	{
	----- 	
	}
	}emp1;

Structure vs class in C++

In C++, a structure is same as class except the following differences:

1) When deriving a struct from a class/struct, default access-specifier for a base class/struct is public. And when deriving a class, default access specifier is private.
2) Members of a class are private by default and members of struct are public by default.
   For example program 1 fails in compilation and program 2 works fine.

```
#include <stdio.h>
class Test {
   int x; // x is private
};
int main()
{
  Test t;
  t.x = 20; // compiler error because x is private
  getchar();
  return 0;
}

// Program 2
#include <stdio.h>
struct Test {
   int x; // x is public
};
int main()
{
  Test t;
  t.x = 20; // works fine because x is public
  getchar();
  return 0;
}
```

DIFFERENCES B/W PROCEDURE ORIENTED & OBJECT ORIENTED PROGRAMMING

Procedure oriented programming	**Object oriented programming**
1. Emphasis on doing things(algorithm)	1.Emphasis on data rather than procedure
2.Large programs are divided into functions	2.programs are divided into objects
3. Most of the functions share global data	3. Data is hidden through access specifiers
4. Data moves openly around the system	4. Objects communicate trough message

from function to function	passing technique
5. Follows top-down Approach	5. Follows Bottum-up approach
6. Adding new functions and data is somewhat difficult	6. New functions and data can easily added.

FEATURES OF OBJECT ORIENTED PROGRAMMING

The prime purpose of C++ programming was to add object orientation to the C programming language, which is in itself one of the most powerful programming languages.

The core of the pure object-oriented programming is to create an object, in code, that has certain properties and methods. While designing C++ modules, we try to see whole world in the form of objects. For example a car is an object which has certain properties such as color, number of doors, and the like. It also has certain methods such as accelerate, brake, and so on.

There are a few principle concepts that form the foundation of object-oriented programming:

Object:

This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.

Class:

When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

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.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

Encapsulation:

Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

Inheritance:

One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

Polymorphism:

The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

Overloading:

The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

CLASSES:

A Class is a template or blue print. A *class* is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions. An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable. Classes are generally declared using the keyword class, with the following format:

```
class class_name
 {
        access_specifier_1:
        member1;
        access_specifier_2:
        member2;
        ...
} object_names;
```

Where class_name is a valid identifier for the class, object_names is an optional list of names for objects of this class. The body of the declaration can contain members, that can be either data or function declarations, and optionally access specifiers. All is very similar to the declaration on data structures, except that we can now include also functions and members, but also this new thing called *access specifier*. An access specifier is one of the following three keywords: private, public or protected. These specifiers modify the access rights that the members following them acquire:

1. **private members** of a class are accessible only from within other members of the same class or from their *friends*.
2. **protected members** are accessible from members of their same class and from their friends, but also from members of their derived classes.

3. Finally, **public members** are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before one other class specifier automatically has private access. Forexample:

```
class CRectangle
{
        int x, y;
        public:
        int z;
        void set_values (int,int);
        int area (void);
} rect;
int main()
{
rect.set_values(10,10);
rect.z=area();
cout<<" area is"<<rect.z;
}
```

Declares a class (i.e., a type) called CRectangle and an object (i.e., a variable) of this class called rect. This class contains four members: two data members of type int (member x and member y) with private access (because private is the default access level) and two member functions with public access: set_values() and area(), of which for now we have only included their declaration, not their definition.

How to access Class members:

We can refer within the body of the program to any of the public members of the object rect as if they were normal functions or normal variables, just by putting the object's name followed by a dot (.) and then the name of the member. For example:

```
rect.z=area();
rect.set_values (3,4);
myarea = rect.area();
.
```

The only members of rect that we cannot access from the body of our program outside the class are x and y, since they have private access and they can only be referred from within other members of that same class.
Here is the complete example of class CRectangle:

```
// classes example
#include <iostream>
using namespace std;
class CRectangle {
int x, y;
public:
```

```
int z;
void set_values (int,int);
int area () {return (x*y);}
};
void CRectangle::set_values (int a, int b) {
x = a;
y = b;
}
int main () {
CRectangle rect;
rect.set_values (3,4);
rect.z= rect.area();
cout << "area: " <<rect.z;
return 0;
}
```

Output:

area: 12

Define Member functions:

The most important new thing in this code is the operator of scope (::, two colons) included in the definition of set_values(). It is used to define a member of a class from outside the class definition itself. You may notice that the definition of the member function area() has been included directly within the definition of the CRectangle class given its extreme simplicity, whereas set_values() has only its prototype declared within the class, but its definition is outside it. In this outside declaration, we must use the operator of scope (::) to specify that we are defining a function that is a member of the class CRectangle and not a regular global function.

The scope operator (::) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, in the function set_values() of the previous code, we have been able to use the variables x and y, which are private members of class CRectangle, which means they are only accessible from other members of their class.

The only difference between defining a class member function completely within its class or to include only the prototype and later its definition, is that in the first case the function will automatically be considered an inline member function by the compiler, while in the second it will be a normal (not-inline) class member function, which in fact supposes no difference in behavior.

Members x and y have private access. By declaring them private we deny access to them from anywhere outside the class. This makes sense, since we have already defined a member function to set values for those members within the object: the member function set_values(). Therefore, the rest of the program does not need to have direct access to them. Perhaps in a so simple example as this, it is difficult to see an utility in protecting those two variables, but in greater projects it may be very important that values cannot be modified in an unexpected way (unexpected from the point of view of the object).

CONSTRUCTORS AND DESTRUCTORS

Constructor is a special function of a class, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class, and cannot have any return type; not even void. We are going to implement CRectangle including a constructor:

The purpose of constructor is to initialize variables or assign dynamic memory for objects during their process of creation to become operative and to avoid returning unexpected values during their execution.

Constructors are two types: 1. Default constructor 2. Parameterized constructor.

```
// example: class constructor
#include <iostream>
using namespace std;
class CRectangle {
int width, height;
public:
CRectangle (int,int);     //parameterized constructor
int area ()
{
return (width*height);
}
};
CRectangle::CRectangle (int a, int b) {
width = a;
height = b;
}
int main ()
{
CRectangle rect (3,4);
CRectangle rectb (5,6);
cout << "rect area: " << rect.area() << endl;
cout << "rectb area: " << rectb.area() << endl;
return 0;
}
rect area: 12
rectb area: 30
```

As you can see, the result of this example is identical to the previous one. But now we have removed the member function set_values(), and have included instead a constructor that performs a similar action: it initializes the values of x and y with the parameters that are passed to it. Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created:

```
CRectangle rect (3,4);
CRectangle rectb (5,6);
```

Constructor Types:

1. Parameterized constructors

Constructors that can take arguments are termed as parameterized constructors. The number of arguments can be greater or equal to one(1). For example:

```
class Example
{
    int x, y;
  public:
    Example();
    Example(int a, int b); // Parameterized constructor
};
Example :: Example()
{
}
Example :: Example(int a, int b)
{
    x = a;
    y = b;
}
```

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly. The method of calling the constructor implicitly is also called the *shorthand* method.

```
Example e = Example(0, 50); // Explicit call

Example e(0, 50);          // Implicit call
```

2. Default constructors

If the programmer does not supply a constructor for an instantiable class, most languages will provide a *default constructor*. The behavior of the default constructor is language dependent. It may initialize data members to zero or other same values, or it may do nothing at all. In C++ a default constructor is required if an array of class objects is to be created. Other languages (Java, C#, VB .NET) have no such restriction.

3. Copy constructors

Copy constructors define the actions performed by the compiler when copying class objects. A copy constructor has one formal parameter that is the type of the class (the parameter may be a reference to an object). It is used to create a copy of an existing object of the same class. Even though both classes are the same, it counts as a conversion constructor

Constructors cannot be called explicitly as if they were regular member functions. They are only executed when a new object of that class is created.

DESTRUCTOR:

The *destructor* fulfills the opposite functionality. It is automatically called when an object is destroyed, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using the operator delete.

The destructor must have the same name as the class, but preceded with a tilde sign (~) and it must also return no value. The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.

```
// example on constructors and destructors
#include <iostream>
using namespace std;
class CRectangle
{
int *width, *height;
public:
CRectangle (int,int);
~CRectangle ();
int area () {return (*width * *height);
}
};
CRectangle::CRectangle (int a, int b)
{
width = new int;
height = new int;
*width = a;
*height = b;
}
CRectangle::~CRectangle ()
{
delete width;
delete height;
}
int main () {
CRectangle rect (3,4), rectb (5,6);
cout << "rect area: " << rect.area() << endl;
cout << "rectb area: " << rectb.area() << endl;
return 0;
}
rect area: 12
rectb area: 30
```


FRIEND FUNCTIONS

In principle, private and protected members of a class cannot be accessed from outside the class in which they are declared. However, this rule does not affect *friends*. Friends are functions or classes declared as such. If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword friend:

```
// friend functions
#include <iostream>
using namespace std;
class CRectangle {
int width, height;
public:
void set_values (int, int);
int area () {return (width * height);}
friend CRectangle duplicate (CRectangle);
};
void CRectangle::set_values (int a, int b) {
width = a;
height = b;
}
CRectangle duplicate (CRectangle rectparam)
{
CRectangle rectres;
rectres.width = rectparam.width*2;
rectres.height = rectparam.height*2;
return (rectres);
}
int main () {
CRectangle rect, rectb;
rect.set_values (2,3);
rectb = duplicate (rect);
cout << rectb.area();
return 0;
}
```

Output:

24

The duplicate function is a friend of CRectangle. From within that function we have been able to access the members width and height of different objects of type CRectangle, which are private members. Notice that neither in the declaration of duplicate() nor in its later use in main() have we considered duplicate a member of class CRectangle. It isn't! It simply has access to its private and protected members without being a member.

The friend functions can serve, for example, to conduct operations between two different classes. Generally, the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the same class to perform operations with them. Such as in the previous example, it would have been shorter to integrate duplicate() within the class CRectangle.

FRIEND CLASSES

Just as we have the possibility to define a friend function, we can also define a class as friend of another one, granting that first class access to the protected and private members of the second one.

```
// friend class
#include <iostream>
using namespace std;
class CSquare;
class CRectangle {
int width, height;
public:
int area ()
{
return (width * height);
}
void convert (CSquare a);
};
class CSquare {
private:
int side;
public:
void set_side (int a)
{
side=a;
}
friend class CRectangle;
};
void CRectangle::convert (CSquare a) {
width = a.side;
height = a.side;
}
int main ()
{
CSquare sqr;
CRectangle rect;
sqr.set_side(4);
rect.convert(sqr);
cout << rect.area();
return 0;
}
```

Output:

16

In this example, we have declared CRectangle as a friend of CSquare so that CRectangle member functions could have access to the protected and private members of CSquare, more concretely to CSquare::side, which describes the side width of the square. You may also see something new at the beginning of the program: an empty declaration of class CSquare. This is

necessary because within the declaration of CRectangle we refer to CSquare (as a parameter in convert()).

The definition of CSquare is included later, so if we did not include a previous empty declaration for CSquare this class would not be visible from within the definition of CRectangle.
Consider that friendships are not corresponded if we do not explicitly specify so. In our example, CRectangle is considered as a friend class by CSquare, but CRectangle does not consider CSquare to be a friend, so CRectangle can access the protected and private members of CSquare but not the reverse way. Of course, we could have declared also CSquare as friend of CRectangle if we wanted to. Another property of friendships is that they are *not transitive*: The friend of a friend is not considered to be a friend unless explicitly specified.

Dynamic Memory Management

A standard C++ array data structure is fixed in size once it's created. The size is specified with a constant at compile time. Sometimes it's useful to determine the size of an array *dynamically* at execution time and then create the array. C++ enables you to control the *allocation* and *deallocation* of memory in a program for objects and for arrays of any built-in or user-defined type. This is known as dynamic memory management and is performed with the operators new and delete. We'll use these capabilities to implement our Array class in the next section.

The new operator to dynamically allocate (i.e., reserve) the exact amount of memory required to hold an object or array at execution time. The object or array is created in the free store (also called the heap)—*a region of memory assigned to each program for storing dynamically allocated objects*. Once memory is allocated in the free store, you can access it via the pointer that operator new returns. When you no longer need the memory, you can return it to the free store by using the delete operator to deallocate (i.e., release) the memory, which can then be *reused* by future new operations.

***Obtaining Dynamic Memory with* new**

Consider the following statement:

```
cout << x++ << endl;
Time *timePtr = new Time;
```

The new operator allocates storage of the proper size for an object of type Time, calls the default constructor to initialize the object and returns a pointer to the type specified to the right of the new operator (i.e., a Time *). If new is unable to find sufficient space in memory for the object, it indicates that an error occurred by throwing an exception.

***Releasing Dynamic Memory with* delete**

To destroy a dynamically allocated object and free the space for the object, use the delete operator as follows:

```
delete timeptr;
```

This statement first *calls the destructor for the object to which* timePtr *points, then deallocates*
the memory associated with the object, returning the memory to the free store.

Initializing Dynamic Memory

You can provide an initializer for a newly created fundamental-type variable, as in which initializes a newly created double to 3.14159 and assigns the resulting pointer to ptr. The same syntax can be used to specify a comma-separated list of arguments to the constructor of an object. For example,
Time *timePtr = new Time(12, 45, 0);

initializes a new Time object to 12:45 PM and assigns the resulting pointer to timePtr.

***Dynamically Allocating Arrays with* new []**

You can also use the new operator to allocate arrays dynamically. For example, a 10-element integer array can be allocated and assigned to gradesArray as follows:

int *gradesArray = new int[10];

which declares int pointer gradesArray and assigns to it a pointer to the first element of a dynamically allocated 10-element array of ints. The size of an array created at compile time must be specified using a constant integral expression; however, a dynamically allocated array's size can be specified using *any* non-negative integral expression that can be evaluated at execution time.

***Releasing Dynamically Allocated Arrays with* delete []**

To deallocate the memory to which gradesArray points, use the statement

delete [] gradesArray;

STATIC CLASS MEMBERS

There is an important exception to the rule that each object of a class has its own copy of all the data members of the class. In certain cases, only *one* copy of a variable should be *shared* by *all* objects of a class. A static data member is used for these and other reasons. Such a variable represents "class-wide" information (i.e., a property that is shared by all instances and is not specific to any one object of the class).

Scope and Initialization of static Data Members

Although they may seem like global variables, a class's static data members have class scope. Also, static members can be declared public, private or protected. A fundamental- type static data member is initialized by default to 0. If you want a different initial value, a static data member can be initialized *once*. A static const data member of int or enum type can be initialized in its declaration in the class definition. However, all other static data members must be defined *at global namespace scope* (i.e., outside the body of the class definition) and can be initialized only in those definitions—again, the next version of the C++ standard will allow initialization where these variables are declared in the class definition. If a static data member is an object of a class that provides a default constructor, the static data member need not be initialized because its default constructor will be called.

Accessing* static *Data Members

A class's private and protected static members are normally accessed through the class's public member functions or friends. *A class's* static *members exist even when no objects of that class exist.* To access a public static class member when no objects of the class exist, simply prefix the class name and the scope resolution operator (::) to the name of the data member.

To access a private or protected static class member when *no* objects of the class exist, provide a public static member function and call the function by prefixing its name with the class name and scope resolution operator. *A* ***static member function*** is a service of the *class, not* of a specific object of the class.

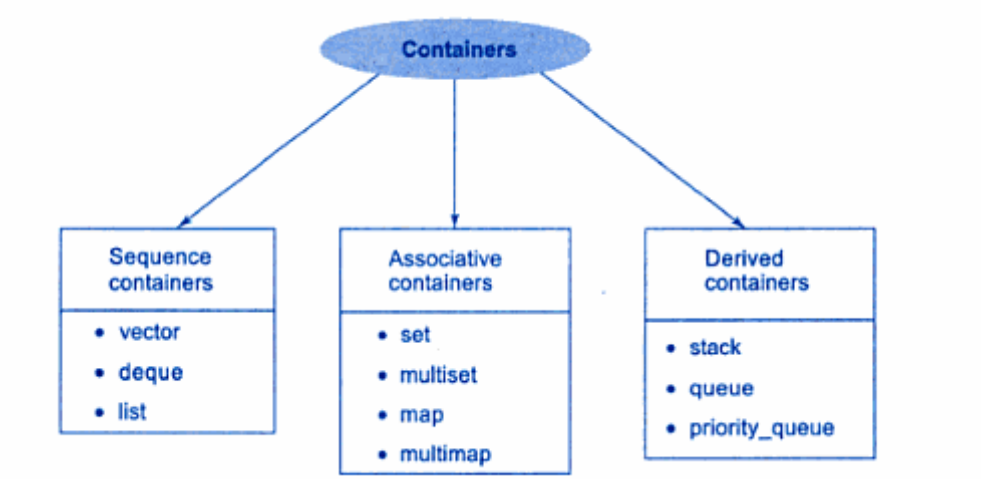
```
class Foo {
        public:
        Foo()
        {
        ++numFoos;
        cout << "We have now created " << numFoos << " instances of the Foo class\n";
        }
        static int getNumFoos()
        {
        return numFoos;
        }
        private:
        static int numFoos;
};
int Foo::numFoos = 0; // allocate memory for numFoos, and initialize it
int main() {
        Foo f1;
        Foo f2;
        Foo f3;
        cout << "So far, we've made " << Foo::getNumFoos() << " instances of the Foo
class\n";
}
```

Here getNumFoos() function is a static member of Foo class. So it can access without object using Class name and Scope Resolution operator.

CONTAINER CLASS

A class that is used to hold objects in memory or external storage is often called a *container class*. A container class acts as a generic holder and has a predefined behavior and a well-known interface. When it contains a group of mixed objects, the container is called a heterogeneous container; when the container is holding a group of objects that are all the same, the container is called a homogeneous container.

The STL(Standard Template Library) container s are divided into three major categories—sequence containers, associative containers and container adapters.



The *sequence containers* represent *linear* data structures, such as vectors and linked lists. *Associative containers* are *nonlinear* containers that typically can locate elements stored in the containers quickly. Such containers can store sets of values or key/value pairs. The sequence containers and associative containers are collectively referred to as the *first-class containers.* There are other container types that are considered "near containers"—C-like pointer-based arrays , bitsets for maintaining sets of flag values for performing highspeed mathematical vector operations These types are considered "near containers" because they exhibit capabilities similar to those of the first-class containers, but do not support all the first-class-container capabilities. Type string supports the same functionality as a sequence container, but stores only character data.

Derived containers are also called as container adapters. Stack, queues and priority queues can be created from different sequence containers. The derived constructors do not support Iterators and therefore we cannot use them for data manipulations. they support two member functions called push and pop for inserting and deleting operations.

Container	Description	Header file	Iterator
vector	A dynamic array. Allows insertions and deletions at back. Permits direct access to any element	<vector>	Random access
list	A bidirectional, linear list. Allows insertions and deletions anywhere.	<list>	Bidirectional
deque	A double-ended queue. Allows insertions and deletions at both the ends. Permits direct access to any element.	<deque>	Random access
set	An associate container for storing unique sets. Allows rapid lookup. (No duplicates allowed)	<set>	Bidirectional
multiset	An associate container for storing non-unique sets. (Duplicates allowed)	<set>	Bidirectional
map	An associate container for storing unique key/value pairs. Each key is associated with only one value (One-to-one mapping). Allows key-based lookup.	<map>	Bidirectional
multimap	An associate container for storing key/value pairs in which one key may be associated with more than one value (one-to-many mapping). Allows key-based lookup.	<map>	Bidirectional
stack	A standard stack. Last-in-first-out(LIFO).	<stack>	No iterator
queue	A standard queue. First-in-first-out(FIFO).	<queue>	No iterator
priority–queue	A priority queue. The first element out is always the highest priority element.	<queue>	No iterator

ITERATORS:

In object-oriented computer programming, an iterator is an object that enables a programmer to traverse a container, particularly lists. Various types of iterators are often provided via a container's interface. Note that an iterator performs traversal and also gives access to data elements in a container, but does not perform iteration (i.e., not without some significant liberty taken with that concept or with trivial use of the terminology). An iterator is behaviorally similar to a database cursor.

The primary purpose of an iterator is to allow a user to process every element of a container while isolating the user from the internal structure of the container. This allows the container to store elements in any manner it wishes while allowing the user to treat it as if it were a simple sequence or list. An iterator is best visualized as a pointer to a given element in the container, with a set of overloaded operators to provide a set of well-defined functions:

Operator* — Dereferencing the iterator returns the element that the iterator is currently pointing at.
Operator++ — Moves the iterator to the next element in the container. Most iterators also provide Operator– to move to the previous element.

Operator== and Operator!= — Basic comparison operators to determine if two iterators point to the same element. To compare the values that two iterators are pointing at, deference the iterators first, and then use a comparison operator.

Operator= — Assign the iterator to a new position (typically the start or end of the container's elements). To assign the value of the element the iterator is point at, deference the iterator first, then use the assign operator.

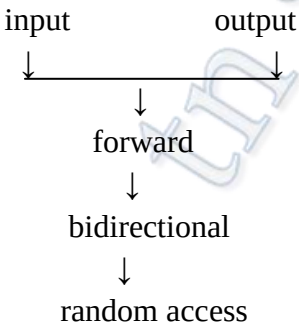
Iterator Functions:
advance() : Advance iterator (function template)distance Return distance between iterators (function template)
begin() : Iterator to beginning (function template)
end() : Iterator to end (function template)
prev() : Get iterator to previous element (function template)next Get iterator to next element (function template)

ITERATOR CATEGORIES

There are five categories of Iterators in STL and the Standard C++ Library:

1. **Input iterators** allow algorithms to advance the iterator and give "read only" access to the value.
2. **Output iterators** allow algorithms to advance the iterator and give "write only" access to the value.
3. **Forward iterators** combine read and write access, but only in one direction (i.e., forward).
4. **Bidirectional iterators** allow algorithms to traverse the sequence in both directions, forward and backward.
5. **Random access iterators** allow jumps and "pointer arithmetics."

Each category adds new features to the previous one. The iterator categories obey the following order:



PROXY CLASSES

Two of the fundamental principles of good software engineering are *separating interface from implementation* and *hiding implementation details*. We strive to achieve these goals by defining a class in a header and implementing its member functions in a separate implementation file. However, *headers do contain a portion of a class's implementation and*

hints about others. For example, a class's private members are listed in the class definition in a header, so these members are visible to clients, even though the clients may not access the private members. *Revealing a class's* private *data in this manner potentially exposes proprietary information to clients of the class.* We now introduce the notion of a proxy class that allows you to hide even the private data of a class from clients of the class. Providing clients of your class with a *proxy class* that knows only the public interface to your class enables the clients to use your class's services without giving the clients access to your class's implementation details.

Implementing a proxy class requires several steps. First, we create the class definition for the class that contains the proprietary implementation we would like to hide. Our example class, called Implementation, is shown in Prog1. The proxy class Interface is shown in Prog2-3. The test program and sample output are shown in Prog4. Class Implementation (Prog1) provides a single private data member called value (the data we would like to hide from the client), a constructor to initialize value and functions setValue and getValue.

Prog1

```
 // Implementation class definition.
class Implementation
 {
 public:
 // constructor
 Implementation( int v ): value( v ) // initialize value with v
{
 // empty body
 } // end constructor Implementation
 // set value to v
void setValue( int v )
{
 value = v; // should validate v
 } // end function setValue
 // return value
int getValue() const
 {
 return value;
 } // end function getValue
 private:
int value; // data that we would like to hide from the client
}; // end class Implementation
```

We define a proxy class called Interface (Prog2) with an identical public interface (except for the constructor and destructor names) to that of class Implementation. The proxy class's only private member is a pointer to an Implementation object. Using a pointer in this manner allows us to hide class Implementation's implementation details from the client. When a class definition uses only a pointer or reference to an object of another class (as in this case), the class header for that other class (which would ordinarily reveal the private data of that class) is *not* required to be included with #include. This is because the compiler doesn't need

to reserve space for an *object* of the class. The compiler *does* need to reserve space for the *pointer* or *reference.* The sizes of pointers and references are characteristics of the hardware platform on which the compiler runs, so the compiler already knows those sizes. You can simply declare that other class as a data type with a *forward class declaration* before the type is used in the file.

Prog2

```
 // Proxy class Interface definition.
 // Client sees this source code, but the source code does not reveal
 // the data layout of class Implementation.
class Interface
 {
 public:
 Interface( int ); // constructor
void setValue( int ); // same public interface as
int getValue() const; // class Implementation has
 ~Interface(); // destructor
 private:
// requires previous forward declaration (line 6)
Implementation *ptr;
 }; // end class Interface
```

The member-function implementation file for proxy class Interface Prog3 is the only file that includes the header Implementation.h containing class Implementation. The file Interface.cpp (Prog3) is provided to the client as a precompiled object code file along with the header Interface.h that includes the function prototypes of the services provided by the proxy class. Because file Interface.cpp is made available to the client only as object code, the client is not able to see the interactions between the proxy class and the proprietary class. The proxy class imposes an extra "layer" of function calls as the "price to pay" for hiding the private data of class Implementation. Given the speed of today's computers and the fact that many compilers can *inline* simple function calls automatically, the effect of these extra function calls on performance is often negligible.

Prog3:

```
  // Implementation of class Interface--client receives this file only
 // as precompiled object code, keeping the implementation hidden.
 #include "Interface.h" // Interface class definition
#include "Implementation.h" // Implementation class definition
 // constructor
 Interface::Interface( int v ): ptr( )// initialize ptr to point to
 { // a new Implementation object
 // empty body
 } // end Interface constructor
// call Implementation's setValue function
void Interface::setValue( int v )
{
ptr->setValue( v );
```

```
} // end function setValue
 // call Implementation's getValue function
 int Interface::getValue() const
 {
return ptr->getValue();
 } // end function getValue
 // destructor
 Interface::~Interface()
 {
delete ptr;
} // end ~Interface destructor
```

Notice that only the header for Interface is included in the client code (line 4)—there is no mention of the existence of a separate class called Implementation. Thus, the client never sees the private data of class Implementation, nor can the client code become dependent on the Implementation code.

Prog4

```
 // Hiding a class's private data with a proxy class.
 #include <iostream>
using namespace std;
int main()
 {
Interface i( 5 ); // create Interface object
 cout << "Interface contains: " << " before setValue" << endl;
i.setValue( 10 );
cout << "Interface contains: "  << " after setValue" << endl;
} // end main
```

FUNCTION OVERLOADING:

C++ enables several functions of the same name to be defined, as long as they have different signatures. This is called function overloading. The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call. Function overloading is used to create several functions of the *same* name that perform similar tasks, but on *different* data types.

The following program uses overloaded square functions to calculate the square of an int and the square of a double. The compiler chooses the proper function to call, based on the type of the argument. The last two lines of the output window confirm that the proper function was called in each case.

```
 #include <iostream>
 using namespace std;
// function square for int values
int square( int x )
{
cout << "square of integer " << x << " is ";
```

```
return x * x;
} // end function square with int argument
// function square for double values
double square( double y )
{
cout << "square of double " << y << " is ";
return y * y;
} // end function square with double argument

 int main()
{
 cout << square( 7 ) ; // calls int version
 cout << endl;
cout << square( 7.5 ); // calls double version
 cout << endl;
} // end main
```

Output:
square of integer 7 is 49
square of double 7.5 is 56.25

Observe the following three examples carefully to better understand function overloading.

```
// Overloading Example (1)
double geometric_mean( int, int );
// (2)
double geometric_mean( double, double );
// (3)
double geometric_mean( double, double, double );
geometric_mean( 10, 25 );                    // Will call (1):
geometric_mean( 22.1, 421.77 );              // Will call (2):
geometric_mean( 11.1, 0.4, 2.224 );          // Will call (3):
```

Under some circumstances, a call can be ambiguous, because two or more functions match with the supplied arguments equally well. Example, supposing the declaration of geometric_mean above:

```
geometric_mean(7, 13.21);
```

The above line is is an error, because (1) could be called and the second argument casted to an int, and (2) could be called with the first argument casted to a double. None of the two functions is unambiguously a better match.

```
geometric_mean(1.1, 2.2, 3);
```

This will call (3) too, despite its last argument being an int, Because (3) is the only function which can be called with 3 arguments

Overloading resolution

Please beware that overload resolution in C++ is one of the most complicated parts of the language. This is probably unavoidable in any case with automatic template instantiation,

user defined implicit conversions, built-in implicit conversation and more as language features. So do not despair if you do not understand this at first go. It is really quite natural, once you have the ideas, but written down it seems extremely complicated.

OPERATOR OVERLOADING

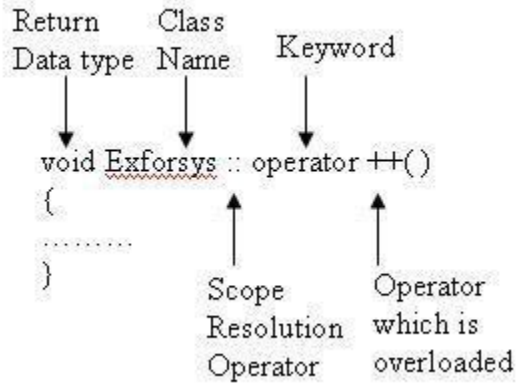
Operator overloading is a very important feature of Object Oriented Programming. It is because by using this facility programmer would be able to create new definitions to existing operators. In other words a single operator can perform several functions as desired by programmers.

Operator Overloading – Unary operators

As said before operator overloading helps the programmer to define a new functionality for the existing operator. This is done by using the keyword **operator**.

For example

Suppose we have a class say Exforsys and if the programmer wants to define a operator overloading for unary operator say ++, the function is defined as



Inside the class Exforsys the data type that is returned by the overloaded operator is defined as

```
class Exforsys
{
   private:
   …….
   public:
   void operator ++( );
   ……
};
```

The important steps involved in defining an operator overloading in case of unary operators are namely:

- Inside the class the operator overloaded member function is defined with the return data type as member function or a friend function.
- If the function is a member function then the number of arguments taken by the operator member function is none.

- If the function defined for the operator overloading is a friend function then it takes one argument.

Now let us see how to use this overloaded operator member function in the program

```
#include <iostream.h>
class Exforsys
{
   private:
   int x;
   public:
   Exforsys( ) { x=0; }       //Constructor
   void display();
   void Exforsys ++( );          //overload unary ++
};

void Exforsys :: display()
{
   cout<<"\nValue of x is: " << x;
}

void Exforsys :: operator ++( ) //Operator Overloading for operator ++
                                        defined
{
   ++x;
}

void main( )
{
   Exforsys e1,e2;         //Object e1 and e2 created
   cout<<"Before Increment"
   cout <<"\nObject e1: "<<e1.display();
   cout <<"\nObject e2: "<<e2.display();
   ++e1;  //Operator overloading applied
   ++e2;
   cout<<"\n After Increment"
   cout <<"\nObject e1: "<<e1.display();
   cout <<"\nObject e2: "<<e2.display();
}
```

The output of the above program is:

```
Before Increment
Object e1:
Value of x is: 0
Object e1:
Value of x is: 0
```

```
Before Increment
Object e1:
Value of x is: 1
Object e1:
Value of x is: 1
```

In the above example we have created 2 objects e1 and e2 of class Exforsys. The operator ++ is overloaded and the function is defined outside the class Exforsys.

When the program starts the constructor Exforsys of the class Exforsys initialize the values as zero and so when the values are displayed for the objects e1 and e2 it is displayed as zero. When the object ++e1 and ++e2 is called the operator overloading function gets applied and thus value of x gets incremented for each object separately. So now when the values are displayed for objects e1 and e2 it is incremented once each and gets printed as one for each object e1 and e2.

Operator Overloading – Binary Operators

Binary operators, when overloaded, are given new functionality. The function defined for binary operator overloading, as with unary operator overloading, can be member function or friend function.

The difference is in the number of arguments used by the function. In the case of binary operator overloading, when the function is a member function then the number of arguments used by the operator member function is one (see below example). When the function defined for the binary operator overloading is a friend function, then it uses two arguments.

Binary operator overloading, as in unary operator overloading, is performed using a keyword operator.

Binary operator overloading example:

```
#include <iostream.h>
class Exforsys
{
private:
int x;
int y;

public:
Exforsys()                  //Constructor
{ x=0; y=0; }

void getvalue( )            //Member Function for Inputting Values
{
cout << "\n Enter value for x: ";
```

```
cin >> x;
cout << "\n Enter value for y: ";
cin>> y;
}

void displayvalue( )          //Member Function for Outputting Values
{
cout <<"value of x is: " << x <<"; value of y is: "<<y
}

Exforsys operator +(Exforsys);
};

Exforsys Exforsys :: operator + (Exforsys e2)
//Binary operator overloading for + operator defined
{
int x1 = x+ e2.x;
int y1 = y+ e2.y;
return Exforsys(x1,y1);
}

void main( )
{
Exforsys e1,e2,e3;              //Objects e1, e2, e3 created
cout<<\n"Enter value for Object e1:";
e1.getvalue( );
cout<<\n"Enter value for Object e2:";
e2.getvalue( );
e3= e1+ e2;                     //Binary Overloaded operator used
cout<< "\nValue of e1 is:"<<e1.displayvalue();
cout<< "\nValue of e2 is:"<<e2.displayvalue();
cout<< "\nValue of e3 is:"<<e3.displayvalue();
}
```

The output of the above program is:

```
Enter value for Object e1:
Enter value for x: 10
Enter value for y: 20
Enter value for Object e2:
Enter value for x: 30
Enter value for y: 40
Value of e1 is: value of x is: 10; value of y is: 20
Value of e2 is: value of x is: 30; value of y is: 40
Value of e3 is: value of x is: 40; value of y is: 60
```

In the above example, the class Exforsys has created three objects e1, e2, e3. The values are entered for objects e1 and e2. The binary operator overloading for the operator „+‟ is declared as a member function inside the class Exforsys. The definition is performed outside the class Exforsys by using the scope resolution operator and the keyword operator.

The important aspect is the statement:

e3= e1 + e2;

The binary overloaded operator „+‟ is used. In this statement, the argument on the left side of the operator „+‟, e1, is the object of the class Exforsys in which the binary overloaded operator „+‟ is a member function. The right side of the operator „+‟ is e2. This is passed as an argument to the operator „+‟ . Since the object e2 is passed as argument to the operator‟‟+‟‟ inside the function defined for binary operator overloading, the values are accessed as e2.x and e2.y. This is added with e1.x and e1.y, which are accessed directly as x and y. The return value is of type class Exforsys as defined by the above example.

There are important things to consider in operator overloading with C++ programming language. Operator overloading adds new functionality to its existing operators. The programmer must add proper comments concerning the new functionality of the overloaded operator. The program will be efficient and readable only if operator overloading is used only when necessary.

Some operators cannot be overloaded:
Scope resolution operator denoted by ::
Member access operator or the dot operator denoted by .
Conditional operator denoted by ?:
Pointer to member operator denoted by .*

TEXT BOOKS:

1. Deitel and Deitel, “C++, How To Program”, Fifth Edition, Pearson Education, 2005.
2. Mark Allen Weiss, “Data Structures and Algorithm Analysis in C++”, Third Edition, Addison-Wesley, 2007.

REFERENCES:

1. Bhushan Trivedi, “Programming with ANSI C++, A Step-By-Step approach”, Oxford University Press, 2010.
2. Goodrich, Michael T., Roberto Tamassia, David Mount, “Data Structures and Algorithms in C++”, 7th Edition, Wiley. 2004.
3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, "Introduction to Algorithms", Second Edition, Mc Graw Hill, 2002.
4. Bjarne Stroustrup, “The C++ Programming Language”, 3rd Edition, Pearson Education, 2007.
5. Ellis Horowitz, Sartaj Sahni and Dinesh Mehta, “Fundamentals of Data Structures in C++”, Galgotia Publications, 2007.