C++ Learning

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 7.8)

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 12.1. **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.

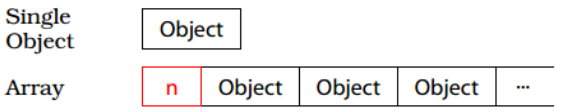
It may have occurred to you that functions returning the raw resource inside an RAII class are **contrary to encapsulation**. That's true, but it's not the design disaster it may at first appear, it hides what clients don't need to see, but it makes available those things that clients honestly need to 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 Section6.3, Section6.5。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 4.6，4.7，7.6。
- **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 7.6。

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 8.2。
- `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。

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 继承与面向对象设计

6.1 确定你的 public 继承塑模出 is-a 关系

public继承意味着 is-a 关系，适用于 base classes 身上的每一件事情一定也适用于 derived class 身上。例如: 企鹅是不会飞的鸟，但大部分鸟会飞。这样三层继承关系比较合适:

```
class Bird {
  ... // no fly function is declared
};
class FlyingBird: public Bird {
public:
  virtual void fly();
  ...
};
class Penguin: public Bird {
  ... // no fly function is declared
};
```

当然，也有一种处理方式是为企鹅定义 fly 函数，令它产生一个运行期错误:

```
class Penguin: public Bird {
public:
  virtual void fly() { error("Attempt to make a penguin fly!"); }
  ...
};
```

Section 4.1说过：**好的接口可以防止无效的代码通过编译**，所以第一种是更好的选择。

6.2 避免遮掩（hide）继承而来的姓名

关于 name hiding 看 Section 12.6。

有时候你并不想继承 base class 的所有函数，在 public 继承下是绝对不可能的，但是在 private 继承下是可以的:

```
class Base {
public:
  virtual void mf1();
```

```
  virtual void mf1(int);
  ... // as before
};
class Derived: private Base {
public:
  virtual void mf1(){ Base::mf1(); }
};
...
Derived d;
int x;
d.mf1(); // fine, calls Derived::mf1
d.mf1(x); // error! Base::mf1() is hidden
```

这里无法用 using，因为它会令所有基类同名函数在继承类中都可见。我们将其称之为 **inline 转交函数（forwarding function）**。

假如继承结合 template，就将面对“继承名称被遮掩”的一个全然不同的形式，到时候再讲，可见 Section 7.3。

6.3 区分接口继承和实现继承

看下面这个例子：

```
class Shape {
public:
  virtual void draw() const = 0;
  virtual void error(const std::string& msg);
  int objectID() const;
  ...
};
class Rectangle: public Shape { ... };
class Ellipse: public Shape { ... };
```

将函数定义为 pure virtual 函数，impure virtual 函数，non-virtual 函数分别对应着你想要 derived class 继承的东西：

- 只继承接口，每一个 shape 继承类都有其独自的 draw 实现。

- 继承接口和一份缺省实现，每一个继承类可以重写，但如果不重写，也可以使用基类的实现。
- 继承接口和一份强制实现，每一个继承类都不应该修改该实现。

这里有一个小 tip 注意一下：pure virtual 函数也是可以定义的，继承类可以用 Base::virtualFunction 调用。

这样可以避免经验不足的 class 设计者常犯的两个错误：

- 将所有函数声明为 non-virtual，这使得继承类无法进行特化，析构函数也会带来问题。
- 将所有成员函数都声明为 virtual，有时候这样是对的，例如 interface class，但是这也可能是设计者缺乏坚定立场的前兆。某些函数就是不该在继承类中重新被定义，就应该将它声明为 non-virtual。

6.4 考虑 virtual 函数以外的其他选择

```
class GameCharacter {
public:
  virtual int healthValue() const;
};
```

还有两种设计可以替代这种 virtual 函数：

- **借助 non-virtual interface（NVI）来实现 template method 模式**：

```
class GameCharacter {
public:
  int healthValue() const{ // derived classes do not redefine
    ... // do “before” stuff — see below
    int retVal = doHealthValue(); // do the real work
    ... // do “after” stuff — see below
    return retVal;
}
private:
  virtual int doHealthValue() const{ // derived classes may redefine this
    ... // default algorithm for calculating character’ s health
  }
};
```

我们把这个 non-virtual 函数称为 virtual 函数的外包装（wrapper）。这种设计的优缺点在于：

- 可以在 wrapper 之前和之后做一些事情，例如验证函数的先决条件和事后条件等等，如果让客户直接调用 virtual 函数就没有办法做到这个事情。
- 但是 NVI 设计涉及在继承类内重新定义若干个继承类并不会调用的 virtual 函数。
- 在 NVI 手法下并不一定需要 virtual 函数是 private 的，假如继承类也需要调用基类函数，就必须是 protected。有时候 virtual 函数甚至一定得是 public，例如析构函数，这样就不能实施 NVI 手法了。

- 第二个是**藉由 Function Pointers 实现 Strategy 模式**。

```
class GameCharacter; // forward declaration
// function for the default health calculation algorithm
int defaultHealthCalc(const GameCharacter& gc);
class GameCharacter {
public:
  typedef int (*HealthCalcFunc)(const GameCharacter&);
  explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc): healthFunc(hcf){}
  int healthValue() const
  { return healthFunc(*this); }
  ...
private:
  HealthCalcFunc healthFunc;
};
```

这种设计模式提供了某些有趣弹性：

- 同一类型的不同实体可以有不同的健康计算函数。
- 某一直任务的健康计算函数可以在运行期变更，即这些计算函数并未访问对象的 non-public 成分。如果计算需要用到 non-public 成分，就需要弱化 class 的封装，例如声明 friend 或者提供一些 public 访问函数等。

- 还可以藉由function完成 **strategy 模式**：

```
class GameCharacter; // as before
```

```
int defaultHealthCalc(const GameCharacter& gc); // as before
class GameCharacter {
public:
  typedef std::function<int (const GameCharacter&)> HealthCalcFunc;
  explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc):healthFunc(hcf){}
  int healthValue() const
  { return healthFunc(*this); }
  ...
private:
  HealthCalcFunc healthFunc;
};
```

std::function使得“健康计算函数”具有更大的弹性:

```
short calcHealth(const GameCharacter&); //non-int return type
struct HealthCalculator { // class for health
  int operator()(const GameCharacter&) const // calculation function objects
  { ... }
};
class GameLevel {
public:
  float health(const GameCharacter&) const;
};
class EvilBadGuy: public GameCharacter { // as before
  ...
};
GameLevel currentLevel;
EvilBadGuy ebg(std::bind(&GameLevel::health,currentLevel,_1));
```

GameLevel::health实际上有两个参数,有一个隐式参数 *this,如果我们将它作为 a 将抗击孙函数,就必须用某种方式转换它,使它不再接受两个参数,于是我们将currentLevel作为第一个参数输入,这个就是std::bind做的事情。

- 最后一点就是可以将集成体系内的 virtual 函数替换为另一个集成体系内的 virtual 函数,比如,关于健康计算可以创建一个继承体系,人物部分再用 std::function实现 strategy 模式即可。

6.5 绝不重新定义继承而来的 non-virtual 函数

```
class B {
public:
  void mf();
};
class D: public B {
public:
  void mf(); // hides B::mf; see Item 33
};

D x;
B *pB = &x;
D *pD = &x;
pB->mf(); // calls B::mf
pD->mf(); // calls D::mf
```

6.6 绝不重新定义继承而来的缺省参数值

virtual 函数是动态绑定，但是缺省参数值却是静态绑定：

```
class Shape {
public:
  enum ShapeColor { Red, Green, Blue };
  virtual void draw(ShapeColor color = Red) const = 0;
};
class Rectangle: public Shape {
public:
// notice the different default parameter value — bad!
virtual void draw(ShapeColor color = Green) const;
...
};
class Circle: public Shape {
public:
virtual void draw(ShapeColor color) const;
...
```

```
};

Circle *pc = new Circle;
Shape *pr = new Rectangle;
pc->draw(); // Error! No default parameter!
pr->draw(); // calls Rectangle::draw(Shape::Red)!
```

虽然 pr 执行的是 Rectangle 的 draw，但是缺省参数值却是来自 Shape class 的。这是编译器为了保证运行期效率所采用的行为。

但是我们真的要每一个继承类都要重新写一下相同的缺省参数值，类似下面这种的吗?

```
class Rectangle: public Shape {
public:
  virtual void draw(ShapeColor color = Red) const;
};
```

这样会代码重复，并且又有相依性，一旦基类缺省值作了修改，所有继承类都要修改。一种替代设计就是 Section 6.4讲到的 NVI 手法：令 base class 内的一个 public non-virtual 函数调用 private virtual 函数，后者可被 derived class 重新定义。

```
class Shape {
public:
  enum ShapeColor { Red, Green, Blue };
  void draw(ShapeColor color = Red) const{
    doDraw(color); // calls a virtual
  }
private:
  virtual void doDraw(ShapeColor color) const = 0;
};
class Rectangle: public Shape {
public:
  ...
private:
  virtual void doDraw(ShapeColor color) const;
};
```

6.7 通过 composition 塑模出 has-a 关系或者“根据某物实现出”关系

composition 指的是某种类型的对象内含有其他类型的对象，同义词还有 layering, containment, aggregation, embedding。

程序中的对象相当于你塑造的某种事物，这属于应用域（application domain）部分，还有一些对象则是纯粹实现细节上的人工制品，例如缓冲区 buffers，互斥器 mutexes，查找树 search trees 等等，这些相当于实现域（implementation domain），当复合发生在这两个区域内，就表现出 has-a 或者 is-implemented-in-terms-of 关系。**has-a 和 is-a 很好区分，但是 is-a 和 is-implemented-in-terms-of 比较麻烦**。例如，当你想利用 list 实现一个 set，这就不能用继承关系，而是要用复合。

6.8 明智而审慎使用 private 继承

private 继承实际上意味着 implemented-in-terms-of。D 以 private 形式继承 B，意味着 D 对象是根据 B 对象实现而得，并没有其他意义了。

Section 6.7提到 compo sition 也有同样的意义，那么怎么做取舍呢？答案很简单：**尽可能使用复合，必要时才使用 private 继承，主要是当 protected 成员或者 virtual 函数牵扯进来的时候，还有一种激进情况是空间方面的利害关系**。

假设我们的程序涉及Widgets，我们需要涉及某种定时器来收集它的数据，此时我们有工具：

```
class Timer {
public:
  explicit Timer(int tickFrequency);
  virtual void onTick() const; // automatically called for each tick
};
```

此时我们有两种设计：

- **private 继承：**

  ```
  class Widget: private Timer {
  private:
    virtual void onTick() const; // look at Widget usage data, etc.
  };
  ```

- **public 继承 + 复合：**

```
class Widget {
private:
  class WidgetTimer: public Timer {
  public:
    virtual void onTick() const;
  };
  WidgetTimer timer;
  ...
};
```

第二种设计是更好的，原因在于：

- 首先，你之后可能会想设计Widget使它得以拥有继承类，同时你可能会想阻止继承类重新定义onTick，这种想法只有第二种设计可以做到。
- 你可能会想将 Widget 的编译依存性降低，那么我们就可以把 WidgetTimer 移出 Widget 之外，Widget 内含一个 WidgetTimer 指针即可，这样 Widget 就不再需要 include 任何与 Timer 有关的东西。

之前提到的激进情况针对于空白类，即内部没有任何数据，包括 non-static 成员，virtual 函数，也没有 virtual base class：

```
class Empty {};
class HoldAnInt {
private:
  int x;
  Empty e;
};
sizeof(HoldsAnInt) > sizeof(int) !!!
```

你会发现很神奇的事情是，HoldsAnInt的 size 会大于一个 int，因为 size(Empty)=1，有时候编译器为了齐位，size(Empty)可能等于size(int)。

但是如果我们继承 Empty：

```
class HoldsAnInt: private Empty {
private:
  int x;
};
```

可以发现`sizeof(HoldsAnInt)==sizeof(int)`，这就是空白基类最优化（EBO，empty base optimization）。

现实中的 empty class 并不一定是 empty，它们可能包含 typedefs，enums，static 成员变量或者 non-virtual 函数。STL 中就有许多技术用途的 empty class，EBO 使得这样的继承很少增加继承类的大小。

6.9 明智而审慎使用多重继承

最先要认清的一件事是程序有可能会从多个 base class 里继承相同名称，从而导致歧义（ambiguity）：

```
class BorrowableItem {
public:
  void checkOut(); // check the item out from the library
};
class ElectronicGadget {
private:
  bool checkOut() const;
};
class MP3Player: public BorrowableItem, public ElectronicGadget{ ... };
MP3Player mp;
mp.checkOut(); // ambiguous!
mp.BorrowableItem::checkOut();
```

多重继承中最要命的是钻石型多重继承：

```
class File { ... };
class InputFile: public File { ... };
class OutputFile: public File { ... };
class IOFile: public InputFile, public OutputFile{ ... };
```

这时候有一个问题：base class 内的成员变量是否要经由每一条路径复制？从一个角度说，对于 File 中的变量 fileName，IOFile 应该有两份 fileName 成员变量，但另一个角度说，IOFile 对象只该有一个文件名称。**C++ 默认第一种，如果想要第二种，只需要让 InputFile 和 OutputFile 进行 virtual 继承即可**。

从正确行为观点看，public 继承应该总是 virtual 的，但是 virtual 继承需要付出很多代价：

- virtual 继承的那些 class 所产生的对象往往比 non-virtual 的大。
- 初始化规则复杂且不直观：virtual base 初始化责任是由 most derived class 负责的。

但是还是非必要不使用 virtual base，如果使用就尽可能避免在 base class 中防放置数据。

7 模板与泛型编程

7.1 了解隐式接口和编译期多态

前面讲的都是显式接口（explicit interface）和运行期多态（runtime polymorphism），即 virtual 函数。但是 template 而言是隐式接口和编译期多态。这类似于“哪一个重载函数该被调用（编译期发生）”和“哪一个 virtual 函数该被绑定（运行期发生）”的差异。

7.2 了解 typename 的双重意义

在 template 声明式中 class 和 typename 没有不同。但有一些情况必须要用 typename。

```
template<typename C> // print 2nd element in
void print2nd(const C& container){ // this is not valid C++!
  if (container.size() >= 2) {
    C::const_iterator iter(container.begin());
    ++iter;
    int value = *iter;
    std::cout << value;
  }
}
```

template 内出现依赖于某个 template 参数，称之为**从属名称**（dependent names），反之为**非从属名称（non-dependent names）**。如果从属名称在 class 内呈嵌套状，则称作 **嵌套（nest）从属名称**，`C::const_iterator`就是这种名称。

嵌套从属名称可能会导致解析（parsing）困难：如果 C 有个 static 成员变量而碰巧命名为`const_iterator`呢？那么，想要告诉编译器这是一个类型的话，就必须在他之前加上关键字`typename`。

一般性规则很简单：任何时候你想在 template 中指涉一个嵌套从属类型名称，就必须加上关键字`typename`，例如：

```
template<typename C> // typename allowed (as is  “class” )
void f(const C& container, // typename not allowed
typename C::iterator iter); // typename require
```

但该规则的一个例外是：`typename`不可以出现在 base class list 内的嵌套从属类型名称之前，也不可以在 member initialization list 中作为 base class 的修饰符，例如：

```
template<typename T>
class Derived: public Base<T>::Nested { // base class list: typename not allowed
public:
  explicit Derived(int x) : Base<T>::Nested(x) // base classidentifier in mem. init. lis
  {
    typename Base<T>::Nested temp; // use of nested dependent type
  }
};
```

最后再看一个typename的例子:

```
template<typename IterT>
void workWithIterator(IterT iter){
  typename std::iterator_traits<IterT>::value_type temp(*iter);
  ...
}
```

7.3 学习处理模板化基类内的名称

```
class CompanyA {
public:
  ...
  void sendCleartext(const std::string& msg);
  void sendEncrypted(const std::string& msg);
  ...
};

template<typename Company>
class MsgSender {
public:
  ... // ctors, dtor, etc.
  void sendClear(const MsgInfo& info){
    std::string msg;
    create msg from info;
    Company c;
    c.sendCleartext(msg);
```

```
  }
  void sendSecret(const MsgInfo& info){ ... }
};
```

如果我们想每次发信息的时候要记录某些东西，我们会想出这样的解法：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  void sendClearMsg(const MsgInfo& info){
    write "before sending" info to the log;
    sendClear(info); // call base class function;
    // this code will not compile!
     write "after sending" info to the log;
  }
  ...
};
```

这里我们换了一个不同的名称，这样既可以避免遮掩继承而得的名称，也避免重新定义一个继承而得的 non-virtual 函数。但是，以上代码无法通过编译，问题在于，编译器在遇到 class template LoggingMsgSender 定义式时，并不知道它继承什么样的 class，更明确地说，编译器没办法知道它是否有 sendClear函数。因此，**编译器往往拒绝在 templatized base class 内寻找继承而来的名称**。

编译器编译期会去解析与模板无关的名称（都是静态绑定），运行期再去解析与模板相关的名称，但不会进入 templatized base class 内寻找名称。

```
void g() { cout<<"gobal::g()"<<endl; }

template<class T> class X {
public:
  void g() { cout<<"X::g()"<<endl; }
};

//Use of undeclared identifer g if global g() has been commented out.
template<class T> class Y : public X<T> {
public:
```

```
  void f() { g(); this->g(); }
};

template<> class Y<char> : public X<char> {
public:
  void f() { g(); this->g(); }
};

Y<char> y;
y.f(); \\ X::g() X::g()
Y<int> y;
y.f(); \\ global::g() X::g()
```

我们有三种方法可以让 C++ “不进入 templatized base classes 观察” 的行为失效:

- 在 base class 函数调用动作之前加上this->:

```
void sendClearMsg(const MsgInfo& info){
  write "before sending" info to the log;
  this->sendClear(info); // okay, assumes that
  // sendClear will be inherited
  write "after sending" info to the log;
}
```

- 使用using声明式:

```
using MsgSender<Company>::sendClear; // tell compilers to assume that sendClear is
void sendClearMsg(const MsgInfo& info){
  ...
  sendClear(info); // okay, assumes that sendClear will be inherited
  ... //
}
```

- 指出被调用的函数位于 base class 内部:

```
void sendClearMsg(const MsgInfo& info){
  ...
```

```
    MsgSender<Company>::sendClear(info);
    ...
  }
```

但这样会关闭 virtual 绑定行为。

上述每一个解法都在**对编译器承诺，base class template 的任何特化版本都将支持其一般版本所提供的借口**。

7.4 将与参数无关的代码抽离 templates

```
template<typename T, std::size_t n>
class SquareMatrix {
public:
  ...
  void invert(); // invert the matrix in place
};
```

当 n 不同时会产生多个 invert() 函数，这个是重复的，因此我们可以这样解决：

```
template<typename T>
class SquareMatrixBase {
protected:
  SquareMatrixBase(std::size_t n, T *pMem) : size(n), pData(pMem) {}
  void setDataPtr(T *ptr) { pData = ptr; } // reassign pData
  ...
private:
  std::size_t size; // size of matrix
  T *pData; // pointer to matrix values
};

template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
public:
  SquareMatrix() : SquareMatrixBase<T>(n, 0), /pData(new T[n*n])
  { this->setDataPtr(pData.get()); } //give a copy of it to the base class
private:
```

```
  boost::scoped_array<T> pData; // see Item 13 for info on
};
```

7.5 运用成员函数模板接受所有兼容类型

真实指针做得很好的一件事是支持隐式转换：Derived class 指针可以隐式转换为 base class 指针，non-const 对象指针可以转换为 const 对象的指针等等。但是智能指针比较麻烦，因为同一个 template 的不同 instantiations 之间没有固有关系。

我们可以使用 member function templates：

```
template<typename T>
class SmartPtr {
public:
  template<typename U>
  SmartPtr(const SmartPtr<U>& other)
  : heldPtr(other.get()) { ... }
  T* get() const { return heldPtr; }
  ...
private:
  T *heldPtr;
};
```

这个行为只有当坐在某个隐式转换的时候才能通过编译，这正是我们想要的。

除了构造函数，这种方法也支持赋值操作。但有一点要注意：如果你声明 member template 用于拷贝构造或赋值操作，你还是要声明正常的拷贝构造函数和拷贝赋值操作符。

```
template<class T> class shared_ptr {
public:
  template<class Y>
  explicit shared_ptr(Y * p);
  template<class Y>
  shared_ptr(shared_ptr<Y> const& r);
  template<class Y>
  explicit shared_ptr(weak_ptr<Y> const& r);
  template<class Y>
  explicit shared_ptr(auto_ptr<Y>& r); //there is not const!
```

```
  template<class Y>
  shared_ptr& operator=(shared_ptr<Y> const& r);
  template<class Y>
  shared_ptr& operator=(auto_ptr<Y>& r);
```

从某个内置指针或者智能指针类型另一个shared_ptr类型是不允许的。auto_ptr是并未被声明为 const 的。

7.6 需要类型转换时将模板定义为非成员函数

Section 7.6讨论过为什么只有 non-member 函数才有能力在所有实参上实施隐式类型转换，在这里我们将其模板化了。

```
template<typename T>
class Rational {
public:
  Rational(const T& numerator = 0, const T& denominator = 1);
  const T numerator() const;
  const T denominator() const;
};
template<typename T>
const Rational<T> operator*(const Rational<T>& lhs, const Rational<T>& rhs)
{ ... }

Rational<int> oneHalf(1, 2);
Rational<int> result = oneHalf * 2; // error! won' t compile
```

在调用 operator* 的时候，编译器并不知道我们想要调用什么函数，它需要想出什么函数被 operator* 的 template 具现化出来，所以需要先算出 T 是什么，但是 **template 的实参推导规程并不考虑“通过构造函数而发生的”隐式类型转换**。

解决办法就是在 template class 内使用 friend 声明式：

```
template<typename T>
class Rational {
public:
  ...
```

```
  friend const Rational operator*(const Rational& lhs, const Rational& rhs);
};
template<typename T>
const Rational<T> operator*(const Rational<T>& lhs, const Rational<T>& rhs)
{ ... }
```

当oneHalf被声明为一个Rational<int>的时候，class Rational<int>也被具现化出来，同时 friend 函数也就被自动声明出来，它此时只是一个函数而非函数模板，因此编译器调用他的时候就可以使用隐式转换函数了。

但此时能通过链接，但是无法编译，因为这个函数只被声明于类内部，并没有被定义出来，**最简单的可行办法就是将operator*本体合并至声明式内部**。

这个技术的一个趣味点是：我们虽然用 friend，却与其传统用途“访问 class 的 non-public”成分毫不相干。为了让类型转换发生于所有实参上，我们需要 non-member 函数；为了令函数被自动具现化，需要声明在 class 内部，而在 class 内部声明一个 non-member 函数的唯一办法就是 friend。

正如 Section 5.5所说，定义在 class 内部的函数都会成为 inline，为了减少这种 inline 声明带来的冲击，我们可以让operator*不作任何事情，让它去调用辅助函数：

```
template<typename T> class Rational;
template<typename T>
const Rational<T> doMultiply( const Rational<T>& lhs, const
Rational<T>& rhs);
template<typename T>
class Rational {
public:
  ...
  friend const Rational<T> operator*(const Rational<T>& lhs,
const Rational<T>& rhs)
  { return doMultiply(lhs, rhs);
  ...
};
```

7.7 使用 traits classes 表现类型信息

STL 有五种迭代器：

- **Input 迭代器**只能向前移动，一次一步，客户只能读取它们所指的东西，且只能读取一次，例如istream_iterators
- **Output 迭代器**类似，例如ostream_iterators，他们俩只适合一次性操作算法（one-pass algorithms）
- **forward** 迭代器可以做前面的事情，也可以读或写一次以上，使得它们可以用于多次性操作算法，例如单向 linked list 的迭代器，
- **Bidirectional 迭代器**可以双向移动，例如 STL 的 list, set, multiset, map, multimap 的迭代器。
- **random access 迭代器**，它最有威力，因为它可以执行迭代器算术：在常量时间内向前或向后跳跃人以距离，例如 vector, dequem string 的迭代器。

对于这五种分类，STL 分别提供专属的卷标结构（tag struct）加以确认：

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag: public input_iterator_tag {};
struct bidirectional_iterator_tag: public forward_iterator_tag {};
struct random_access_iterator_tag: public bidirectional_iterator_tag {};
```

如果我们想实现 advance 函数来将迭代器移动某个距离：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d){
  if (iter is a random access iterator) {
    iter += d; // use iterator arithmetic
  }
  else {
    if (d >= 0) { while (d--) ++iter; }
    else { while (d++) --iter; }
  }
}
```

这种做法必须先判断 iter 是不是 random access 迭代器，这就是 traits 要做的事情：它们允许你在编译期间取得某些类型信息。

```
template < ... > // template params elided
class deque {
public:
  class iterator {
  public:
    typedef random_access_iterator_tag iterator_category;
    ...
  };
  ...
};

template<typename IterT>
struct iterator_traits {
  typedef typename IterT::iterator_category iterator_category;
  ...
};
```

这对用户自定义类型行得通，但这对指针行不通，因为指针不可能嵌套 typedef，因此还有第二部分：

```
template<typename T> // partial template specialization
struct iterator_traits<T*> // for built-in pointer types
{
  typedef random_access_iterator_tag iterator_category;
  ...
};
```

现在就应该知道如何设计一个 traits class 了：

1. 确认若干你希望将来可取得的类型相关信息。例如对迭代器而言，我们希望取得它的分类。
2. 为该信息选择一个名称。
3. 提供一个 template 和一组特化版本，内含你支持的类型相关信息。

现在就有了 advance 的伪代码：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d){
  if (typeid(typename std::iterator_traits<IterT>::iterator_category) ==
  typeid(std::random_access_iterator_tag))
  ...
}
```

但是它还会有问题，if语句是在运行期才会核定，我们希望它能在编译器就完成（C++17有了if constexpr），我们也可以选择重载：

```
template<typename IterT, typename DistT>
void doAdvance(IterT& iter, DistT d, std::random_access_iterator_tag){
  iter += d;
}
template<typename IterT, typename DistT>
void doAdvance(IterT& iter, DistT d, std::bidirectional_iterator_tag){
  if (d >= 0) { while (d--) ++iter; }
  else { while (d++) --iter; }
}
template<typename IterT, typename DistT>
void doAdvance(IterT& iter, DistT d, std::input_iterator_tag){
  if (d < 0 ) {
    throw std::out_of_range("Negative distance"); // see below
  }
  while (d--) ++iter;
}
```

这里也就是继承关系带来的一项红利。于是我们可以实现出 advance 函数：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d){
  doAdvance(iter, d, typename  std::iterator_traits<IterT>::iterator_category());
}
```

我们现在总结一下如何使用 traits class：

1. 建立一组重载函数或者函数模板，彼此间的差异只在于各自的 traits 参数，令每个函数实现与 tarits 信息对应。

2. 建立一个控制函数或者函数模板，它调用上述重载函数并传递 traits class 所提供的信息。

不光iterator_traits在用，iterator_category还提供灵四分迭代器相关信息，其中最有用的是value_type，可以参考 Section 7.2。另外还有char_traits, numeric_limits。

7.8 认识 template 元编程

Template metaprogramming（TMP，模板元编程）是编写 template-based C++ 程序并执行于编译器的过程。TMP 有两个作用：可以让事情变得更容易，也可以将工作从运行期转移到编译期，从而使得某些错误在编译期就可以找到，并且程序会更高效，比如上一节的 typeid 和 traits 的比较。

```
void advance(std::list<int>::iterator& iter, int d){
  if(typeid(std::iterator_traits<std::list<int>::iterator>::iterator_category)
     == typeid(std::random_access_iterator_tag)) {
    iter += d; // error! won' t compile
  }
  else {
    if (d >= 0) { while (d--) ++iter; }
    else { while (d++) --iter; }
  }
}
```

即便 if 语句判断是错的，内部代码还是会被检查，由于 list 的 iterator 不存在 += 运算，所以无法通过编译。

TMP 还可以用来**递归模板具现化**：

```
template<unsigned n>
struct Factorial {
  enum { value = n * Factorial<n-1>::value };
};
template<>
struct Factorial<0> {
  enum { value = 1 };
};
std::cout << Factorial<5>::value;
```

下面举 TMP 的三个重要例子：

- 确保量度单位准确，比如将一个质量变量赋值给一个速度变量是错误的但是一个距离变量除以时间变量再赋值给速度变量是成立的，如果使用 TMP，就可以保证在编译期程序中所有量度单位的组合都正确。
- 表达式模板可以优化矩阵运算，见 Section 12.5：

```
typedef SquareMatrix<double, 10000> BigMatrix;
BigMatrix m1, m2, m3, m4, m5;
BigMatrix result = m1 * m2 * m3 * m4 * m5;
```

- 生成各种 design pattern。

8 定制 new 和 delete

8.1 了解 new-handler 的行为

当operator new抛出异常的时候,他会先调用一个客户指定的错误处理函数,称为new-handler(并非全部事实，真正做的事情会更复杂一点，见 Section 8.3)。为了指定这个函数，客户必须调用 set_new_handler，这是声明于<new>的函数：

```
namespace std {
  typedef void (*new_handler)();
  new_handler set_new_handler(new_handler p) throw();
}
```

set_new_handler的参数是个指针指向operator new无法分配足够内存的时候调用的函数，返回值也是个指针，指向 set_new_handle被调用前正在执行的哪个new-handler函数。可以这样使用：

```
void outOfMem()
{
  std::cerr << "Unable to satisfy request for memory\n";
  std::abort();
}
int main()
{
  std::set_new_handler(outOfMem);
  int *pBigDataArray = new int[100000000L];
  ...
}
```

当operator new无法满足内存申请时，它会不断调用 new-handler函数，直到找到足够内存。一个设计良好的 new-handler函数必须做到：

1. 让更多内存可被使用。实现该策略的一个做法是程序一开始执行就分配一大块内存，当 new-handler 第一次被调用再将它们还给程序使用。

2. 安装另一个 new-handler。如果这个 new-handler 无法取得更多可用内存，或许他知道另外哪个 new-handler 有此能力，就可以让它替换自己。为达到此目的，做法之一是 new-handler 修改“会影响其行为”的 static 数据，namespace 数据或 global 数据。

3. 卸除 new-handler：将 null 传给set_new_handler，operator new则会在不成功时抛出异常。
4. 抛出bad_alloc异常，它会被传播到内存索求处。
5. 不返回，调用abort或者exit。

若想在 class 中使用，只需令 class 提供自己的set_new_handler和 operator new即可。

```
class Widget {
public:
  static std::new_handler set_new_handler(std::new_handler p) throw();
  static void* operator new(std::size_t size) throw(std::bad_alloc);
private:
  static std::new_handler currentHandler;
};
std::new_handler Widget::currentHandler = 0;

std::new_handler Widget::set_new_handler(std::new_handler p) throw(){
  std::new_handler oldHandler = currentHandler;
  currentHandler = p;
  return oldHandler;
}
```

最后，Widget的operator new做以下事情：

1. 调用标准set_new_handler，告知 Widget 的错误处理函数，这会为Widget安装 global new-handler
2. 调用 global operator new，执行实际内存分配。如果分配失败，global operator new会调用Widget的 new-handler。
3. 如果最终无法分配足够内存，就会抛出bad_alloc异常，在此情况下，Widget的operator new必须恢复原本的 glocal new-handler然后再传播该异常。为确保原本的 new-handler 总能被重新安装回去，Widget将 global new-handler视为资源并遵守 Section 3.1，运用资源管理对象防止资源泄露。
4. 如果 global operator new能够分配足够内存，则会返回一个指针，指向分配所得。Widget析构函数会管理 glocal new-handler，它会自动将operator new被调用前的 new-handler恢复回来。

```
class NewHandlerHolder {
public:
  explicit NewHandlerHolder(std::new_handler nh) : handler(nh) {}
  ~NewHandlerHolder() { std::set_new_handler(handler); }
private:
  std::new_handler handler;
  NewHandlerHolder(const NewHandlerHolder&); // prevent copying
  NewHandlerHolder& operator=(const NewHandlerHolder&);
};

void* Widget::operator new(std::size_t size) throw(std::bad_alloc){
  // install Widget' s new-handler
  NewHandlerHolder h(std::set_new_handler(currentHandler));
  return ::operator new(size); // allocate memory or throw
}
```

Widget的客户应该类似这样使用其new-handler：

```
// decl. of func. to call if mem. alloc. for Widget objects fails
void outOfMem();

/ set outOfMem as Widget' s new-handling function
Widget::set_new_handler(outOfMem);

// if memory allocation fails, call outOfMem
Widget *pw1 = new Widget;

// if memory allocation fails, call the global new-handling function (if there is one)
std::string *ps = new std::string;

Widget::set_new_handler(0);
Widget *pw2 = new Widget;
// if mem. alloc. fails, throw an exception immediately.
```

实现这个方案的代码并不因 class 的不同而不同，所以可以提取出来复用：

```
template<typename T> //  “mixin-style”  base class
```

```
class NewHandlerSupport {
public:
  static std::new_handler set_new_handler(std::new_handler p) throw();
  static void* operator new(std::size_t size) throw(std::bad_alloc);
  ...
private:
  static std::new_handler currentHandler;
};

template<typename T>
std::new_handler NewHandlerSupport<T>::set_new_handler(std::new_handler p) throw(){
  std::new_handler oldHandler = currentHandler;
  currentHandler = p;
  return oldHandler;
}
template<typename T>
void* NewHandlerSupport<T>::operator new(std::size_t size) throw(std::bad_alloc){
  NewHandlerHolder h(std::set_new_handler(currentHandler));
  return ::operator new(size);
}

// this initializes each currentHandler to null
template<typename T>
std::new_handler NewHandlerSupport<T>::currentHandler = 0;

class Widget: public NewHandlerSupport<Widget> {
  ...
};
```

Widget继承自一个模板化的 base class，后者又以Widget作为类型参数，这种模式称为**curiously recurring template pattern; CRTP**，可以理解为Do It For Me。

8.2 了解 new 和 delete 的合理替换时机

有三个常见理由：

1. **用来检测运用上的错误**。如果 new 所得内存 delete 时不行失败，就会内存泄漏，多次 delete 则会导致不确定行为。如果 operator new 持有一串动态分配的地址，delete 将地址从中一走，倒是很容易检测出来该错误。

   此外各种各样的变成错误可能导致数据 overruns（写入点在分配区块尾端之后）或者 underruns。如果我们自行定义 operator news，便可以超额分配内存，用额外空间放置特定的 signatures，operator delete 可以检查上述 signatures 是否原封不动，如果不是就表示发生了 overrun 或者 underrun，这时候就可以 log 下这个事实。

2. **为了强化效能**。编译器自带的 new 和 delete 主要为了一般目的，它们不但可以被长时间执行的程序接受也可以被少于 1s 的程序接受；它们必须处理大块、小块、大小混合型内存；它们还必须接纳各种分配形态；它们必须考虑破碎问题。所以有时候定制版本性能会更好，速度快，所需内存少。

3. **为了收集使用上的统计数据**。分配区块的大小分布如何？寿命分布如何？倾向于 FIFO 还是 LIFO 还是随机次序来分配和归还？运用形态是否随时间改变？最大动态分配量是多少?

4. **增加分配和归还的速度**。

5. **降低缺省内存管理器带来的空间额外消耗**。可以看 **Boost::Pool 程序库**

6. **弥补缺省分配器的非最佳齐 suboptimal alignment 位**。

7. **将相关对象成簇集中**。

8. **为了获得非传统的行为**。

以下是一个简单的少有问题的检测 overruns 或 underruns 的定制版 new：

```
typedef unsigned char Byte;
// this code has several flaws
void* operator new(std::size_t size) throw(std::bad_alloc){
  using namespace std;
  size_t realSize = size + 2 * sizeof(int);
  // signatures will also fit inside
  void *pMem = malloc(realSize);
  if (!pMem) throw bad_alloc();
  *(static_cast<int*>(pMem)) = signature;
  *(reinterpret_cast<int*>(static_cast<Byte*>(pMem)+realSize-sizeof(int))) = signature;
```

```
  // return a pointer to the memory just past the first signature
  return static_cast<Byte*>(pMem) + sizeof(int);
}
```

首先，它没有满足 Section 8.3所说的反复调用 newhandler 函数。另外，这里还涉及到**齐位（alignment）**。

C++ 要求 operator new 返回的指针都有适当的对齐，malloc 就是在这样的要求下工作，但是我们还加了一个偏移量。如果我们企图获得一个 double 所用的内存，但是偏移量只有 4 bytes，这会导致程序崩溃或执行速度变慢。

8.3 编写 new 和 delete 时需要固守常规

non-member operator new 的伪码：

```
void* operator new(std::size_t size) throw(std::bad_alloc){
  using namespace std;
  if (size == 0) { // handle 0-byte requests
    size = 1; // by treating them as 1-byte requests
  }
  while (true) {
    attempt to allocate size bytes;
    if (the allocation was successful)
      return (a pointer to the memory);
    // allocation was unsuccessful; find out what the current new-handling function is
    new_handler globalHandler = set_new_handler(0);
    set_new_handler(globalHandler);
    if (globalHandler) (*globalHandler)();
    else throw std::bad_alloc();
   }
}
```

这里注意几点：

- 要处理 0 字节的情况
- 内存申请失败的话需要无限循环来尝试分配内存。
- 其中将 new-handling 设为 null 后又可恢复原样，因为我们没有办法直接获得 new-handling 函数指针，这样很有效。

- Class 专属版本还应该处理 “比正确大小更大的（错误）申请”，因为 operator new 会被 derived class 继承。

```
void* Base::operator new(std::size_t size) throw(std::bad_alloc){
  if (size != sizeof(Base))
    return ::operator new(size);
}
```

撰写 delete 唯一需要记住的事情就是 **C++ 保证删除 null 指针永远安全**。以下为 non-mamber delete 的伪码：

```
void operator delete(void *rawMemory) throw(){
  if (rawMemory == 0) return; // do nothing if the null pointer is being deleted
  deallocate the memory pointed to by rawMemory;
}
```

如果是 class 版本则需要多加一个检查删除数量：

```
void Base::operator delete(void *rawMemory, std::size_t size) throw(){
  if (rawMemory == 0) return; // check for null pointer
  if (size != sizeof(Base)) {
    ::operator delete(rawMemory);
    return;
  }
  deallocate the memory pointed to by rawMemory;
  return;
}
```

8.4 写了 placement new 也要写 placement delete

```
Widget *pw = new Widget;
```

假如 new 调用成功，但是 Widget 的默认构造函数却抛出异常，那么该内存必须取消并恢复旧观，否则会造成内存泄漏，在这个时候客户没有能力归还内存，因此这个责任交给了 C++ 运行期系统上，它会去调用 new 对应的 delete 版本，但是前提是他必须知道改调用哪一个。

对于非正常形式的 new，它要去寻找参数个数和类型都相同的 delete。因此：声明 **placement new** 必须要同时声明一个 **placement delete**，否则不会调用任何一个 delete 从而导致内存泄漏。

另外，也要必须小心避免让 class 专属的 new 掩盖客户其他的 new（包括正常版本的 new）（normal new 的话再写一个函数调用，base new 就用 using）：

```
class Base {
public:
  ...
  static void* operator new(std::size_t size, std::ostream& logStream) throw(std::bad_al
  ...
};
Base *pb = new Base; // error! the normal form of operator new is hidden
Base *pb = new (std::cerr) Base; // fine, placement new.

class Derived: public Base {
public:
  ...
  static void* operator new(std::size_t size) throw(std::bad_alloc);
  ...
};

Derived *pd = new (std::clog) Derived; // error! Base' s placement new is hidden
Derived *pd = new Derived; // fine
```

缺省情况下，C++ 在 global 作用域内提供以下形式的 operator new：

```
void* operator new(std::size_t) throw(std::bad_alloc); // normal new
void* operator new(std::size_t, void*) throw(); // placement new
void* operator new(std::size_t, const std::nothrow_t&) throw(); //nothrow new, return nu
```

9 类型推导

9.1 理解模板类型推导

```
template<typename T>
void f(ParamType param);
f(expr);
```

在编译期间，编译器会用expr推到两个类型：T以及 ParamType。这两种类型通常是不一样的，因为ParamType经常带修饰符，例如const或者reference。T类型的推导不仅仅取决于expr，也取决于ParamType的类型，分为三种情况：

- ParamType是指针或者引用类型，但不是通用引用 (Universal reference，ParamType带有&&修饰符，见 Section 12.2).
- ParamType是通用引用.
- ParamType既不是指针也不是引用。

以下将分类进行讨论：

- 当ParamType是指针或者引用类型，但不是通用引用时，类型推导原理为：
  1. 如果expr类型是引用的话，忽略引用。
  2. 通过令expr

  和ParamType类型一致来决定T。例如：

```
template<typename T>
void f(T& param);
void g(const T& param); // param is now a ref-to-const

int x = 27;        // x is an int
const int cx = x;  // cx is a const int
const int& rx = x; // rx is a reference to x as a const int

f(x);  // T is int, param's type is int&
f(cx); // T is const int, param's type is const int&
f(rx); // T is const int, param's type is const int&
```

```
g(x); // T is int, param's type is const int&
g(cx); // T is int, param's type is const int&
g(rx); // T is int, param's type is const int&
```

- ParamType为通用引用时，类型推导（见 Section ??）如下：

  1. 如果expr为左值，则T和ParamType均为左值引用，**这是唯一一种T被推导为引用的情况**，另外值得注意的是，虽然ParamType带有&&，**但还是被推导为左值引用**。
  2. 如果expr为右值，那么推导和第一种情况类似。

  例如：

```
template<typename T>
void f(T&& param); // param is now a universal reference

int x = 27; // as before
const int cx = x; // as before
const int& rx = x; // as before

f(x);  // x is lvalue, so T is int&,param's type is also int&
f(cx); // cx is lvalue, so T is const int&, param's type is also const int&
f(rx); // rx is lvalue, so T is const int&, param's type is also const int&
f(27); // 27 is rvalue, so T is int, param's type is therefore int&&
```

- ParamType既不是指针也不是引用时，这意味着param 将会传递输入参数的副本，因此此时的类型推导如下：

  1. 如果expr是一个引用，则忽略引用部分。
  2. 如果忽略引用之后是const的，也将const忽略。如果是volatile(见 Section ??) 的也要忽略。

  例如：

```
template<typename T>
void f(T param); // param is now passed by value
```

```
int x = 27;
const int cx = x;
const int& rx = x;
f(x);  // T's and param's types are both int
f(cx); // T's and param's types are again both int
f(rx); // T's and param's types are still both int
```

这样做的原因是cx和rx不能被更改不代表param 不能被更改,另外,要意识到**const和volatile只有在按值传入参数的时候会被忽略**。

如果expr是一个指向const对象的const指针的话，**指针的const会被忽略**:

```
// ptr is const pointer to const object
const char* const ptr = "Fun with pointers";

// type deduced for param will be const char*
f(ptr);
```

在许多情况下，数组会退化为指向第一个元素的指针。当数组被传入到一个 by-value 参数的模板中呢？

```
template<typename T>
void f(T param); // template with by-value parameter

const char name[] = "J. P. Briggs"; // name's type is const char[13]
f(name); // name is array, but T deduced as const char*
```

但是如果expr是一个引用呢?

```
template<typename T>
void f(T& param); // template with by-reference parameter

f(name); // T is deduced to be const char [13]
//paramType is const char (&)[13]
```

这种语法使得我们可以利用模板来推导出数组元素个数:

```
template<typename T, std::size_t N>
constexpr std::size_t arraySize(T (&)[N]) noexcept{
  return N;
}
```

将函数声明为constexpr可以让结果在编译器就可以使用，同时也能让编译器生成更好的代码 (见 Section 11.8, 11.9):

```
int keyVals[] = { 1, 3, 7, 9, 11, 22, 35 }; // keyVals has 7 elements
int mappedVals[arraySize(keyVals)]; // so does mappedVals
std::array<int, arraySize(keyVals)> mappedVals; // mappedVals'size is 7
```

同样的，**函数类型也会退化到 function 指针**:

```
void someFunc(int, double); // someFunc is a function with type
void(int, double)

template<typename T>
void f1(T param); // in f1, param passed by value
template<typename T>
void f2(T& param); // in f2, param passed by ref

f1(someFunc); // param deduced as ptr-to-func;
// type is void (*)(int, double)
f2(someFunc); // param deduced as ref-to-func;
// type is void (&)(int, double)
```

9.2 理解 auto 类型推导

当一个变量用auto声明的时候，auto扮演模板中的 T的作用，变量的类型修饰符可以看做ParamType。

同样的，auto也分为三个情况:

- 类型修饰符是指针或引用，但不是通用引用。
- 类型修饰符是通用引用。
- 类型修饰符既不是指针也不是引用。

```
auto x = 27;        // case 3 (x is neither ptr nor ref)
const auto cx = x;  // case 3 (cx isn't either)
const auto& rx = x; // case 1 (rx is a non-universal ref)

auto&& uref1 = x;  // uref1's type is int&
auto&& uref2 = cx; // uref2's type is const int&
auto&& uref3 = 27; // uref3's type is int&&

const char name[] = "R. N. Briggs"; // name's type is const char[13]
auto arr1 = name; // arr1's type is const char*
auto& arr2 = name; // arr2's type is const char (&)[13]

void someFunc(int, double); //void(int, double)
auto func1 = someFunc; // func1's type is void (*)(int, double)
auto& func2 = someFunc; // func2's type is void (&)(int, double)
```

可以看到，auto类型推导和模板类型推导一致。只有一种情况下，它们是不同的：

```
int x1 = 27;
int x2(27);
int x3 = { 27 };
int x4{ 27 };
```

它们都是一个结果：值为 27 的int，但是如果我们换成auto 的话，对于后两个而言，会定义一个只包含一个元素 27 的类型为 std::initializer_list<int>的对象。

```
auto x1 = 27; // type is int, value is 27
auto x2(27); // ditto
auto x3 = { 27 }; // type is std::initializer_list<int>, value is { 27 }
auto x4{ 27 }; // ditto
```

花括号初始化的时候，auto会推断为 std::initializer_list<int>，假如花括号内部类型不相同，则会报错：

```
auto x5 = { 1, 2, 3.0 }; // error! can't deduce T for std::initializer_list<T>
```

然而对于模板而言，使用花括号的话类型推导会失败：

```
auto x = { 11, 23, 9 };
template<typename T>
void f(T param);
f({ 11, 23, 9 }); // error! can't deduce type for T

template<typename T>
void f(std::initializer_list<T> initList);
f({ 11, 23, 9 });
```

C++14 中auto可以用在函数返回类型中，也可以在 lambda 表达式的参数声明中使用，然而**它们使用的是模板类型推导而不是auto类型推导**。

```
auto createInitList(){
  return { 1, 2, 3 }; // error: can't deduce type.
}

std::vector<int> v;
auto resetV = [&v](const auto& newValue) { v = newValue; };
resetV({ 1, 2, 3 }); // error! can't deduce type.
```

9.3 理解 decltype

decltype会告诉你表达式的类型：

```
const int i = 0; // decltype(i) is const int
bool f(const Widget& w); // decltype(w) is const Widget&
// decltype(f) is bool(const Widget&)

struct Point {
int x, y; // decltype(Point::x) is int
}; // decltype(Point::y) is int
Widget w; // decltype(w) is Widget

if (f(w)) … // decltype(f(w)) is bool

template<typename T> // simplified version of std::vector
```

```
class vector {
public:
…
T& operator[](std::size_t index);
…
};
vector<int> v; // decltype(v) is vector<int>
if (v[0] == 0) … // decltype(v[0]) is int&
```

在 C++11 中decltype主要用来声明那些返回值依赖于参数类型的函数模板。例如，假如函数要返回vector<bool>中的一个元素，然而 operator[]并不会返回bool&而是一个新的object，这里的关键是返回值是依赖于容器的：

```
// works, but requires refinement.
template<typename Container, typename Index>
auto authAndAccess(Container& c, Index i) -> decltype(c[i]){
  authenticateUser();
  return c[i];
}
```

这就是 C++11 中的**返回值类型后置语法**。在 C++14 中可以只留下 auto：

```
template<typename Container, typename Index>
auto authAndAccess(Container& c, Index i){
  authenticateUser();
  return c[i]; // return type deduced from c[i]
}
```

这里auto按照模板类型推导，然而这里是存在问题的，虽然 operator[]返回T&，但是在推导时引用会被忽略，因此下面的代码是无法编译的：

```
std::deque<int> d;
authAndAccess(d, 5) = 10;
```

C++14 中，我们可以这样改进，我们称之为decltype推导方法：

```
// still have refinement.
template<typename Container, typename Index>
```

```
decltype(auto) authAndAccess(Container& c, Index i){
  authenticateUser();
  return c[i];
} // return type T&.
```

这种推导方式也可以用作初始化表达式:

```
Widget w;
const Widget& cw = w;
auto myWidget1 = cw; // myWidget1's type is Widget
decltype(auto) myWidget2 = cw; // myWidget2's type is const Widget&
```

上面说到的 refinement 实际上是说, 如果我想传入一个右值, 例如一个临时的 Container, 这是无法编译通过的, 因此我们还可以做出改进:

```
template<typename Container, typename Index> // final C++14 version
decltype(auto) authAndAccess(Container&& c, Index i){
  authenticateUser();
  return std::forward<Container>(c)[i];
}

template<typename Container, typename Index> // final C++11 version
auto authAndAccess(Container&& c, Index i)
-> decltype(std::forward<Container>(c)[i]){
  authenticateUser();
  return std::forward<Container>(c)[i];
}
```

这里我们使用了`std::forward`, 参见 Section 12.2.

当然,在极少数情况下`decltype`会得不到你期望的类型:如果对一个名称使用`decltype`是没有问题的, 但是如果对于一个比名称复杂的左值表达式, `decltype`都会返回左值引用:

```
decltype(auto) f1(){
  int x = 0;
  return x; // decltype(x) is int, so f1 returns int
}
decltype(auto) f2(){
```

```
  int x = 0;
  return (x); // decltype((x)) is int&, so f2 returns int&
}
```

9.4 了解如何去看类型推导

有以下几种方法：

- 通过 IDE 编辑器来看。
- 通过编译器诊断：

  ```
  template<typename T> // declaration only for TD;
  class TD; // TD == "Type Displayer"
  TD<decltype(x)> xType;
  ```

  编译器会报错：

  ```
  error: 'xType' uses undefined class 'TD<int>'
  ```

- 运行期也可以输出：

  ```
  #include <boost/type_index.hpp>
  std::cout << typeid(x).name() << '\n'; // display types for x
  using boost::typeindex::type_id_with_cvr;
  cout << "T = " << type_id_with_cvr<T>().pretty_name();
  ```

10 auto

10.1 相比于显式类型声明，更倾向于用 auto

- 使用auto可以避免忘记初始化：

  ```
  int x1; // potentially uninitialized
  auto x2; // error! initializer required
  ```

- 也可以在声明与迭代器相关的局部变量时省点笔墨：

```
template<typename It> // as before
void dwim(It b, It e){
  while (b != e) {
    // typename std::iterator_traits<It>::value_type
    auto currValue = *b;
  }
}
```

- 也可以表示那些只有编译器才知道的类型:

```
auto derefUPLess =  [](const std::unique_ptr<Widget>& p1,
                       const std::unique_ptr<Widget>& p2)
{ return *p1 < *p2; };
```

在 C++14 中这种特性得到了延续:

```
auto derefUPLess =  [](const auto& p1, const auto& p2)
{ return *p1 < *p2; };
```

上边的函数的类型也可以写作:

```
std::function<bool(const std::unique_ptr<Widget>&,
                   const std::unique_ptr<Widget>&)>
derefUPLess =  [](const std::unique_ptr<Widget>& p1,
                      const std::unique_ptr<Widget>& p2)
{ return *p1 < *p2; };;
```

它可以用来表示任何可调用的 object。

但是这样的话即使抛开语法的冗长，也需要重复参数类型，另外，使用 `std::function`与使用`auto` 不同: 一个持有闭包的`auto`声明变量与 closure 具有相同的类型，因此它只使用 closure 所需的内存。持有 closure 的`std::function`声明变量的类型是`std::function` object 的实例化模板，通常会比前者占用更多的内存，并且通过`std::function`调用一个 closure 会更慢。

另外，`auto`也可以避免"type shortcuts":

```
std::unordered_map<std::string, int> m;
for (const std::pair<std::string, int>& p : m){
  // do something with p
}
```

这看起来很正确，但是却存在问题：std::unordered_map的 key 类型是const的，所以 Hash 表中的std::pair的类型是 std::pair<const std::string, int>。因此，编译器会创造一个临时 object 绑定在 p 上，之后再销毁。所以为了避免这种意外情况，我们可以：

```
for (const auto& p : m){
  // as before
}
```

当然，auto也不是完美的，有的时候类型推断会得到意料之外的结果，可以看 Section 9.2,10.2. 另外，在编程中，有时如果为了可读性更好，也可以少使用一点auto。

10.2 当 auto 推断出意料之外的类型时请使用显式类型初始化

有些时候auto类型推导与你期望的结果不同，例如：如果我有一个输入Widget输出std::vector<bool>函数：

```
std::vector<bool> features(const Widget& w);
Widget w;
bool highPriority = features(w)[5];
processWidget(w, highPriority);
```

这样看起来没有问题，但如果我们换成auto类型声明呢?

```
auto highPriority = features(w)[5];
processWidget(w, highPriority); // undefined behavior!
```

此时highPriority的类型不再是bool而是 std::vector<bool>::reference了。因为 std::vector<b 不是容器，而是一个压缩形式的bool，一个字节表示一个bool，然而 C++ 不允许引用一个字节，因此有了 std::vector<bool>::reference像一个bool&，为了表现像一个bool&，它存在向bool的隐式转换。

此时highPriority指向了一个临时变量所占据的一个字节，在语句过后，features(w)这个临时变量销毁，highPriority相当于变成了一个悬挂指针，因此后续会出现 undefined behavior。

std::vector<bool>::reference是一种**代理类**：为了模拟和增强某些其他类型的行为而存在的类。代理类被用于多种用途：

- std::vector<bool>::reference是用于提供 std::vector<bool>的operator[]操作返回一个指向字节的引用的幻觉的。

- 智能指针用于资源管理。
- 表达式模板。

11 Modern C++

11.1 构造 object 时注意区分 () 和 {}

初始化分为三种：括号、等号或者大括号，有时候还可能等号和大括号同时出现：

```
int x(0); // initializer is in parentheses
int y = 0; // initializer follows "="
int z{ 0 }; // initializer is in braces
```

对于用户定义的类而言，区分赋值和初始化是很重要的，因为会调用不同的函数：

```
Widget w1; // call default constructor
Widget w2 = w1; // not an assignment; calls copy ctor
w1 = w2; // an assignment; calls copy operator=
```

C++98 无法直接初始化一个具有特定值的 STL 容器，但是 C++11 有了通用初始化 (uniform initialization) 就可以了：

```
std::vector<int> v{ 1, 3, 5 };
```

大括号也可以用来为 non-static 数据成员提供默认初始值：

```
class Widget {
private:
  int x{ 0 }; // fine, x's default value is 0
  int y = 0; // also fine
  int z(0); // error!
};
```

不可拷贝的对象（例如`std::atomics`，见 Section 11.4）也可以使用大括号或者括号来初始化，但不能用等号：

```
std::atomic<int> ai1{ 0 }; // fine
std::atomic<int> ai2(0);   // fine
std::atomic<int> ai3 = 0;  // error!
```

另外，大括号初始化的一个新颖的特点就是**它禁止 built-in 类型的隐式 narrowing 转换**：

```
double x, y, z;
int sum1{ x + y + z }; // error! sum of doubles may not be expressible
as int
int sum2(x + y + z); // okay (value of expression truncated to an int)
int sum3 = x + y;  // ditto
```

大括号初始化的另一个值得注意的特性是它不受 C++ 最令人烦恼的解析的影响。C++ 规则的一个副作用是，**任何可以解析为声明的东西都必须被解释为声明**，当开发人员想要默认构造一个对象，但无意中最终声明了一个函数时，最令人烦恼的解析最常困扰着他们。问题的根源在于，如果你想调用带有参数的构造函数，你可以这样做：

```
Widget w1(10); // call Widget ctor with argument 10
Widget w2(); // most vexing parse! declares a function named w2 that returns a Widget!
Widget w3{}; // calls Widget ctor with no args
```

当然大括号初始化也有缺点，在于有的时候会出现令人惊讶的行为，例如 Section 9.2中auto类型推导 std::initializer_lists，另外，类如果存在以 std::initializer_lists为参数的构造函数，它永远是更高优先级的：

```
class Widget {
public:
  Widget(int i, bool b);
  Widget(int i, double d);
  Widget(std::initializer_list<long double> il);
};

Widget w1(10, true); // calls first ctor
Widget w2{10, true}; // uses braces, but now calls std::initializer_list ctor
// (10 and true convert to long double)
Widget w3(10, 5.0); // calls second ctor
Widget w4{10, 5.0}; // uses braces, but now calls
std::initializer_list ctor
// (10 and 5.0 convert to long double)
```

甚至说拷贝构造和移动构造也会被它拦截：

```
class Widget {
public:
  ...
  operator float() const; // convert to float
};
Widget w5(w4); // uses parens, calls copy ctor
Widget w6{w4}; // uses braces, calls std::initializer_list ctor
(w4 converts to float, and float converts to long double)
Widget w7(std::move(w4)); // uses parens, calls move ctor
Widget w8{std::move(w4)}; // uses braces, calls std::initializer_list ctor
```

如果出现 narrowing conversions 会直接报错，而不是找其他匹配的构造函数：

```
class Widget {
public:
  Widget(int i, bool b); // as before
  Widget(int i, double d); // as before
  Widget(std::initializer_list<bool> il);
};
Widget w{10, 5.0}; // error! requires narrowing conversions
```

如果无法与std::initialize_list的参数相转化，才会去找其他匹配的构造函数：

```
class Widget {
public:
  Widget(int i, bool b); // as before
  Widget(int i, double d); // as before
  Widget(std::initializer_list<std::string> il);
};
Widget w1(10, true); // uses parens, still calls first ctor
Widget w2{10, true}; // uses braces, now calls first ctor
Widget w3(10, 5.0); // uses parens, still calls second ctor
Widget w4{10, 5.0}; // uses braces, now calls second ctor
```

这里还有有趣的一点，如果使用空的大括号，则默认调用默认构造函数，如果想调用空的std::initializer_list就要用两个大括号：

```
Widget w1; // calls default ctor
```

```
Widget w2{}; // also calls default ctor
Widget w3(); // most vexing parse! declares a function
Widget w4({}); // calls std::initializer_list ctor with empty list
Widget w5{{}}; // ditto
```

所以要注意区分括号和大括号初始化，典型的例子就是std::vector：

```
std::vector<int> v1(10, 20); // 10-element std::vector, all elements 20
std::vector<int> v2{10, 20}; // 2-element std::vector, element values are 10 and 20
```

尤其是在写模板的时候，对于内部的 object 要考虑好要用什么初始化，这是件很困难的事情。

11.2 倾向于使用 nullptr 而不是 0 或者 NULL

如果 C++ 发现自己在只能使用指针的上下文中看到 0，它会不情愿地将 0 解释为空指针，NULL也是一个整型，同理，如果遇到 f 重载了指针类型和整型，就会出现错误：

```
void f(int); // three overloads of f
void f(bool);
void f(void*);
f(0); // calls f(int), not f(void*)
f(NULL); // might not compile, but typically calls f(int). Never calls f(void*)
```

而nullptr类型为std::nullptr_t，它可以隐式转换为任何指针类型，它绝对不会转换为整型。它还会让语义更好理解，例如：

```
auto result = findRecord( /* arguments */ );
if (result == 0) {
  ...
}

if (result == nullptr) {
  ...
}
```

当我们引入模板的时候nullptr表现得更好：

```
int f1(std::shared_ptr<Widget> spw); // call these only when
double f2(std::unique_ptr<Widget> upw); // the appropriate
```

```
bool f3(Widget* pw); // mutex is locked

std::mutex f1m, f2m, f3m; // mutexes for f1, f2, and f3
using MuxGuard = std::lock_guard<std::mutex>;

{
  MuxGuard g(f1m); // lock mutex for f1
  auto result = f1(0); // pass 0 as null ptr to f1
} // unlock mutex

{
  MuxGuard g(f2m); // lock mutex for f2
 auto result = f2(NULL); // pass NULL as null ptr to f2
} // unlock mutex

{
  MuxGuard g(f3m); // lock mutex for f3
  auto result = f3(nullptr); // pass nullptr as null ptr to f3
} // unlock mutex
```

可以看到前两个是可以运行的，如果用nullptr替换是通过不了的，我们希望只有输入正确的指针类型才可以运行，所以我们可以使用模板：

```
template<typename FuncType,
typename MuxType,
typename PtrType>
decltype(auto) lockAndCall(FuncType func, MuxType& mutex, PtrType ptr){
  MuxGuard g(mutex);
  return func(ptr);
}

auto result1 = lockAndCall(f1, f1m, 0);    // error!
auto result2 = lockAndCall(f2, f2m, NULL); // error!
auto result3 = lockAndCall(f3, f3m, nullptr); // fine
```

11.3 相对于 typedef 更偏好 alias 声明

在很多时候 alias 声明和typedef做的事情相同，例如：

```
typedef void (*FP)(int, const std::string&);
using FP = void (*)(int, const std::string&);
```

但是当模板引入后就不一样了，因为 alias 声明可以模板化，而 typedef只能嵌套在模板化 struct 中：

```
template<typename T> // MyAllocList<T>
using MyAllocList = std::list<T, MyAlloc<T>>;

template<typename T>
class Widget {
private:
  MyAllocList<T> list; // no "typename",
};



template<typename T>
struct MyAllocList {
  typedef std::list<T, MyAlloc<T>> type;
};
template<typename T>
class Widget { // Widget<T> contains a MyAllocList<T>
private:
  typename MyAllocList<T>::type list; // typename
};
```

编译器知道第一个MyAllocList<T>一定是一个类型,但是不知道 MyAllocList<T>::type是类型还是MyAllocList的某个特化中的其他东西，例如，它认为可能会有这种可能：

```
template<>
class MyAllocList<Wine> {
private:
  enum class WineType{ White, Red, Rose };
```

```
  WineType type; // in this class, type is a data member!
};
```

如果您完成过任何模板元编程 (TMP)，那么您几乎肯定会遇到需要获取模板类型参数并从中创建修改后的类型的需要。例如，给定某种类型T，您可能希望删除T包含的任何常量引用限定符，例如，您可能希望将 const std::string&转换为std::string等等。

C++11 在<type_traits中提供了这样一些 type trait:

```
std::remove_const<T>::type      // yields T from const T
std::remove_reference<T>::type // yields T from T& and T&&
std::add_lvalue_reference<T>::type // yields T& from T
```

而在 C++14 中对于每一个这样的操作都使用了 alias 声明:

```
std::remove_const_t<T>          // yields T from const T
std::remove_reference_t<T>      // yields T from T& and T&&
std::add_lvalue_reference_t<T> // yields T& from T
```

实际上实现就是:

```
template <class T>
using remove_const_t = typename remove_const<T>::type; // typename
```

11.4 相比于无作用域的 enums 更偏向于有作用域的 enums

在 C++98 中,枚举器中的名称包含在包含enum的作用域中,称为无作用域的enum(unscoped),因此不能有相同的名称在同一个作用域中:

```
enum Color { black, white, red };
auto white = false; // error!
```

- C++11 添加了 scoped enum，使得**名字不会泄露**:

  ```
  enum class Color { black, white, red };
  auto white = false; // fine, no other
  Color c = white; // error! no enumerator named
  // "white" is in this scope
  Color c = Color::white; // fine
  auto c = Color::white; // also fine
  ```

- 另外，scoped enum为强类型枚举，而 unscoped enum 可以隐式转换为整型，**有的时候会造成语义曲解**：

```
enum Color { black, white, red };
std::vector<std::size_t> primeFactors(std::size_t x);
Color c = red;
if (c < 14.5) { // compare Color to double (!)
  auto factors = primeFactors(c); // compute prime factors of a Color (!)
}
```

```
enum class Color { black, white, red };
std::vector<std::size_t> primeFactors(std::size_t x);
Color c = red;
if (c < 14.5) { // error! cannot compare Color and double
  auto factors = primeFactors(c); // error!
}
```

如果你真的想这么做的话，需要：

```
if (static_cast<double>(c) < 14.5) {
  auto factors = primeFactors(static_cast<std::size_t>(c));
}
```

- scoped enum 第三个优点是可以**前置声明**：

```
enum Color; // error!
enum class Color; // fine
```

这样做的目的是出于每个enum都有一个依赖的整型，这取决于编译器，例如：

```
enum Color { black, white, red };
```

编译器可能会选择char，因为这三个都可以这样表示，然而当取值范围很大的时候，例如：

```
enum Status { good = 0, failed = 1, incomplete = 100,
 corrupt = 200, indeterminate = 0xFFFFFFFF};
```

这时候编译器就会选一个很大的整型，然而如果我们如果再加一个 Status 的话，**编译器就需要对整个系统重新进行编译！**然而引入 scoped enum前置声明就可以解决这个问题：

```
enum class Status; // forward declaration
void continueProcessing(Status s); // use of fwd-declared enum
```

编译器默认依赖类型是int，也可以做修改：

```
enum class Status: std::uint32_t;
```

但有一个情况是需要用 unscoped enum的，那就是与 std::tuple相关的时候：

```
// name email reputation
using UserInfo = std::tuple<std::string,std::string,std::size_t> ;
UserInfo uInfo; // object of tuple type
auto val = std::get<1>(uInfo); // get value of field 1
```

这样看起来是很不舒服的，所以可以：

```
enum UserInfoFields { uiName, uiEmail, uiReputation };
UserInfo uInfo;
auto val = std::get<uiEmail>(uInfo);
```

但是如果使用 scoped enum的话看起来会很冗长：

```
enum class UserInfoFields { uiName, uiEmail, uiReputation };
UserInfo uInfo;
auto val = std::get<static_cast<std::size_t>(UserInfoFields::uiEmail)>(uInfo);
```

如果想做简化的话就饿可以写一个函数将枚举器转化为它的依赖整型，并且保证在编译期就可以使用，那它必须是constexpr函数 (见 Section 11.9)，另外，我们也要声明它为nonexcept的，因为我们知道它肯定不会出现异常。

```
template<typename E> // C++14
constexpr auto toUType(E enumerator) noexcept{
  return static_cast<std::underlying_type_t<E>>(enumerator);
}
auto val = std::get<toUType(UserInfoFields::uiEmail)>(uInfo);
```

11.5 相比于私有化不定义函数更倾向于 deelted 函数

在 C++98 中如果相拒绝用户调用类的拷贝构造或者拷贝赋值，一般采用 private化，在 C++11 中开始采用delete修饰符，这样做的好处有：

- 即便是类内部或者友元函数想要调用拷贝构造也会报错，
- 习惯上deleted函数被声明为public，因为 C++ 会优先检查可调性（accessibility），如果声明为private，可能只会报错说调试图调用private函数，而忽略了重点。
- deleted函数还有一个重要优势在于**任何函数都可以是 deleted的**：

```
bool isLucky(int number);      // original function
bool isLucky(char) = delete;   // reject chars
bool isLucky(bool) = delete;   // reject bools
bool isLucky(double) = delete; // reject doubles and floats
// C++ prefers converting float to double rather than int
```

- deleted函数的另一个小技巧是避免使用本该不存在的模板实例化：有两种特殊指针，一个是void*，因为它无法解引用也没办法进行加减；另一个是char*，因为它通常表示指针指向一个 C 风格的字符串而不是单独的一个字符。

```
template<typename T>
void processPointer(T* ptr);
template<>
void processPointer<void>(void*) = delete;
template<>
void processPointer<char>(char*) = delete;
```

然而要注意的是，如果在一个 class 内定义一个函数模板，如果你想避免某些特例化，必须在 namespace scope 下定义特例化deleted函数才可以，因为模板特例化都必须写在 namespace scope 中：

```
class Widget {
public:
  template<typename T>
  void processPointer(T* ptr){ ... }
private:
```

```
    template<> // error!
    void processPointer<void>(void*);
};
template<>
void Widget::processPointer<void>(void*) = delete;
```

11.6 给每个重写函数声明 override

重写（override）和重载（overload）没有任何关系：

```
class Base {
public:
  virtual void doWork();
};
class Derived: public Base {
public:
  virtual void doWork(); // override
};
std::unique_ptr<Base> upb = std::make_unique<Derived>();
upb->doWork(); // derived class function is invoked
```

重写必须要满足：

- 基类函数必须是虚函数，
- 名称必须相同，除非是析构函数，
- 参数类型必须一致，
- 函数的 const 性必须一致，
- 返回类型和异常修饰符必须是相容的，
- C++11 中还加入了函数的引用修饰符必须是一致的：

```
class Widget {
public:
  void doWork() &;  // only when *this is an lvalue
  void doWork() &&; // only when *this is an rvalue
```

```
};
Widget makeWidget(); // factory function (returns rvalue)
Widget w;
w.doWork(); // calls Widget::doWork for lvalues
makeWidget().doWork(); // calls Widget::doWork for rvalues
```

下面四种函数都可以编译通过，但肯定不是你想要的：

```
class Base {
public:
  virtual void mf1() const;
  virtual void mf2(int x);
  virtual void mf3() &;
  void mf4() const;
};
class Derived: public Base {
public:
  virtual void mf1();                // const
  virtual void mf2(unsigned int x); // paramType
  virtual void mf3() &&;             // reference qualifier
  void mf4() const;                  // no virtual
};
```

C++ 中引入了override修饰词，如果都加上的话，程序是无法编译的，这样就能确保你声明不会出错，这样做比靠编译器提醒你靠谱得多，如果您正在考虑更改基类中虚函数的 signature，它还可以帮助您衡量后果。

关于引用修饰符补充一点它使用的地方：

```
class Widget {
public:
  using DataType = std::vector<double>;
  DataType& data() { return values; }
private:
  DataType values;
};
auto vals1 = w.data();
```

```
auto vals2 = makeWidget().data();
```

可以看到第二个拷贝构造是不必要的，我们希望使用移动拷贝构造，所以有：

```
class Widget {
public:
  using DataType = std::vector<double>;
  DataType& data() & { return values; } // return lvalue
  DataType data() && { return std::move(values); } // return rvalue
private:
  DataType values;
};
```

11.7 相对于 iterators 更偏好 const_iterator

const_iterator就是无法改变其所指向的东西的迭代器，C++14 中提供了非成员函数来获取迭代器，例如std::cbegin, std::rbegin等等，但是 C++11 中只提供了std::begin和std::end。在编写更通用的 code 时，更偏好于这些非成员函数：

```
template<typename C, typename V>
void findAndInsert(C& container, const V& targetVal, const V& insertVal) {
  using std::cbegin;
  using std::cend;
  auto it = std::find(cbegin(container), cend(container), targetVal);
  container.insert(it, insertVal);
}

// C++11 cbegin
template <class C>
auto cbegin(const C& container)->decltype(std::begin(container)){
  return std::begin(container);
}
```

对于一个const容器，只会返回const_iterator。

11.8 如果不会产生异常就将函数声明为 noexcept

C++11 开始关于异常，一个函数只有两种情况：可能有异常、不存在异常。一个函数是否声明为noexcept和是否声明为const一样重要，因为它能让编译器产生更好的代码：

如果在运行时，异常离开 f，则违反了 f 的异常规范。使用 C++98 异常规范，调用堆栈将展开到 f 的调用者，并且在执行一些与此处不相关的操作后，程序执行将终止。对于 C++11 异常规范，运行时行为略有不同：堆栈仅可能在程序执行终止之前展开。展开调用堆栈和可能展开调用堆栈之间的差异对代码生成有着惊人的巨大影响。在 noexcept 函数中，如果异常将传播到函数之外，优化器不需要将运行时堆栈保持在不可展开状态，也不必确保 noexcept 函数中的对象在异常时以与构造相反的顺序销毁。

一个典型的例子就是移动操作：

```
std::vector<Widget> vw;
Widget w;
vw.push_back(w);
```

在 C++98 中如果 vector 的 size 等于 capacity 时，会分配更大的内存然后对每个元素进行拷贝，这样有很强的异常安全性，但 C++11 中如果直接将拷贝换为移动的话，如果移动某个元素发生了异常，是无法复原的，因此违背了异常安全性，所以 C++11 中采取的措施是确保移动不出现异常才会用移动构造，否则就用拷贝构造，判断的依据就是move操作是否为noexcept的。

另一个典型例子就是 STL 算法库中的swap函数，它是否为 noexcept取决于用户定义的swap是否为 nonexcept:

```
template <class T, size_t N>
void swap(T (&a)[N], T (&b)[N]) noexcept(noexcept(swap(*a, *b)));

template <class T1, class T2>
struct pair {
void swap(pair& p) noexcept(noexcept(swap(first, p.first))
                && noexcept(swap(second, p.second)));
};
```

这些函数叫做条件性noexcept。

在 C++11 中所有 memory deallocation 函数和所有析构函数都是隐式 noexcept的

11.9 尽可能用使用 constexpr

constexpr修饰一个 object 的时候意味着它是常量并且编译期间可知，例如数组的尺寸，模板参数，枚举值都是编译期间可知的，但是 constexpr function会更有趣一点：如果它用 compile-time 常量调用的时候就会返回 compile-time 常量，否则就会返回 runtime 值，这样很好的避免了 duplication：

```
constexprint pow(int base, int exp) noexcept { ... }
constexpr auto numConds = 5;
std::array<int, pow(3, numConds)> results;
auto baseToExp = pow(base, exp); // call pow function at runtime
```

在 C++11 中constexpr函数最多只能有一句 return，但 C++14 放宽了要求：

```
constexpr int pow(int base, int exp) noexcept{
  return (exp == 0 ? 1 : base * pow(base, exp - 1));
}
constexpr int pow(int base, int exp) noexcept{
  auto result = 1;
  for (int i = 0; i < exp; ++i) result *= base;
  return result;
} // C++14
```

如果有一个类满足它的构造函数和其他成员函数都是constexpr的，例如：

```
class Point {
public:
  constexpr Point(double xVal = 0, double yVal = 0) noexcept: x(xVal), y(yVal){}
  constexpr double xValue() const noexcept { return x; }
  constexpr double yValue() const noexcept { return y; }
  // C++14
  constexpr void setX(double newX) noexcept { x = newX; }
  constexpr void setY(double newY) noexcept { y = newY; }
private:
  double x, y;
};
```

那么这个类的 object 也可以是constexpr的。

11.10 让 const 成员函数是线程安全的

```
class Polynomial {
public:
  using RootsType = std::vector<double>;
  RootsType roots() const{
    if (!rootsAreValid) {
      ...
      rootsAreValid = true;
    }
    return rootVals;
  }
private:
  mutable bool rootsAreValid{ false };
  mutable RootsType rootVals{}; // on initializers
};
```

当有两个线程求根的时候,这样做是线程不安全的,最简单的解决办法就是加入一个mutex:

```
class Polynomial {
public:
  RootsType roots() const{
    std::lock_guard<std::mutex> g(m); // lock mutex
    ...
    return rootVals;
  } // unlock mutex
private:
  mutable std::mutex m;
};
```

这里值得一提的是**std::mutex只能进行移动，因此这样做的话会导致Polynomial类不再能进行拷贝，只能进行移动**。

很多时候，我们也可以采用更经济的方法std::atomic（一个其他线程保证看到其操作不可分割地发生的类），例如我们有时候想看一个函数被调用了多少次：

```
class Point {
public:
```

```
  double distanceFromOrigin() const noexcept {
    ++callCount; // atomic increment
    return std::sqrt((x * x) + (y * y));
  }
private:
  mutable std::atomic<unsigned> callCount{ 0 };
  double x, y;
};
```

当然，std::atomic也是只能移动的类。

```
class Widget {
public:
int magicValue() const{
  if (cacheValid) return cachedValue;
  else {
    auto val1 = expensiveComputation1();
    auto val2 = expensiveComputation2();
    cachedValue = val1 + val2; // uh oh, part 1
    cacheValid = true; // uh oh, part 2
    return cachedValue;
  }
}
private:
  mutable std::atomic<bool> cacheValid{ false };
  mutable std::atomic<int> cachedValue;
};
```

这时候虽然可以运行，但是效率上有一些问题：当第二个线程调用 magicValue的时候，看到cacheValid是false，它就会等到第一个线程结束的时候立刻再去跑一遍。

这个时候我们只需要调换一下 part 1 和 2 的顺序就可以，但是如果更多的变量和逻辑加入进来该怎么办呢?

所以这里的经验是如果只有单个变量或者内存需要同步，就使用 std::atomic，否则用mutex。

11.11 了解特殊的成员函数生成

special member functions 指的是 C++ 愿意自己创造的函数：默认构造函数、析构函数、拷贝构造、拷贝赋值、移动构造、移动赋值函数。

移动构造和移动赋值和其他的行为类型：它们会在被需要的时候产生，默认移动构造会移动构造每个非静态成员（包括基类部分的），移动赋值同理。这里要注意一点的是，memberwise 移动是指那些支持移动操作的数据成员进行移动，否则就进行拷贝，Section **??**会详细来讲这里的细节。

与拷贝操作不同的是，拷贝构造和拷贝赋值是独立的，声明一个不会影响编译器默认生成另一个。然而**移动操作却不是独立的，声明了一个会阻止编译器默认生成另一个**。

另外，**如果类已经明确声明了一个拷贝操作，那么默认移动操作就不会生成**。另一个方向也是成立的，**如果类已经明确声明了一个移动操作，那么默认拷贝操作就不会生成（delete）**。

The Rule of Three 规定，如果声明复制构造函数、复制赋值运算符或析构函数中的任何一个，则应声明所有这三个函数。它源于这样的观察：接管复制操作的含义的需要几乎总是源于执行某种资源管理的类，并且这几乎总是意味着

- 在一个复制操作中完成的任何资源管理可能需要在另一复制操作中完成，
- 类析构函数也将参与资源的管理（通常是释放它）。要管理的经典资源是内存，这就是为什么所有管理内存的标准库类（例如，执行动态内存管理的 STL 容器）都声明“三巨头”：复制操作和析构函数。

C++11 中如果用户定义了析构函数或者拷贝操作，就不会生成默认的移动操作，因此合成移动操作只有满足三个条件时才会产生：

- 类中没有声明拷贝操作，
- 没有声明移动操作，
- 没有声明析构函数。

同样的道理也适用于拷贝操作，如果想用默认的操作，可以添加后缀 `=default`.

另外，请注意，规则中没有关于成员函数模板的存在阻止编译器生成特殊成员函数的内容：

```
class Widget {
  template<typename T> // construct Widget from anything
  Widget(const T& rhs);
  template<typename T>
```

```
  Widget& operator=(const T& rhs);
};
```

编译器仍然会自动生成拷贝操作和移动操作，在 Section **??**会将其中的原因。

12 Appendix

12.1 三五法则

- 需要析构函数的类也需要拷贝构造函数和拷贝赋值函数。
- 需要拷贝操作的类也需要赋值操作，反之亦然。需要拷贝操作代表这个类在拷贝时需要进行一些额外的操作。赋值操作 = 先析构 + 拷贝，所以拷贝需要的赋值也需要。反之亦然。
- 析构函数不能是私有的或删除的，否则成员便无法销毁。
- 如果一个类有私有的或不可访问的析构函数，那么其默认和拷贝构造函数会被定义为删除的。
- 如果一个类有const或引用成员，则不能使用默认的拷贝赋值操作。

12.2 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。

12.3 万能引用

首先理解引用折叠，引用折叠会发生在模板实例化、auto类型推导、创建和运用typedef和别名声明、以及decltype语境中。

引用折叠规则：

- 所有右值引用折叠到右值引用上仍然是一个右值引用。如X&& &&折叠为X&&。
- 所有的其他引用类型之间的折叠都将变成左值引用。如X& &, X& &&, X&& &折叠为X&。可见左值引用会传染，沾上一个左值引用就变左值引用了。根本原因：在一处声明为左值，就说明该对象为持久对象，编译器就必须保证此对象可靠（左值）。
- 当万能引用（T&& param）绑定到左值时，由于万能引用也是一个引用，而左值只能绑定到左值引用。因此，T会被推导为 T&类型。从而param的类型为T& &&，引用折叠后的类型为T&。
- 当万能引用（T&& param）绑定到右值时，同理，右值只能绑定到右值引用上，故T会被推导为T类型。从而 param 的类型就是T&&（右值引用）。

12.4 std::function

12.5 std::bind

12.6 表达式模板

12.7 重载、重写、重定义

- **重载**（overload）：函数名相同，函数的参数个数、参数类型或参数顺序三者中必须至少有一种不同。函数返回值的类型可以相同，也可以不相同。发生在一个类内部。const关键字可以用于对重载函数的区分。
- **重定义**：也叫做隐藏，子类重新定义父类中有相同名称的非虚函数（参数列表可以不同），指派生类的函数屏蔽了与其同名的基类函数。发生在继承中。
- **重写**（override）：也叫做覆盖，一般发生在子类和父类继承关系之间。子类重新定义父类中有相同名称和参数的虚函数。

重写需要注意：

- 被重写的函数不能是 static 的。必须是 virtual 的

- 重写函数必须有相同的类型，名称和参数列表
- 重写函数的访问修饰符可以不同。尽管 virtual 是 private 的，派生类中重写改写为 public,protected 也是可以的。

重定义规则如下：

- 如果派生类的函数和基类的函数同名，但是参数不同，此时，不管有无 virtual，基类的函数被隐藏。
- 如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有 vitual 关键字，此时，基类的函数被隐藏（如果相同有 virtual 就是重写覆盖了）。

重定义存在的意义是避免基类修改某些实现导致继承类正常运行的代码出现异常行为。
但是如果 upcast，即用基类指针指向继承类 object，就只能调用基类的函数。
如果想继续使用基类的函数，需要在继承类中加入 `using Base::f`。

```
class Base {
public:
  virtual int f() const {}
  virtual void f(string) const {}
  virtual void g() const {}
};

class Derived2 : public Base {
public:
  // Overriding a virtual function:
  // string version has been hidden!
  int f() const {
    cout << "Derived2::f()\n";
    return 2;
  }
};

class Derived3 : public Base {
public:
  // Cannot change return type:
```

```
  //void f() const{ cout << "Derived3::f()\n";}
};

class Derived4 : public Base {
public:
  // Change argument list:
  // two base::f have been hidden
  int f(int) const {
    cout << "Derived4::f()\n";
    return 4;
  }
};

Derived4 d4;
x = d4.f() // error!
Base *base = new Derived4;
base -> f(1) // error! Derived version is unavailable.
br -> f();
br -> f("string"); // Base version available.
```

12.8 virtual 实现原理

虚函数实现多态的机制，严格来说是动态多态，是在出现运行的时候实现的。

虚函数的实现原理：每个虚函数都会有一个与之对应的虚函数表，该虚函数表的实质是一个指针数组，存放的是每一个对象的虚函数入口地址。对于一个派生类来说，他会继承基类的虚函数表同时增加自己的虚函数入口地址，如果派生类重写了基类的虚函数的话，那么继承过来的虚函数入口地址将被派生类的重写虚函数入口地址替代。那么在程序运行时会发生动态绑定，将父类指针绑定到实例化的对象实现多态。

12.9 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。
-