Inheritance

Inheritance

- Inheritance is a mechanism of reusing and extending existing classes without modifying them, thus producing hierarchical relationships between them.
- That enables creating new classes from existing ones.
- Furthermore, it is a powerful tool that enables us to model problems in a more logical way.
- Inheritance is almost like embedding an object into a class.
- Suppose that you declare an object x of class A in the class definition of B. As a result, class B will have access to all the public data members and member functions of class A.
- However, in class B, you have to access the data members and member functions of class A through object x.

Example


```
#include <iostream>
using namespace std;
class A {
 int data;
public:
 void f(int arg) {
  data = arg;
 }
 int g() { return data; }
};
class B {
public:   A x;
};
int main() {
 B obj;
 obj.x.f(20);
 cout << obj.x.g() << endl;
 //   cout << obj.g() << endl;
}
```

- In the main function, object obj accesses function A::f() through its data member B::x with the statement obj.x.f(20).
- Object obj accesses A::g() in a similar manner with the statement obj.x.g().
- The compiler would not allow the statement obj.g() because g() is a member function of class A, not class B.

Inheritance(cont.)

- Inheritance lets you include the names and definitions of another class's members as part of a new class.
- The class whose members you want to include in your new class is called a base class.
- Your new class is derived from the base class.
- The new class contains a sub-object of the type of the base class.

Example


```
#include <iostream>
using namespace std;
class A {
  int data;
public:
  void f(int arg) { data = arg; }
  int g() { return data; }
};
class B : public A {…};
int main() {
  B obj;
  obj.f(20);
  cout << obj.g() << endl;
}
```

- Class A is a base class of class B.
- The names and definitions of the members of class A are included in the definition of class B.
- Class B inherits the members of class A.
- Class B is derived from class A.
- Class B contains a sub-object of type A.

Derivation

- When you derive a class, the derived class inherits class members of the base class. You can refer to inherited members (base class members) as if they were members of the derived class.

```
class Base {
  public: int a,b;
};
class Derived : public Base {
  public: int c;
};

int main() {
 Derived d;
 d.a = 1; // Base::a
 d.b = 2; // Base::b
 d.c = 3; // Derived::c
}
```

Derivation - Redefine

- If you redefine base class members in the derived class, you can still refer to the base class members by using the :: (scope resolution) operator.

Example


```
class Base {
public:  char* name;
       void display() { cout << name << endl; }
 };

class Derived: public Base {
   public:  char* name;
      void display() { cout << name << ", " << Base::name << endl; }
 };

int main() {
  Derived d;
  d.name = "Derived Class";
  d.Base::name = "Base Class";
  // call Derived::display()
  d.display();
  // call Base::display()
  d.Base::display();
}
```

Derived Class, Base Class
Base Class

Derivation using pointer

- You can use a pointer or a reference to a derived class object in place of a pointer or reference to its base class.
- For example, you can pass a pointer or reference to a derived class object D to a function expecting a pointer or reference to the base class of D.
- You do not need to use an explicit cast to achieve this; a standard conversion is performed.
- You can implicitly convert a pointer to a derived class to point to an accessible unambiguous base class.
- You can also implicitly convert a reference to a derived class to a reference to a base class.

Example


```
class Base {
public: char* name;
        void display() { cout << name << endl; }
};
class Derived: public Base {
public: char* name;
        void display() { cout << name << ", " << Base::name << endl;
  }
 };

int main() {
  Derived d;
  d.name = "Derived Class";
  d.Base::name = "Base Class";
  Derived* dptr = &d;
  Base*  bptr = dptr; // standard conversion from Derived* to Base*
  bptr->display(); // call Base::display()
 }
```

Base Class

Protected members

- A protected nonstatic base class member can be accessed by members and friends of any classes derived from that base class by using one of the following:
  - A pointer to a directly or indirectly derived class
  - A reference to a directly or indirectly derived class
  - An object of a directly or indirectly derived class

Protected members (cont.)

- If a class is derived privately from a base class, all protected base class members become private members of the derived class.
- If you reference a protected nonstatic member x of a base class A in a friend or a member function of a derived class B, you must access x through a pointer to, reference to, or object of a class derived from A. However, if you are accessing x to create a pointer to member, you must qualify x with a nested name specifier that names the derived class B.

Example

The compiler would and would not allow apply for g() except for the following: The compiler allows i = 2 because it is equivalent to this->i = 2.

```
class A {
public:
protected: int i;
};
class B : public A {
  friend void f(A*, B*);
void g(A*);
 };
void f(A* pa, B* pb) {
// pa->i = 1;
pb->i = 2;
// int A::* point_i =
  &A::i;
int A::* point_i2 = &B::i;
 }
```

The compiler would not allow pa->i = 1 because pa is not a pointer to the derived class B.

The compiler would not allow int A::* point_i = &A::i because i has not been qualified with the name of the derived class B.

```
void B::g(A* pa) {
// pa->i = 1;
i = 2;
// int A::* point_i =
  &A::i; int A::*
  point_i2 = &B::i;
}

void h(A* pa, B* pb) {
// pa->i = 1;
// pb->i = 2;
}
```

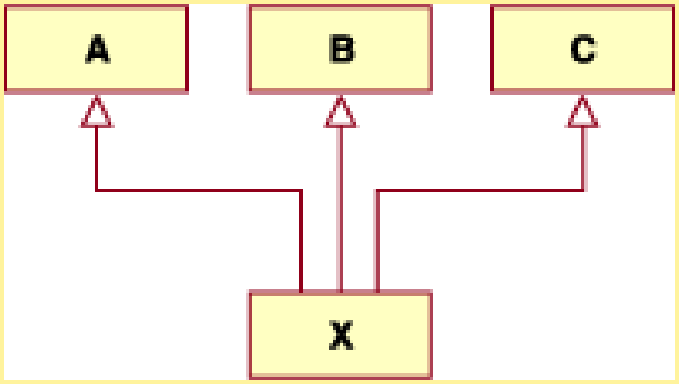
Function h() cannot access any of the protected members of A because h() is neither a friend or a member of a derived class of A.

Access control of base class members

- When you declare a derived class, an access specifier can precede each base class in the base list of the derived class. This does not alter the access attributes of the individual members of a base class as seen by the base class, but allows the derived class to restrict the access control of the members of a base class.
- You can derive classes using any of the three access specifiers:
  - In a **public** base class, public and protected members of the base class remain public and protected members of the derived class.
  - In a **protected** base class, public and protected members of the base class are protected members of the derived class.
  - In a **private** base class, public and protected members of the base class become private members of the derived class.
- In all cases, private members of the base class remain private. Private members of the base class cannot be used by the derived class unless friend declarations within the base class explicitly grant access to them.

Multiple inheritance

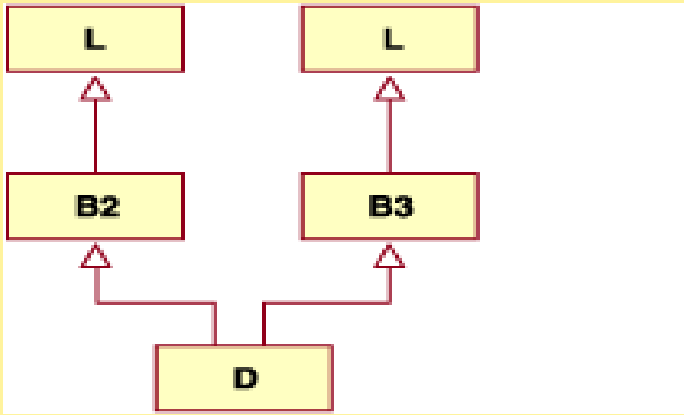

```
class A { /* ... */ };
 class B { /* ... */ };
 class C { /* ... */ };
class X : public A, private B, public C { /* ... */ };
```



Multiple inheritance (cont.)

- However, a derived class can inherit an indirect base class more than once, as shown in the following example:

```
class L { /* ... */ };
// indirect base class
class B2 : public L { /* ... */ };
class B3 : public L { /* ... */ };
class D : public B2, public B3 { /* ... */ }; // valid
```



What is inherited from the base class?

- In principle, a derived class inherits every member of a base class except:
  - its constructor and its destructor
  - its operator=() members
  - its friends
- Although the constructors and destructors of the base class are not inherited themselves, its default constructor (i.e., its constructor with no parameters) and its destructor are always called when a new object of a derived class is created or destroyed.
- If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class:
  derived_constructor_name (parameters) : base_constructor_name (parameters) {...}

Example


```
class mother {
public:   mother () { cout << "mother: no parameters\n"; }
       mother (int a) { cout << "mother: int parameter\n"; }
 };
class daughter : public mother {
public:   daughter (int a) { cout << "daughter: int
  parameter\n\n"; }
};
class son : public mother {
public:  son (int a) : mother (a) { cout << "son: int
  parameter\n\n"; }
 };
int main () {
  daughter cynthia (0);
  son daniel(0);
  return 0;
}
```

```
mother: no parameters
daughter: int parameter

mother: int parameter
son: int parameter
```

Overloading member functions from base and derived classes

- A member function named f in a class B will hide all other members named f in the base classes of A, regardless of return types or arguments.
- Example:

```
class A {
  void f() { }
};
class B : public A {
  void f(int) { }
};
int main() {
  B obj_B;
  obj_B.f(3);
  // obj_B.f();
}
```

The compiler would not allow the function call obj_B.f() because the declaration of void B::f(int) has hidden A::f().

Introducing - using


```
class A {
  void f() { }
};
class B : public A {
  using A::f;
  void f(int) { }
 };
int main() {
  B obj_B;
  obj_B.f(3);
  obj_B.f();
}
```

To overload, rather than hide, a function of a base class A in a derived class B, you introduce the name of the function into the scope of B with a using declaration.
The example is the same as the previous example except for the using declaration using A::f:

Example 2


```
class A {
  void f() { }
  void f(int) {
    cout << "void A::f(int)" << endl;
  }
};
class B : A {
  using A::f;
  void f(int) { cout << "void B::f(int)" << endl; }
};

int main() {
 B obj_B;
 obj_B.f(3);
}
```

void B**::f(int**)

Suppose that you introduce a function f from a base class A ,a derived class B with a using declaration, and there exists a function named B::f that has the same parameter types as A::f.
Function B::f will hide, rather than conflict with, function A::f.

Ambiguous base classes

- When you derive classes, ambiguities can result if base and derived classes have members with the same names.
- Access to a base class member is ambiguous if you use a name or qualified name that does not refer to a unique function or object.
- The declaration of a member with an ambiguous name in a derived class is not an error.
- The ambiguity is only flagged as an error if you use the ambiguous member name.

Example


```
class B1 {
public: int i; int j;
        void g(int) { }
};
class B2 {
public: int j;
        void g() { }
};
class D : public B1, public B2 {
public: int i;
};
int main() {
   D dobj;
   D *dptr = &dobj;
   dptr->i = 5;
   // dptr->j = 10;
   dptr->B1::j = 10;
   // dobj.g();
   dobj.B2::g();
}
```

The statement dptr->j = 10 is ambiguous because the name j appears both in B1 and B2.

The statement dobj.g() is ambiguous because the name g appears both in B1 and B2, even though B1::g(int) and B2::g() have different parameters.

The statement dobj.B2::g(); resolve ambiguity by qualifying a member with its class name using the scope resolution (::) operator.

Virtual functions

- By default, C++ matches a function call with the correct function definition at compile time.
- This is called static binding.
- You can specify that the compiler match a function call with the correct function definition at run time; this is called dynamic binding.
- You declare a function with the keyword virtual if you want the compiler to use dynamic binding for that specific function.

Default : no virtual


```
#include <iostream>
using namespace std;
class A {
  void f() {
    cout << "Class A" << endl;  }
};
class B: public A {
  void f() {
    cout << "Class B" << endl;  }
 };
void g(A& arg) { arg.f(); }
int main() {
B x;
g(x);
}
```

- When function g() is called, function A::f() is called, although the argument refers to an object of type B.
- At compile time, the compiler knows only that the argument of function g() will be a reference to an object derived from A; it cannot determine whether the argument will be a reference to an object of type A or type B.
- However, this can be determined at run time.

The following is the output of the above example:

```
Class A
```

Virtual definition


```
#include <iostream>
using namespace std;
class A {
  virtual void f() {
  cout << "Class A" << endl; }
};
class B: public A {
  void f() {
  cout << "Class B" << endl;  }
};
void g(A& arg) { arg.f(); }
int main() {
  B x;
  g(x);
}
```

- **This example is the same as the previous example, except that A::f() is declared with the virtual keyword.**
- **The virtual keyword indicates to the compiler that it should choose the appropriate definition of f() not by the type of reference, but by the type of object that the reference refers to.**

<u>The following is the output of the above example:</u>

```
Class B
```

Overloading virtual functions


```
class A {
  virtual void f() { cout << "void A::f()" << endl; }
  virtual void f(int) { cout << "void A::f(int)" << endl; }
 };
class B : A {
  void f(int) { cout << "void B::f(int)" << endl; }
};

int main() {
  B obj_B;
  B* pb = &obj_B;
  pb->f(3);
  pb->f();
}
```

```
void B::f(int)
void A::f()
```

Overloading virtual functions(cont.)

דוגמא 1

```
class A{
public:
    virtual void f() {cout<<"fa"<<endl;}
    void k()         {cout<<"ka"<<endl;}
};

class B : public A{
public:
    virtual void f() {cout<<"fb"<<endl;}
    void k()         {cout<<"kb"<<endl;}
};

class C : public B{
public:
    virtual void f()  {cout<<"fc"<<endl;}
    virtual void k() {cout<<"kc"<<endl;}

};
```

```
int main() {
    C c;
    c.f();      // fc (regular call)
    c.k();      // kc (regular call)
    A& a1 = c;
    a1.f();     // fc (f virtual in A)
    a1.k();     // ka (k not virtual in A)
    B& b1 = c;
    b1.f();     // fc (f virtual in A)
    b1.k();     // kb (k not virtual in B)
    B b2 = c;
    b2.f();     // fb (slicing)
    b2.k();     // kb (slicing)
    A a2 = c;
    a2.f();     // fa (slicing)
    a2.k();     // ka (slicing)
    A *a3 = &c;
    a3->f(); // fc (f virtual in A)
    a3->k(); // ka (k not virtual in A)
    B *b3 = &c;
    b3->f(); // fc (f virtual in B)
    b3->k(); // kb (k not virtual in B)
}
```