Chapter 14 Composition, More on Inheritance, and Class Templates

14.1 Composition (1): classes with object members

- A class can take objects of other classes as its data members. We say the object of the latter class is *contained* in an object of the former class. This relationship is termed *composition*, or *has-a* relationship (different from *inheritance*, or *is-a*).
- The difference between *has-a* and *is-a* is that, the former does NOT inherit the interfaces of the base class. In other words, base-class interfaces are hidden unless explicitly called on purpose. With this in mind, we can say the derived class does NOT acquire all the features (implementation and interfaces) of the base class, so it is not "a kind of" the base class.
- Initializing contained objects: you MUST use an initialization list to invoke their constructors. Note that this syntax is different from initializing inherited members.
  - For **inherited members**, constructors use the **class name** of the base class to invoke a specific base-class constructor. If you omit it, the base-class default constructor will be invoked instead. The class name is followed by a parenthesis pair enclosing argument values the base-class constructor needs.
  - For **object members**, constructors use the **member name**. If you omit it, the member object's class default constructor will be invoked instead. The member name is followed by a parenthesis pair enclosing argument values the member object's class constructor needs.

```
class A {
public:
  A(); A(int a_in, const char *x_in);
  int a; std::string x; void show() { std::cout << a << std::endl; }
};
A::A() { a = 0; x = ""; }
A::A(int a_in, const char *x_in) { a = a_in; x = x_in; }
// B interited "A" members,
// and has two "std::string" objects as its members
class B : public A {
public:
  B(int a_in, char *x_in, int b_in, char *y_in, std::string *z_in);
  int b; std::string y; std::string *z;
};
B::B(int a_in, char *x_in, int b_in, char *y_in, std::string *z_in)
  : A(a_in, x_in), /* initialize inherited member: use class name */
  y(y_in),         /* initialize object member: use member name */
  z(z_in)          /* initialize object member: use member name */
{ b = b_in; }      /* take care of the rest */
// declare a B object: B ob = B(1,"a",2,"b",new std::string("bb"));
// remeber to free the momory allocated by "new": delete ob.z;
```

  - About the initialization order: items in the initialization list will be initialized in the order in which they are declared in the class declaration, NOT in the order in which they appear in the

initialization list. This feature could have important implications if the code uses the value of one member to initialize another.

- Using members of contained objects inside the class scope: using them the same way as using an ordinary object - access it through the member object's name: `memberObject.memberOfMember`. Of course, only `public` members are accessible.

  > Of course, if the member object is denoted by a pointer, then you should access its member in this fashion: `ptrMemberobject->membersOfMember`.
  >
  > Plus, `.`, `->` are of the same precedence and left-to-right associativity.

- Using members of contained objects from outside: using them the same way as using an ordinary object, e.g. `object.memberObject.membersOfMember`.

  ```
  1  std::cout << ob.x.size() << " " << ob.z->size() << endl;
  ```

  Note that you cannot "skip": `object.memberOfMember` is categorically wrong.

14.2 Composition (2): private/protected inheritance

- A *has-a* relationship does not have to be implemented by composition. Private and protected inheritance can be used to realize it too.
- With private inheritance, `public` and `protected` members of the base class become `private` members of the derived class. With protected inheritance, these two kinds of members become `protected` members of the derived class.
- A useful summary: composition adds an object of a class as a **named member object**, whereas private/protected inheritance adds an object of a class as an **unnamed inherited object**. These two is collectively termed *subobject* in some treatments.
- Initializing private-ly/protected-ly inherited class members: do it the same way as with other kinds of inheritance - call the class name, a.k.a. the constructor name, in the initialization list.
- Using members of private-ly/protected-ly inherited class inside the class scope: access them the same way as with ordinary members - directly call their member name. If, however, the inherited member has the same name as another member (this is allowed), you should access the inherited one with class resolution operator: `BaseClassName::member`.

  > This is no different from public inheritance, as discussed in Chapter 13.

- Implementing *has-a* relationship: composition or private/protected inheritance?
  - Most programmers prefer composition, in that
    - It is easy to follow since explicitly named objects are declared to represent the contained classes.
    - It allows you to contain multiple objects of the same class.
    - Inheritance can raise problems since a class may inherit more than one class, in which case, you may have to deal with issues as separate base classes having cognominal members or base classes sharing a common ancestor.
  - However, private/protected inheritance does offer some new features beyond composition.
    - Inheritance allows you to access the inherited class's `protected` members, be they data members or methods.

    - Inheritance allows you to overload the inherited class's `virtual` functions. However, private inheritance makes the function, among all other inherited members, `private`, and protected inheritance `protected`.

14.3 Multiple inheritance (MI)

- Syntax: listing all the base-classes with their respective inheritance access identifier, in the inheritance list, e.g. `class Derived : pubilc Base1, protected Base 2 {};`

  > If you omit the inheritance access identifier for a base class, the compiler sets adopts private inheritance for this base class.

- MI is problematic as it is effective. Two chief problems are:
  - Inheriting different methods with the same name from different base classes. For example,

    ```
    class Singer { public: void show() {...} ...};
    class Waiter { public: void show() {...} ...};
    class SingerWaiter : public Singer, public Worker
    {...}; // And "SingerWaiter" does NOT have its own version of
           // void show().
    ```

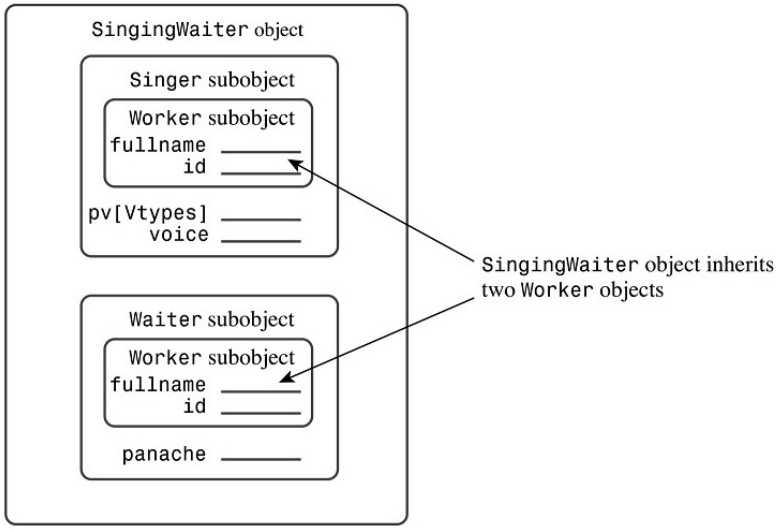
    - In this case, if there is a `SingerWaiter` object `osw` and a call `osw.show();` is made, which `show()` should the program execute - the `Singer` version or the `Waiter` version?
    - A workaround is using a scope-resolution operator: `osw.Singer::show();`.
    - If you want the program to execute the both, you can use `::` to call the both (probably inside `SingerWriter`'s own version of `show()`). However, it rises a problem if `Singer` and `Waiter` have a common ancestor `Worker`, because it may, say, displays `Worker` portion's data twice. To remedy this, you have to recode `Singer` and `Waiter`'s `show()`: strip the `Worker` portion from these two methods and capsulate it in another method.

- Inheriting multiple instances of a class via different immediate base classes that share one base class. For example,

  ```
  class Worker {...};
  class Singer : public Worker {..}; class Waiter : public Worker {..};
  class SingerWaiter : public Singer, public Worker {...};
  ```

  In this case, one `SingerWaiter` object inherits two `Woker` objects, through `Singer` and `Waiter` respectively.

  - One of the problems caused by this situation is that, suppose there is an object `osw` of class `SingerWaiter`. This statement becomes ambiguous: `Worker *pw = &ed;` because normally such an assignment sets a base-class pointer to the address of the base-class portion within the derived object, but there are two portions bearing the identity of `Woker` instead of one.
  - You could work around the problem by using explicit type cast, e.g. `Worker *pw1 = (Singer *)&osw;` and `Woker *pw2 = (Waiter *)&osw;`. But this trick certainly complicates the technique of using an array of base-class pointers to refer to a variety of objects - an application of polymorphism.
  - The technique of virtual base classes provides a solution. However, in some circumstances, programmers need multiple copies of a class.

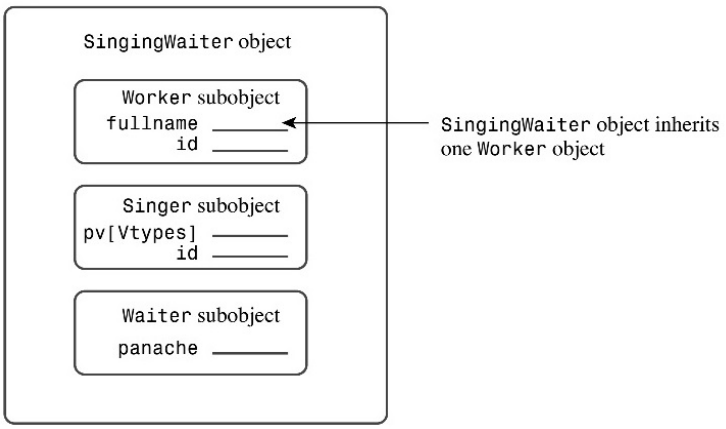


- Virtual base classes
  - The technique of virtual base classes allow an object which is derived from multiple base classes that themselves share a common base class to inherit just one copy of that shared base class.
  - Syntax: add keyword `virtual` in the inheritance list - either before or after the inheritance access identifier, in ALL the immediate base classes' declarations.

```
1 class Singer : virtual public Worker {...};
2 class Waiter : public virtual Worker {...};
```

    Then you can define `SingerWaiter` class as before:

```
1 class SingerWaiter : public Singer, public Worker {...};
```



  - Why the term "virtual"? Actually it is different form virtual functions (see Chapter 13). Reusing this keyword is just the result of resisting adding new keywords, in fear of invalidating some existing programs.
  - The concept of virtual base class should NOT be confused with abstract base class. Any class (that is not `final`) can be inherited as a virtual base class, whereas the latter refers to a special kind of class that cannot be instantiated as they have pure virtual functions.
  - Note that `Singer` and `Waiter` can still function as they do before; their usages are NOT affected by inserting `virtual` in their respective inheritance list.

- This technique is unfriendly to expandability, because when you decide to declare a new class `SingerWaiter` later on, you have to go back to the declaration of `Singer` and `Waiter` to tamper with their initialization list. And sometimes it is a bad idea to do so because these two existing classes might be in a library.
- Having virtual base classes requires a new approach to class constructors. With non-virtual base classes, you just list the immediate base classes' constructors in the initialization list.

```
// WITHOUT using virtual base classes
SingerWaiter::SingerWaiter(...) : Singer(...), Waiter(...) {...}
```

But, because now `Worker` is inherited by `Singer` and `Waiter` as a `virtual` base class, the information provided by `Singer`'s and `Waiter`'s constructors will NOT be passed to invoke a `Worker` constructor. Thus, you have to explicitly list the constructor for `Worker` as well:

```
// using virtual base classes
SingerWaiter::SingerWaiter(...)
  : Worker(...), Singer(...), Waiter(...) {...}
```

> NOTE: though you have explicitly invoked a `Worker` constructor, you still CANNOT omit the information about the `Worker` portion when invoking `Singer`'s and `Waiter`'s constructors, since you have to comply with these two constructors' prototypes.

- Mixed virtual and non-virtual base classes: when a class inherits a particular base class through several virtual paths and several non-virtual paths, that class has one base-class subobject to represent all the virtual paths, and a base-class subobject to represent each non-virtual path.
  - For example, if class `B` is a virtual base class to `X` and `Y`, and is a non-virtual base class to `Z` and `W`, and class `M` inherits `X`, `Y`, `Z` and `W`, then class `M` would contain three subobject of `B`.

14.4 Class Templates

- The solution to code reusing is not just confined to inheritance and composition. *Class template*s provide another. A class template has two points:
  - Replace the class declaration with a template class declaration, and
  - Replace method definitions with template method definitions.

> The template feature for classes and functions, is the implementation of one of the corner stone of C++ - generic programming (the other is OOP).

- It is important to realize that templates are NOT real class declarations and method definitions; they are just instructions to the compiler about how to generate them. As a result, templates are not compiled until used, thus you cannot place templates in a separate source file because that source file has nothing to compile.
- Declaring and implementing a template class:
  - **Step 1:** prefix the template class declaration with `template <typename Type>`.
  - **Step 2:** prefix the template method name in its definition header with `template <typename Type>`, AND qualify the template method name with `ClassTemplateName<typename Type>::` (instead of `ClassName::` as before).

- `Type` is your choice of symbol of the parameterized type to denote a real type, be it a built-in type, a compound type, or a class type, and the keyword `typename` can be replaced by the keyword `class`.

```
// in a header file
template <typename Type> class Stack {
private: int top; enum {MAX = 10}; Type items[MAX];
public: Stack(); bool isempty(); bool push(const Type &item); ...
        void show(Stack<Type> &) const; // NOT void show(Stack &) const;
};
template <typename Type> Stack<Type>::Stack() {top = 0;}
template <typename Type> bool Stack<Type>::isempty() {return top == 0;}
template <typename Type> bool Stack<Type>::push(const Type &item) {...}
```

Note that in the example above, `Stack<Type>` and expressions like `Stack<int>` and `Stack<string *>` are class names, NOT `stack` itself, which is a template name.

- Using a template class: instead of using `Stack myStack;` to declare an object, you should specify the real type: `Stack<int> myStack;`.
  - This is different from using a function template, for which the compiler uses the argument types to a template function to figure out what kind of type to use.

```
template <typename T> void swap(T &a, T&b) // template function
{ T temp = a; a = b; b = temp;}
// use:
int i = 1, j = 2; char x = 'g', y = 'h'; swap(i,j); swap(x,y);
```

- Non-type parameters: as with function templates, you can use a non-type parameter in the angular bracket of a class template. This feature is useful to generate a class template that has variable-size (instead of fixed-size) arrays, as with the usage if non-type parameters in function templates.

```
// with function templates
template <unsigned N, unsigned M>
int compare(const char (&a)[N], const char (&b)[M]); // varaible-size
// use: compare("hi","mom");
```

```
// with class templates
template <class T, int N> class Heap {
private: T arr[N]; // varaible-size
public: Array() {}; explicit Array(const T &); ...
};
template <class T, int N> Heap<T,N>::Heap(const T &v) {...}
// use: Heap<double, 12> myHeap; Heap<Singer,20> myChoir;
```

- Template versatility
  - As with regular classes, template classes (though they are not classes by definition) can serve as base classes and contained classes, and they can be specific types to other templates. For example,

```
template <typename T> class Array {...};
// inheritance:
template <typename U> class ListArray : public Array<U> {...};
// composition:
template <typename V> class Stack {private: Array<V> arr; ...};
// as specific type:
Array <Stack<int>> myArray; // an array-of-stacks-of-'int'
```

- You can use class templates recursively, for example,

```
Heap<Heap<int,10>, 50> myHeap;
```

This makes `myHeap` an object which contains 50 elements, and each element are objects with 10 `int` elements. The equivalent ordinary way would have this declaration: `int myHeap[50][10];`, but this ordinary declaration does not comply with OOP's doctrine.

- As with function templates, you can use more than one parameterized types:

```
template <class T, class U> class Pair { private: T a; U b; ...};
template <class T, class U> Pair<T,U>::some_method(...) {...}
```

- Default parameterized type - a feature often utilized by STL:

```
template <class U, class V = int> class Slot {...}; ...
Slot<double, double> s1; // U is 'double', V is 'double'
Slot<double> s2;         // U is 'double', V is 'int' by default
```

  - This feature is NOT available for function templates.

- Member templates: a template can be a member of a structure, class, or template class. The STL requires this feature to fully implement its design.

```
template <typename T> class beta {
private:
  template <typename V> class hold { //-----//
  private: V value;                          //
  public: hold(V v = 0): value(v) {}         // template data member
          void show() const {...}            //
  }; //-----------------------------------//
  hold<T> q; hold<int> n;
public:
  beta(T t, int i): q(t), n(i) {}
  template<typename U> U blab(U u, T t) {..};// template method
  void show() const {q.show(); n.show();}
};
```

Or, the detailed definition of the template class and template method can be moved to the outside, like this

```
template <typename T> class beta {
private:
  template <typename V> class hold; // forward declaration
  hold<T> q; hold<int> n;
public:
  beta(T t, int i): q(t), n(i) {}
  template<typename U> U blab(U u, T t); // template method
  void show() const {q.show(); n.show();}
};
template <typename T> template <typename V> class beta<T>::hold {
private: V value;
public: hold(V v = 0): value(v) {}
        void show() const {...}
};
template <typename T> template <typename U>
  U beta<T>::blab(U u, T t) {...}
```

When using, `beta<double> guy(5.0,3);` makes `T` represent `double`; and `std::cout << guy.blab(10,2.5) << std::endl;` makes `U` represent `int` (template functions can deduce the argument types, see 8.5 and 8.6).

- Templates can be used as parameterized types, just like a regular parameterized type.

```
template <template <typename T> class Thing, typename U> class Item {
private: Thing<int> s1; Thing<U> s2; U t;
public: // assumes the "Thing" class has push() and pop() methods
  bool push(int a, double *x) {return s1.push(a) && s2.push(x);}
  bool pop(int &a, double *x) {return s1.pop(a) && s3.pop(x);}
};
```

Suppose you have this declaration: `Item<Array,char *> pet;`. Then the `Thing<int>` and `Thing<double *>` in the declaration above will be instantiated as `Array<int>` and `Array<char *>`.

- Template specializations
  - Class templates are like function templates in that you can have implicit instantiations, explicit instantiations, and explicit specializations, collectively known as *specialization*s.
  - A template describes a class or a function in terms of a general type, whereas a specialization is a class or function declaration generated by a specific type.
  - Implicit instantiation: using the template, not generate until an object is needed
    - The class templates examples above all use *implicit instantiation*. That is, they are used to declare an object indicating the desired type, and the compiler generates a specialized class declaration using the recipe provided by the template.
    - The compiler does NOT generate an implicit instantiation of a template class until it needs an object.

```
Heap<double, 30> *ptr;    // a pointer, no object is needed yet
pt = new Heap<double, 30> // now an object is needed here
```

  - Explicit instantiation: using the template, generate right right away

- If you explicit declare an class (not an object) using specific type(s), the compiler generates a specialized class declaration using the recipe provided by the template immediately, NOT waiting for any object declaration.

```
template class Heap<string, 100>; // explicit instantiation
```

- Explicit specialization: not using the template
  - An explicit specialization circumvents the class template's declaration, and directs the compiler to generate an class according to the separately-provided class declaration recipe.
  - A specialized class template declaration has the following form:

```
template <> class Heap<string, 100> {...};
class Heap<string, 100> {...};  // alternative form
```

Therefore,

```
Heap<int,100> scoresA;     // use the general template recipe
Heap<string, 50> scoresB;  // use the general template recipe
Heap<string, 100> scoresC; // use the explicit specialization
```

- Partial specialization is also allowed, so you can provide modified code for these specialization. This is a kind of "incomplete" explicit specialization.

```
// general template
template <class T, class U> class Pair {...};
// partial specialization
template <class T> class Pair<T, int> {...};
// ... and this is an complete explicit specialization
template <> class Pair<char, int> {...};
```

So the pattern here is that the first `<>` indicates parameterized types that are not yet specified, and the second `<>` indicates specific types and leaves a place for the not-yet-specified type.

- You can even partially specialized a kind of pointers or references, like this

```
// general template
template <typename T> Drawer {...};
// pointer partial specialization
template <typename T*> Drawer {...};
// use:
Drawer<char> dr1; // use the general template
Drawer<char *> dr2; // use the partial specialization
```

14.5 More on class templates

- Template aliases (C++11): a convenient way.

```
typedef std::array<double,12> arrd12;
arrd12 myArray;
// old way: std::array<double,12> myArray;
```

Since C++11, another approach is introduced, so that a template alias can be used as a template:

```
template<typename T> using arrtype = std::array<T,12>;
arrtype<int> myArray1;    // array of 12 'int's
arrtype<char *> myArray2; // array of 12 'char *'s
```

C++11 extends the `using =` syntax to non-templates too.

```
typedef const char *pc;        // typedef
using pc = const char *;       // using =
typedef int (*pfarr[3])(int);  // typedef (array-of-pointer-to-function)
using pfarr = int (*[3])(int); // using =
```

The new form is more readable because it separates the type name from type information more clearly.

- Friend functions and class templates.
  - There are three kinds of friend functions of template classes:
    - Non-template friend functions.
    - Bound template friend functions, meaning the type of the friend is determined by the type of the class when a class in instantiated.
    - Unbound template friend functions, meaning that all specializations of the friend functions are friends to each specialization of the class.
  - Non-template friend functions to template classes: similar to friend functions to classes. Nothing strange.

    ```
    template <class T> class HasFriend1 { ...
    friend void counts(); // friend to all HasFriend instantiations
    };
    ```

  - Bound template friend functions to template class: declare friends template outside the class template, and build friendship inside. The friend template type parameters are the template class type parameters.

    ```
    template <class T> void counts(); // declare friend's template before
    template <class T> void report(T &);
    template <class TT> class HasFriend2 { ...
    friend void counts<TT>();         // build friendship
    friend void report<>(HasFriendT<TT> &);
    };
    ```

As with regular function templates, the `<>` in the declarations inside the class identifies these as function template's specializations, to be more specific, explicit instantiation. In the case of `report()`, the `<>` can be left empty because the type argument can be deduced from the function's argument: `HasFriendT<TT>`. Of course, you can choose to fill the angular bracket: `friend void report<HasFriendT<TT>>(HasFriendT<TT> &);`.

- Unbound template friend functions to template classes: declare friends inside the class template. The friend template type parameters are NOT the template class type parameters.

```
template <typename T> class HasFriend3 { ...
template <class C, class D, class E> friend void show(C &, D &, E);
};
```

```
//example of using:
HasFriend3<double> hfd; HasFriend<int> hfi;
show(hfd, hfi, "a");
// the compiler generates an implicit instantiation as a friend:
// void show(HasFriend<double> &, HasFriend<int> &, const char *);
```

□