Efficient C++

目录

1 Accustoming yourself to C++

1.1 Prefer `const, enum, inline` to `#define`

- `#define` will be confusing if you get an error during complication.
- Use of `constant` yield smaller code than using a `#define`.

When replacing `#defines` with constants, there are two spectial cases:

- Defining constant pointers": `const char* const authorName = "xxx''`
- Concerning class-specific constants. To limit the scope od a constant to a class, you must make it a member, to ensure there is at most one copy of the constant, you must make it a `static` member.

```
class Game{
private:
  static const int Nums = 5;
  int scores[Nums];
  ...
};
```

If it is an integral type (integers, chars, bools), we do not need provide a definition. Otherwise we need a separate definition in **implementation file**:

```
const int Game::Nums; // definition of Nums, no initial value here.
```

There is no way to carete a class-specific constant using a `#define`.

- In another way, we can use "enum hack":

```
class Game{
private:
  enum {Nums = 5};
  int scores[Nums];
  ...
};
```

- The enum hack behaves in some ways more like a `#define`, for example, it's legal to take the address of a const but not to an enum.
- Also, enums never result in unnecessary memory allocation.
- the enum hack is a fundamental technique of template metaprogramming. (see Section **??**)

Another common misuse of the `#define` is using it to implement macros that look like functions but that do not incur the overhead of a function call. For example:

```
#define CALL_WITH_MAX(a,b) f( (a) > (b) ? (a) : (b) )
```

You have to remember to parenthesize all the arguments in the macro body. Oherwise you will run into trouble. However, even you get that right, the weird things can happen:

```
int a = 5, b = 0;
CALL_WITH_MAX(++a, b);    // a is incremented twice
CALL_WITH_MAX(++a, b+10); // a is incremented once
```

Now we can use a template for an inline function in Section 5.5

```
template <class T>
inline void callWithMax(const T& a, const T &b){
  f(a > b ? a : b);
} // because we donot know what T is, we pass by reference-to-const.
```

1.2 Use `const` whenever possible

The `const` keyword is remarkably versatile. For pointers, you can specify whethwe the pointer itself is const or the data is const:

```
char greeting[] = "Heello";
const char *p = greeting;  //non-const pointer, const data
char *const p = greeting;  //const pointer, non-const data
const std::vector<int>::iterator iter = vec.begin(); // iter acts like a T *const;
++ iter;  // error! iter is const.
std::vector<int>::const_iterator cIter = vec.begin(); // iter acts like a const T*
*cIter = 10; // error! *cIter is const.
```

Some of the most powerful uses of `const` stem from its application to function declarations:

- **Having a function return a constant value** sometimes can reduce the incidence of the clinent errors without giving up safety or efficiency. For example: In Section 4.7

  ```
  const Rational operator*(const Rational &lhs, const Rational &rhs);
  ```

  We can avoid the unintentional error like `if(a * b = c)`. Such code would be illegal if a and b were of a built-in type. **One of the hallmarks of good user-defined types is that they avoid gratuitous incompatibilities with built-in types.** (see Section 4.1)

- The purpose of **const on member functions** is to identify which member functions may be invoked on `const` objects, which are important for two reasons:

  - They make the interface of a class easier to understand. It is important to know which functions may modify an object.
  - They make it possible to work with `const` objects. There is a fundamental ways to improve performance, which is **pass objects by reference-to-const**, which will explains in Section 4.3.

- **Member functions differing only in `constness`** can be overloaded, which is an important feature of C++. For example:

  ```
  class TextBlock{
  public:
    const char &operator[](std::size_t position) const;
    char &operator[](std::size_t position);
  };
  ```

  By overloading and giving the different versions different return types, you can have `const` and non-`const` TextBlocks handled differently.

- There are two prevailing notions of `const`: **bitwise const** and **logical const**. Bitwise constness is easy to understand, for logical constness, here is an example"

```
class CTextBlock{
public:
  std::size_t length() const;
private:
  char *pText;
  std::size_t textLength;
  bool lengthIsValid;
};
std::size_t CTextBlock::length() const{
  if(!lengthIsValid){
    textLength = std::strlen(pText); // error!
    lengthIsValid = true;   // error!
  }
  return textLength;
}
```

Now the solution is simple: `mutable` frees non-static data members from the constraints of bitwise constness.

- **Avoiding Duplication in `const` and `non-const` member functions**. Sometimes `operator[]` in `TextBlock` not only returns a reference to the character, it also performed bounds checking, logged access information, etc. Putting all this in both functions yields more compilation time, maintenance and code-bloat. It is possible to move all codes into a seperate member function (private).

  There is an another way. That is, you want to have one version of `operator[]` call the other one.

```
class TextBolock{
public:
  ...
  const char &operator[](std::size_t position) const{ ... }
  char &operator[](std::size_t position){
    return const_cast<char&>(static_cast<const TextBlock&>(*this)[position]);
  }
```

The one that removes `const` can be accomplished only via `const_cast`. Though casting

is such a bad idea as a general rule (see 5.2), but code duplication is no picnic either. It is determined by you, but this technique is worth knowing.

1.3 Make sure that objects are initialized before they are used

If you are in the C part of C++ and initialization would probably incur a runtime cost, initialization is not guaranteed to take place. This explains why array (from C part of C++) isn't necessarily guaranteed to have its contents initialized but a vector is. The best way is to **always initialize objects before you use them.**

- For built-in types, you need to do this manually, for else, the responsibility for initialization falls on constructors. However, **do not confuse assignment with initialization.**

```
class ABEntry{
public:
  ABEntry(const std::string &name, const stdLLstring &address);
private:
  std::string theName;
  std::string theAddress;
  int numTimesConsulted;
};
ABEntry::ABEntry(const std::string &name, const std::string &address){
  theName = name;          \\ these are all assignments, not initializations.
  theAddress = address;
  numTimesConsulted = 0;
}
```

Their default constructors were automatically called prior to entering the ABEntry constructor. But this is not true for numTimesConsulted because it is a built-in type. For it, there is no guarantee it was initialized at all prior to its assignment.

A better way is to use the member initialization list instead of assignments:

```
ABEntry::ABEntry(const std::string &name, const std::string &address)
: theName(name), theAddress(address), numTimesConsulted(0)
{} // these are now all initializations.
```

It is more efficient because default consturctions were wasted before.

For objects of built-in type, ifor consistency, it is often best to initialize everthing via member initialization. There is a policy of **always listing every data member on the initialization list**.

```
ABEntry::ABEntry(): theName(), theAddress(), numTimesConsulted(0)
{}
```

Sometimes initialization list must be used for built-in types. For example: data members that are `const` or `references` that must be initialized. (See 2.1).

When class has multiple constructors, we can omit entries in the lists for data members where assignment works as well as true initialization, moving the assignments to a single function that all constructors call.

One aspect of C++ that is not fickle is **the order in which an object's data is initialized**. This order is always the same: base classes, derived clasees, data members in the order in which they are declared. (**When initialize an array, declare the size first.**)

- **The order of initialization of non-local static objects difined in different translation units** is important.

  A `static` **object** is one that exists from the time it's constructed until the end of the program, including global objects, objects declared `static` inside classes, functions or at file scope. `Static` objects inside functions are known as local static objects.

  If initialization of a non-local static object in one translation unit uses a non-local static object in a different translation unit, **the object it uses could be uninitialized.** For example,

```
class FileSystem{
public:
  std::size_t numDisks() const;
  ...
};
extern FileSystem tfs;
```

In another file, there is

```
class Directory{
public:
  Directory(params);
  ...
};
Directory::Directory(params){
  std::size_t disks = tfs.numDisks();
  ...
}
```

Further suppose the client decides to create a single Diretory object `tempDir`, `tfs` and `tempDir` are created by different people at different times in different source files. How can you be sure that `tfs` will be initialized before `tempDir`? You can't because **the relatice order of initilization of non-local static objects defined in different translation units is undefined.**

Fortunately, a small design change eliminates the problem entirely. All that has to be done is to **move each non-local static object into its own function, where it's declared `static`**. This approach is founded on C++'s guarantee that local static objects are initilized when the object's definition is first encountered during a call to that function.

```
FileSystem &tfs(){
  static FileSystem fs;
  return fs;
}
Directory &tempDir(){
  static Directory td(params);
  return td;
}
```

The reference-returning functions dictated by this scheme are always simple, which makes them excellent candidates for inlining. On the other hand, it make them problematic in multithreaded systems. One way to deal with it is to invoke all the reference-returning functions during the single-threaded startup portion of the program.

To avoid using objects before they are initialized, you need to do three things:

- manually initialize non-member objects of built-in types
- use member initialization lists to initialize all parts of an object
- design around the initialization order uncertainty that afflicts non-local static objects defined in separate translation units.

2 Constructors, destructors, assignment operators

2.1 Know what functions C++ silently writes and calls

Compilers will declare their own versions of a copy constructor, copy assignment, destructor if you don't declare them. Further more, if you declare no constructors, compliers will also declare a default constructor. All these functions are `public and inline` (See 5.5).

```
class Empty
public:
  Empty() { ... } // default constructor
  Empty(const Empty& rhs) { ... } // copy constructor
  ~Empty() { ... } // destructor — see below for whether it' s virtual
  Empty& operator=(const Empty& rhs) { ... } // copy assignment operator
};
```

- The generated destructor is **non-virtual** (See 2.3) unless it's for a class inheriting from a base class that declares a virtual destructor,
- The copy constructor and assignment simply copy each non-static data member.
- If you want to support copy assignment in a class **containing a reference or const member**, you must define the copy assignment operator yourself.
- Compilers reject implicit copy assignment operators in derived classes that inherit from **base classes declaring the copy assignment operator** `private`.

2.2 Explicitly disallow the use of compilergenerated functions you do not want.

Use `=delete` or declare `private` and give no implementations if you don't want a class to support a particular kind of copy functionality.

2.3 Declare destructors virtual in polymorphic base classes.

A particluar example is get function of a factory, we will get a pointer to a base class. If we declare destructors non-virtual, **the object will be partially destroyed**.

- If a class does not contain virtual functions, that often indicates it is not meant to be used as a base class, so making the destructor virtual is usually a bad idea because the size of the class will increase. **Declare a virtual destructor in a class if and only if that class contains at least one virtual function.**
- The implementation of virtual functions requires that objects carry information that can be used at runtime to determine which virtual functions should be invoked on the object. This information typically takes the form of a pointer called a `vptr`. The `vptr` points to an array of function pointers called a `vtbl`; each class with virtual functions has an associated `vtbl`. When a virtual function is invoked on an object, the actual function called is determined by following the object's `vptr` to a `vtbl` and then looking up the appropriate function pointer in the `vtbl`.
- Not all base classes are designed to be used polymorphically. **Neither the standard string type, for example, nor the STL container types are designed to be base classes at all,** much less polymorphic ones.
- If we want an abstract base class, we can declare a pure virtual destructor, but **we must provide a definition for the pure virtual destructor.**

2.4 Prevent exceptions from leaving destructors

- Destructors should never emit exceptions. If functions called in a destructor may throw, the destructor should catch any exceptions, then swallow them or terminate the program.
- If class clients need to be able to react to exceptions thrown during an operation, the class should provide a regular (i.e., non-destructor) function that performs the operation.

2.5 Never call virtual functions during construction or destruction

- **During base class construction of a derived class object, the type of the object is that of the base class.** Not only do virtual functions resolve to the base class, but the parts of the language using runtime type information (e.g., `dynamic_cast` (see 5.2) and `typeid`) treat the object as a base class type.
- The same reasoning applies during destruction.
- It's not always so easy to detect calls to virtual functions during construction or destruction.

```
class Transaction {
public:
  Transaction() { init(); } // call to non-virtual...
  virtual void logTransaction() const = 0;
  ...
private:
void init() {
  ...
  logTransaction(); // ...that calls a virtual!
}
};
```

2.6 Have assignment operators return a reference to *this

One of the interesting things about assignments is that you can chain them together:

```
int x, y, z;
x = y = z = 15; // chain of assignments.
x = (y = (z = 15)); //equivalent.
```

The way this is implemented is that **assignment returns a reference to its left-hand argument**. This convention applies to all assignment operators.

```
class Widget {
public:
  ...
  Widget &operator=(const Widget &rhs){
    ...
    return *this;
  }
  Widget& operator+=(const Widget& rhs) // the convention applies to
  {                                     // +=, -=, *=, etc.
    ...
    return *this;
  }
Widget& operator=(int rhs)              // it applies even if the
```

```
  {                                        // operator' s parameter type
  ...                                      // is unconventional
  return *this;
  }
};
```

2.7 Handle assignment to self in operator=

If you follow the advice of 3.1 and 3.2, you'll always use objects to manage resources, and you'll make sure that the resource-managing objects behave well when copied.

If you try to manage resources yourself, however (which you'd certainly have to do if you were writing a resourcemanaging class), you can fall into the trap of **accidentally releasing a resource** before you're done using it. For example,

```
Widget& Widget::operator=(const Widget& rhs) // unsafe impl. of operator=
{
  delete pb; // stop using current bitmap
  pb = new Bitmap(*rhs.pb);                  // start using a copy of rhs' s bitmap
  return *this;
}
```

Now, the self-assignment problem here is that itself holds a pointer to a deleted object! There are three ways to prevent this error.

- check for assignment to self via an identity test at the top of operator=.

  ```
  Widget& Widget::operator=(const Widget& rhs){
    if (this == &rhs) return *this; // identity test: if a self-assignment, do nothin
    delete pb;
    pb = new Bitmap(*rhs.pb);
    return *this;
  }
  ```

  This works but there is also exception-unsafe. If the "new Bitmap" expression yields an exception (either because there is insufficient memory for the allocation or because Bitmap's copy constructor throws one), the Widget will end up holding a pointer to a deleted Bitmap. You can't safely delete them. You can't even safely read them.

- 5.4 explores exception safety in depth, but in this Item, it suffices to observe that in many cases, a careful ordering of statements can yield exception-safe code.

```
Widget& Widget::operator=(const Widget& rhs)
{
  Bitmap *pOrig = pb; // remember original pb
  pb = new Bitmap(*rhs.pb); // point pb to a copy of rhs' s bitmap
  delete pOrig; // delete the original pb
  return *this;
}
```

Now, if "`new Bitmap`" throws an exception, `pb` remains unchanged.

- If you're concerned about efficiency, use the technique known as "**copy and swap**", which is described in 5.4.

```
class Widget {
  ...
  void swap(Widget& rhs); // exchange *this' s and rhs' s data, see Item 29 for det
};
Widget& Widget::operator=(const Widget& rhs){
  Widget temp(rhs); // make a copy of rhs' s data
  swap(temp); // swap *this' s data with the copy' s
  return *this;
}
```

- Make sure that any function operating on more than one object behaves correctly if two or more of the objects are the same.

2.8 Copy all parts of an object

When you're writing a copying function, be sure to

- copy **all** local data members,

```
class Customer {
public:
```

```
  ...
  Customer(const Customer& rhs);
  Customer& operator=(const Customer& rhs);
  ...
private:
  std::string name;
  Date lastTransaction;
};

Customer::Customer(const Customer& rhs)
: name(rhs.name){} // copy rhs' s data  Data has been forgotten!!!

Customer& Customer::operator=(const Customer& rhs)
{
  name = rhs.name; // copy rhs' s data
  return *this;    //  DATE!
}
```

- invoke the appropriate copying function in **all base classes**.

```
class PriorityCustomer: public Customer { // a derived class
public:
  ...
  PriorityCustomer(const PriorityCustomer& rhs);
  PriorityCustomer& operator=(const PriorityCustomer& rhs);
  ...
private:
  int priority;
};
PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
: Customer(rhs), // invoke base class copy ctor
  priority(rhs.priority){}
PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
```

```
  Customer::operator=(rhs); // assign base class parts
  priority = rhs.priority;
  return *this;
}
```

- if you find that your copy constructor and copy assignment operator have similar code bodies, **eliminate the duplication by creating a third member function that both call**.

3 Resource Management

This chapter begins with a straightforward object-based approach to resource management built on C++'s support for constructors, destructors, and copying operations. Experience has shown that disciplined adherence to this approach can all but eliminate resource management problems. The chapter then moves on to Items dedicated specifically to memory management. These latter Items complement the more general Items that come earlier, because objects that manage memory have to know how to do it properly.

3.1 Use objects to manage resources

To make sure that the resource is always released, we need to put that resource inside an object whose destructor will automatically release the resource when control leaves domain.

```
void f(){
  std::auto_ptr<Investment> pInv(createInvestment()); // call factory function
   ...
} // automatically delete pInv via auto_ptr' s dtor
```

This simple example demonstrates the two critical aspects of using objects to manage resources:

- **Resources are acquired and immediately turned over to resource-managing objects**: the resource returned by `createInvestment` is used to initialize the `auto_ptr` that will manage it. In fact, the idea of using objects to manage resources is often called **Resource Acquisition Is Initialization (RAII)**.
- **Resource-managing objects use their destructors to ensure that resources are released**.

See more details of smart pointers in Section 6.1. **Pay attention to smart pointer to aynamically allocated arrays.**

If you need to craft your own resource-managing classes, that's not terribly difficult to do, but it does need to consider of Section 3.2 and 3.3.

3.2 Think carefully about copying behavior in resource-managing classes.

What should happen when an RAII object is copied? Most of the time, you'll want to choose one of the following possibilities:

- **Prohibit copying**: declare the copying operations private.
- **Reference-count the underlying resource**: See shared_ptr.
- **Copy the underlying resource**: copying a resource-managing object performs a "deep copy".
- **Transfer ownership of the underlying resource**: See unique_ptr.

3.3 Provide access to raw resources in resource-managing classes.

In a perfect world, you'd rely on such classes for all your interactions with resources, never sullying your hands with direct access to raw resources. But the world is not perfect. There are two general ways to do it: **explicit conversion** and **implicit conversion**.

- shared_ptr and unique_ptr both offer a get member function to perform an **explicit conversion**, i.e., to return (a copy of) the raw pointer inside the smart pointer object.

```
FontHandle getFont(); // from C API — params omitted for simplicity
void releaseFont(FontHandle fh); // from the same C API

class Font { // RAII class
public:
  explicit Font(FontHandle fh) // acquire resource; use pass-by-value,because the C
  : f(fh){}
  ~Font() { releaseFont(f ); } // release resource
  FontHandle get() const { return f; } // explicit conversion function
  ... // handle copying (see Item 14)
private:
  FontHandle f; // the raw font resource
};
```

- However, sometimes we might find the need to explicitly request such conversions off-putting enough to avoid using the class . That, in turn,would increase the chances of leaking fonts.
- The alternative is to have Font offer an implicit conversion function to its FontHandle, which makes calling into the C API easy and natural:

```
class Font {
public:
  operator FontHandle() const // implicit conversion function
  { return f; }
  ...
};
```

- The downside is that implicit conversions increase the chance of errors. For example, a client might **accidently** create a `FontHandle` when a `Font` was intended:

  ```
  Font f1(getFont());
  FontHandle f2 = f1;
  ```

  When `f1` is destroyed, the font will bereleased, and `f2` will dangle.

In general, explicit conversion is safer, but implicit conversion is more convenient for clients.

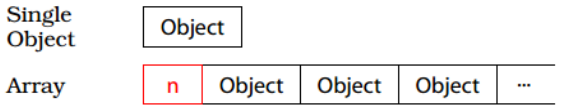
It may have occurred to you that functions returning the raw resource inside an RAII class are **contrary to encapsulation**. That's true, but it's not the design disaster it may at first appear, it hides what clients don't need to see, but it makes available those things that clients honestly need to access.

3.4 Use the same form in corresponding uses of new and delete

```
std::string *stringArray = new std::string[100];
...
delete stringArray;
```

At the very least, 99 of the 100 string objects pointed to by stringArray are unlikely to be properly destroyed, because **their destructors will probably never be called**.

The memory layout for single objects is generally different from the memory layout for arrays. In particular, the memory for an array usually includes the size of the array, thus making it easy for delete to know how many destructors to call.



The rule is simple: if you use [] in a `new` expression, you must use [] in the corresponding `delete` expression. If you don't use [] in a `new` expression, don't use [] in the matching `delete` expression.

3.5 Store newed objects in smart pointers in standalone statements.

Suppose we have a function to reveal our processing priority and a second function to do some processing on a dynamically allocated Widget in accord with a priority:

```
int priority();
void processWidget(shared_ptr<Widget> pw, int priority);
```

Consider now a call to `processWidget`:

```
processWidget(new Widget, priority());
```

It won't compile. `shared_ptr`'s constructor taking a raw pointer is `explicit`, so there's no implicit conversion. The following code, however, will compile:

```
processWidget(shared_ptr<Widget>(new Widget), priority());
```

Although we're using object-managing resources everywhere here, **this call may leak resources**.

Before `processWidget` can be called, then, compilers must generate code to do these three things:

- Call `priority`
- Execute `new Widget`
- Call the `shared_ptr` constructor

If `new Widget` expression executed before `priority()`, and the call to `priority()` yields an exception, the pointer returned from `new Widget` will be lost.

The way to avoid problems like this is simple: use a separate statement to create the `Widget` and store it in a smart pointer, then pass the smart pointer to `processWidget`:

```
shared_ptr<Widget> pw(new Widget);
processWidget(pw, priority());
```

4 设计与声明

4.1 让接口容易被正确使用，不易被误用

- 首先必须考虑客户可能做出什么错误：

```
class Date {
public:
  Date(int month, int day, int year);
  ...
};
```

这就很容易以错误的次序传递参数。

- 可以通过导入新类型来进行预防。

```
class Month {
public:
  static Month Jan() { return Month(1); } // functions returning all valid
  ... // why these are functions, not other member functions
private:
  explicit Month(int m); // prevent creation of new Month values
};
```

- 预防客户错误另一个办法是限制类型内什么事情可以做，什么事情不可以做，常见的就是加上const，可以看 Section 1.2。
- 另一个一般性准则是尽量令 types 的行为与内置 types 一致，
- 任何接口如果要求客户必须记得做某些事，就是有着不正确使用的可能，例如假如有一个factory函数，返回一个指针，那就有可能没有删除指针或者二次删除。

```
Investment* createInvestment();
```

我们可以返回智能指针：

```
shared_ptr<Investment> createInvestment();
```

假设 class 设计者要将指针传递给一个getRidOfInvestment来进行删除指针，那就要将它绑定为shared_ptr的删除器：

```
shared_ptr<Investment> createInvestment(){
  shared_ptr<Investment> retVal(static_cast<Investment*>(0),
getRidOfInvestment);
  retVal = ...; // make retVal point to the correct object
  return retVal;
}
```

同时，智能指针有一个特别好的性质就是消除所谓的"cross-DLL prolbem"，这个问题发生于对象在动态链接程序库 (DLL) 只不过被`new`创建，却在另一个 DLL 中被 delete，这会导致运行期错误。因此经常用来自动解除互斥锁 (mutex)。

4.2 设计 class 犹如设计 type

- **新 type 的对象应该如何被创建和销毁？**这会影响到构造函数、析构函数、内存分配函数和释放函数 (`operator new, operator new[], operator delete, operator delete[]`，见 Section 8。
- **初始化和赋值该有什么样的差别？**不要混淆了初始化和赋值。
- **新 type 的对象如果被按值传递，意味着什么？**拷贝构造函数用来定义一个 type 的 pass-by-value 该如何实现。
- **什么是新 type 的"合法值"？**
- **新 type 需要配合某个继承图系 (inheritance graph) 吗？**注意 virtual 和 non-virtual 的影响，见 Section??, Section??。如果你允许其他 class 继承该 class，那会影响你所声明的函数，尤其是析构函数，是否是 virtual，见 Section 2.3。
- **新 type 需要什么样的类型转换？**如果你希望允许类型 `T1`被隐式转换为`T2`类型，就必须在`T1`内部写一个类型转换函数 (`operator T2`),或者在`T2`写一个`non-explicit-one-argument`的构造函数。
- **什么样的操作符和函数对于新 type 而言是合理的？**这决定你讲为你的 class 声明哪些函数，其中某些是 member，某些不是，见 Section 4.6，4.7，??。
- **什么样的标准函数应该驳回？**必须声明为`private`，见 Section 2.2。
- **谁该取用新 type 的成员？**这帮助你决定哪些成员为 `public`、`protected`、`private`，以及哪些 class 或者 function 应该是`friend`，以及将它们嵌套于另一个之内是否合理。

- **什么是新 type 的未声明接口（undeclared interface）？**它对效率、异常安全性（Section 5.4）以及资源运用（多任务锁定和动态内存）提供何种保证？
- **新 type 有多么一般化？**如果是定义一整个 types 家族，应该考虑模板。
- **你真的需要一个新 type 吗？**如果只是定义新的 derived class 以便为既有的 class 添加机能，说不定单纯定义几个 non-member 函数更好。

4.3 以 pass-by-reference-to-const 替换 pass-by-value

- 以 by reference-to-const 方式传递参数可以减少拷贝构造和析构的开销
- 也可以避免 slicing（对象切割）的问题，即当一个 derived class 对象以 by value 方式传递并被视为一个 base class 对象。

  references 在编译器底层往往以指针形式实现，因此 pass-by-reference 通常意味着真正传递的是指针。当然，如果是内置类型、STL 的迭代器或者函数对象，pass-by-value 效率更高。

4.4 必须返回一个 object 时，别妄想返回其 reference

一个典型的例子就是重载算术运算符的时候不能返回 reference，否则就会传递一些 references 指向并不存在的对象。如果函数内部创建一个指针，就会出现内存泄漏，如果函数内部创建一个 local static 对象，那 (a*b)==(c*d) 永远都会成立。

因此，**绝对不要返回 pointer 或者 reference 指向一个 local stack 对象或 heap-allocated 对象**，或者指向一个 local static 对象而有可能同时需要多个这样的对象，Section 1.3 已经为“在单线程环境中合理返回 reference 指向一个 local static 对象”提供了一份设计示例。

4.5 将成员变量声明为 private

如果成员变量不是`public`，客户唯一能够访问对象的办法就是成员函数，这样可以保证一致性，另外，这样封装可以为“所有可能的实现”提供弹性，例如，你可以随时修改 class 的实现方式，客户只需要重新编译，遵循 Section 5.6甚至可以不用重新编译。另外，这也可以使得成员变量被读或者被写时轻松告诉其他对象，可以验证 class 的约束条件以及函数的前提和时候状态等等。

即便成员变量是protected，如果将其取消，也会有大量不可预知的代码受到破坏，需要重写、重新测试等等，因此，**从封装的角度，只有两种访问权限：private（提供封装）和其他（不提供封装）**。

4.6 偏好 non-member、non-friend 来替换 member 函数

```
class WebBrowser {
public:
  ...
  void clearCache();
  void clearHistory();
  void removeCookies();
  ...
  void clearEverything(); // calls clearCache, clearHistory,and removeCookies
};
void clearBrowser(WebBrowser& wb){
  wb.clearCache();
  wb.clearHistory();
  wb.removeCookies();
}
```

哪一个clearBrowser更好呢?

面向对象守则要求数据尽可能地被封装，如果某些东西被封装，它就不再可见，因此，越多函数可以访问数据，数据的封装性就越低。因此，当二者都可以提供相同的机能时，选择 non-member 做法可以减少编译依赖度，增加 class 的包裹弹性、技能扩充性。

另外一点，一个像WebBrowser这样的 class 会拥有大量便利函数，某些与书签有关，某些与 cookie 有关等等，通常大多数客户指对其中某些感兴趣，一个自然的做法是将它们分别声明于不同的头文件的相同命名空间中，这也是标准库的组织方式，我们只需要 include 所需要的东西即可：

```
// header “webbrowser.h” — header for class WebBrowser itself
// as well as “core” WebBrowser-related functionality
namespace WebBrowserStuff {
  class WebBrowser { ... };
  ... // “core” related functionality
}
```

```
// header “webbrowserbookmarks.h”
namespace WebBrowserStuff {
  ... // bookmark-related convenience functions
}

// header “webbrowsercookies.h”
namespace WebBrowserStuff {
... // cookie-related convenience functions
}
```

4.7 如果所有参数都需要类型转换，请采用 non-member 函数

考虑如果为Rational类重载算术运算符，如果定义为 member 函数，

```
Rational oneEight (1,8);
Rational result = oneEight * 2; // Correct!
Rational result = 2 * oneEight; // Error!
```

因为只有被列入参数列表内的参数才可以进行隐式类型转换，所以当需要支持混合式算术运算的时候，必须将其列为 non-member 函数，同时尽可能避免将其声明为 friend 函数。

如果你需要为某个函数的所有参数（包括 this 指针所指的隐藏参数）进行类型转换，这个函数就必须是 non-member。该条款并不完全成立，等牵扯到 template 的时候才完整，见 Section **??**。

4.8 考虑写一个不抛异常的 swap 函数

标准库的swap函数十分典型：

```
namespace std {
  template<typename T> // typical implementation of std::swap;
  void swap(T& a, T& b) // swaps a’ s and b’ s values
  {
    T temp(a);
    a = b;
    b = temp;
  }
```

```
}
```

有些时候这样做效率很低，最典型的例子就是一些“以指针指向一个对象，内涵真正数据”的那种类型，实际上它们的swap只需要交换指针就可以了。

```
namespace std {
  template<>
  void swap<Widget>(Widget& a, Widget& b){
    swap(a.pImpl, b.pImpl); // to swap Widgets, swap their pImpl pointers;
  } // this won' t compile
}
```

这里试图访问 Widget 类的 private 成员，因此无法编译，正确做法是：

```
class Widget { // same as above, except for the
public: // addition of the swap mem func
  ...
  void swap(Widget& other){
     using std::swap; // the need for this declaration is explained later in this Item
     swap(pImpl, other.pImpl); // to swap Widgets, swap their
  } // pImpl pointers
  ...
};
namespace std {
  template<> // revised specialization of
  void swap<Widget>(Widget& a,Widget& b){
     a.swap(b); // to swap Widgets, call their
  } // swap member function
}
```

然而当 Widget 和 WidgetImpl 都是模板类的时候还是会出现错误，因为模板函数是不支持偏特化的，即下面的写法是不合法的：

```
namespace std {
template<typename T>
  void swap<Widget<T> >(Widget<T>& a, Widget<T>& b)
  { a.swap(b); }
}
```

然而我们又不能在 std namespace 里面添加新东西，所以我们应该将特化的 swap函数和 Widget 类定义在一起：

```
namespace WidgetStuff {
  ... // templatized WidgetImpl, etc.
  template<typename T> // as before, including the swap
  class Widget { ... }; // member function
  ...
  template<typename T> // non-member swap function;
  void swap(Widget<T>& a,Widget<T>& b) // not part of the std namespace
  {
     a.swap(b);
  }
}
```

现在，每次调用swap的时候都会找到该专属版本。最后，如果你调用 swap，请确定包含一个using声明式，保证 std::swap可见：

```
template<typename T>
void doSomething(T& obj1, T& obj2)
{
  using std::swap;
  swap(obj1, obj2); // call the best swap for objects of type T
  ...
}
```

另外，成员版swap绝不可以抛出异常，因为swap的一个最好应用就是帮助 class 提供强烈的异常安全性保障，Section 5.4会提供所有细节，而它基于一个假设：成员版的swap绝不抛出异常。

5 实现

5.1 尽可能延后变量定义式的出现时间

只要定义了一个变量，就要承受构造成本和析构成本，有的时候可能会有某个变量从未被使用，但仍会耗费成本，例如：

```
std::string encryptPassword(const std::string& password){
  using namespace std;
  string encrypted;
  if (password.length() < MinimumPasswordLength) {
    throw logic_error("Password is too short");
  }
  return encrypted;
}
```

如果有个被丢出，它就真的没被使用，因此最好延后encrypted的定义式：

```
std::string encryptPassword(const std::string& password){
  using namespace std;
  if (password.length() < MinimumPasswordLength) {
    throw logic_error("Password is too short");
  }
  string encrypted;
  return encrypted;
}
```

但是这还不够紧实，因为encrypted虽然定义但没有初值，调用的是默认构造函数，Section 1.3解释了为什么通过默认构造函数构造比直接构造时指定初值的效率差。

```
std::string encryptPassword(const std::string& password){
  string encrypted(password); // define and initialize via copy
  encrypt(encrypted);
  return encrypted;
}
```

因此“尽可能延后”是指尽可能延后到这份定义能够直接给它初值实参为止。如果涉及循环的话，最好是放在内部，因为对程序的可理解性和易维护性更好，如果追求效率的话就要分析赋值成本和构造析构成本哪个更高。

5.2 少做转型动作

转型共有五种形式：

- **旧式转型 (old-style casts)**：

```
(T)expression //将expression转型为T
T(expression)
```

- `const_cast<T>(expression)`：通常被用来 cast away the constness，它也是唯一由此能力的转型操作符。
- `dynamic_cast<T>(expression)`：主要用来执行 safe downcasting，也是用来决定某对象是否归属集成体系中的某个类型，它是唯一无法用旧式语法执行的动作，也是耗费重大运行成本的动作。
- `reinterpret_cast<T>(expression)`：意图执行地执行低级转型，实际动作可能取决于编译器，例如将一个 pointer to int 转型为 int，这一类在低级代码之外很少见，本书只在讨论如何针对原始内存 (raw memory) 写出一个调试用的分配器 (debugging allocator) 使用，见 Section **??**。
- `static_cast<T>(expression)`：用来强迫隐式转换，例如将 non-const 对象转换为 const 对象，以及上述转换的反向，例如 type* 转换为 void*，pointer-to-derived 转为 pointer-to-base 等等，但是它无法将 const 转为 non-const。

注意，**任何一种类型转换往往真的令编译器编译出运行期间执行的代码**，例如：

```
class Base { ... };
class Derived: public Base { ... };
Derived d;
Base *pb = &d; // implicitly convert Derived* to Base*
```

这种情况下会有一个偏移量(offset)在运行期被施加于这`Derived*`指针身上来获得正确的`Base*`指针值，这说明，单一对象可能拥有一个以上的地址，因此你应该避免做出“object 在 C++ 中如何布局”的假设，例如，**将 object 地址转型为`char*`后进行指针算术，这几乎总会导致无定义行为**。

关于转型还有一个有趣的事情：我们很容易写出似是而非的代码，例如

```
class Window { // base class
public:
```

```
  virtual void onResize() { ... } // base onResize impl
  ...
};
class SpecialWindow: public Window { // derived class
public:
  virtual void onResize() { // derived onResize impl;
    static_cast<Window>(*this).onResize(); // this doesn' t work!
    ...
  }
  ...
};
```

此时它是在**当前对象的 base class 成分的副本**上调用 `Window::onResize`，然后再在当前对象身上执行 `SpecialWindow`专属动作，如果两个`onResize`修改了对象内容，那么 base class 成分并没有更改，但 derived class 成分却更改了。解决的办法就是：

```
class SpecialWindow: public Window {
public:
  virtual void onResize() {
    Window::onResize(); // call Window::onResize on *this
    ...
  }
  ...
};
```

因此，如果你发现自己打算类别转换，这就是一个警告信号：你可能正在把局面发展到错误的方向上，如果你用的是`dynamic_cast`更是如此。

首先，`dynamic_cast`执行速度相当慢，例如至少会进行多次的 `strcmp`调用来比较 class 名称。之所以需要`dynamic_cast`通常是因为你想执行 derived class 操作函数，但手上只有一个指向 base 的 pointer 或者 reference，有两个一般性做法可以避免这个问题：

- 使用容器并在其中存储直接指向 derived class 对象的指针（通常是智能指针），当然这种做法使你无法在同一个容器内存储指针指向所有可能的派生类，因此就需要多个容器。
- 另一个做法就是在 base class 内提供 virtual 函数做你想对各个派生类做的事情。

5.3 避免返回“handles”指向对象内部成分

```
class Rectangle {
public:
  ...
  Point& upperLeft() const { return pData->ulhc; }
  Point& lowerRight() const { return pData->lrhc; }
  ...
};
```

这两个函数被声明为 const 成员函数，是不想让客户修改的，但却又返回 reference 指向 private 内部数据。这告诉我们两点：**成员变量的封装性最多只等于“返回其 reference”的函数的访问级别**。

如果返回的是指针或者迭代器，情况是相同的，这些我们统称为 **handles**，返回一个“代表对象内部数据”的 handle 就会带来降低封装性的风险。因此需要将函数加上`const`。

```
class Rectangle {
public:
  ...
  const Point& upperLeft() const { return pData->ulhc; }
  const Point& lowerRight() const { return pData->lrhc; }
  ...
};
```

但有时候还会存在问题，更明确地说，**它可能导致 dangling handles**，比如如果返回一个指针，可能就会不小心析构了。

但这不以为绝对不可以让成员函数返回 handle，有时候必须这么做，例如，`operter[]`就是返回 references 指向容器内的数据。

5.4 为异常安全而努力是值得的

假设有个 class 用来表现夹带背景图案的 GUI 菜单，这个 calss 希望用于多线程，所以有互斥器`mutex`作为并发控制用：

```
class PrettyMenu {
public:
  void changeBackground(std::istream& imgSrc); // change background image
```

```
private:
  Mutex mutex; // mutex for this object
  Image *bgImage; // current background image
  int imageChanges; // # of times image has been changed
};
```

下面是changeBackground函数的一个可能实现:

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  lock(&mutex); // acquire mutex (as in Item 14)
  delete bgImage; // get rid of old background
  ++imageChanges; // update image change count
  bgImage = new Image(imgSrc); // install new background
  unlock(&mutex); // release mutex
}
```

从异常安全性的观点来看，这个函数很糟糕，因为它没有满足异常安全的两个条件：即当异常被抛出时，带有异常安全性的函数会:

- **不泄露任何资源**。上述代码一旦new出现异常，unlock 就不会执行，互斥器就永远被 hold 住了。
- **不允许数据败坏（corrupted）**。如果new出现异常，bgImage就会指向一个已被删除的对象，imageChanges也被累加，但新图像并没有成功安装。

解决资源泄露很容易，可以看 Section 3.1,3.2。

在专注解决数据败坏之前，先定义一些术语：异常安全函数（Exception-safe functions）提供以下三个保证之一:

- **基本承诺：**如果异常被抛出，程序内任何事物仍然保持在有效状态，所有对象都处于内部前后一致的状态。举个例子，如果有异常抛出，PrettyMenu对象可以继续拥有原背景图像或者拥有一个默认图像。
- **强烈保证：**如果异常被抛出，程序状态不改变，即如果函数失败，程序会回复到“调用函数之前”的状态。和这样的函数共事是最容易的。
- **不抛掷（nothrow）保证：**承诺不抛出异常，因为它们总是能够完成承诺的功能。作用于内置类型（int，指针等等）的所有操作都提供 nothrow 保证。

```
int doSomething() noexcept;
int doSomething() throw();
```

这并不是说doSomething绝对不会抛出异常，而是说抛出异常是严重的错误。

因此，我们目前的抉择是，我们该为我们所写的函数提供哪一种保证？从异常安全性的观点看来，nothrow函数很棒，但是大部分函数而言，抉择往往在基本保证和强烈保证之间。

对于changebackground而言，提供强烈保证不困难：不仅将 Image*改为智能指针，还重新排列了一下语句次序，在更换图像之后才累加imageChanges。

```
class PrettyMenu {
  ...
  std::tr1::shared_ptr<Image> bgImage;
  ...
};
void PrettyMenu::changeBackground(std::istream& imgSrc){
  Lock ml(&mutex);
  bgImage.reset(new Image(imgSrc));
  ++imageChanges;
}
```

另外，还有一个一般化的设计策略可以做到强烈保证：**copy and swap**：为你打算修改的 object 做出一个副本，然后在副本上修改，如果有任何动作抛出异常，原 object 则保持不变。待所有改变成功后，再将副本与原 object 进行 swap 。实现上通常是将所有“隶属对象的数据”从原 object 放进另一个对象内，然后赋予原 object 一个指针指向哪个所谓的实现对象（implementation object，即副本），这种手法通常被叫做 **pimpl idiom**，可见 Section 5.6。对PrettyMenu而言，写法如下：

```
struct PMImpl {
  std::tr1::shared_ptr<Image> bgImage;
  int imageChanges;
};
class PrettyMenu {
private:
  Mutex mutex;
  std::tr1::shared_ptr<PMImpl> pImpl;
};
```

```
void PrettyMenu::changeBackground(std::istream& imgSrc){
  using std::swap; // see Item 25
  Lock ml(&mutex); // acquire the mutex
  std::tr1::shared_ptr<PMImpl> pNew(new PMImpl(*pImpl));
  pNew->bgImage.reset(new Image(imgSrc)); // modify the copy
  ++pNew->imageChanges;
  swap(pImpl, pNew); // swap the new data into place
} // release the mutex
```

但是并非所有函数都能够这样来实现“强烈保证”。另外，它并不能保证整个函数是有强烈的异常安全性的：

```
void someFunc()
{
  f1();
  f2();
}
```

只要其中一个无法提供异常安全保证，整体就无法提供异常安全保证。**函数提供的“异常安全保证”通常最高值等于各个函数的“异常安全保证”中的最弱者**。

5.5 透彻理解 inline 的里里外外

inline的好处有：

- 减少了函数调用所导致的额外开销。
- 编译器最优化机制被设计用来浓缩那些“不含函数调用”的代码，因此编译器用能力对 inline 函数本体执行语境相关最优化。
- 如果对各种函数都用inline的话，目标代码（object code）会增加，可能会导致额外的换页行为（paging），降低指令高速缓存装置的击中率（instruction cache hit rate）以及伴随而来的效率。反过来讲，如果 inline 函数本体很小，那么编译器产出的 object code 可能会更小，就会更好的效率。

inline只是对编译器的申请，也可以隐式提出，隐式方式就是将函数定义在 class 定义中，friend 函数同样适用。

```
class Person {
public:
  ...
  int age() const { return theAge; }
private:
  int theAge;
};

template<typename T> // an explicit inline
inline const T& std::max(const T& a, const T& b)
{ return a < b ? b : a; }
```

template 和 inline 函数都要放在头文件中，因为编译器要将其具现化或者替换函数调用的时候必须知道函数长什么样子。同时，**编译器拒绝将太复杂的函数 inlining（例如带有循环和递归），对于所有 virtual 函数的调用也都会拒绝 inlining**。幸运的是：如果编译器无法 inline，会给你一个警告信息（见 Section **??**）。

还有些时候编译器虽有意愿 inline 某个函数，但也还可能会为该函数生成一个函数本体：

- 如果程序要取某个 inline 函数的地址，编译器就必须为此函数生成一个函数本体，编译器通常不对通过函数指针调用的 inline 函数实施 inlining。即 inline 函数的调用有可能被 inlined，也可能不被 inlined。
- 有时候编译器会生成构造函数和析构函数的 outline 副本，这样它们就可以获得函数指针用在 array 内部元素的构造和析构过程中。

然而，声明 inline 也是有成本的：

- 首先是代码膨胀。
- 其次，**析构函数和构造函数往往是 inline 的糟糕候选人**，虽然表面上构造函数是空的或者很少，实际上所有对象都要初始化以及异常处理都是构造函数要提供的行为，如果是继承类的构造函数 inline，那么基类的构造函数代码也会被插入进来。
- 另外，inline 函数无法随着程序库的升级而升级：假设 f 是一个 inline 函数，如果 f 改变了，所有涉及到 f 的程序都需要重新编译。如果 f 不是 inline 函数，程序只需要重新链接就好。
- 最后，大部分调试器对 inline 函数都束手无策，因为无法对并不存在的函数设立断点。

5.6 将文件间的编译依存关系降到最低

```
class Person {
public:
  Person(const std::string& name, const Date& birthday, const Address& addr);
  std::string name() const;
  std::string birthDate() const;
  std::string address() const;
  ...
private:
  std::string theName; // implementation detail
  Date theBirthDate; // implementation detail
  Address theAddress; // implementation detail
};
```

一旦某个类型发生改变，所有含Person的文件都要重新编译。我们可以采用 **pimpl idiom (pointer to implementation)** 的设计：

```
class PersonImpl; // forward decl of Person impl. class
class Date; // forward decls of classes used in
class Address; // Person interface
class Person {
public:
  Person(const std::string& name, const Date& birthday, const Address& addr);
  std::string name() const;
  std::string birthDate() const;
  std::string address() const;
  ...
private:
  shared_ptr<PersonImpl> pImpl;
};
```

在这样的设计下，Person的客户就完全与其他类的实现细节相分离了。这个分离的关键在于以“声明的依存性”替换“定义的依存性”，那正是编译依存性最小化的本质：**让头文件尽可能自我满足，万一做不到，则让它与其他文件内的声明式（而非定义式）相依**。其他每一件事都源自于这个简单的设计策略：

- 如果使用 object reference 或者 pointer 就可以完成任务，就不要使用 object。
- 尽量以 class 声明式替换 class 定义式：当你声明一个函数而它要用到某个 class 时，并不需要该 class 的定义，加一个前置 class 声明就可以。

```
class Date;
Date today();
void clearAppointments(Date d); //不需要Date的定义式
```

- 为声明式和定义式提供不同的头文件。

像上面的 pimpl idiom 的 classes 经常被称为 **Handle classes**，另一个制作 Handle class 的方法是令Person成为一种特殊的抽象基类，称为 **Interface class**。

```
class Person {
public:
  virtual ~Person();
  virtual std::string name() const = 0;
  virtual std::string birthDate() const = 0;
  virtual std::string address() const = 0;
  static shared_ptr<Person> create(const std::string& name, const Date& birthday, const
  ...
};

class RealPerson: public Person {
public:
  RealPerson(const std::string& name, const Date& birthday, const Address& addr)
  : theName(name), theBirthDate(birthday), theAddress(addr){}
  virtual ~RealPerson() {}
  std::string name() const; // implementations of these
  std::string birthDate() const; // functions are not shown, but
  std::string address() const; // they are easy to imagine
private:
  std::string theName;
  Date theBirthDate;
  Address theAddress;
```

```
};

shared_ptr<Person> Person::create(const std::string& name, const Date&birthday, const Ad
{
  return shared_ptr<Person>(new RealPerson(name, birthday,addr));
}
```

当然，它们也都是有成本代价的，handle classes 会增加内存大小，会为每一次访问增加一层间接性，也要承受初始化，释放的额外开销和异常可能。

Interface classes 每个函数都是`virtual`，所以每次函数调用都要付出一个间接跳跃（indirect jump）成本（见 Section 2.3），另外也要承担 vptr 带来的额外内存。

最后，**Handle classes 和 Interface classes 脱离 inline 函数就没有什么作为**，因为只有 inline 函数本体才必须要置于头文件内部，它们正是特别被设计用来隐藏实现细节的。当它们导致速度或大小差异过于重大以至于 class 之间的耦合并不成为关键时，我们就用具象类（concrete class）替换 handle class 和 interface class。

6 Appendix

6.1 智能指针

shared_ptr使用了引用计数(use count)技术，当复制个 shared_ptr对象时，被管理的资源并没有被复制，而是增加了引用计数。当析构一个shared_ptr对象时，也不会直接释放被管理的的资源，而是将引用计数减一。当引用计数为 0 时，才会真正的释放资源。shared_ptr可以方便的共享资源而不必创建多个资源。

unique_ptr则不同。unique_ptr独占资源，不能拷贝，只能移动。移动过后的unique_ptr实例不再占有资源。当 unique_ptr被析构时，会释放所持有的资源。

```
unique_ptr<T> b = std::move(a);
```

weak_ptr可以解决shared_ptr所持有的资源循环引用问题。weak_ptr在指向shared_ptr时，并不会增加 shared_ptr的引用计数。所以weak_ptr并不知道 shared_ptr所持有的资源是否已经被释放。这就要求在使用 weak_ptr获取shared_ptr时需要判断 shared_ptr是否有效。

```
struct Boo;
struct Foo{
    std::shared_ptr<Boo> boo;
};
struct Boo{
    std::shared_ptr<Foo> foo;
};
```

Foo中有一个智能指针指向Goo，而Goo中也有一根智能指针指向Foo，这就是循环引用，我们可以使用 weak_ptr来解决这个问题。

关于智能指针的一些 tips:

- 智能指针抛开类型 T，是线程安全的，因为智能指针底层使用的引用计数是 atomic的原子变量，原子变量在自增自减时是线程安全的，这保证了多线程读写智能指针时是安全的。
- 尽可能不要使用裸指针初始化智能指针，因为可能存在同一个裸指针初始了多个智能指针，在智能指针析构时会造成资源的多次释放。
- 不要从智能指针中使用get函数返回裸指针去为另一个智能指针赋值，因为如果返回的裸指针被释放了，智能指针持有的资源也失效了，对智能指针的操作是未定义的行为。

- shared_ptr和unique_ptr都可以用reset 函数来释放智能指针,但shared_ptr没有release函数，unique_ptr的release函数会返回内置指针，并将自身置为空。
- 与uniqu_ptr不同,删除器不是shared_ptr类型的组成部分。假设,shared_ptr<T> sp2(q,deleter2)尽管 sp1和sp2有着不同的删除器,但两者的类型是一致的,都可以被放入vector<shared_ptr<T>>类型的同一容器里。
- 与std::unique_ptr不同，自定义删除器不会改变 std::shared_ptr的大小。其始终是F指针大小的两倍。
- shread_ptr和unique_ptr都可以持有数组,但是这里需要在shared_ptr构造时传入deleter,用来销毁持有的数组,默认删除器不支持数组对象,而unique_ptr无需此操作,因为unique_ptr重载了unique_ptr(T[])。**所有不是new分配内存的资源都要给shared_ptr传递一个删除器**。
- 不可以使用静态对象初始化智能指针，因为静态对象的生命周期和进程一样长，而智能指针的析构的时候会导致静态资源被释放。这会导致未定义的行为。
- 不要将this指针返回给shared_ptr。当希望将 this指针托管给shared_ptr时,类需要继承自 std::enable_shared_from_this,并且从 shared_from_this()中获得shared_ptr指针,否则有 double free 的风险。这样做可以的原因是继承自 std::enable_shared_from_this后,内部数据结构会有 shared count 和 weak count，每次创建新的share_ptr都会从这里获取，这样就只有一个控制块，无论怎样增加和减少，都只有一份计数。
- weak_ptr对弱引用计数的获取，实际上是对 shared_ptr的引用计数的获取。弱引用计数最少是 1，不会出现 0。

6.2 万能引用

6.3 零碎知识点

- 普通左值引用无法指向右值，但常量左值引用可以指向右值，常用于拷贝构造函数。