OOPS

Inheritance

One of the most important concepts in object-oriented programming is that of inheritance.

Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application.

This also provides an ***opportunity to reuse the code functionality*** and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class.

This existing class is called the ***base*** class, and the new class is referred to as the ***derived*** class.

The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Base class access control

When one class inherits another, it uses this general form:

```
class derived-class-name : access base-class-name {
    // ...
}
```

Here access is one of the three keywords: ***public, private, or protected***.

The access specifier determines how elements of the base class are inherited by the derived class.

When the access specifier for the inherited base class is public all public members of the base class become public members of the derived class.

If the access specifier is private all public members of the base class become private members of the derived class.

In either case, any private members of the base class remain private to it and are inaccessible by the derived class.

> It is important to understand that if the access specifier is private public members of the base become private members of the derived class, but these are still accessible by member functions of the derived class.

If the access specifier is not present, it is private by default if the derived class is a class. If the derived class is a struct, public is the default access.

```
#include <iostream>

using namespace std;
class base {
   int x;

  public:
   void setx(int n) { x = n; }

   void showx() { cout << x << "\n"; }
};
// Inherit as public
class derived : public base {
   int y;

  public:
   void sety(int n) { y = n; }

   void showy() { cout << y << "\n"; }
};
int main() {
   derived ob;
   ob.setx(10);  // access member of base class
   ob.sety(20);  // access member of derived class
```

```
   ob.showx();   // access member of base class
   ob.showy();   // access member of derived class
   return 0;
}
```

Output

```
10
20
```

Here the variation of the program, this time derived inherits base as private, which causes error.

```
// This program contains an error
#include <iostream>

using namespace std;
class base {
   int x;

  public:
   void setx(int n) { x = n; }

   void showx() { cout << x << "\n"; }
};
// Inherit base as private
class derived : private base {
   int y;

  public:
   void sety(int n) { y = n; }

   void showy() { cout << y << "\n"; }
};
int main() {
   derived ob;
   ob.setx(10);  // ERROR now private to derived class
   ob.sety(20);  // access member of derived class - OK
   ob.showx();   // ERROR now private to derived class
   ob.showy();   // access member of derived class - OK
   return 0;
}
```

As the comments in this (incorrect) program illustrates, both showx() and setx() become private to derived and are not accessible outside it.

In other words, ***they are accessible from within the derived class***. Keep in mind that ***showx()*** and ***setx()*** are still public within base, no matter how they are inherited by some derived class.

This means that an object of type base could access these functions anywhere.

Using protected members : As you know from the preceding section, a derived class does not have access to the private members of the base class. However, there will be times when you want to keep a member of a base class private but still allow a derived class access to it. To accomplish this, C++ includes the protected access specifier. The protected access specifier is equivalent to the private specifier with the sole exception that protected members of a base class are accessible to members of any class derived from that base. Outside the base or derived classes, protected members are not accessible.

The protected access specifier can occur any where in the class declaration, although typically it occurs after the (default) private members are declared and before the public members.

The full general form of a class declaration is shown here:

```
class class-name {
    // private members
    protected: //optional
    // protected members
```

```
    public:
    //public members
};
```

When a protected member of a base class is inherited as public by the derived class, it becomes a protected member of the derived class.

If the base class is inherited as private, a protected member of the base becomes a private member of the derived class.

A base class can also be inherited as protected by a derived class. When this is the case, public and protected members of the base class become protected members of the derived class (of course, private members of the base remain private to it and are not accessible by the derived class).

The following program illustrates what occurs when protected members are inherited as public:

```
#include <iostream>

using namespace std;
class base {
  protected:  // private to base
   int a, b;  // but still accessible by derived
  public:
   void setab(int n, int m) {
      a = n;
      b = m;
   }
};
// Inherit as public
class derived : public base {
   int c;

  public:
   void setc(int n) { c = n; }
   // this function has access to a and b from base
   void showabc() { cout << a << " " << b << " " << c << "\n"; }
};
int main() {
   derived ob;
   // a and b are not accessible here because they
   // are
   // private to both base and derived.
   ob.setab(1, 2);
   ob.setc(3);
   ob.showabc();
   return 0;
}
```

Output:

```
1 2 3
```

If base is inherited as protected, its public and protected members become protected members of derived and therefore are not accessible within the main( ), i.e. the statement:

```
ob.setab(1, 2);// would create a compile-time error.
```