Chapter 12 More about Classes

12.1 Static members

- Classes sometimes need members, be they data or methods, that are associated with the class, rather than with individual objects.
  - For instance, a particular model of device has some characteristic parameters, so it is more natural to associate these parameters with the class that represents the model of device than with individual objects. And, it wastes memory to associate the same values with each object.
  - `static` data members are also useful to track the number of objects of the class type, by changing a `static` counter variable's value in constructors.
  - `static` methods are useful to represent methods that are aimed at the class, not at the individual objects, such as altering some class-wide flags that controls some aspects of the class interfaces' behavior (e.g. formatting, unit conversion).
- Declaring `static` members: prefixing member declarations (NOT definitions) with `static`.

```
class Account{
public:
  void calculate() {amount += amount * interestRate;}
  static double rate() {return interestRate;}
  static void rate(double);
private: std::string owner;  double amount;
  static double interestRate; static double initRate();
};
void Account::rate(double newRate) {...} // NO "static" prefixed
```

- Static members, be they data or methods, exist outside any object. They reside in the static memory, just like other static storage variables, and only have one copy.
- Static methods do not have `this` pointer (because they are not bound to objects, there is no object for `this` to point to), and they cannot be declared as `const` method.
- Using `static` members: we can use an object, reference, pointer, and the class name, to access them outside the class scope, in accordance with the access control law (`private`, `protected`, `public`), no matter they are data members or methods.

```
double r = Account::rate();  // access directly through the class name
Account ac; Account *pac = &ac; Account &rac = ac;
r = ac.rate();  r = rac.rate();  // access through objects/references
r = pac->rate();                 // access through pointers
```

Of course, `static` members can be accessed the same way as we do with non-`static` members, inside the class scope.

> From the outside, non-`static` members, data or methods, can only be accessed through an object, or a reference or pointer to an object. Inside derived class scope, however, you can access base-class non-`private` members, `static` or not, directly through the base-class name.

- Because `static` data members are NOT part of individual objects of the class type, they are not defined when we create objects of the class.
    - As a result, they are not initialized by class's constructors. Moreover, in general we may not initialize `static` data members inside the class definitions, neither.
    - `static` data members have to be initialized outside any function, to ensure that they continue to exist until the whole program completes.
- Initializing `static` data member:
    - You may initialize a `static` data member outside the class declarations, preferably in the source file containing methods definitions. You must declare its type at the front, to distinguish initialization and assignment.

```
double Account::interestRate = 0.018;  // initialization
// you can access this private member because "Account::" makes the
// remainder of the definition in the scope of the class.
```

> The initialization should NOT be in the header that contains the class declaration, because it may violates the one-definition-rule (see Chapter 9) as the header may be included more than once.

    - **Exception:** for a `static` data member that is a `const` of integral or enumeration type, the initialization can (NOT "must") be placed inside the class.

```
static const int nMonth = 12; // in the class declaration
```

      But this syntax is not recommended to use. It is much better to manage all the static data members (integral or not) and methods definitions in one file:

```
const int Account::nMonth = 12; // outside the class declaration.
```

12.2 If you use `new` / `new[]` in constructors ...

- Whenever you use `new` / `new[]` in a constructor to allocate memory, you must use `delete` / `delete[]` in the destructor to free that memory.

> a. It is ok to use either `delete` or `delete[]` with a null pointer.
>
> b. `new type[1]` allocates the same amount of memory as `new type` does. Their difference is that the pointer returned by the former call can only be handled by `delete[]` and the latter one only by `delete`.

- The uses of `new` / `new[]` and `delete` / `delete[]` should be compatible.

> If a destructor frees memory by applying `delete` / `delete[]` to a pointer that is a class member, EVERY constructor for that class should initialize that pointer, either by using `new` / `new[]` or by setting the pointer to the null pointer.

- If there are multiple constructors, all should use `new` / `new[]` the same way - either with or without brackets. There is only one destructor, so all constructor have to be compatible with that destructor.
- C++11 null pointer: `nullptr`

- In C++98, the literal `0` has two meanings: (1) numeric value 0, and (2) null pointer. This polysemy makes it difficult for the reader and compiler to distinguish between the two. In short, it lacks clarity.
- Sometimes, programmers use `(void *)0` to identify a null pointer. However, the null pointer itself may have a nonzero internal representation. Other programmers use `NULL`, a C macro defined to represent the null pointer.
- C++11 provides a better solution by introducing a new keyword `nullptr` to denote the null pointer. You can still use the old styles as before, of course.

12.3 The copy constructor

- A *copy constructor* is a constructor that is used to copy an object to a newly created object. It is used during initialization, including passing function arguments by value and not during ordinary assignment. Like other constructors, it should be `public`.
- Prototype: `ClassName(const ClassName &);`. There are some varieties:
  - Dropping the `const` qualifier or replacing it with `volatile` or `const volatile` is permitted, though seldom used. Plus, the argument MUST be a reference;
  - It is permitted to provide more arguments, provided that (1) they are not the first argument, AND (2) they have default values. For instance, `Salad(const Salad &, char = 'L');` is a copy constructor, but `Salad(const Salad &, char);` is NOT a copy constructor (it is a regular constructor).
- If there is no copy constructor defined for a class, the compiler synthesizes one.

  > See, these methods can be automatically provided by the compiler:
  >
  > | **methods** | **provided if (and only if)** |
  > | --- | --- |
  > | default constructor | there is no any constructor |
  > | copy constructor | there is no any copy constructor |
  > | copy-assignment operator | there is no assignment operator |
  > | move constructor | some special conditions are met (Ch. 18) |
  > | move-assignment operator | some special conditions are met (Ch. 18) |
  > | destructor | there is no destructor |
  > | address operator `&` | there is no address operator |

- A copy constructor is invoked whenever a new object is created and initialized to an existing object of the same class. This happens in several situations.
  - (Obvious) When you explicitly initialize a new object to an existing one. i.e.

```
class Account {...}; Account ac, ac_; ...
Account ac1 = ac;                 // #1
Account ac2(ac);                  // #2
Account ac3 = Account(ac);        // #3
Account *pac4 = new Account(ac);  // #4
ac_ = ac;   // invokes the overloaded assignment operator, NOT the
            // copy constructor, because it is NOT an initialization.
```

Depending on the implementation of the compiler, the middle two may use a copy constructor directly to create `ac2` and `ac3`, or they may use a copy constructor to generate temporary objects whose contents are `ac`, and then assign them to newly created objects `ac2` and `ac3`.

- (Less obvious) When a program generates copies of an object - viz. (1) when passing an object to a function by value, or (2) when returning an object by value, or (3) (depending on the compiler) when generating temporary objects to hold intermediate results, as in `Vector v4 = v1 + v2 + v3;`

  > Passing or returning an object by value is a bad idea unless necessary. It takes time to execute the copy constructor and space to store the new.

- The default copy constructor, namely the copy constructor provided by the compiler when there is no copy constructor, performs a member-by-member copying, a.k.a. *memberwise copying*, of all non-`static` data members.
  - If a data member is itself a class object, the copy constructor for that class is invoked.
  - `static` members are never copied. They reside in the static memory of the program (pre-allocated, fixed-size memory block).
  - Note (1): if a non-`static` data member is an array, memberwise copy may be troublesome.
    - For instance, in the class `StringBad` there is a data member `char s[5];`. When copying object `str1` to `str2`, the program executes this statement in effect: `str2.s = str1.s;`.
    - What it actually does is copying an address value (a array's name represents the address of its first element). So in fact there is still one, instead of two, copy of the array contents. `str1.s` and `str2.s` has the same value - the address of the sole copy's first element.
    - An even worse problem is that if you use `delete[]` to free the memory in the destructor. Since `str1.s` and `str2.s` has the same address, the destructor's statement `delete[]s;` will delete the same block of memory twice, causing a runtime error.
  - Note (2): if you declared a `static` data member in the class definition to keep track of the number of existing objects, the count may be wrong because a default copy constructor, provided by the compiler, does not have the task of counting.
    - This issue becomes apparent when you have your destructor counts objects it destroys. You will notice in some occasions the count of creation is smaller than the count of destruction.

  > More about copying: unnecessary copying wastes time and space, so there is several copying mechanisms: *Shallow copying* is copying the address, instead of the contents, of the to-be-copied object. *Deep copying* is copying the contents. C/C++ uses shallow copying for arrays (by associating the array's name to its first element's address), and deep copying for others.
  >
  > MATLAB uses a *lazy copying* - copy the address by default. It postpones deep copying until a modification instruction is made to the new copy.
  >
  > Python and Java uses shallow copying unless instructed otherwise.

- Explicit copy constructor
  - You can provide your own copy constructor so that the compiler will not generate its own. It is your responsibility to ensure each non-`static` data member is copied.
  - If a class contains a non-`static` data member that are pointers, it is in your discretion to choose from shallow copying and deep copying. Particularly for arrays and pointers initialized by `new` / `new[]`, it is not unusual to mistakenly perform shallow copying while the intention is deep copying - copying the pointed-to data instead of copying the pointer themselves.
- Some compilers, in some circumstances, can bypass the copy constructor by replacing the operation of copying with assignment.

12.4 Assignment operators

- Not all the problems listed above (shallow copying and wrong creation count) can be blamed on the default copy constructor. You have to look at the default *assignment operator*, too.
- Just as C allows structure assignment, C++ allows class object assignment, by overloading assignment operator `=`. If you do not define your own version, the compiler automatically generates a default one. *Assignment operator* in some literature is referred to as *copy-assignment operator*.
- Prototype: `ClassName & operator=(const ClassName &);` Yes you read it right - the return type is a reference, consistent with built-in type assignment.
- An overloaded assignment operator is invoked when you assign an object to another.

```
1  Account ac, ac1; ...
2  ac1 = ac;          // invoke the overloaded assignment
3  Account ac2 = ac; // invoke the copy constructor, possibly the
4                     // assignment operator, too
```

- Like a copy constructor, a default assignment operator of a class performs memberwise copying. If a data member itself is a class object, the program invokes that class's assignment operator. `static` data members are unaffected.
- Explicit assignment operator. Its procedure is similar to explicit copy constructor, but there are some differences;
  - The function returns a non-`const` reference to the invoking object;

    > By returning an object, chained assignment is supported, as with a built-in type: `a = b = c;`. In function notation, it is `a.operator=(b.operator=(c));`.

  - Because the target object (the left object, i.e. the assignee) might refer to previously allocated data (allocated when the object is declared), the function should use `delete` / `delete[]` to free former obligation;
  - The function should protect against assigning to itself; otherwise, the freeing of memory described previously could make the object's content unreadable before they are assigned.

12.5 An example combining `new`, copy constructor, and assignment operator

- Class declaration (preferably in a header) (use *header guard*s: `#ifndef-#define-#endif`)

```
class Person {
private: char *name; static int personCount;
public: Person(); Person(const char *s); Person(const Person &);
        Person &operator=(const Person &);
        ~Person();
        static void showPersonCount();
};
```

- Class implementation (preferably in a source file) (`#include <cstring>` , `#include <iostream>` , and the header that contains the class declaration)

```
int Person::personCount = 0;    // static data member
```

```
Person::Person()               // the default constructor
{
  name = new char[1]; name[0] = '\0';
  personCount++; std::cout << this << std::endl;
}
```

```
Person::Person(const char *s)
{
  int len = std::strlen(s);
  name = new char[len+1]; // "strlen" does not count in trailing '\0's
  std::strcpy(name, s);   // the input 's' ("...") has '\0' at its end
  personCount++; std::cout << this << std::endl;
}
```

```
Person::Person(const Person &p)  // the copy constructor
{
  int len = std::strlen(p.name);
  name = new char[len+1]; // "strlen" does not count in trailing '\0's
  //name = p.name; // WRONG. Shallow copying - copying the address
  std::strcpy(name, p.name); // correct. Deep copying
  personCount++; std::cout << this << std::endl;
}
```

```
Person & Person::operator=(const Person &p)
{
  if (this == &p) return *this; // self-assignment is singled out
  delete[]name;  // free the former obligation
  int len = std::strlen(p.name);
  name = new char[len+1]; // "strlen" does not count in trailing '\0's
  //name = p.name; // WRONG. Shallow copying
  std::strcpy(name, p.name); // correct. Deep copying
  // NO "personCount++;" because it is not creating a new object
  return *this;
}
```

```
Person::~Person()
{
  delete[]name;    // required. The compiler does NOT do it for you
  personCount--; std::cout << this << " ";
  std::cout << "deleted, " << personCount << " left." << std::endl;
}
```

```
#include <iostream>
void Person::showPersonCount()
{ std::cout << personCount << std::endl; }
```

- Use the class:

```
#include "Person.h" // the class declaration header
int main()
{
  Person p1 = "John Major", p2;
  Person::showPersonCount();
  Person p3 = Person(p1);
  Person::showPersonCount();
  p2 = p1;
  Person::showPersonCount();
}
```

- Output in the command line:

```
008FFB48
008FFB3C
2
008FFB30
3
3
008FFB30 deleted, 2 left.
008FFB3C deleted, 1 left.
008FFB48 deleted, 0 left.
```

- Caution: some IDEs (e.g. mine) may issue an error message complaining that `strcpy` is unsafe, probably recommending using another function `strcpy_s`. You can get rid of this error message by configuring the preprocessor with parameter `_CRT_SECURE_NO_WARNINGS`. You may search the detailed procedure online.
- Rules of thumb:
  - Classes that need destructors probably need copy and assignment;
  - Classes that need copy probably need assignment, and vice versa.
  - A class that acts like a variable may need:
    - A copy constructor that performs deep copying, not just the address;
    - A destructor to free any memory allocated by `new`;
    - A copy-assignment operator to free the object's any existing memory allocated by `new` and deep copy the contents from its assigner.

- A class that acts like a pointer may need:
  - A copy constructor that performs shallow copying;
  - A copy-assignment operator that performs shallow-copying;
  - A static data member used to perform reference count (to count how many pointers point to same block of memory).

12.6 More varieties about copy-control methods

- Copy constructor, assignment operator, and destructor, together with move constructor and move-assignment operator, are collectively termed *copy-control method*s. They are responsible for managing copies of a class.

  > **Seven special methods**: These 5 copy-control methods, plus the default constructor and the address operator, are methods that can be generated by the compiler if some conditions are met.

- Using `= delete`
  - We can explicitly prevent copying or assignment by postfixing `= delete` to the corresponding function's declaration (NOT the function header of the definition).

    > Omitting these functions cannot prevent copying and assignment, as the compiler will automatically generate default ones if they are absent. If they are explicitly **declared-and-deleted**, the compiler does not generate them.

  - `= delete` can be applied to other methods as well; it is NOT confined to copy-control methods, i.e. we may use it to guide the function-matching process.
  - You should NOT apply `= delete` to the destructor. Otherwise there is no any destructor for the class to use.
    - However, the compiler might not complain if you delete the destructor; it just prohibits you from declare an object of that class, and from freeing a dynamically allocated object of that class (but allocating one is allowed).
  - An example:

    ```
    class A{
    public:
      A(); ~A();
      A(const A &) = delete;             // no copy
      A &operator=(const A &) = delete;  // no assignment
    };
    ```

- Using `= default`
  - We can explicitly request the compiler to generate a default constructor OR a copy-control method on its own by postfixing `= default` to the corresponding function's declaration OR function header of the definition.
  - `= default` can only be applied to default constructor and copy-control methods.
  - Applying `= default` to the declaration makes the function `inline`; applying it to the definition makes the function non-`inline`.

    > You cannot use `= default` at both places.

- An example:

```
class B{
public:
  B() = default;                       // default constructor
  B(const B &) = default;              // default copy constructor
  B &operator=(const B &);
  ~B() = default;                      // default destructor
};
B & B::operator=(const B &) = default;// default assignment operator
```

- A copy-control method is generated by the compiler as deleted, when it is impossible to copy, assign, or destroy a member of the class, respectively.

  > For example, a generated destructor is defined as deleted by the compiler if the class has a member whose own destructor is deleted or is inaccessible (e.g. `private`).

- `private` copy control: apart from using `= deleted`, declaring copy-control methods as `private` is another technique to turn them off. But there is some differences:
  - Declaring a methods in the `private` section prevent user code from calling it, but friends and members of the class can still call it provided that the called method is defined.
  - To prevent friends and class members from calling methods, you may declare the methods but do NOT define it.

    > It is legal to declare, but not define, a method which is not virtual.

  - Therefore, to thoroughly prevent copy and assignment, you can declare the corresponding method as `private` and provide no definition for them.

```
class A{
private:
  A(const A &); A &operator=(const A &);  // no copy nor assignment
public: A(); ~A();
};
// no definitions for the copy and assignment functions.
```

  - An attempt to call a **declared-but-undefined** member made by a friend or a class member results in a link-time error: unresolved external symbol.
  - An attempt to call a inaccessible (e.g. `private`) member made by user code results in a compile-time error: cannot access private member.
  - An attempt to call a **declared-but-undefined**, accessible member made by user code results in a compile-time error: unresolved external symbol.

□