Encapsulation, Inheritance and Polymorphism In C++

Contributed by faribasiddiq () On 25th January, 2014
This is an article on Encapsulation, Inheritance and Polymorphism In C++ **in** C++.
Rated 5.00 By 5 users
Object oriented programming is a design philosophy. In programming paradigm, object oriented programming is the process by which a problem is decomposed into a number of entities called object and then data and functions are built around these objects. The main advantages of OOP are:

- Modularization of the problem
- Easier maintenance
- Faster development
- Low cost development
- High quality development

In this article, we are going to discuss on three building blocks of object oriented programming:

Encapsulation

Encapsulation is one of the building blocks in OOP concept. It performs the operation of data hiding. It is a way to customize access to different portion of the data. Some data are hidden behind the access methods and some are open to all. But which data are open and which are hidden? How can we achieve this?

We can control the access on data by defining the access modifier. Three commonly used access modifier are public, private and protected.

Access specifier: Here is a list of different access modifier and their accessibility.

Public : To any member function in a class as well as to member objects.
Protected : To the class which has defined it and any derived classes.
Private : Only to the class which has defined it.

Let's clarify the encapsulation concept with a C++ program. Consider the following program:
Code:

```
class Player {
        string mName;
        int mAge;
        public:
                    string gameType;
};
```

```
int main() {
        Player p1;
        p1.mName = "C Ronaldo";
        p1.mAge = 25;
        p1.gameType = "Football";
        return 0;
}
```

Output:
Code:

```
Error 1 error C2248: 'Player::mName' : cannot access private member declared in class 'Playe
Error 2 error C2248: 'Player::mAge' : cannot access private member declared in class 'Player
```

Look at the errors given by the compiler. Player object p1 cannot access mName and mAge members of the player class. Why does it happen? Because these are the private member of the class. But we haven't declared these variables as private. So why can't we access them? If you don't declare any modifier, by default, it is considered as private member of the class. So, mName and mAge become the private member of the Player class. Any private members of a class cannot be accessed directly from class object using dot operator. The other variable gameType is declared as public member. It can be easily accessed by p1 object.

So, how can we access the private member of a class from the class object? We can do this by the creating public member function of that class. Member functions are the access point or interface by which a class object can access values of private members.

Consider the following example:

Code:

```
class Player {
        string mName;
        int mAge;
        public:
                string gameType;
        public:
                void setInfo(string str, int age) {
                mAge = age;
                mName = str;
        }
        void showInfo() {
                cout << "Name :" << mName << " " << "Age :" << mAge   << "GameType:" << game
        }
};
int main() {
        Player p1;
        p1.setInfo("C Ronaldo",25);
        p1.gameType = "Football";
        p1.showInfo();
        return 0;
}
```

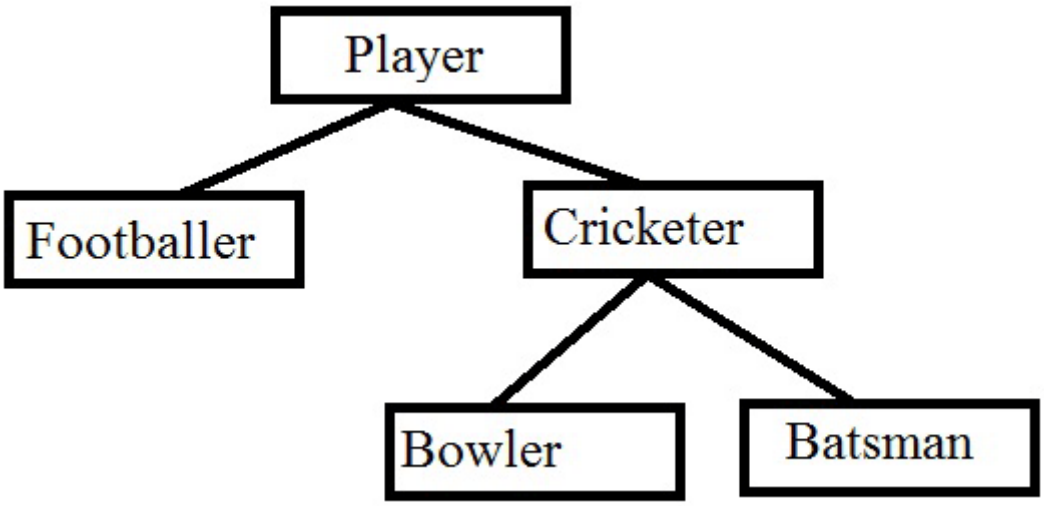
The above program runs as expected. Here, the private members are accessed through a member function setinfo(). This is an example of encapsulation. We can restrict the access to private data from other part of a program as needed. Other access modifier is protected where we can declare members functions or variables as protected and those protected members can be accessed in the derived class, which is inheritance.

Inheritance

In general term, Inheritance is the process to inherit the properties or behavior from an existing instance. It is one of the most important building blocks in object oriented programming. In OOP, inheritance is the process of inheriting the properties and methods of an existing class and making a new class with some extra properties and methods.

Let's understand with an example.

Consider a class, Player. A player has certain properties and behavior. Now we may want to more precisely define different types of players. So, for different game, we can create different player classes: Footballer, Cricketer, RugbyPlayer etc. Here, Player is the base class and Footballer, Cricketer, RugbyPlayer etc are the derived class from Player. These classes inherit the properties and behavior of the class Player and then add some specific properties and behavior of their own. The Cricketer class can further be inherited by other classes like: Bowler, Batsman, Wicketkeeper etc. Base class and derived class are also known as superclass and subclass.



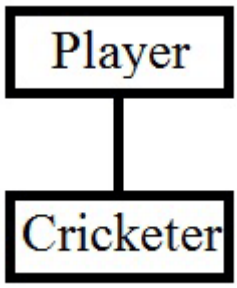
Types of inheritance

Inheritance can be of many types.

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance

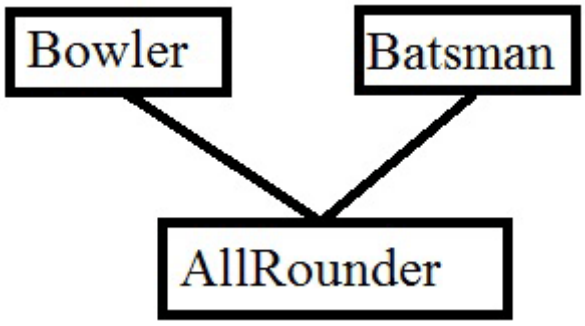
Single Inheritance

When a derived class inherits properties and behaviors of only one base class, it is called single inheritance. Look at the following figure. Cricketer is a derived class from the base class Player.



Multiple Inheritance

When a derived class inherits properties and behaviors of more than one base class, it is called multiple inheritance. In following figure, AllRounder is a derived class from two base classes: Bowler and Batsman.

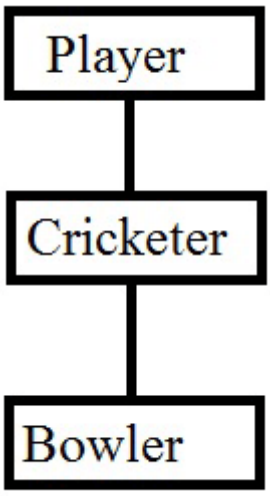


Hierarchical Inheritance

When properties and behaviors of one base class are inherited by more than one derived class, it is called hierarchical inheritance. In following figure, Bowler and Batsman are two derived classes from same base class Cricketer.

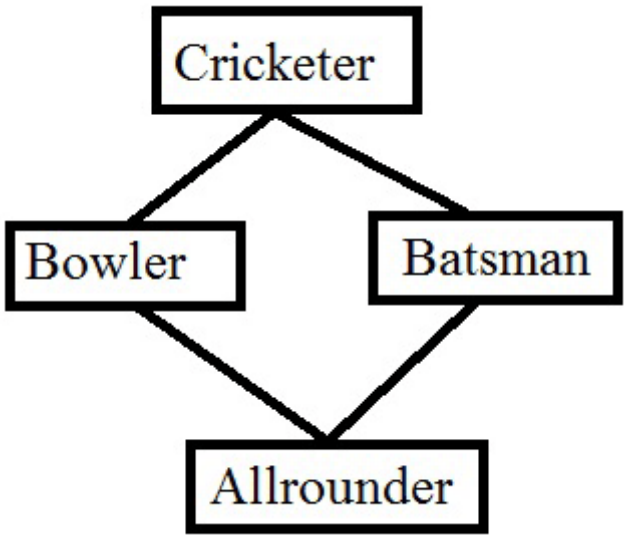
Multilevel Inheritance

When properties and methods of a derived class are inherited by another class, it is called multilevel inheritance. In following figure, Cricketer is derived class from Player base class. Then Cricketer acts as base class for the Bowler class.



Hybrid Inheritance

It is combination of multiple and multilevel inheritance.



Why Inheritance?

Inheritance is of great use for re-usability and extensibility of a module. If we have already defined a class and we need another class with all the characteristics of that class along with some extra properties and behavior, we can use the concept of inheritance.

Consider the previous example. First we have thought of a class named Player. Then we need the Footballer,Cricketer, RugbyPlayer classes. If you analyze, you can get the point that every Footballer, Cricketer, RugbyPlayer are indeed a player. They must hold all the properties and behavior of a player. So, you don't need to make the class from the scratch. Rather, you can extend the player class and add individual properties and behavior of a Footballer, Cricketer,RugbyPlayer in these class.

C++ syntax of inheritance of a class is: `class derived_class_name: access_specifier base_class_name`

The access level can be public, protected and private. Let us see how the access level differ in inheritance?
Public: Public and protected members of base class become public and protected members of derived class respectively, while private members of base class remain private to the base class only and members of the derived class cannot access the derived class.
Private: Public and protected members of base class become private members of derived class. Private members of base class remain private to base class members and can't be accessed from derived class.
Protected: Public and protected members of base class become protected members of derived class. Private members of base class remain private to base class members and can't be accessed from derived class.

A sample showing how inheritance works.

Code:

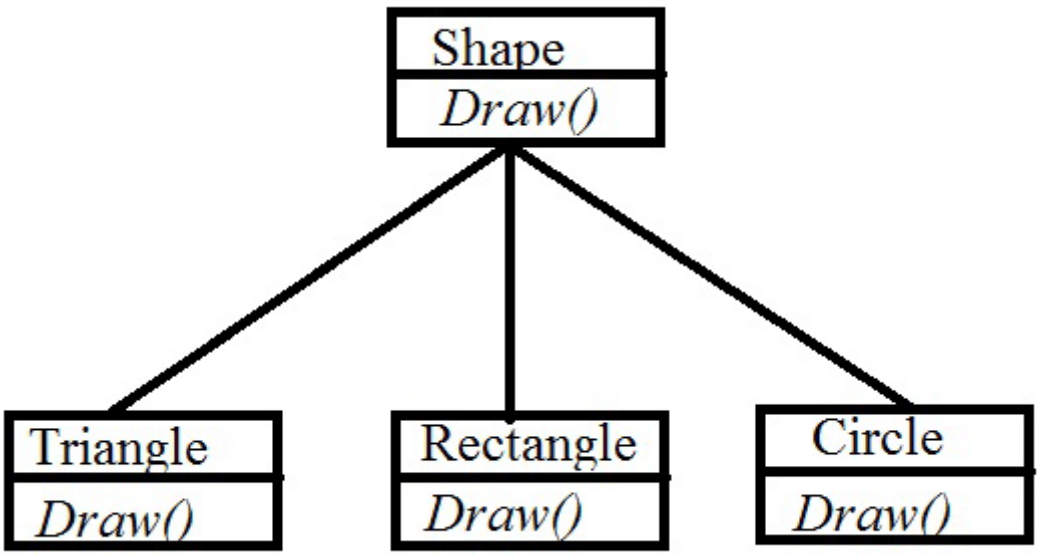
```
class Player {
        protected:
                        int mAge;
        string mName;
        public:
                void playerInfo() {
                        cout << "Name:" << mName << " " << "Age" << mAge << endl;
        }
```

```
        void setInfo(string str, int age) {
                mAge = age;
                mName = str;
        }
};
class Footballer : public Player {
        protected:
                int goalScore;
        double scoringAvg;
        public :
                void setFootballerInfo(int numOfGoal, double avgGoal) {
                goalScore = numOfGoal;
                scoringAvg = avgGoal;
        }
        void footballerInfo() {
                cout << "Name:" << mName << " " << "Age" << mAge << " " << "Scored Goal:"
        }
};
class Cricketer : public Player {
```

In the example above, Player is the base class. Footballer and Cricketer class are the derived class from the Player class. Object of Footballer class f1 can access the mName and mAge variables of player class through the Player class function setinfo. Cricketer class object c1 can perform the same operation. Due to public inheritance of the Player class, both Footballer and Cricketer class object can access the public function setInfo of Player class and inherit protected member mName and mAge.

Polymorphism

Polymorphism is another building block of object oriented programming. The philosophy that underlies is “one interface, multiple implementations.” It allows one interface to control access to a general class of actions. Polymorphism can be achieved both in compile time and run time.



Polymorphism through virtual function

Virtual function : While declaring a function as virtual, the keyword virtual must precede the signature of the function. Every derived class redefines the virtual function for its own need.

Uses of virtual function enable run time polymorphism. We can use base class pointer to point any derived class object. When a base class contains a virtual function and base class pointer points to a derived class object as well as the derived class has a redefinition of base class virtual function, then the determination of which version of that function will be called is made on run time. Different versions of the function are executed based on the different type of object that is pointed to.

The following example shows polymorphism through virtual function.

Code:

```
class Player {
        public:
                virtual void showInfo() {
                        cout << "Player class info" << endl;
        }
};
class Footballer : public Player {
        public :
                void showInfo() {
                        cout << "Footballer class info" << endl;
        }
};
class Cricketer : public Player {
        public :
                void showInfo() {
                        cout << "Cricketer class info" << endl;
        }
};
int main() {
        Player *pPl,p1;
        pPl = &p1;
        Footballer f1;
        Cricketer c1;
        pPl->showInfo();
        pPl = &f1;
        pPl->showInfo();
        pPl = &c1;
        pPl->showInfo();
        system("pause");
```

In the above example, Player is the base class, Footballer and Cricketer are the derived class from Player. Virtual function showInfo is defined in Player class. Then it is redefined in Footballer and Cricketer class. Here, pPl is the Player class pointer, p1, f1 and c1 are Player, Footballer and Cricketer class object.

At first, pPl is assigned the address of p1, which is a base class object. If showInfo is now called using pPl, showInfo function of base class is executed. Next, pPl points to address derived class (Footballer & Cricketer) . If showInfo is called now, the redefined showInfo function of Footballer & Cricketer class are executed. The key point is, which version of the showInfo function will be executed depends on which object is currently pointed by base class pointer. This decision is taken in run time, so it is an example of a run time polymorphism. This type of runtime polymorphism using virtual function is achieved by the base class pointer.

Function overloading

One way of achieving polymorphism is function overloading. When two or more functions share the same name with different parameter list, then this procedure is called function overloading and the functions are called overloaded function.

The following example shows polymorphism using function overloading.

Code:

```
class Player {
        string mName;
        int mAge;
        string mGameType;
        public:
                void setInfo(string str) {
                mName = str;
                cout << "Name :" << mName << endl;
        }
        void setInfo(string str, int age) {
                mAge = age;
                mName = str;
                cout << "Name :" << mName << " " << "Age :" << mAge << endl;
        }
        void setInfo(string str, int age, string game) {
                mAge = age;
                mName = str;
                mGameType = game;
                cout << "Name :" << mName  << " " << "Age :" << mAge << " " << "Game Type:
        }
};
int main() {
        Player p1;
        p1.setInfo("John Sena");
        p1.setInfo("C Ronaldo",25);
        p1.setInfo("J Kallis",38,"Cricket");
        return 0;
}
```

In this example, three functions have same name setInfo but different parameter list is defined for each. One function takes only one string parameter, another takes one string and one integer parameter and the last one takes two string and one integer as parameter. When we call setinfo function from Player class object p1, compiler looks at the argument list. It matches the argument list with the signature of three different funciton named setinfo, then one of the function is called according to the match. For example, when p1.setInfo("John Sena") is used, out of the three setinfo fucntion, the one with signature void setInfo(string str) is called and this one is executed. When p1.setInfo("C Ronaldo",25)is used, out of the three setinfo function, the one with signature void setInfo(string str, int age)is called and this one is executed.

Encapsulation, Inheritance and Polymorphism are the three concepts which must be needed to know while approaching to object oriented programming. In this article, we tried to clarify the basic knowledge of these concepts. Hopefully, you have enjoyed it.