Chapter 13 Class Inheritance

13.1 An overview

- *Class inheritance* is a key aspect of OOP. This mechanism makes class designing more analogous to the reality and enhances code reusability.

  > The other key aspects of OOP is: *data abstraction*, that separate interface from implementation (realized by classes); *dynamic binding*: that use objects of different classes while ignoring the details of how they differ (realized by virtual functions).

- Classes related by inheritance form a hierarchy. The class on which a new class is built upon is called the *base class*, or *parent class*; the new class is called the *derived class*, or *child class*.
- The derived class inherits ALL the members of the base class, except constructors, the destructor, and assignment operators. The derived class may have its own additional members, among which its own constructors are **required** to have, be it explicitly provided or compiler-generated.

  > Friendships are NOT inherited, since friends are NOT member. Therefore, a friend of the base/derived class cannot access the derived/base class.

- In the memory, the derived-class's additional non-`static` data member is appended to the inherited non-`static` data member.

  > Static data members are stored in the heap (belongs to the data area). Member functions, together with other non-member functions, are stored elsewhere (the code area). These are ignored by `sizeof`.
  >
  > Plus, if a class is empty, `sizeof` returns 1.

- The *is-a* relationship: the derived class "is a kind of" the base class. To be more specific, public inheritance implements an *is-a* relationship because the derived class acquires all the features (implementations and interfaces BOTH) of the base class.

  > A sparrow is a kind of bird; a swallow is a kind of bird; a wolf is a kind of mammal; a proton is a kind of baryon; a CEO is a kind of employee.

- Basic format of declaring class inheritance:
  - `class DerivedClass : derivation_list {...};`

    derivation list: `access_specifier BaseClass1, access_specifier BaseClass2, ...`
  - *Access specifier*s: `public`, `protected`, `private` (by default). These three keywords have subtly different meanings from their usages inside a class declaration body.
  - A class can be made uninheritable by postfixing keyword `final` to the class name: `class ClassName final {...};`. (For another usage of `final`, see 13.3)
  - The forward declaration (see 11.2) of a derived class should NOT have its derivation list; just forward-declare it as a regular class: `class DerivedClass;`.
- When a program constructs a derived-class object, it first constructs a base-class object by calling the base-class constructor BEFORE entering the body of the derived-class constructor.

- A derived-class object's inherited data have to be initialized by the base-class constructor. Syntactically it is realized by a initializer list in which contains the name of the base-class followed by parentheses enclosing values the base-class constructor needs (do NOT include the types).

```
DerivedClass::DerivedClass(all_arguments)
  : BaseClass(base_val1, base_val2, ...)
{ /* take care of the rest of the arguments */ }
```

- If your intended base-class constructor is a default one, a.k.a. the constructor that takes no argument, that term in the initializer list can be omitted.

```
DerivedClass::DerivedClass(all_arguments) : BaseClass() {...} // ok
DerivedClass::DerivedClass(all_arguments) {...}               // ok
```

> Initializer list can be used ONLY in constructors (including copy-constructors).

- When a program destroys a **non-dynamic storage** derived-class object, it first invokes the destructor of the derived class, and then invokes the destructor of the base class. It is in the opposite order used to construct an object. If the to-be-destroyed derived-class object is of **dynamic storage**, then things get involved.
  - If the pointer to this object is of the derived-class type, then the process is the same as described above;
  - If the pointer to this object is of the base-class type (it is ok, see 13.2), and the base-class destructor is `virtual` (see 13.3), then the process is the same, too;
  - If the pointer to this object is of the base-class type, and the base-class destructor is non-`virtual`, then only the base-class destructor is called. This process leads to a memory leak.
- When a program copy or assign a derived-class object to another:
  - If the derived class does not have explicit copy-constructor or assignment-operator: it automatically first invokes the copy-constructor or assignment-operator of the base class, and then automatically invokes the compiler-generated copy-constructor or assignment-operator of the derived class.
  - If the derived class has explicit copy-constructor or assignment-operator: the program ONLY invokes the derived-class one. So you should manually initialize the base-class portion. Like this:

```
Derived::Derived(const Derived &d): Base(d)    // copy-constructor
{...}  // Base(d) only sets the Base part (see 13.2)
Derived &Derived::operator=(const Derived &d)  // assignment-operator
{ Base::operator=(d) /* assign the Base part */ ; ...}
```

- A derived-class can redefine (overload) a base-class method, by declaring its own cognominal method with a different definition. It is a form of *compile-time polymorphism* (see13.3).
- A derived class does NOT have direct access to the `private` members of the base class, though it does inherits them (it can be verified by `sizeof`); it has to invoke non-`private` base-class methods to access base-class `private` members.
- `protected` members: a blend of `private` and `public`.
  - Like `private`, `protected` members are inaccessible outside the class scope;

    - Like `public`, `protected` members are accessible inside the class scope, including class members and friends.
- Access specifiers in inheritance: the purpose of derivation access specifiers is to control access that users of the derived class - including other classes derived from the derived class - have to the members inherited from the base class.
    - `public` inheritance: `class DerivedClass : public BaseClass {...};`

base-class members	becomes ... in derived-class
`private`	(only accessed through base-class methods)
`protected`	`protected`
`public`	`public`

    - `protected` inheritance: `class DerivedClass : protected BaseClass {...};`

base-class members	becomes ... in derived class
`private`	(only accessed through base-class methods)
`protected`	`protected`
`public`	`protected`

    - `private` inheritance (default): `class DerivedClass : private BaseClass {...};`

base-class members	becomes ... in derived class
`private`	(only accessed through base-class methods)
`protected`	`private`
`public`	`private`

    - Exempting individual members: sometimes we need to change the access level of a member that a derived class inherits. We can do so by providing a `using` declaration in your intended section. However, this technique can be applied ONLY to names that the derived class is allowed to directly access.

        > It applies to **name**s, which means that it is just `using Base::name;`, regardless of the name is a data member name or a method name.

```
class Base {public: int size; void show(int);};
class Derived : private Base
{ public: using Base::size; protected: using Base::show; };
```

13.2 Special relationships ensured by public inheritance

- A pointer or reference to a base-class type CAN be bound to an object of a class derived from that base class with public inheritance.

> It is transitive: a pointer or reference to a base-class type can be bound to an object of a class publicly derived from a class publicly derived from that base class, and so forth.

- The special rule above has an **important implication**: when a pointer or reference to a class type is declared, the actual type of the object it points/refers to is unknown to the compiler - it might be either the class type itself, or a class type derived from it.
- However, a base-class pointer/reference can ONLY access base-class members. The accessibility is also controlled by access control (e.g. `private`), of course.

```
class Base {public: int x;}; class Derived : public Base {public: int y;};
int main()
{
  Derived od; od.x = 1; od.y = 2;
  Base *p = &od; // allowed. Base * <-- &Derived
  Base &r = od;  // allowed. Base & <-- Derived
  std::cout << p->x << endl;// allowed. 'x' is a member of class 'Base'
  std::cout << r.y << endl; // error. 'y' is not a member of class 'Base'.
}
```

- The rule above is termed *derived-to-base conversion*. (Yes, you read it right; it is NOT base-to-derived). It is also termed *upcasting* - figuratively the derived class is more detailed, thus more down-to-earth, whereas the base class is aloof and detached.
- Static type and dynamic type: two concepts
  - The *static type* of a variable or other expression is always known at compile time - it is the type with which a variable is declared or that an expression yields.
  - The *dynamic type* of a variable or other expression is unknown until runtime.
  - The dynamic type of an expression that is neither a reference nor a pointer is always the same as the expression's static type. But for pointers and references, the dynamic type might differs from the static type. For instance, in the code snippet above, pointer `p`'s static type is `Base`, but its dynamic type is `Derived` since it points to an object of `Derived` class.
- A pointer/reference to a derived-class type CANNOT be implicitly bound to an object of base-class. But this *downcasting* is allowed by C-style forced type casting and `static_cast` - an explicit C++ type casting.

```
Base ob;
Derived *pd = &ob;                          // error
Derived *pd = static_cast<Derived *>(&ob); // ok, though not recommended
std::cout << pd->y << std::endl;  // ok, but the output is garbage
```

- The conversion rule provides some commonly-used techniques:
  - You can assign a derived-class object to a base-class object (the derived-class own additional members are lost, of course). Because an implicit assignment operator of the base class is at work: `BaseClass &operator=(const BaseClass &);`, and its formal argument (`BaseClass &`) can take a derived-class object or its reference.

```
Derived od; Base ob;
ob = od;                        // assign: Base <-- Dirived
```

- You can copy a derived-class object to initialize a base-class object (the derived-class own additional members are lost, of course). Because an implicit copy constructor of the base class is at work: `BaseClass(const BaseClass &);`, and its formal argument (`BaseClass &`) can take a derived-class object or its reference.

```
Derived od;
Base ob1 = od;                  // initialize: Base <-- Dirived, #1
Base ob2(od);                   // initialize: Base <-- Dirived, #2
Base ob3 = Base(od);            // initialize: Base <-- Dirived, #3
Base *ob4 = new Base(od);       // initialize: Base <-- Dirived, #4
```

- In the two techniques above, it is worth noting that only the base-class portion is successfully assigned or copied.
  - Because in the assignment or initialization statement, the invoking object is a base-class object, and therefore the working assignment operator or copy constructor belongs to the base class.
  - In these case, we say the derived-class portion is *sliced down*.

13.3 Polymorphism and virtual functions

- *Polymorphism* is a key idea of C++. This word means "many forms" is Greek. To be specific, it means the feature that an interface can have more than one implementation. C++'s polymorphism has two kinds:
  - *Compile-time polymorphism*, or *early binding*, or *static binding*: the program knows which implementation to use during compile-time. It is realized by overloading.
  - *Runtime polymorphism*, or *late binding*, or *dynamic binding*: the program does not know which implementation to use until runtime. It is realized by calling virtual functions through references or pointers.

> Virtual functions is NOT overloaded functions. These two are mutually exclusive.

- A virtual function is a member function that seems to behave differently for the derived class than it does for the base class - "It has multiple behaviors".

> However, virtual function is not the only mechanism to achieve this pattern of behavior. Function overloading can do that, too - a base-class method can be redefined by a cognominal derived-class method. Plus, you can still call the overloaded cognominal base-class method by using `BaseClassName ::`.

- The difference between virtual and non-virtual member functions: a virtual function is a member function the base class expects its derived classes to override; a non-virtual one is a member function the base class expects its derived class to inherit.
- A member function is made virtual be prefixing the keyword `virtual` to its declaration (NOT its definition): `virtual returnType methodName(arg_list);`
  - Postfixing `= 0` to a virtual function's declaration makes it a *pure virtual function*. For a pure virtual function, the `virtual` prefix in its declaration can be dropped.

> Non-member functions CANNOT be virtual.

- A non-pure virtual function MUST have a definition - even an empty one is accepted. A pure virtual function does NOT need one, however, as its definition is ignored outright.

> A non-virtual function, be it a member or not, does not need a definition unless the program uses it. (But the compiler has no way to determine if a non-pure virtual function is used.)

- Calls to virtual functions **might be** resolved at runtime.
  - If a virtual function call is plain - neither through reference nor pointer, the call is bound at compile time, because the dynamic type of the invoking object is the static type of that object, which is known to the compiler.
  - If a virtual function call is made through a reference or a pointer, the call is bound at runtime, because the dynamic type of the invoking object is unknown to the compiler - it might be the class type the reference/pointer claims to be bound to, or might be that class's derived class type. This embodies runtime polymorphism.
- Subtle yet crucial is the difference between overriding a base-class virtual function and overloading a base-class non-virtual function, for the derived class.
  - If a derived-class overrides a base-class virtual function, then calling the function through a reference/pointer that is of the base-class type but bound to a derived-class object results in calling the derived-class version.
  - If a derived-class overloads a base-class non-virtual function, then calling the function through a reference/pointer that is of the base-class type but bound to a derived-class object results in calling the base-class version.

```
class BaseV { public: virtual void show();};  // use virtual function
void BaseV::show() { std::cout << "BaseV virtual show()!\n"; }
class DerivedV : public BaseV { public: void show();};
void DerivedV::show() { std::cout << "DerivedV show()!\n"; }
// use:
DerivedV dV; BaseV bV; BaseV *pBaseV = &dV;   // Base * <-- &Derived
bV.show();      // output: BaseV virtual show()!
pBaseV->show(); // output: DerivedV show()!
```

```
class BaseO { public: void show();};       // use function overloading
void BaseO::show() { std::cout << "BaseO virtual show()!\n"; }
class DerivedO : public BaseO { public: void show();};
void DerivedO::show() { std::cout << "DerivedO show()!\n"; }
// use:
DerivedO dO; BaseO bO; BaseO *pBaseO = &dO;// Base * <-- &Derived
bO.show();      // output: BaseO virtual show()!
pBaseO->show(); // output: BaseO virtual show()!
```

  - The compiler distinguishes these two situations by examining the called member function's `virtual`-ness at compile-time.
- Details of overriding a base-class virtual function:
  - When a derived class overrides a virtual function, it may, but is NOT required to, repeat the `virtual` keyword. Once a function is declared as `virtual`, it remains `virtual` in the derived classes, and derived classes of derived classes, and so on.
  - A derived-class function that overrides the base-class virtual function must have the same argument type(s) as the base-class virtual function.

    - A derived-class function that overrides the base-class virtual function must have the same return type as the base-class virtual function, with one exception.
        - The exception is: if the base-class virtual function returns a pointer/reference bound to the base class, then the derived-class function that overrides the virtual function is allowed to return a pointer/reference bound to the derived class. Plus, of course, it can still return a base-class pointer/reference.
    - An unrelated, regular derived-class member function can have the same name as the base-class virtual function, provided it has a different argument list.
- Postfixing `final` to a virtual function declaration prevents that function from being overridden by derived classes. (For another usage of `final`, see 13.1)
    - It is useful when you want to prevent a derived-class virtual function with which that class overrides the base-class virtual function, from being overridden by derived classes of that derived class.
- Postfixing `override` to the derived-class member function explicitly indicates its purpose of overriding a base-class virtual function.
    - By doing so, the compiler will complain if the overriding is invalid, say, there is no such virtual function in the base class, or, the argument list is wrong. So this technique is helpful to mistake-proofing.
- Circumventing the virtual mechanism
    - In some cases, we want to prevent dynamic binding of a call to virtual function; we can to force the call to use the base-class version, by using the scope operator.

      ```
      class Base {public: virtual void show() {...}};
      class Derived : public: Base {public void show() {...}};
      Derived od; Base *pBase = &od; Base &rBase = od;
      pBase->show();  rBase.show();             // call the Derived version
      pBase->Base::show(); rBase.Base::show();  // call the Base version
      ```

    - Another scenario in which this technique is applied is that the derived-class version itself needs to call the base-class version.

      ```
      void Derived::show() { Base::show(); /* do something else */ }
      ```

      In such case, if you inadvertently omit the `Base::` part, the function above becomes an infinite recursion.
- Constructors CANNOT be `virtual`. A derived class does NOT inherit base-class constructors, so actually there is no point to making them `virtual`, anyway.
- Virtual destructor
    - The base-class destructor should be explicitly declared as `virtual`, if a `delete` operator is applied to a pointer that claims to be bound to this class actually points to a derived-class object.
    - By doing so, when you `delete` a base-class pointer that points to a derived-class object, the derived-class version of destructor is called (and the base-class version is called automatically afterwards). If you do not make the base-class destructor `virtual`, then only the base-class destructor is called.

```
class BaseV{public: virtual ~Base() = default; };// virtual destructor
class BaseN{public: ~Base() = defualt; }; // non-virtual destructor
class DerivedV : public BaseV {}; class DerivedN : public BaseN {};
BaseV *p = new DerivedV; BaseN *q = new DerivedN;
delete p; // the Derived version of destructor is called, and the Base
          // version is called automatically afterwards
delete q; // only the Base version of destructor is called
```

- The mechanism of dynamic storage dictates that it is the programmer's responsibility to manage the memory. However, if the storage class of the derived-class object is static or automatic, doing so is unnecessary - the compiler will choose the correct version to call during compile-time.

```
class Base{public: ~Base() = default; }; // non-virtual destructor
class Derived : public Base {};
Base od = Derived;   // automatic storage
// When the compiler destroys the object, the Derived version of
// destructor is called, and the Base version is called automatically
// afterwards. No need for manual interference.
```

- It is recommended to make a base-class destructor `virtual`, regardless of whether there would be a base-class pointer bound to a derived-class object.
  - Doing so makes the code more easy to be expanded, especially for code that is meant to be a part of a library.
  - Moreover, it is not uncommon to make the destructor of any class `virtual`, regardless whether the class is to be inherited and whether the class needs an explicit destructor.

- The overhead of virtual function mechanism:
  - If a class contains virtual functions, then each object of that class has an additional hidden data member, which is a pointer to that class's *virtual function table* (VTBL). The VTBL contains addresses of these virtual function's instructions.
  - If a class is derived from a base class which has virtual functions, then the derived class has its own copy of the base-class VTBL, and each derived-class object has a hidden member that points to the derived-class VTBL.
  - If the derived class overrides a base-class virtual function, then the corresponding entry in the derived-class VTBL is changed to the new version's address. If the derived class adds another virtual function, then its VTBL gets a new entry that stores the address of the new virtual function.
  - In short, the overhead of virtual function mechanism is:
    - Each object has its size increased to hold a pointer;
    - For each class, an array-of-pointer (the VTBL) is allocated in the memory. Each pointer has an address of a virtual function's instructions.
    - For each virtual function call, there is an extra step of looking up the address in the VTBL.

13.4 Abstract base classes (ABCs)

- If a class has pure virtual functions (see 13.3), then the class CANNOT be instantiated (but can still be inherited). That is, the class cannot be used to declare an object. This class is termed *abstract base class* (ABC), or *abstract class*.

> An example in which an abstract class makes sense: men and women have many things in common, yet conceptually it is awkward to derive a Woman class from a Man class, and so is deriving a Man from a Woman. The most natural way is to define a Human class as a base class, and derive Man and Woman from Human, respectively. However, there is no such object that is an instance of Human - biologically a person is either male or female. So the Human class should be abstract so that it can be inherited but not be instantiated.

- A pure virtual function does NOT need a definition; its definition, if provided, is ignored anyway.
- A derived class of an ABC must override the latter's all pure virtual functions, or the derived class itself is still an ABC, which is uninstantiable.
- ABCs and "interface contract"
    - One school of thought holds that if you design an inheritance hierarchy of classes, the only concrete classes, namely the instantiable classes, should be those that never serve as a base class. This approach tends to produce cleaner designs with fewer complications.
    - Before designing an ABC, you first have to develop a model of what classes are needed to represent a programming problem and how they relate to one another.
    - The ABC methodology is a much more systematic, disciplined way to approach inheritance than the more ad hoc, spur-of-the-moment approaches beginners might take.
    - One way of thinking about ABCs is to consider them an enforcement of interface. An ABC demands that its pure virtual functions be overridden in any concrete derived classes - forcing the derived class to obey the rules of interface the ABC has set.
    - This model is common in *component-based programming* paradigms, in which the use of ABCs allows the component designer to create an “interface contract” where all components derived from the ABC are guaranteed to uphold **at least** the common functionality specified by the ABC.

13.5 Some observations

- Observations on using base-class methods
    - A derived object automatically uses inherited base-class methods if the derived class has not redefined (overloaded) the method.
    - A derived-class constructor automatically invokes the base-class default constructor if you don’t specify another constructor in the initialization list.
    - A derived-class constructor explicitly invokes the base-class constructor specified in the initialization list.
    - Derived-class methods can use the scope-resolution operator to invoke public and protected base-class methods.
- What is not inherited: constructors (including copy-constructors), the destructor, assignment-operators.
- `Private` members, be it a data member or a method, are inherited but can only be accessed by base-class non-`private` methods.
- Class function summary (**bold** words require attentions)

Function	Inherited	Member or friend	Generated by default	Can be `virtual`	Return type
constructors	**NO**	member	**YES**	**NO**	**NO**
destructor	**NO**	member	**YES**	yes	**NO**
`=`	**NO**	member	**YES**	yes	yes
`&`	yes	either	**YES**	yes	yes
conversion	yes	member	no	yes	yes
`()`	yes	member	no	yes	yes
`[]`	yes	member	no	yes	yes
`->`	yes	member	no	yes	yes
`op=` (e.g. `+=`, `&=`, `>>=`)	yes	either	no	yes	yes
`new`	yes	`static` **member**	no	**NO**	`void*`
`delete`	yes	`static` **member**	no	**NO**	`void`
other operators	yes	either	no	yes	yes
other methods	yes	member	no	yes	yes
friends	**NO**	friend	no	no	yes

- Destructor considerations: a base class destructor should be `virtual`. That way, when you `delete` a derived-class object via a base-class pointer, the program uses the derived-class destructor followed by the base-class destructor, rather than using only the base-class destructor.
- Passing an object by value versus passing by reference
  - In general, if you write a function using an object argument, you should pass the object by reference rather than by value.
  - One reason for this is efficiency. Passing an object by value involves generating a temporary copy, which means calling the copy constructor and then later calling the destructor. Calling these functions takes time, and copying a large object can be quite a bit slower than passing a reference.
  - If the function does not modify the object, you should declare the argument as a `const` reference.
  - Another reason for passing objects by reference is that, in the case of inheritance using virtual functions, a function defined as accepting a base-class reference argument can also be used successfully with derived classes.
- Returning an object directly versus returning a reference
  - Some class methods return objects. Sometimes a method must return an object, but if it is not necessary, you should use a reference instead.
  - Returning an object directly is analogous to passing an object by value: both processes generate

temporary copies. Similarly, returning a reference is analogous to passing an object by reference: both the calling and the called function operate on the same object.
    - However, it is not always possible to return a reference. A function should not return a reference to a temporary object created in the function because the reference becomes invalid when the function terminates.
    - As a rule of thumb, if a function returns a temporary object created in the function, you should not return a reference. If a function returns an object that was passed to it by reference/pointer, you probably should return a reference.
- The *dominance* rule in member name ambiguity resolution
    - If a derived class does NOT overload or override a base-class member, calling that member's name within the derived-class scope or with a derived-class object results in calling the most recently defined base-class version of that member in the inheritance hierarchy.
    - If the derived class does overload or override it, calling that member's name is calling the derived-class version, and other version(s) can be called by using `::`.
    - This rule pays no attention to access identifiers, which means calling a member does NOT guarantee accessing the member.
- The `this` pointer to a derived-class object can be converted to a base-class type. Therefore you can use `(Base &)(*this)` within the derived-class scope in lieu of the name of the base-class "object" that is integrated into the derived-class object, and this "object" is pointed to by `(Base *)this`. This technique is also used in calling a friend of the base class with a derived-class object.

□