Object Oriented Programming in C++
Basics of OOP

In this section we describe the three most important areas in object oriented programming: encapsulation, inheritance and polymorphism.

1. INTRODUCTION

OOP was developed as a result of limitations in earlier approaches to programming. The basic idea was to combine data and the functions that operate on that data into a single unit called an object.

In OOP, when you approach a problem you should no longer be thinking about how to divide it into functions. Rather you will be considering what objects to use. Thinking about objects should result in a close match between objects in your program and objects in the real world.

In this section we will investigate the three basic ideas that form the basis of OOP: Encapsulation, Inheritance and Polymorphism.

1.1 Introduction to Encapsulation

The concept of encapsulation embodies the hiding of information (both data and code): local variables are hidden in functions, private members are hidden in classes. As a result, external direct access to the data and functions by name can be prevented. Therefore the programmer has a greater freedom in the choice of object names, and the probability of programming errors is reduced at the expense of stricter control of data access by the compiler.

1.2 Introduction to Inheritance

This is a concept in OOP that promotes code reuse through the inheritance mechanism. A new class is derived from an existing one called the base class. The derived class reuses the base class members and can add too and alter them. Inheritance avoids the redevelopment and testing of code that already exists. The inheritance relationship is hierarchical and is a method of coping with complexity.

1.3 Introduction to Polymorphism

This concept of C++ can be expressed by the following comparison. The inscription Do it!! on a signboard or pointer is executed by a programmer and a mountain skier in different ways, depending on what sort of object is indicated by the pointer. If it points in the direction of a mountain then it means: “Ascend the mountain and ski down it”, and if it points in the direction of a computer it means: “write a program”.
2. INHERITANCE

To understand the notion of a derived class we give the following two examples.

```cpp
#include <iostream.h>

class coordinate {public: int N;};
class x_cord {public : coordinate C;};

void main()
{
    x_cord X;
    X.C.N = 1;
    cout << X.C.N;
}
```

```cpp
#include <iostream.h>

class coordinate {public: int N;};
class x_cord : public coordinate{};

void main()
{
    x_cord X;
    X.N = 1;
    cout << X.N;
}
```

The left hand box indicates how the class `x_cord` uses class `coordinate` as one of its member data items. In the right box it is indicated how the code of `coordinate` is reused via inheritance. The : after the type name `x_cord` means *is derived from*. In general the inheritance is carried out with the following statement `class derivedClass : <access-specifier> baseClass`. Even though the data member in `coordinate` is of public nature, once inherited it becomes private. Hence one has to specify the `<access-specifier>` when inheriting. In general there are three kinds of inheritances; public, protected and private. With public inheritance, every object of a derived class may also be treated as an object of that base class’ derived class. With protected access, derived classes and their friends can access protected base class members, whereas non-friend and non-derived-class-member functions cannot.

Note that a derived class cannot access the private members of its base class; allowing this would violate the concept of encapsulation in C++. A derived class however has access to the public and protected members of its base class. That is, the base class members that should not be accessed within the derived class is defined as private. However a derived class can access private members of the base class via the interfaces public and protected provided by the base class.

Now we look closely at the different inheritance types. public inheritance is the most commonly used type of inheritance. For this type the public and protected members of the base class are inherited as public and protected respectively. However friend functions are not inherited.

The protected access ensures that the public and protected members of the base class are inherited as protected members of the derived class.

If the inheritance is private, both public and protected members of the base class become private members in the derived class.

We now look at what inheritance means in terms of storage allocation.
A derived class can inherit its own members too. For example `class base` and `class derived: base`. The class `derived` will have its members and all the members of `base` as well. Now we are faced with a problem of differentiating between the member names of the `base` class and the same named members in the `derived` class. In general, it is not a good idea to have members with the same name in a derived class. However, using the scope operator we can differentiate the members of different classes.

In the following program we show how to differentiate members with the same name in `base` and `derived` classes. Note that these member of the `derived` class can be accessed implicitly (`D.N`) or explicitly (`D.x_coord::N`) while the members that are inherited can only be accessed explicitly (`D.coordinate::N`). If there was no name clash, all members of the `derived` class can be accessed implicitly.

```cpp
#include <iostream.h>
class coordinate {public: int N;}; // base class
class x_coord : public coordinate{public: int N;}; // derived class
void main()
{
    coordinate B;
x_coord D;
    B.N=1; // coordinate member
    D.N=sizeof(x_coord)/sizeof(coordinate); // x_coord member
    D.coordinate::N=3; // inherited member
    cout << B.N
         << D.N // implicit access
         << D.x_coord::N // same as above
         << D.coordinate::N; // explicit access, this
                           // cannot be accessed // implicitly.
}
```

When adding a member function to a derived class under the same circumstances we have the same problem. This is illustrated in the following example.

```cpp
#include <iostream.h>

The `base` class contains an array of 10 characters and the storage allocation of it is 10 as indicated. We note also that the size of the `derived` class is also 10. It becomes clear that the class `derived` inherits member `M` in the sense that it is contained the same way as if it was declared in the form `class derived char M[10];`.

```
In order to distinguish an inherited member with the same name in the body of a member function, it is also necessary to use the scope resolution operator as shown in the program on the left. Here a constructor initializes both members that have the same name \( M \), one of which is inherited from the base class \( \text{base} \), and the other of which is a proper member of the class \( \text{derived} \).

3. POLYMORPHISM

Polymorphism is the genie in OOP, taking instructions from a client and properly interpreting its wishes. A polymorphic function has many forms:

1. **Coercion** (ad hoc polymorphism): A function or operator works on several different types by converting their values to the expected type.

2. **Overloading** (ad hoc polymorphism): A function is called based on its signature defined as the list of argument types in its parameter list. For example \( \text{cout} \) alters the type of output according to the input parameter type.

3. **Inclusion** (pure polymorphism): A type is a sub type of another type. Functions available for the base type will work on the sub type. Such a function can have different implementations that are invoked by a run-time determination of sub type. The virtual function call that uses pointers fall into this category.

4. **Parametric polymorphism** (pure polymorphism): The type is left unspecified and is later substantiated. Manipulation of generic pointers and templates provides this in C++.

We will see how these things tie in with real programming concepts now.

3.1 Inheriting Functions

All member functions of a base class are inherited by a derived class in the same way as data members. A special case arises when a derived class contains its own member function having the same name and a list of arguments as a member function of the base class.
```cpp
#include <iostream.h>
class base {
public:
    void Say();
};
void base::Say() {
    cout << "Base Say()" << endl;
}
class derived: public base {
public:
    void Say();
};
void derived::Say() {
    cout << "Derived Say()" << endl;
}
void main()
{
    derived D, *pD;
    D.Say();
    D.derived::Say();
    D.base::Say();
pD=&D;
pD->Say();
pD->derived::Say();
pD->base::Say();
}

// Replacement for main
// in the above program
void main()
{
    derived D;
    base *pB;
pB = &D;
pB-> Say();
pB->base::Say();
}
```

In this program we show how member functions with identical names are inherited by another class. A member function inherited from the base class may be invoked only with the use of the scope resolution operator. The name of the inherited function is hidden by the function of the derived class that has the same name. It is seen that the member function of the derived class type can be accessed without the scope resolution operator. The program also illustrates the use of pointers to access the member functions within the class. The scope is again defined to resolve the ambiguity of the function `Say()` that is a member function of the class derived and the inherited class base. There is another method that can be used to address member functions of classes using pointers.

In this program a pointer is defined to the base class and it is used to point to the derived class. In doing so the conversion `derived *' to `base *' occurs. That is, both pB->Say() and pB->base::Say() refer to the member function base::Say(). However, pB->derived::Say() will generate an error since derived is not a base class of base.

The above program indicated that the type of pointer determines which function it is addressing within the class. The base class within the derived class can be accessed by the above controlled pointer conversion. However the opposite is not possible. That is if a base class object is created using base B and a pointer to the derived class is set to point to the base class as derived *pD=&B, the program will generate an error since `base *' cannot be converted to `derived *'.

Now our next question is why did pB=&D work. This works since the base class is a subset of the derived class, hence anything that can be accessed using a pointer to the derived class can also be explicitly accessed using a pointer to the base class. This is known as the ‘default pointer conversion’ in classes.
The above process can also be achieved by using explicit pointer conversion. However one is not encouraged to use explicit pointer conversion unless it is absolutely necessary. This is carried out using explicitly type casting using \texttt{(type *)} notation.

The foundation of polymorphism in C++ is based on a mechanism known as the \textit{virtual functions}. It is important to understand the difference between an ordinary member function and a virtual member function. Virtual functions are called using a pointer to the base class as described earlier.

```
#include <iostream.h>

class base {
public:
  virtual void Say();
};

void base::Say() {
  cout << "Base Say()" << endl;
}

class derived : public base {
public:
  void Say();
};

void derived::Say() {
  cout << "Derived Say()" << endl;
}

void main()
{
  derived D;
  base *pB;
  pB=&D;
  pB->Say();
  pB->base::Say();
}
```

In this program the default pointer conversion occurs differently. In the previous case \texttt{pB->Say()} was the same as \texttt{base::Say()}. However now with the definition of \texttt{virtual} in the base class function, \texttt{pB->Say()} refers to \texttt{derived::Say()}. Hence if one wants to access the function \texttt{Say()} in the base class it has to be addressed using \texttt{pB->base::Say()}. Furthermore \texttt{pB->derived::Say()} will generate an error since \texttt{derived} is not a base class of \texttt{base}. Hence it is clear that a virtual function is a member function that is called via a pointer to a public base class. The actual member function invoked is decided by the class type of the actual object that is addressed by the pointer. The following example shows you how to put this into practice.

For example consider the \texttt{shape} class which is used as the base class for a number of different shapes like the square, rectangle, arc and so on. Each of these classes can have its own function to compute the area. To obtain the areas of a several of these shapes, you can create an array of pointers to all the shapes in the program with a statement like \texttt{shape * array[n]}. Choosing the appropriate array member \texttt{array[i]->area();} one can get the appropriate area. That is if the pointer in \texttt{array} points to a circle, the functions returns the area of a circle. But, all the different classes must be derived from the same base class and the base class member function must be virtual.

### 3.2 Using Virtual Functions

The scenario handle by virtual functions is as follows. Imagine it is necessary to develop a function which has a pointer to an object as an argument and it is necessary that the function
should take different actions depending upon the type of the object: base or derived.

```cpp
class base {
public:
    void NV() {
        cout << "Base NonVirtual \n";
    }
};

class derived : public base {
public:
    void NV() {
        cout << "Derived NonVirtual \n";
    }
};

void UseB(base *pB) {
    pB->NV();
}

void UseD(derived *pD) {
    pD->NV();
}

void main()
{
    base B;
    derived D;
    UseB(&B);
    UseD(&D);
}
```

In the first program, two functions are presented for the pointer conversion. In the second program the default pointer conversion happens and the appropriate function within the class can be called via the same function call. The programs can also be set up to be called via references rather than pointers. Then the function calls would be `Use(B)` and `Use(D)`. Where as the function definition will hold `void Use(base *pB)` rather than a pointer and within this function definition one can use `B.V();` to call the virtual function inside the class.

4. MULTIPLE INHERITANCE

We note the following

1. A derived class can itself serve as a base class.
2. A class can be derived from any number of base classes.

The first item is similar to the concept of nested classes. The second item is similar to having multiple class definitions within a class. The following programs illustrate the use of the above concepts. The first program illustrates concept of indirect inheritance. With the inheritance hierarchy the class `point` inherits all the members of `coordinate`. The variable calls `P.N`, `P.coordinate::N`, `P.x_coord::N` and `P.point::N` all refer to the same variable. However the call `P.N` is the most useful.

In the second program one cannot use variable calls such as `P.N` since `N` is ambiguous inside the class `point` and so is `P.coordinate::N`. A base class cannot appear more than once in
the multiple inheritance list. Each comma separated base class will be preceded by its access specifier.

```cpp
class coordinate {public: int N;};
class x_coord : public coordinate {
};
class point : public x_coord {
};
void main()
{
  coordinate C;
  x_coord X;
  point P;
  C.N=1;
  X.N=2;
  P.N=3;
  cout << C.N << X.N << P.N << endl;
  cout << P.N
  << P.coordinate::N
  << P.x_coord::N
  << P.point::N;
}
```

Next we look at what happened to virtual function in a multiple inheritance environment.

```cpp
class coordinate {
public:
  int N;
  virtual void Say() {
    cout << N << " coordinate::N" << endl;
  }
};
class x_coord : public coordinate {
public:
  void Say() {
    cout << N << " x_coord::N" << endl;
  }
};
class point : public x_coord {
public:
  void Say() {
    cout << N << " point::N" << endl;
  }
};
class any : public point {
public:
  void Say() {
    cout << N << " any::N" << endl;
  }
};
void Say(coordinate &C) {
  C.Say();
}
void main()
{
  coordinate C;
  x_coord X;
  point P;
  any A;
  C.N=1;
  X.N=2;
  P.N=3;
  A.N=4;
  Say(C);
  Say(X);
  Say(P);
  Say(A);
}
```

In this program a non member global function `Say` is used to process objects of various types. The argument it takes is a ‘reference to coordinate’. Due to the existence of the virtual function feature in the body of the global function `Say()`, the function `Say()` which is the member function of the same class as the object that is passed into the global function is called. This is confirmed by the output of the program;

1 coordinate::N  
2 x_coord::N  
3 point::N  
4 any::N

If we discard the keyword `virtual` from the program, the absence of the virtual function feature will result in calling in all three calls to `Say()`, the version corresponding to class `coordinate` and the outputs of the program will be;

1 coordinate::N  
2 coordinate::N  
3 coordinate::N  
4 coordinate::N
It is seen that in all cases the base class function is called with the appropriate variable \( N \). When a pointer to the base class points to the derived class, the variable \( N \) is the one in the derived class, since the variable in the base class has to be addressed using the scope `base::N`.

What is most impressive about the virtual function mechanism is that if you extend the hierarchy of derived class, you need not change the general function for processing, which in our example is the global function `Say()`. The class `Any` shows that it is sufficient in a new derived class to introduce the necessary virtual function `Say()`, and it will be called automatically with `Say(A)` where \( A \) is an object type of the derived class.

6. CONSTRUCTING DERIVED CLASS OBJECTS

In this section we look at how constructors of the base classes are used to create a derived class object.

When an object of a class derived from one base class is defined, then first a base class constructor is called, after which a derived class constructor is called. When an object of a derived class is destroyed, the destructors are called in reverse order to that of the constructors. The program to the right will illustrate this. The base constructor followed by the derived constructor will be called at the creation of the object.

An implicit call of a base constructor has a significant drawback since only a default constructor without arguments can be called in this manner. That is if no default constructor is available in the base class, an error will be generated at the compilation time. However C++ provides a method to specify which base constructor is to be called at object creation. The following program illustrates how the base constructor is called with an initializer list.

```
#include <iostream.h>

class base {
    public:
        base() {cout << "Base constructor\n";}
        ~base() {cout << "Base destructor\n";}
};

class derived : public base {
    public:
        derived() : base (1) {
            cout << "Derived constructor\n";
        }
};

void main()
{
    cout << "Main Start\n";
    derived D;
    cout << "Main end\n";
}
```

Had we used `derived() : base () {
...` as the constructor definition in `derived`, the constructor would have called the default constructor of the base class `base()` rather than the overloaded constructor `base(int i)`.

The output of this program is

```
1 Int constructor in base
Default constructor in Derived
```
However for the latter case the default constructor should be defined within the base class.

We now extend the information we have gathered to include constructor calls for multiple inheritances as well.

```cpp
#include <iostream.h>
class coordinate {
public:
  int N;
  coordinate(int n) {N=n;}
};
class x_coord : public coordinate {
public:
  x_coord(int n) : coordinate(n){}
};
class y_coord : public coordinate {
public:
  y_coord(int n) : coordinate(n){}
};
class point : public x_coord, public y_coord {
public:
  point(int nx, int ny) : x_coord(nx), y_coord(ny){}
};
void main()
{
  point P (1,2);
  cout << P.x_coord::N
       << P.y_coord::N;
}
```

In this program all the classes use an overloaded constructor. In a multiple inheritance scenario, the constructors which are required to be called are listed, separated by commas, after the colon in the constructor definition. Note that only the constructor of a direct base class can appear in this list. That is, the indirect base class coordinate cannot appear in this list. The initializers are passed in via point P (1,2). These initializers are passed to the constructors x_coord and y_coord, which in turn passes to the constructor coordinate. As a result P.x_coord::N and P.y_coord::N assume the values 1 and 2.

7. TEMPLATES AND GENERIC PROGRAMMING IN C++

C++ uses the keyword `template` to provide *parametric polymorphism*. Parametric polymorphism allows the same code to be used with respect to different types where the type is a parameter in the code body. The code is written generically. An important use of this technique is found in writing generic *container classes*. In this section we consider function and class template. Function templates can be used to create a group of related, overloaded functions. Overloaded function are normally used to perform similar operations on different types of data. This is a powerful feature in OOP.

7.1 Function Templates

Consider a function `Say()` which is to output the values of the variables of different types to the monitor with a new line. From the methods we know so far we can achieve this objective by writing a set of overloaded functions with the same name and different types of argument. Another approach is to use the following template.
template<class Type>
void Say(Type obj) {
    cout << endl << obj;
};// void main()
    { 
    Say('1');
    Say(2);
}

Here the keywords template and class are keywords and the word Type can be any C++ data type. It is seen that any type of object can be passed into the function and the appropriate type is understood and the cout function called. The function can be set to return a generic type too. For example Type Say(Type obj) is acceptable. However, if there is a class defined as class NewClass, then the function call Say(NewClass) will generate an error since cout is undefined for a type NewClass. However this issue can be circumvented by defining a friend function operator<<() within the class which uses operator overloading. As seen from the above discussion, template functions and overloading are intimately related. The related functions generated from a function template all have the same name, so the compiler uses overloading resolution to invoke the proper function. template<class Type> void Say(Type obj) cout << endl << obj; ; void main() Say('1'); Say(2);

7.2 Class Templates

Similar to a function template, a class template defines a generic class that accepts a parameter.

The parameter T is used to pass in the data type. Note the syntax of the external function definition. The first line of the member function definition coincides with the first line of the class template declaration template <class T>. You can also see that the name of the parameter T is attached to the class name Point. This parameter may be used in an argument list and in the body of the function. The rest of the function definition is normal. The expression Point<int> pi replaces all instances of T with int in the template definition.

When using templates, static members are not universal, but are specific to each instant of the class object.

If this is the template, the instances

test<int> a;
test<double> b;

will have distinct static variable instances test<int>::count and test<double>::count.
That is, each template class created from class template has its own copy of each static data member of the class template; all objects of that template class share that one static data member. And as with static data members of non-template classes, static data members of template classes must be initialized at file scope. Each template class gets its own copy of the class template static member functions.

Both classes and function templates can have several class arguments. For example template <class T1, class T2> is possible. Other possible template arguments include constant expression, function names, and character strings. For example template <class T, int n> is possible. If such a class definition is called assign, then it is called using assign<double,2> x,y; and so on.

```
template <class T>
class newClass {
 public:
 friend void universalFriend();
 friend vect<T> instantiated(vect<T> v);
 ....
};
```

A template class can contain friend functions too. A friend function that does not use a template specification is universally a friend of all objects created of the template class. If a friend function that incorporates template arguments is specifically a friend of its object class. This makes sense because when a parameterization is used in the friend function, it depends on the type of parameter and cannot be a friend of an object with a different parameter type.

Dynamic allocation of memory using generic parameterization is achieved using \texttt{ptr = new T[size]}. Templates and inheritance are closely related. A class template can be derived from either a template class or a non-template class. A template class can be derived from a class template. A non-template class can be derived from a class template.

8. TYPE CASTING

Apart from implicit type conversions and ad hoc polymorphism, C++ also supports explicit cast conversions. Following are some of the cast operators.

- \texttt{static_cast\langle type\rangle(variable)}
- \texttt{reinterpret_cast\langle type\rangle(\&variable)}
- \texttt{const_cast\langle type\rangle(variable)}
- \texttt{dynamic_cast\langle type\rangle(variable)}

\texttt{static_cast} is used when the conversion is well defined and portable. \texttt{reinterpret_cast} is used when the cast is system dependent as in a reference. The \texttt{dynamic_cast} can be used to determine the type at run time. for example,

```cpp
Base* ptr;
Derived* dptr = dynamic_cast<Derived*>(ptr);
```

will ensure safe type casting. The cast converts the pointer value \texttt{ptr} to a \texttt{Derived*}. If the conversion is inappropriate, a value zero, the NULL pointer is returned. This is known as a down-cast.
9. THE STANDARD TEMPLATE LIBRARY (STL)

STL is the C++ standard library providing generic programming for many standard data structures and algorithms. It provides containers, iterators and algorithms that support a standard for generic programming. At the onset we will explore what containers, iterators and algorithms are.