常见类型	函数	面向对象	容器	模板与泛型编程	内存管理	其它
变量 字符串与数组	函数	类 重载运算与类型转换 继承体系	容器 容器适配器 泛型算法	模板与泛型编程	内存管理	输入输出

变量

1.类型

- 算术类型
    - 整形
        - 包括char和bool在内
    - 浮点型
        - 单精度
        - 双精度
        - 扩展精度
- 空类型（void）

使用建议：

- 使用int执行整数运算，超过范围用long long，因为long一般和int大小一样
- 浮点运算用double，float通常精度不够而且双精度和单精度的计算代价相差无几。long double提供的精度通常没必要，而且运算时的消耗也不容忽视

2.大小

- **字节**：内存可寻址的最小块，大多数计算机将内存中的每个字节与一个数字(地址)关联起来。C++中，一个字节要至少能容纳机器基本字符集中的字符；
- **字**：一般是32比特(4字节)或64比特(8字节)

在不同机器上有所差别，对于C++标准(P30)：

- 一个char的大小和机器字节一样；
- bool大小未定义；
- int至少和short一样大；
- long至少和int一样大；
- long long(C++11)至少和long一样大

3.signed与unsigned

> 除了bool和扩展字符型外，都可以分为signed和unsigned；char可以表现为signed char和unsigned char，具体由编译器决定；

- **unsigned减去一个数必须保证结果不能是一个负值，否则结果是取模后的值**（比如，很多字符串的长度为无符号型，在for循环非常容易出现 `str.length() - i >= 0` 这种表达，如果i比字符串长度大，那么就会引发错误）

- **signed会转化为unsigned（切勿混用signed和unsigned)**
- **溢出**
  - **赋给unsigned超过范围的值**：结果是初始值对无符号类型表示值总数取模后的余数
  - **赋给signed超过范围的值**：结果未定义，可能继续工作、崩溃、生成垃圾数据

4.类型转换

4.1 隐式转换与显式转换

- **隐式转换**
  - **整形的隐式转换**：多数表达式中，比int小的整形首先提升为较大整形
  - **数组转成指针**
  - **指针的转换**：0,nullptr转成任意指针，任意指针转void
  - **转换时机**：
    - 拷贝初始化
    - 算术或关系运算
    - 函数调用时
- **显示转换**
  - **命名强制类型转换** `cast-name<type>(expression)`
    - **static_cast**：只要不包含底层const，都可以使用。适合将较大算术类型转换成较小算术类型
    - **const_cast**：只能改变底层const，例如指向const的指针(指向的对象不一定是常量，但是无法通过指针修改)，如果指向的对象是常量，则这种转换在修改对象时，结果未定义
    - **reinterpret_cast**：通常为算术对象的位模式提供较低层次上的重新解释。如将int*转换成char*。很危险！
    - **dynamic_cast**：一种动态类型识别。转换的目标类型，即type，是指针或者左右值引用，主要用于基类指针转换成派生类类型的指针(或引用)，通常需要知道转换源和转换目标的类型。如果转换失败，返回0（转换目标类型为指针类型时）或抛出bad_cast异常（转换目标类型为引用类型时）

- **旧式强制类型转换** `type (expr)` 或 `(type) expr`

> 旧式强制类型转换与const_cast，static_cast，reinterpret_cast拥有相似行为，如果换成const_cast，static_cast也合法，则其行为与对应命名转换一致。不合法，则执行与reinterpret_cast类似的行为

4.2 算术转换

- 既有浮点型也有整形时，整形将转换成相应浮点型
- 整形提升：bool,char,signed char,unsigned char,short,unsigned short所有可能值能存于int则提升为int，否则提升为unsigned int
- signed类型相同则转换成相同signed类型中的较大类型
- unsigned类型大于等于signed类型时，signed转换成unsigned
- unsigned类型小于signed类型时：
  - 如果unsigned类型所有值能存在signed类型中，则转换成signed类型
  - 如果不能，则signed类型转换成unsigned类型

5.初始化与赋值

> 很多语言中二者的区别几乎可以忽略，即使在C++中有时这种区别也无关紧要，所以特别容易把二者混为一谈

C++中初始化和赋值是2个完全不同的操作

- **显示初始化**：创建变量时的赋值行为
  - **拷贝初始化**： `int a = 0;`
  - **直接初始化**： `int a(0);`
  - **初始值列表**： `int a = {0};` 或 `int a{0};`
- **默认初始化**（程序）
  - 局部变量

        - non-static：内置类型非静态局部变量不会执行默认初始化
        - static：如果没有初始值则使用值初始化
    - 全局变量：内置类型全局变量初始化为0
- **值初始化**
    - 内置类型的值初始化为0
    - `container<T> c(n)` 只指定了容器的大小，未指定初始值，此时容器内的元素进行值初始化
    - 使用初始值列表时，未提供的元素会进行值初始化
    - 静态局部变量会使用值初始化

6.声明与定义

- 声明：
    - `extern 类型 变量名字;`
- 声明 + 定义：
    - `类型 变量名字;`
    - `extern 类型 变量名字 = 值;`（如果在函数内则会报错）

声明不会分配存储空间，定义会分配存储空间

7.作用域

访问被同名局部变量覆盖的全局变量：`::变量名`（不管有多少层覆盖，都是访问全局）

8.复合类型

8.1 引用

- **本质**：引用并非对象，它只是为对象起了另一个名字
- **形式**： `int &a = b;`

理解与使用：

- 非常量引用不能绑定到字面值或表达式的计算结果
- 一般来说，引用类型和绑定的对象类型需严格匹配
- 程序把引用和其初始值绑定到一起（对引用的操作都在其绑定的对象上进行）因此一旦初始化完成，无法另引用重新绑定到另外一个对象。因此必须初始化
- 引用本身并非对象，故不能定义引用的引用

8.2 指针

- 指针不同于引用，指针本身就是一个对象
- 因为引用不是对象，没有实际地址，所以不能定义指向引用的指针
- 指针是一个对象，所以存在对指针的引用
- 一般来说，指针类型和指向的对象类型也需严格匹配
- 编译器并不负责检查试图拷贝或以其它方式访问无效指针
- 和试图使用未经初始化的变量一样，使用未经无效指针的后果无法估计
- **空指针**：不指向任何对象（不要混淆空指针和**空类型(void)**的指针）
  - `int *p1 = nullptr; (C++11)`
  - `int *p2 = 0;`
  - `int *p3 = NULL; //#include cstdlib`
- 把int变量直接赋给指针是错误的，即使变量的值恰好等于0
- **空类型(void)**指针用于存放任意对象的地址

8.3 复合类型的声明

1）非数组与复合类型的声明

从右到左分析

```
int *&r = p; //r是一个引用，引用一个int指针p
```

变量的定义包括一个基本数据类型和一组声明符。同一条语句中，虽然基本数据类型只有一个，但是声明的形式却可以不同：

```
int* p1, p2; //p1是一个int*，p2是一个int
```

2）数组与复合类型的复杂申明

从数组名字开始，由内到外分析（数组的维度紧跟着被声明的名字，所以由内到外阅读比从右到左好多了）

- 数组与指针的复杂申明

  ```
  int (*Parray)[10] = &arr; //Parry是一个指针，指向一个含有10个int的数组
  ```

- 数组与引用的复杂申明

  ```
  int (&arrRef)[10] = arr; //arrRef是一个引用，引用一个含有10个int的数组
  ```

- 数组与指针及引用的混合复杂申明

  ```
  int *(&arry)[10] = ptrs; //arry是一个引用，引用一个包含10个int指针的数组
  ```

9.const

1）const对象

- const对象必须初始化，因为创建后const对象的值就不能再改变，初始值可以是任意复杂的表达式

```
const int i = get_size(); //运行时初始化
const int j = 42;         //编译时初始化
```

- 只能在const类型的对象上执行不改变其内容的操作
- 当以编译时初始化的方式定义一个const对象时，编译器将在编译过程中把用到该对象的地方替换成对应值
- 默认状态下，const对象仅在文件内有效。多个文件的同名const对象等同于在不同文件中定义了独立的变量
- 要在多个文件之间共享同一个const对象，需在定义和声明时都加上extern

2）const的引用（常量引用）

- 不能修改所绑定的对象
- 和非常量引用不同，常量引用可以使用字面值或任意表达式作为初始值（原因:绑定了一个临时量常量）

3）指针与const

- **指向常量的指针**(并不一定要指向常量，只是为了说明无法修改所指的对象)

```
const int *a = &b;
```

- **const指针(常量指针)**：不能修改指针，将一直指向一个地址，因此必须初始化。但是指向的对象不是常量的话，可以修改指向的对象

```
int *const a = &b;
const double *const pip = &pi; //pip是一个常量指针，指向的对象是一个双精度浮点型常量
```

4）顶层const与底层const

- **顶层const**：无法修改指针本身（顶层是一种直接的关系）

```
const int ci = 123;
int *const a = &b;
```

- **底层const**：无法修改所指的对象（底层是一种间接的关系）
  - 用于声明引用的const都是底层const

10.constexpr与常量表达式

- **常量表达式**：在“编译过程”就能确定结果的表达式。
  - 包括：
    - 字面值
    - 常量表达式初始化的const对象
  - 以下不是常量表达式

```
int s = 123;
const int sz = get_size();
```

- **constexpr变量(C++11)**：变量声明为contexpr类型，编译器会检查变量的值是否是个常量表达式

```
constexpr int mf = 20          //20是常量表达式
constexpr int limit = mf + 1;  //mf + 1是常量表达式
const int sz = size();         //只有当size是一个constexpr函数时，声明才正确
```

- **constexpr函数**：这种函数足够简单以使编译时就可以计算其结果
- **字面值类型**：能使用constexpr声明的类型应该足够简单，称为字面值类型

- 包括
  - 算数类型
  - 引用 & 指针
    - constexpr的指针初始值必须是nullptr，0或存储于某个固定地址中的对象
    - 一般来说全局变量和静态局部变量的地址不变
    - constexpr指针，constexpr只对指针有效，与指针所指对象无关
    - `constexpr const int *p = &i //p`是常量指针，指向整形常量`i`
- 不包括
  - 自定义类型
  - I/O 库
  - string字符串

11.类型别名

1. typedef： `typedef double wages;`
2. using（C++11)： `using SI = Sales_item;`

const与指针的类型别名使用时，还原别名来理解const的限定是错误的

12.auto

- 编译器根据初始值判断变量类型
- 必须初始化
- 一条语句声明多个变量（只能有一个基本类型，const int和int不算1个类型）

```
auto i = 0, *p = &i;   //正确
auto sz = 0, pi = 3.14 //错误
```

- 初始值为引用时，类型为所引对象的类型
- auto一般会忽略掉顶层const，底层const会保留下来
- 如果希望判断出的auto是一个顶层const，在auto前加const
- 还可以将引用的类型设为auto，此时原来的初始化规则仍然适用

13.decltype

- 希望根据表达式判定变量类型，但不用表达式的值初始化变量
- `decltype(f()) sum = x;` `f()` 并不会被调用， `sum` 为 `f()` 的返回类型
- 引用从来都作为其所指对象的同义词出现，只有在decltype处是一个例外
- 如果表达式的结果对象能作为一条赋值语句的左值，则表达式将向decltype返回一个引用类型

```
decltype(*p) c; //错误，c是int &，必须初始化
```

- 变量加上括号后会被编译器视为一个表达式

```
decltype((i)) d; //错误，d是int &，必须初始化
```

字符串与数组

1.字符串

> 字符串也是一种顺序容器

```
#include<string>
using std::string
```

1.1 初始化

> 默认初始化为空串

- **拷贝初始化**
  - =
  - 允许使用以空字符结束的字符数组来初始化
- **直接初始化**
  - ()
  - 如果传入一个 char* ：
    - 1) 同时传入了长度，则拷贝 char* 指向字符数组的指定长度的字符
    - 2) 没有传入长度，则 char* 指向的字符数组必须以空字符结尾
  - 示例：
    - string s(c,n); //s包含n个字符c
    - string s(cp,n); //s是cp指向的数组(cp为char *)中前n个字符的拷贝
    - string s(s2,pos2); //s是string s2从下标pos2开始的字符的拷贝。若pos2>s2.size()，则行为未定义(会抛出out_of_range异常)
    - string s(s2,pos2,len2); //s是string s2从pos2下标开始，len2个字符的拷贝。若pos2>s2.size()，则行为未定义(会抛出out_of_range异常)。不管len2多长，至多拷贝到结尾

1.2 大小

返回类型： `size::size_type`

无符号类型，注意与带符号数的运算

- 判断是否为空串
  - `if(str.size() == 0)`
  - `if(str.empty())`
  - `if(str == "")`

1.3 常见操作

- **访问**
  - **遍历**
    - 不需修改： `for(auto c : s)` 或 `for(decltype(s.size()) i = 0; i < s.size( ); i++ )`
    - 需要修改： `for(auto &c : s)` 或 `for(decltype(s.size()) i = 0; i < s.size( ); i++ )`
  - **访问某个字符**
    - 下标运算符： `str[pos]` ，接收的参数类型为 `size::size_type` 。返回“引用”，所以可以修改。越界结果不可预知
    - `str.at(pos)` ：会检查下标 `pos` 是否有效
    - 迭代器
- **转化为字符数组**
  - `c_str()`
- **获得子串**
  - `s.substr(pos)` ：返回从pos开始的尾串。如果超出范围会抛出out_of_range异常
  - `s.substr(pos,n)` ：返回从pos开始，长度为n的子串。超出范围则返回剩余所有部分
- **修改**
  - **插入**
    - `s.append(str)` ：在字符串末尾插入str指向的字符串
    - `s.insert(pos,n,c)` ：在pos之前插入n个字符c
    - `s.insert(pos,cstr)` ：在pos之前插入字符指针cstr指向的字符串

        - `s.insert(pos1,s2,pos2,n)` ：在s的pos1位置插入s2从pos2开始的n个字符
    - **删除**
        - `s.erase(pos,n)` ：从pos位置开始，删除n个字符，若n过大，则删完从pos开始的剩余字符
    - **替换**
        - `s.replace(pos,n,str)` ：将pos位置开始的n个字符删除，然后在pos位置处插入str指向的字符串
- **搜索**
    - 搜索成功返回 `string::size_type` 类型的下标；搜索失败返回 `string::npos`
        - `string::npos` ：static变量， `const string::size_type` 类型，初始化为 `-1` 。由于是一个unsigned类型，因此这个初始值意味着 `npos` 等于任何string最大的可能大小
    - `s.find(args)` ：查找s中 `args` 第一次出现的位置
    - `s.rfind(args)` ：在s中查找args中任何一个字符最后一次出现的位置（反向查找）
    - `s.find_first_not_of(args)` ：在s中查找第一个不在args中的字符
    - `s.find_last_not_of(args)` ：在s中查找最后一个不在args中的字符（反向查找）
- **比较**
    - `s.compare(args)` ：可以传入字符串或字符指针，以及位置，长度等
- **数值转换**
    - **数值转字符串**
        - `to_string(val)` ：val可以是任何算术类型
    - **字符串转数字**（ `p` 是 `size_t` 类型变量，保存 `s` 中第一个非数值字符的下标，默认为 `0` ； `b` 表示转换所用的基数，默认为 `10` ）
        - **转成整形**
            - `stoi(s,p,b)`
            - `stol(s,p,b)`
            - `stoul(s,p,b)`
            - `stoll(s,p,b)`
            - `stoull(s,p,b)`
        - **转成浮点数**
            - `stof(s,p)`

- stod(s,p)
- stold(s,p)

2.数组

> 数组的元素为对象，不存在引用的数组（ `int &refs[10] = ...;` 错误）

2.1 初始化

- 默认情况下，数组的元素被默认初始化
- 字符数组可以使用字符串字面值初始化
- 不允许直接使用数组拷贝和赋值

2.2 大小

维度必须是一个常量表达式。类型定义为 `size_t` ，定义在头文件 `cstddef` 中

2.3 遍历

数组遍历可以用 `for(auto i : array)` ，但是对于指针不行，即 `array` 不能是指针

- 数组迭代器：可以通过如下调用获取数组迭代器，函数定义在 `iterator` 头文件
  - begin(array)
  - end(array)

2.4 auto与decltype

```
int ia[ ] = {0,1,2,3,4...9};
auto ia2(ia); //ia2是一个整形指针，指向ia的第一个元素
```

```
decltype(ia) ia3 = {0,1,2,3,4...9}  //decltype(ia)返回的类型是由10个整数构成的数组
```

2.5 多维数组

> 严格来说，C++中没有多维数组，通常所说的多维数组其实是数组的数组

1）初始化

```
int [2][3] = {{1,2,3},{4,5,6}};
int [2][3] = {1,2,3,4,5,6}; //和上面等价
```

从内存分布上来说连续，就像是一维数组，但是并不能用 `int*` 来遍历。因为每3个元素实际上是一个 `int[3]` 类型

4）遍历

如果使用 `for(:)` 形式遍历多维数组，除了最内层循环，其它层循环的控制变量都应该是引用类型

5）类型别名

```
using int_array = int[4];
typedef int int_array[4]; //等价的typedef声明
```

函数

1.函数参数

1.1 形参

1) 形参类型的选择

- **值**：对象越大，拷贝开销越大。同时，如I/O类型的一些类类型根本不支持拷贝指针
- **引用**：c++建议使用引用，只是定义了一个别名。开销很低，如果无需修改，使用 `const type &`

2) const形参与重载

实参初始化形参时，顶层const会被忽略

```
void fcn(const int i);
void fcn(int i);
```

所以在调用fcn时，会出错

3) 数组形参

尽管不能以值传递的方式传递数组，但是可以把形参写成类似数组的形式：

```
//下面三者等价，都是const int*传递数组指针
const int*;
const int[];
const int[10]
```

多维数组时，以指针的形式： `int (*matrix)[10]` 。如果以数组形式： `int matrix[][10]` 。第一个维会被忽略，实际上是指向10个int的数组

4) 函数指针作为形参

和数组类似，不能传值的方式传递函数，但是也可以把参数写成函数形式：

```
//以下两者等价
void useBigger(const string &s1,const string &s2,bool pf(const string &,const string &));
void useBigger(const string &s1,const string &s2,bool (*pf)(const string &,const string &));
```

第一个声明中的 pf 会被编译器转换为(指向)函数(的)指针；

5）可变形参

- **实参类型相同**
  - **inittializer_list**（一种模板类型头文件同名，对象元素永远是常量值，无法改变其中元素的值）
- **实参类型不同**
  - **可变参数模板**
  - **省略符**（省略符形参应该仅仅用于C和C++通用的类型，大多数类类型的对象在传递给省略符形参时都无法正确拷贝）

1.2 默认实参

> 为形参提供默认的实参，可以是字面值，变量或表达式

- 默认实参类型
  - **字面值**
  - **变量**：（局部变量不能作为默认实参。局部变量不会影响被覆盖的(作为默认实参的)全局变量）
  - **表达式**

不必所有形参都指定默认实参，但指定了默认实参后的形参都必须指定

如果调用函数时为默认实参指定值，默认实参会被覆盖。尽可能将默认的参数放在尾部，因为无法跳过前面的默认实参覆盖后面的

2.函数返回

2.1 状态码

> 头文件：

- EXIT_FAILURE
- EXIT_SUCCESS

2.2 数组与函数的指针

> 注意，是"(指向)数组(的)指针"，而不是(指向)数组元素(的)指针

不能返回数组，但是可以通过返回一个(指向)数组(的)指针来访问数组（代码）

不能返回函数，但是可以通过返回一个(指向)函数(的)指针来调用函数。返回类型必须是(指向)函数(的)指针（代码）

3.函数重载

> C中没有重载；main函数不能重载

3.1 判断标准

- **类型别名不算重载**
- **只有返回类型不同不算重载**
- **多指定了变量名也不算重载**
- **不同作用域无法重载**
- **顶层const不算重载**（申明不会报错，传入实参时，顶层const会被忽略，不知道调用哪个，所以会报错）
- **底层const算重载**（代码）

3.2 函数匹配步骤

> 编译器根据实参类型确定应该调用哪一个函数；不同作用域中无法重载，内层会覆盖外层同名函数；内层同名变量也会覆盖外层同名函数

1. **选出候选函数**
   - 函数名相同
   - 调用点可见
2. **选出可行函数**（没有可行函数则报无匹配错误）
   - 参数数量相等
   - 实参形参类型“匹配”
     - 类型相同
     - 能互相转换
3. **寻找最佳匹配**（没找到最佳匹配则报二义性调用错误）
   - 精确匹配
     - 类型完全相同
     - 实参从数组类型转换成相应指针类型
     - 向实参添加或删除“顶层”const
   - 不能精确匹配则涉及到实参向形参的转换（按如下级别转换）
     - const转换
     - 类型提升：较小整形会(忽略short)直接提升成int或更大整形
     - 算术类型转换：所有算术类型转换的级别一样：`void manip(long); void manip(float); manip(3.14);` 会报二义性错误
     - 类类型转换

重载函数的(函数)指针必须与重载函数中的某一个“精确匹配”

4.内联函数

> 内联只是向编译器发出一个请求，编译器可以选择忽略这个请求

5.constexpr函数

> 值能用于常量表达式的函数，但其返回值并不一定是常量表达式，当返回值是常量表达式时，可以用于常量表达式；被隐式声明为内联函数，编译时被替换成结果值；

- 参数类型必须是字面值类型
- 返回类型必须是字面值类型
- 函数体中必须只有一条return语句

6.函数指针

6.1 函数类型与函数指针

```
typedef bool pfT1(const string &s);//函数类型
using pfT2 = bool(*)(const string &s);//函数指针

//三个声明等价
bool (*pf1)(const string &s);
pfT1 *pf2;
pfT2 pf3;

//初始化
bool (*pf1)(const string &s) = ff;
```

6.2 如何赋值

当使用函数名为函数指针赋值时，函数自动转换成指针：

```
//两者等价
pf = lengthCompare;
pf = &lengthCompare;
```

可以使用nullptr与0赋值，这样的函数指针不执行任何函数对象

6.3 如何调用

```
//两者等价
pf(...);
(*pf)(...);
```

6.4 作为形参与返回值

- ”函数类型“的形参会被编译器转换为函数指针
- 如果返回是“函数类型”，编译器不会自动转换成指针类型，所以必要时要加 *

类

1.关键字

> 唯一的区别在于默认访问权限以及默认派生访问说明符不一样

- **struct**
  - 默认访问权限为public；
  - 默认派生访问说明符也为public；
- **class**
  - 默认访问权限为private；
  - 默认派生访问说明符也为private；

2.向前声明

`class Screen;` 只是声明了一个类类型，在定义前是一个不完全类型

3.组成

3.1 友元声明

友元声明允许其他类或函数访问类的非公有成员

- 声明只能出现在类内部（最好在类定义开始或结束前集中声明友元）
- 不受访问说明符的约束
- 友元的声明仅仅指定了访问的权限，而非一个通常意义上的函数声明，应该在类外对友元函数再一次声明
- 友元不具有传递性，也不能继承

非成员函数的友元声明：

```
friend Sales_data add(const Sales_data&,const Sales_data&);
```

类的友元声明：

```
friend class Window_mgr;
```

成员函数的友元申明必须满足如下结构：

1. 先声明友元中的类
2. 然后在当前类声明这个友元类的成员函数
3. 最后再对这个友元类的成员函数进行定义

3.2 访问说明符

> 指定类成员的访问权限，加强类的封装性

一个类可以包含0个或多个访问说明符，对于某个访问说明符能出现的次数也没有严格限制

有效范围直到出现下一个访问说明符或到达类的结尾处为止

第一个访问说明符前的成员都是默认访问权限（struct和class不同）

- public：整个程序内可被访问
- private：可以被类的成员函数访问，不能被类的使用者访问

3.3 类型别名成员

类型别名成员受访问说明符的控制

和成员变量不同，类型别名成员必须先申明再使用，因此通常出现在类开始的地方

```
typedef std::string::size_type pos;
```

3.4 静态成员

> 静态成员与类相关而不与各个对象相关。静态成员存在于任何对象之外，对象中不包含任何与静态数据成员有关的数据。不与任何对象绑定在一起，因此不包含this指针，也就不能声明成const。

静态数据成员可以是不完全类型，甚至可以就是它所属的类类型（非静态数据成员不行，只能声明成所属类类型的指针或引用）

1）访问方式

- 通过作用域运算符访问
- (仍可以)通过对象、引用或指针访问
- 成员函数不需要通过域运算符便可直接访问

2）定义

- **必须在类外定义和初始化每个静态成员**（因为静态数据成员不属于类的任何对象，所以并不是在创建类对象时被定义的，意味着它们不是由类的构造函数初始化的）
- **定义时不能重复使用 `static`**

可以为静态成员提供const整数类型的类内初始值，但要求静态成员必须是字面值常量类型的constexpr：

```
static constexpr int period = 30;
```

此时，如果在类外不会用到period，则可以不在类外定义period。否则，可以定义：

```
constexpr int Account::period; //如果提供了类内初始值，类外定义时不能再指定一个初始值
```

3.5 成员变量

用 `mutable` 修饰的成员是一个**可变数据成员**。可变数据成员不管是常量成员函数(以const结尾)还是非常量成员函数，都能修改其值。即使是类的常量对象，其可变数据成员也是能被修改的

3.6 成员函数

声明必须在类内，定义可以在类外，类内定义隐式为内联函数，类外定义时需要指定 `类名::` 。如果希望类外定义的函数也以为内联函数，可以显示的加上 `inline` 关键字（可以都加，但是最好只在外面定义时加）

this指针作为隐式参数传入。默认情况下，this是指向非常量对象的常量指针

1）构造函数

名字与类名相同，没有返回类型

默认实参： `Sales_data(string s = "") : bookNo(s) { };`

- 如果提供实参，则使用实参初始化bookNo，否则使用默认实参初始化
- 因此效果相当于，同时定义了几种构造函数
- 如果一个构造函数所有形参都指定了默认实参，则该构造函数实际上也相当于默认构造函数(无需任何实参)
- 静态成员可以作为默认实参

1.1 ）默认构造函数

无“需”实参的构造函数，如果定义了其它构造函数，则编译器不会再生成合成的默认构造函数

1.2 ）委托构造函数

使用类的其它构造函数执行自己的初始化过程：

```
Sales_data() : Sales_data("",0,0) { }
Sales_data(string s) : Sales_data(s,0,0) { }
Sales_data(istream &is) : Sales_data() {read(is,*this);}
```

1.3 ）拷贝构造函数

第一个参数是自身类类型的引用，且任何额外参数都有默认值：

```
class Foo {
public:
    Foo(const Foo&);
};
```

> 需要拷贝构造的类也需要赋值操作，反之亦然，但是并不意味着一定需要析构

拷贝构造函数很多情况下会被隐式使用，不应该使用 `explict` 修饰

合成拷贝构造函数：将参数的非静态成员逐个拷贝到正在创建的对象中

- 类类型的成员使用其拷贝构造函数
- 内置类型成员直接拷贝
- 数组成员逐元素拷贝，如元素是类类型则使用其拷贝构造函数

调用时机：

- = 号定义对象时
- **函数调用和函数返回时**：这也解释了为什么第一个参数必须是引用。如果不是引用，在函数调用实参拷贝给形参时，会调用拷贝构造函数构造形参。此时，拷贝构造函数中是一个非引用的形参，因此又会调用拷贝构造函数来构造这个形参...从而造成无限循环
  - 形参类型为非引用类类型的函数的调用时
  - 返回类型为非引用类类型的函数返回时

```
string dots(10,'.') //直接初始化，编译器使用普通的函数匹配；
string s2 = dots //拷贝初始化；
string null_book = "9-999-9" //拷贝初始化；
```

如果涉及隐式转换，如上面的null_book，编译器可以选择跳过拷贝初始化，将其改为直接初始化来构造null_book。这样的话不会调用拷贝构造函数，但是还是要求拷贝构造函数可见（代码）

1.4）移动构造函数

> 从给定对象"窃取"资源而不是拷贝资源

第一个参数是该类型的右值引用，任何额外参数都必须有默认实参：

```
//noexcept承诺这个函数不会抛出异常
StrVec::StrVec(StrVec &&s) noexcept : elements(s.elements) ...
```

不抛出异常的移动构造函数必须标记为 noexcept ：移动操作"窃取"资源，它通常不分配资源。因此，移动操作通常不会抛出任何异常

合成移动构造函数：

- 满足下列条件时，编译器才会合成移动构造函数（与移动赋值运算符相同）
  - 类没有自定义拷贝控制函数
    - 没有自定义"拷贝构造函数"

        - 没有自定义"拷贝赋值运算符"
        - 没有自定义"析构函数"
    - 同时类的每个非static数据成员都可以移动

2）赋值运算符

- 拷贝赋值运算符
- 移动赋值运算符

2.1）拷贝赋值运算符

```
Foo& operator= (const Foo&);
```

= 运算符左侧对象为隐式 *this ，右侧对象作为参数传入，返回左侧对象的引用

标准库要求容器中的类型要具有赋值运算符

> 需要赋值操作的类也需要拷贝构造，反之亦然。但是并不意味着一定需要析构

- **合成拷贝赋值运算符**
    - 对于某些类，合成拷贝赋值运算符运来阻止对象拷贝
    - 如果不是这种情况，会将右侧对象的每个非static成员赋予左侧运算对象的对应成员，通过成员类型的拷贝赋值运算符来完成。对于数组，逐个赋值
- **自定义拷贝赋值运算符**
    - 如果将一个对象赋予它自身（自赋值），赋值运算符必须能正确工作
    - 大多数赋值运算符组合了析构函数和拷贝构造函数的工作

行为像值的类

- 每个对象都有自己成员数据的拷贝，两者相互独立，改变原对象不会影响副本 代码
- 如过类实现了自定义的swap函数，则拷贝赋值运算符可以使用拷贝并交换技术 代码

行为像指针的类

- 对于指针类型的成员直接拷贝，指向相同动态内存 代码

2.2）移动赋值运算符

> 从给定对象"窃取"资源而不是拷贝资源

```
StrVec & StrVec::operator=(StrVec &&rhs) noexcept {...}
```

如果一个类定义了自己的"拷贝构造函数"、"拷贝赋值运算符"、"析构函数"，编译器就不会为它合成移动构造函数或移动赋值运算符

不抛出异常的移动赋值运算符必须标记为 `noexcept` ：移动操作"窃取"资源，它通常不分配资源。因此，移动操作通常不会抛出任何异常

合成移动赋值运算符：

- 满足下列条件时，编译器才会合成移动赋值运算符（与移动构造函数相同）
  - 没有自定义"拷贝构造函数"
  - 没有自定义"拷贝赋值运算符"
  - 没有自定义"析构函数"
- 同时类的每个非static数据成员都可以移动

3）析构函数

不接受参数，所以不能被重载，只有唯一一个

```
~Foo();
```

首先执行函数体，然后销毁成员（按初始化顺序的逆序销毁）：

- 类类型的成员调用其析构函数销毁（智能指针是类类型）
- 内置类型没有析构函数，什么也不需要做
- 隐式销毁一个内置类型的指针成员不会delete它所指的对象

析构函数体自身不直接销毁成员，成员是在析构函数体之后隐含的析构阶段中被销毁的。整个对象销毁过程中，析构函数体作为成员销毁步骤之外的另一部分而进行(析构作为delete表达式的第一步，它并不会释放内存空间)

> 如果一个类需要自定义析构函数，几乎可以肯定它也需要自定义拷贝赋值运算符和拷贝构造函数（如含有动态分配的成员时）

合成析构函数：对于某些类，合成析构函数用来阻止对象析构，如果不是这种情况，合成析构函数的函数体就为空

显示调用析构函数：

```
//可以通过对象、对象的指针、对象的引用调用：
string *sp = new string("a value");
sp->~string( );
```

4）常量成员函数

> 参数列表后加const。这些函数不会修改对象

const可以将this指针修改为指向常量的常量指针。故，类的常量对象和类常量对象的常量引用可以调用这些函数，也只能调用这些函数

类的非常量对象能调用常量成员，但是这种情况下，只能返回一个常量引用，因此不能使用返回结果再进一步调用非常量成员函数。可以通过重载一个与常量成员函数对应的非常量成员函数来解决：代码

5）=default与=delete

- **`=default` ：显示地要求编译器生成合成的版本**
  - **合成的函数将隐式声明为内联的**：只对成员的类外定义使用 `=default` 可以取消内联

    - **只能对编译器能合成的默认构造函数或拷贝控制成员使用**
- `=delete` ：**删除函数不能被调用**
    - **必须出现在第一次声明的时候**
    - **可以对任何函数指定**
    - **删除了析构函数的类型不能定义变量，但能动态分配这种类型对象(不能释放)**
    - 合成的拷贝控制成员可能是删除的
    - **private阻止拷贝**：新标准发布前，通过将拷贝构造函数和拷贝赋值运算符声明为private来阻止拷贝。但是，这种情况下，友元和成员函数仍旧可以拷贝对象。为了防止友元和成员函数拷贝对象，除了声明为private，还必须不定义这些函数。试图访问一个未定以的成员将导致一个链接时错误。这样处理之后，成员函数或友元函数中拷贝对象操作会导致链接时错误

4.初始化

> 构造函数第一行代码执行前，所以成员已经完成了初始化

4.1 显示初始化

- **直接初始化**：`()`
- **拷贝初始化**：`=`
    - 既有移动构造函数也有拷贝构造函数时，使用普通函数匹配规则
    - 没有移动构造函数时，右值也被拷贝
- **列表初始化**：`{}`

4.2 默认初始化

> 使用默认构造函数执行默认初始化，内置类型的成员变量初始化方式跟对象的位置有关 代码

- **局部对象**

    - **非静态局部对象**：调用默认构造函数，内置类型的成员不会默认初始化
    - **静态局部对象**：调用默认构造函数，内置类型的成员值初始化
- **全局对象**：调用默认构造函数，内置类型的成员初始化为0

4.3 值初始化

> 类的值初始化为默认初始化

- `container<T> c(n)` ：只指定了容器的大小，未指定初始值，此时容器内的元素进行值初始化
- 使用初始值列表时，未提供的元素会进行值初始化
- 静态局部变量会使用值初始化

4.4 成员的初始化

1）初始化顺序

成员的初始化顺序与它们在类中出现的顺序一样，如果提供了初始值列表，与初始值列表中的顺序无关

2）初始化步骤

1. **初始值列表**
    - 不一定要为所有成员指定初始值
    - 使用初始值列表和在函数体内分别为成员赋值的区别在于：一个是初始化，一个是先(默认)初始化再赋值
    - const和引用等一些必须初始化的成员必须使用初始值列表
2. **类内初始值**
    - 某些编译器可能不支持类内初始值
    - 对于初始值列表中没提供值的成员，使用类内初始值
3. **默认初始化**

5.作用域与名字查找

5.1 作用域

一个类就是一个作用域

一旦遇到类名，定义的剩余部分就在类的作用域内了（所以对于外部定义的成员函数，要注意返回类型）

5.2 名字查找

1）编译器处理类定义

1. 先编译成员的声明使类可见（成员包括成员变量和成员函数）
2. 再编译函数定义
    - 不管函数体是在类内定义还是类外定义
    - 因为处理完所有申明后才处理函数体，因此函数体内能使用类内所有名字；

2）成员声明中的名字查找

1. 类内声明前：因为在申明中的名字只能在该声明前查找，所以类型定义应该放在所有申明开头（类的开始处)
2. 类外,类定义前（外层作用域）

3）函数定义中的名字查找

1. 在函数体内
    - 比如函数体中使用传入的参数，就是在函数体内查找成功，此时类申明中的同名变量(名字)会被覆盖。如果想访问被覆盖的类申明中的名字(如被覆盖的成员变量)，可以使用 `this->` 或 `class_name::`
    - 也会覆盖类定义外的同名名字。如果想访问被覆盖的类定义外的同名名字(如被覆盖的全局变量)，可以使用 `::name`（这个和函数内访问被覆盖的全局变量一样）
2. 在类的申明中
3. 在函数定义外,定义前

6.类型转换

6.1 隐式类型转换

如果构造函数只接受"一个实参"，则它实际上定义了转换为此类类型的隐式转换机制。这种构造函数被称作转换构造函数，在隐式转换过程中，实际上使用转换构造函数创建了一个临时对象

直接初始化不会触发隐式转换，因此可以使用直接初始化来调用 `explict` 声明的构造函数

- **只允许一步隐式转换**
  - 比如一个类定义了一个传入string参数的构造函数，则不能通过字符串字面值来转换
  -
- **`explict` 可以抑制隐式转换**： `explict Sales_data(istream &);`
  - `explict` 不应重复，即如果在类外部定义构造函数，则不应加 `explict`
  - `explict` 是抑制隐式转换，如果不发生隐式转换，则 `explict` 构造函数能被调用。如直接初始化： 类名 对象名(参数)
  - `explict` 只是抑制隐式转换，可以使用显示转换或跳过类型转换(使用直接初始化)来使用 `explict` 构造函数

7.类对象移动

> 为什么要移动？1）某些情况下，对象拷贝后就立即销毁了。这些情况下，移动而非拷贝会大幅度提升性能；2）I/O类或unique_ptr这样的类包含不能被共享的资源，不能拷贝但可移动

7.1 右值引用

> `&&` 。就是必须绑定到右值的引用。只能绑定到一个将要销毁的对象，可以从绑定到的对象窃取状态

- 也是某个对象的另一个名字

- 不能将一个右值引用绑定到一个左值上
- 不能将一个右值引用绑定到一个右值引用类型的变量上
- 左值持久右值短暂
- `std::move(v)` 可以将变量 `v` 转换为右值引用类型
  - 头文件： `<utility>`
  - 调用 `move` 就意味着承诺：除了对 `v` 赋值或销毁它外，我们将不再使用它，调用 `move` 后，不能对以后源对象的值做任何假设
- 移动后应满足下列条件
  - 移动后原对象不应再指向被移动的资源。否则原对象析构时可能会释放资源，使得移动的内存被释放
  - 移动后的原对象应该处于可析构状态
  - 移动操作还必须保证对象仍然是有效的(用户不应对其值进行任何假设)
    - 有效是指，可以为其赋值或者可以安全地使用而不依赖其当前值(也就是说，之后的代码不会通过移动后的对象来访问已经被移动的数据，即不依赖于移后原对象中的数据)；当从标准库string或容器对象移动数据后，可以对它执行诸如 `empty` 或 `size` 这些操作，但是，我们不知道将会得到什么结果。我们可能期望一个移后原对象是空的，但这并没有保证

重载运算与类型转换

1.重载运算

- 重载为成员函数
  - 某些时候别无选择，**必须作为成员**（ `=` 、 `[]` 、 `()` 、 `->` ）
  - 某些时候作为**成员更好**（ `+=` 、 `-=` 、 `*-` 、 `++` 、 `--` ... ）
- 重载为非成员函数
  - 某些时候作为**普通函数更好**（ `+` 、 `*` ）

- 某些时候**必须作为普通函数**

1.1 重载为成员函数

一个运算对象绑定到隐式的 `this` 指针上，因此运算符函数的(显示)参数比运算对象少1

```
//两者等价
data1 += data2; //间接调用；
data1.operator += (data2); //直接调用；
```

当把运算符定义成成员函数时，它的左侧运算对象必须是运算符所属类的一个对象

1.2 重载为非成员函数

运算符函数是非成员函数时，为了与内置类型的运算符函数区分开来，必须至少含有一个类类型的参数

参数数量和运算对象相等

```
//两者等价
data1 + data2;//间接调用
operator+(data1,data2);//直接调用
```

具有对称性的运算符可能转换任意一端的运算对象，通常应该是非成员函数

如果想提供含有类对象的混合类型表达式，则运算符必须定义成非成员：

```
string u = "hi" + s;
```

如果 + 是string的成员，则产生错误

1.3 不应重载的运算符

- **逻辑与、逻辑或、逗号运算符**
  - 一些运算符指定了运算对象求值的顺序，因为使用重载的运算符本质上是一次函数调用，所以这些关于运算对象求值顺序的规则无法应用到重载的运算符上
  - 这几个运算符的运算对象求值顺序规则无法保留下来
- **`&&` 和 `||`**
  - 重载版本也无法保留内置运算符的短路求值属性，两个运算对象总是会被求值
- **逗号运算符，取址运算符**
  - C++已经定义了这两种运算符用于类类型对象时的特殊含义，所以一般来说不应该被重载

1.4 可被重载的运算符

- 1）输入输出运算符
- 2）算术和关系运算符
- 3）赋值和复合赋值运算符
- 4）下标运算符
- 5）递增递减运算符
- 6）成员访问运算符
- 7）函数调用运算符
- 8）类型转换运算符

1）输入输出运算符

第一个参数为I/O对象的引用，因此**必须重载为非成员函数**。但是通常需要读写类的非公用成员，所以I/O运算符一般被**声明为友元**

1.1）重载的输出(<<)运算符

```
ostream &operator<<(ostream &os,const Sales_data &item);
```

第一个参数是非常量的 ostream 对象的引用，第二个参数一般来说是一个常量引用

1.2）重载的输入(>>)运算符

第一个参数是运算符将要读取的流引用，第二个参数是想要读入到的(非常量)对象的引用

输入运算符必须处理输入可能失败的情况，输出运算符不需要：

- 不必逐个检查每个读取，可以等读取了所以数据后在使用前一次检查
- 在出错后，应该保持对象处于可用状态(不应是一些数据有效，一些无效)
- 可能需要更多数据验证工作。即使流没错误，但是输入的数据格式不满足要求，此时，运算符也应该设置流的条件状态以标示出失败信息

2）算术和关系运算符

2.1）算术运算符

通常定义为非成员以允许对左侧或右侧的运算对象进行转换

因为这些运算一般不需要改变运算对象的状态，所以形参都是常量的引用

如果类定义了算术运算符，一般也会定义一个对应的复合赋值运算符。此时，最有效的方式是使用复合赋值来定义算术运算符

2.2）相等运算符

通常是将两个对象的所有成员进行比较，都相等时才认为两个对象相等

应该具有传递性

2.3）不相等运算符

如果定义了相等运算符也应定义不相当运算符。如果定义了不相等运算符，也应定义相等运算符。其中一个应该把工作委托给另外一个

2.4）小于运算符

如类也有==运算符，定义的关系应与==一致（如果不等则肯定有一个小于另外一个）

3）赋值和复合赋值运算符

3.1）赋值运算符

必须是成员函数，需要访问对象的私有成员

3.2）复合赋值运算符

可以不是成员函数，但是还是倾向于把包括复合赋值在内的所有赋值运算都定义在类的内部

4）下标运算符

> 如表示容器的类，可以通过下标访问容器中的元素

- 必须是成员函数
- 返回访问元素的引用
- 最好同时定义常量版本与非常量版本：当作用于一个常量对象时，下标运算符返回常量引用；

5）递增递减运算符

> 迭代器类中，通常会实现递增递减运算符

并不要求必须是成员函数，但是因为它们改变的正好是所操作对象的状态，所以建议将其定义为成员函数

5.1）前置递增递减

```
StrBlobPtr& operator++();
StrBlobPtr& operator--();
```

返回递增递减后对象的引用

5.2）后置递增递减

为了与前置版本区别开，接受一个额外（不被使用的）int类型的形参，当我们使用后置运算符时，编译器为这个形参提供一个值为0的实参。这个形参的唯一作用就是区分前置版本和后置版本的函数，而不是真的要在实现后置版本时参与运算

```
StrBlobPtr operator++(int);
StrBlobPtr operator--(int);
```

返回递增递减前对象的拷贝

递增递减操作可以直接使用实现的前置版本来完成

显示调用递增递减运算：

```
StrBlobPtr p(a1);
p.operator++(0); //调用后置版本的operator++;
p.operator++( ); //调用前置版本的operator++;
```

调用后置版本，必须为它的参数传递一个值，尽管传入的值通常被运算符函数忽略，但必不可少，因为编译器只有通过它才知道应该使用后置版本

6）成员访问运算符

通常将两者定义为const成员，因为与递增递减运算符不一样，获取一个元素并不会改变对象的状态

6.1）解引用

通常也是类的成员，但并非必须如此

能令 `operator*` 返回任何我们指定的操作。换句话说，我们可以让 `operator*` 返回一个固定值 `42` ，或者打印对象的内容，或者其他

6.2）箭头

必须是类的成员

箭头运算符永远不能丢掉成员访问这个最基本的含义。当重载箭头运算符时，可以改变的是箭头从哪个对象当中获取成员，而箭头获取成员这一事实则永远不变

7) 函数调用运算符

```
struct absInt{
    int operator() (int val) const {
        return val < 0 ? -val : val;
    }
};
```

可以创建一个对象，并且像函数一样调用：

```
absInt absObj;
int ui = absObj(42);
```

函数对象：重载了函数调用运算符的类，创建的对象可以"像函数一样"使用，这样的对象就是函数对象

函数对象通常作为泛型算法的实参，如 `for_each`

标准库定义的函数对象类 (头文件： `<functional>`) ：

- `plus` ：定义了一个函数调用运算符用于对一对运算对象执行 `+` 的操作
- `modulus` ：定义了一个函数调用运算符执行二元 `%` 操作
- `equal_to` ：执行 `==`

默认情况下排序算法使用 `operator<` 将序列按照升序排列。如果要执行降序排列的话，可以传入一个 `greater` 类型的对象，该类产生一个调用运算符并负责执行待排序类型的 `>` 运算，从而实现降序排列

标准库规定其函数对象(即标准库定义的函数对象)对于指针同样适用。一般来说，比较两个无关的指针将产生未定义的行为，然而我们可能会希望通过比较指针的内存地址来sort指针的vector。直接这样做将产生未定义的行为，可以使用 `less` 函数对象类来实现：

```
sort(nameTable.begin( ),nameTable.end( ),less<string*>( ));
```

8）类型转换运算符

```
[explicit] operator type() const; {...}
```

- 既没有实参，也没有形参
- 不能声明返回类型。尽管类型转换函数不负责指定返回类型，但是实际上每个类型转换函数都会返回一个对应类型的值；
- 必须定义成成员函数
- 通常应该是const

有 `explict` 则必须通过显示转换，没有则是隐式转换

8.1）隐式类型转换运算符

```
operator int() const {return val;};
```

因为类型转换运算符是隐式执行的，所以无法给这些函数传递实参，也就不能在类型转换运算符的定义中使用任何形参

尽管编译器一次只能执行一个“用户定义”的类型转换，但是隐式的用户定义类型转换可以置于一个标准(内置)类型转换之前或之后

8.2）显示类型转换运算符

> C++11引入了显示类型转换运算符

```
explicit operator int() const {return val;}
```

编译器不会将一个显示的类型转换运算符用于隐式类型转换。除非表达式被用作条件

向 bool 的类型转换通常用在条件部分，因此 operator bool 一般定义成 explicit 的

2.二义性类型转换

> 转换构造函数与类型转换运算符共同定义了类的类型转换

2.1 转换二义性

1）两个类提供相同的类型转换

A定义了一个接受B类对象的转换构造函数，同时B定义了一个转换目标是A类的类型转换运算符：

```
B b;
A a = f(b);//b是通过B的类型转换运算符转换成A，还是通过A的接受B类型对象的转换构造函数构造？
```

2）类定义了多个转换规则，而这些转换涉及的类型本身可以通过其它类型转换联系在一起

最典型的例子是算术运算符：

```
struct A{
    A(int = 0);
    A(double);
};
long lg;
A a2(lg); //long转换成int还是double?
```

```
short s = 42;
A a3(s); //这种情况下，short进行整形提升的转换级别高于short转换成double，所以会匹配A(int)版本
```

因此，对某个给定的类来说，最好只定义最多一个与算术类型有关的转换规则

下面的例子"转换目标"会出现二义性：

```
struct A{
    operator int() const;
    operator double() const;
};
void f2(long double);
A a;
f2(a); //a转换成int再转换成long double，还是转换成double再转换成long double?
```

2.2 避免转换出现二义性

1. 不要令两个类执行相同类型的转换（如果Foo有一个接受一个Bar类对象的构造函数，则不要在Bar类中再定义转换目标是Foo类的类型转换运算符）
2. 避免转换目标是内置运算类型的类型转换，如果已经定义了一个转换成算术类型的类型转换则
    - 不要再定义接受算术类型的重载运算符
    - 不要定义转换到多种算术类型的类型转换

继承体系

```
class Bulk_quote : public Quote [, ...] {
    ...
};
```

如果是单继承(只继承一个基类)，则没有 `[, ...]` 部分

1.虚函数

> 虚函数是基类希望每个派生类自己定义的函数，对不同的派生类，实现可能不同

在函数声明开头添加 `virtual` 关键字

析构函数一般声明为虚函数：果基类的析构函数不是虚函数，则delete一个指向派生类对象的基类指针将产生未定义的行为。

虚函数在派生类中隐式地也是虚函数：也可以通过指明virtual关键字来说明该函数在派生类中也是虚函数，但是没必要，因为一旦某个函数被声明成虚函数，则所有派生类中它都是虚函数

虚函数必须有定义：即使不被调用也必须定义，因为编译器也无法确定到底会调用哪个虚函数

虚函数与默认实参：虚函数也可以拥有默认实参。如果某次调用使用了默认实参，则该实参值由本次调用的静态类型决定。换句话说，如果我们通过基类的引用或指针调用函数，则使用基类中定义的默认实参。此时传入派生类函数的将是基类函数定义的默认实参。如果派生类函数依赖于此实参，则程序的结果将与我们的预期不符

- **回避虚函数**
  - **如何回避**？使用域作用符： `baseP->Base::fcn(...);` 强迫执行基类中的版本
  - **为何回避**？当一个派生类的虚函数调用它覆盖的基类的虚函数版本时需要回避虚函数。在此情况下，基类的版本通常完成继承层次中所有类型都要做的共同任务，而派生类中的版本需要执行一些与派生类本身密切相关的操作；如果此时没有回避虚函数，运行时该调用将被解析为对派生类版本自身的调用，导致无限循环

1.1 动态绑定

满足以下2个条件时，会发生动态绑定：

1. **通过基类的引用或指针调用**：意味着通过派生类的"引用"或"指针"调用不会发生，实际上派生的"引用"或"指针"也无法指向一个基类对象
2. **调用基类(声明)的虚函数**：意味着调用的函数如果不是虚函数也不会发生

1.2 函数覆盖

> 如果派生类不声明及定义基类的虚函数，则直接使用基类中的版本

通过 `override` 显示声明：override说明符出现在形参列表(包括任何const或引用修饰符)以及尾置返回类型之后。显示声明这个函数是派生类覆盖的存在于基类中的某个虚函数。使得程序员的意图更加清晰的同时让编译器可以为我们发现一些错误。比方说，如果定义了一个函数与基类中虚函数的名字相同但是形参列表不同，这是合法的行为。但是编译器将认为新定义的这个函数与基类中原有的函数是相互独立的。这时，派生类的函数没有覆盖掉基类中的版本。就实际的编程习惯而言，这种申明往往意味着发生了错误，因为我们可能原本希望派生类能覆盖掉基类中的虚函数，但是不小心把形参列表弄错了

通常情况下，覆盖函数必须与虚函数的参数类型及返回类型相同：例外是，当类的虚函数返回类型是类本身的指针或引用时，覆盖函数的返回类型可以与虚函数不同。也就是说，如果 `D` 是 `B` 的派生类，则基类的虚函数返回 `B*` 而派生类的对应函数可以返回 `D*` ，只不过这样的返回类型要求从 `D` 到 `B` 的类型转换是可访问的

1.3 纯虚函数与抽象基类

纯虚函数：分号前包含 `= 0` 的虚函数。纯虚函数无须定义，但是也可以定义

抽象基类：

- 含有纯虚函数的类
- 主要负责定义接口（即一系列纯虚函数）
- 不能(直接)创建对象
- 抽象基类的派生类必须定义纯虚函数，否则依然是抽象基类

2.派生类的构造与拷贝控制

2.1 构造

> 尽管可以在派生类构造函数体内给它的公用或受保护的基类成员赋值，但是最好不要这么做。应该遵循基类的接口，通过调用基类的构造函数来初始化那些从基类中继承而来的成员

1）初始化顺序

1. **如果未显示调用基类构造函数，则对派生类对象的基类部分执行默认初始化**
   - 基类部分调用基类的默认构造函数执行默认初始化
   - 如果显示指明了基类的构造函数，则用指定的构造函数初始化基类部分
   - 对于多继承，基类的构造顺序与派生列表中基类的出现顺序一致，与派生类构造函数初始值列表中基类的顺序无关
2. **然后派生类构造函数初始化派生类部分**

2）继承的构造函数

using生成与基类对应的构造函数：通常情况下，using声明语句只是令某个名字在当前作用域内可见。而当作用于构造函数时，using声明语句将令编译器产生代码。对于基类的每个构造函数，编译器都生成一个与之对应的派生类构造函数。对于多继承，如果多个基类包含相同参数的构造函数，则继承会引发冲突，可以通过在派生类中自定义这个参数相同的版本来解决冲突

- **参数与基类构造函数的参数相同**：另外，如果派生类含有自己的数据成员，则这些成员将被默认初始化
- **不会改变构造函数的访问级别**（这和using一般的用途不同）
- **using声明语句不能指定explict或constexpr**：如果基类的构造函数是explict或constexpr，则继承的构造函数也拥有相同的属性
- **构造函数中的默认实参不会被继承**：当一个基类构造函数含有默认实参时，这些实参并不会被继承。相反，派生类将获得多个继承的构造函数，其中每个构造函数分别省略一个含有默认实参的形参
- **只能继承直接基类的构造函数**：
- **不能继承默认，拷贝和移动构造函数**：如果派生类没有，编译器将为其合成

- **不会继承与派生类自定义形参相同的构造函数**：

2.2 拷贝控制

1）拷贝顺序

- 对于拷贝与移动操作
  - 与构造顺序相同，先处理基类再处理派生类
  - 对多继承中基类的处理，与派生列表中基类顺序相同
- 对于析构操作
  - 与构造顺序相反，先处理派生类再处理基类
  - 对多继承中基类的处理，与派生列表中基类的顺序相反

2）合成的拷贝控制

基类或派生类的合成拷贝控制成员的行为与其他合成的构造函数，赋值运算符或析构函数类似：它们对“类本身”的成员依次进行初始化、赋值、销毁操作。此外，这些合成的成员还负责使用直接基类中对应的操作对一个对象的直接基类部分进行初始化、赋值或销毁

3）用户定义的拷贝控制

3.1）拷贝及移动

```
class Base { ... };
class D : public Base {
public:
    //拷贝构造
    D(const D& d) : Base(d)  /*D的成员初始值*/  {...}
    //移动构造
    D(D&& d) : Base(std::move(d)) /*D的成员初始值*/ {...}
};
```

默认情况下，使用基类的默认"构造函数"初始化派生类对象的基类部分。如果想拷贝(或移动)基类部分，则必须在派生类的构造函数初始值列表中显示地使用基类的拷贝(或移动)构造函数

3.2) 赋值运算符

```
D &D::operatpr=(const D &rhs)
{
    Base::operator=(rhs); //为基类部分赋值
    //...
   return *this;
}
```

如果不显示的调用基类的赋值运算符，则派生类中的基类部分不会被赋值

3.3) 析构

```
class D : public Base{
public:
    ~D() {/*清除派生类成员的操作*/}
};
```

析构函数内不需要显示调用基类的析构函数，在销毁派生类自身成员后，会隐式调用基类的析构函数析构基类的部分

3.防止继承与防止覆盖

- **`final` 关键字防止继承**：在类的定义时，通过在类名后面添加final关键字，可以防止此类被继承。在不想考虑一个类是否适合作为一个基类或不希望一个类作为基类时使用
- **`final` 关键字防止覆盖**：定义成final的函数，如果之后有任何尝试覆盖该函数的操作，都将引发错误

4.static与继承

- 静态成员在整个继承体系中只存在该成员的唯一定义
- 不论从基类派生出多少派生类，每个静态成员只存在唯一实例
- 如果静态成员可访问（即权限允许），则既能通过基类访问也能通过派生类访问（包括基类对象和派生类对象）

5.类型转换与继承

- **静态类型**：编译时已知，是变量声明时的类型或表达式生成的类型
- **动态类型**：变量或表达式的内存中的对象的类型，运行时才知道

只有发生动态绑定时，动态类型才可能与静态类型不同。

只有指针和引用才有动态类型

- 变量或表达式既不是引用也不是指针则静态类型与动态类型一致，都为静态类型
- 指针或引用的这种静态类型与动态类型不一致，产生了动态绑定

5.1 指针或引用的转换

1）存在派生类向基类的转换

之所以存在派生类向基类的类型转换是因为每个派生类对象都包含一个基类部分，而基类的引用或指针可以绑定到该基类部分上

- **可以将基类的指针或引用绑定到派生类的对象上**
  - 对于多继承，编译器不会在派生类向基类的几种转换中进行比较和选择，因为在它看来转换到任意一种基类都一样好。因此，如果重载函数的参数是不同基类的引用，调用时传入一个派生类对象，会产生二义性错误
- 派生类向基类的类型转换也可能会由于访问受限而变得不可行

继承与容器

- **容器中如何存储具有继承关系的不同类？**
  - 答：基类的指针，最好是智能指针。正如我们可以将一个派生类的普通指针转换成基类指针一样，我们也能把一个派生类的智能指针转换成基类的智能指针。
- **如何实现可以动态绑定的元素添加函数？**
  - 首先，基类定义个虚函数。虚函数返回一个动态创建的该类的对象，返回这个对象的指针
  - 然后，每个派生类覆盖这个虚函数，实现自己的版本
  - 元素添加函数中可以根据传回的指针创建智能指针，将智能指针进一步添加到容器中；因此，元素添加函数只需要定义一个基类对象的引用作为参数，就能实现向容器中添加不同的类对象

2）不存在基类向派生类的转换

因为基类不包括派生类的成员，派生类可能调用到自身(不属于)基类的成员，所以基类不能转换成派生类

```
Base b;
Derived *d1 = &b; //错误；
Derived &d2 = b; //错误；
```

即使一个基类指针或引用绑定在一个派生类对象上，也不能执行从基类向派生类的转换：

```
Derived d;
Base *b = &d;
Derived *dp = b; //错误，不能将基类转换成派生类；
```

因为编译器在编译时无法确定某个特定的转换在运行时是否安全，因为编译器只能通过检查指针或引用的静态类型来推断该转换是否合法

5.2 对象之间不存在转换

派生类向基类的自动转换只对指针或引用类型有效。将派生类对象转换成基类对象，可以在基类中定义相应的拷贝或移动操作（此时实际运行的构造函数是基类中定义的那个，只能处理基类自己的成员，派生类部分被"切掉"了； 尽管自动类型转换只对指针或引用类型有效，但是继承体系中的大多数类仍然(显示或隐式)定义了拷贝控制成员。因此，能将一个派生类对象拷贝，移动或复制给一个基类对象；不过这种操作只处理派生类对象的基类部分）

6.访问权限与继承

访问权限与继承

对基类的访问权限由基类自身控制，即使对于派生类的基类部分也是如此。因此，尽管基类的友元不能访问派生类的成员，但却可以通过派生类对象，访问基类的成员

6.1 using修改继承自基类成员的权限

```
class Base{
public:
    std::size_t size() const {return n;}
protected:
    std::size_t n;
};
class Dericed : private Base {
public:
    using Base::size;
private:
    using Base::n;
};
```

只能为那些能访问的(如:不管哪种继承,基类中的private成员都不能访问)名字提供using声明。using可以将"直接"或"间接"基类中的任何可访问成员标记出来，声明中名字的访问权限由using声明语句前的访问说明符决定

- using与继承中的函数重载

- using与构造函数

7.继承中的作用域

> 这里主要是继承中，类与类之间的作用域关系。与单个类的作用域共同组成了整个C++类继承体系的作用域

派生类作用域嵌套在其基类作用域内：正是因为这种继承嵌套的关系，派生类才能像使用自己的成员一样使用基类的成员

7.1 名字查找

1. **先在自身作用域中查找**
2. **然后到基类作用域中查找**

名字查找在编译时进行，静态类型决定了类的哪些成员名字是可见的。如：即使一个基类指针指向的是派生类的对象，但是调用派生类的函数也无法编译通过，因为指针的静态类型是基类的类型，名字查找无法进入派生类的作用域

多继承中的名字查找：

- 对于多继承，会并行的在多个基类中查找
- 对于一个派生类来说，从其多个基类中分别继承名字相同的成员完全合法，只不过在使用这个名字时必须明确指出版本，否则会引发二义性
  - 有时即使派生类继承的两个函数参数列表不同也可能发生错误
  - 即使该名字在一个基类中是私有的，另一个基类中是公用或受保护的同样也可能发生错误

7.2 名字继承与冲突

派生类能重用定义在直接或间接基类中的名字。如果派生类出现同名名字，将会隐藏（注意，"隐藏"不同于虚函数中的"覆盖"）基类中的名字

- 可以通过作用域运算符来访问隐藏的成员。作用域运算符指示从指定的类的作用域开始查找

- 除了"覆盖"继承而来的虚函数之外，派生类最好不要重用其它定义在基类中的名字

不同作用域与无法重载，因此同名会隐藏：因此，派生类中的函数不会重载基类中的成员。即使形参列表不同，派生类也是将基类中的同名函数进行隐藏，而不会重载

- **派生类的成员函数会隐藏基类中所有同名的重载函数**：所以如果派生类希望所有重载版本都对其可见，要么都不隐藏，要么为所有版本定义相应的重载版本
- **如果只想覆盖一个或几个其它保持不变可以用using**：using声明指定一个名字而不指定形参列表，所以一条基类成员函数的using声明语句就可以把该函数的所有重载实例添加到派生类作用域中。此时派生类只需定义其特有的函数就可，而无须为继承而来的其他函数重新定义。基类函数的每个实例在派生类中都必须是可访问的，因为对派生类没有重新定义的重载版本的访问，实际上是对using声明点的访问。

8.虚继承

方式：在派生列表中使用 `virtual` 关键字（可以出现在继承访问说明符之前也可以出现在之后）

8.1 重复继承

> 即菱形继承

- 派生类可以通过它的两个直接基类分别继承同一个间接基类
- 直接继承某个基类，然后通过另一个基类再一次间接继承该类

子对象冗余：默认情况下，派生类含有继承链上每个类对应的子部分。当一个派生类重复继承一个类时，将包含该类的多个子对象

虚继承可以防止子对象冗余：iostream类继承自istream和ostream，这两个类又继承自base_ios； 一个iostream对象肯定希望在同一个缓冲区中进行读写操作，也会要求条件状态能同时反映输入和输出操作的情况。假如在iostream对象中真的包含了base_ios的两份拷贝，则上述的共享行为就无法实现了。类似i/o类中的继承需求，如果不想重复继承某类时，有冗余子对象，可以在继承时使用虚继承。虚继承的目的是令某个类做出声明，承诺愿意共享它的基类

8.2 成员可见性

```
   B
D1 D2
   D
```

假设D1、D2继承自B，D继承自D1、D2，B中含有成员 x ：

- 如果D1、D2都没有 x 的定义，则 x 被解析为B的成员
- 如果D1或D2某一个定义了 x ，则D1或D2中 x 的优先级高于B中的 x （也不会有二义性）
- 如果D1和D2都定义了 x ，则产生二义性；

8.3 构造与拷贝控制

1）构造

在虚派生中，虚基类是由最底层的派生类“负责”初始化的

```
   B
D1 D2
   D
```

- 如果创建D的对象，则虚基类由D负责初始化
- 如果创建D1的对象，则虚基类由D1负责初始化

构造顺序：

1. 先构造虚基类
2. 再构造直接基类
3. 最后构造自身成员；

编译器会按照直接基类的声明顺序对其依次进行检查，以确定其中是否含有虚基类，如果有，则构造虚基类

2）合成的拷贝控制

与构造函数按照完全相同的顺序执行

3）析构

与构造完全相反的顺序

容器

1.容器通用操作

1.1 类型别名

1）迭代器类型

- `iterator`
- `const_iterator`
- `reverse_iterator` ：反向迭代器
- `const_reverse_iterator`

2）容器大小

`size_type` ：无符号整形，能保存容器最大可能大小

3）迭代器距离

`difference_type` ：带符号整形，两个迭代器间的距离

4) 元素类型

- `value_type` ：容器的元素类型
- `reference` ：元素的左值类型，与 `value_type&` 同义
- `const_reference` ：元素的const左值类型，与 `const value_type&` 同义

1.2 构造

- `C c`
- `C c1(c2)`
- `C c(b,e)` ：将迭代器 `(b,e)` 之间的元素拷贝到 `c` ，不包括 `e`
- `C c{a,b,c...}`

1.3 赋值与swap

- `c1 = c2`
- `c1 = {a,b,c...}`
- `a.swap(b)` ：交换 `a` ， `b` 的元素
- `swap(a,b)` ：与 `a.swap(b)` 等价

1.4 大小

- `c.size()`
- `c.max_size()` ： `c` 可保存的最大元素数目
- `c.empty()`

1.5 添加与删除元素

- `c.insert(args)` ：将 `args` 中的元素插入进 `c`
- `c.emplace(inits)` ：使用 `Inits` 构造 `c` 中的一个元素
- `c.erase(args)` ：删除 `args` 指定的元素
- `c.clear()` ：删除 `c` 中的所有元素

1.6 关系运算符

> 关系运算符两边的运算对象必须是相同类型的容器。如果容器元素的类型不支持所用运算符，就不能进行比较

- `==` 、 `!=` ：所有容器都支持
- `<` 、 `<=` 、 `>` 、 `>=` ：无序关联容器不支持

1.7 获取迭代器

带r的版本返回反向迭代器；带c的版本返回const迭代器

不以c开头的版本都是重载过的：

- 如果通过一个常量容器对象调用，则返回一个const迭代器
- 如果通过一个非常量容器对象调用，则返回一个普通迭代器

所以当使用auto时，返回的类型取决于调用的容器对象

- `c.begin()` 、 `c.end()` ： `begin` 与 `end` 相等，则迭代器为空
- `c.cbegin()` 、 `c.cend()`
- `c.rbegin()` 、 `c.rend()`
- `c.crbegin()` 、 `c.crend()`

2.顺序容器

> 为程序员提供了控制元素存储和访问顺序的能力，这种顺序不依赖于元素的值，而是与元素加入容器时的位置相对应

2.1 种类

1. **vector**：随机访问快，尾部之外的位置元素插入删除慢；通常vector是最好的选择，除非有很好的理由选择其它顺序容器
2. **deque**：双端队列，随机访问快，头尾插入删除快
3. **list**：双链表，支持双向访问，任何位置的插入删除都很快
4. **forward_list**：单链表，和list类似。forward_list的设计目标是达到与最好的手写单向链表相当的性能，因此，没有 `size()` 操作； forward_list不支持迭代器递减
5. **array**：大小固定，不能插入删除。与内置数组相比，array是一种更安全，更容易使用的数组类型（ `array<int ,10> a;` ）
6. **string**：随机访问快，尾部插入删除快

2.2 操作

1）定义和初始化

- `C c`
- `C c1(c2)` ： `c1` 、 `c2` 类型必须相同
- `C c1 = c2` ： `c1` 、 `c2` 必须是相同类型。所以如果是array，大小也要相同

- `C c{a,b,c...}` ：任何遗漏的元素值初始化
- `C c = {a,b,c...}` ：任何遗漏的元素值初始化
- `C c(b,e)` ：和初始化列表不同，当传递一个迭代器参数来拷贝一个范围（不包括 `e` ）时，不要求容器类型是相同的，可以不同，能转换就行（array不支持）
- `C c(n)` ：值初始化，explict
- `C c(n,t)` ： `n` 个元素都为 `t`

> 对array使用初始化列表时，列表中的元素个数必须"小于或等于"array的大小

2）赋值与swap

swap通常比赋值要快得多

指向容器的迭代器，引用和指针(除string外)在swap操作后都不会失效，他们仍指向swap操作之前所指向的那些元素，但是swap后，这些元素已经属于不同的容器了

string的swap：与其他容器不容，对一个string调用swap会导致迭代器，引用和指针失效。会真正交换两个string的元素，所需的时间与元素数目成正比

- `c1 = c2` ： `c1` 类型必须相同 `c2` （大小也是array类型的一部分，所以 `c1` 、 `c2` 大小要相同）。赋值后 `c1` 等于 `c2` 的大小
- `c1 = {a,b,c,...}` ：不适用于array
- `a.swap(b)` ：交换a,b的元素
- `swap(a,b)` ：与 `a.swap(b)` 等价

3）assign

> array与关联容器不支持。可以用assgin实现将一个vector中的一段 `char*` 的值赋给一个list中的 `string`

- `c.assign(b,e)` ：将 `c` 中的元素替换成迭代器 `b` 、 `e` 范围的元素， `b` 、 `e` 一般指向不同容器，两个容器类型不同，但是类型兼容(能转换)
- `c.assign(il)` ：将 `c` 中的元素替换为初始化列表 `il` 中的元素
- `c.assign(n,t)` ：将 `c` 中的元素替换成 `n` 个值为 `t` 的元素

4）大小

除了这些操作，还包括：

- `resize(size_type)` ：
  - 扩大：不会改变原有的元素，新元素值初始化
  - 缩小：删除尾部的元素
- `resize(size_type,t)` ：
  - 扩大：新元素的值为 `t`
  - 缩小：直接删除多余的，跟 `t` 没关系

> 缩小容器会使指向被删除元素的迭代器，引用，指针都失效

5）添加元素

> 会改变大小，array不支持。中部添加单个或多个，都是添加到迭代器p之前

- **以下操作添加单个元素**
  - **首部添加**：vector、string不支持，返回 `void`
    - `c.push_front()`
    - `c.emplace_front()`
  - **中部添加**：vector可以通过这些接口在首部插入。插入到迭代器p指向的位置之前，返回新添加元素的迭代器
    - `c.insert(p,t)`
    - `c.emplace(p,args)`
  - **尾部添加**：forward_list不支持，返回 `void`
    - `c.push_back()`
    - `c.emplace_back()`
- **以下操作添加多个元素**：插入到迭代器p指向的位置之前，返回新添加第一个元素的迭代器，范围为空时，返回p
  - `c.insert(p,b,e)`
  - `c.insert(p,n,t)`
  - `c.insert(p,il)` ：il是一个花括号包围的元素值列表

6）删除元素

删除前记得检查容器是否为空

会改变大小，array不支持

除了删除首尾元素，都返回删除元素后一个元素的迭代器；

删除deque中首尾元素外任意元素，都会使迭代器，指针，引用失效。指向vector和string删除元素后面元素的迭代器，指针，引用会失效；

- **删除一个**
  - `c.pop_back()` ：forward_list不支持。返回 `void`
  - `c.pop_front()` ：vector,string不支持。返回 `void`
  - `c.erase(p)` ：删除迭代器 `p` 指定的元素，返回被删除元素后元素的迭代器。若 `p` 是尾元素，则返回尾后迭代器；若p是尾后迭代器，结果未定义；
- **删除多个**
  - `c.erase(b,e)` ：删除迭代器 `b` 、 `e` 范围间的元素， `e` 不会删除。返回最后一个被删除元素后一个元素的迭代器。调用结束后， `b == e`
- 删除所有
  - `c.clear()` ：返回 `void`

forward_list的添加与删除：由于单链表难以访问前驱，所以forward_list是在给定位置后插入或删除元素，因此对于中部的添加与删除，和其它顺序容器接口不同

- **添加**
  - `lis.insert_after(p,t)`
  - `lis.insert_after(p,n,t)`
  - `lis.insert_after(p,b,e)`
  - `lis.insert_after(p,il)`
  - `lis.emplace_after(p,args)`

- **删除**：返回被删除之后元素的迭代器。如果p指向尾元素或是一个尾后迭代器，结果未定义
  - `lis.erase_after(p)` ：删除迭代器 `p` 之后的元素
  - `lst.erase_after(b,e)` ：不包含迭代器 `b` 指向的元素

`lst.before_begin()` 返回链表首元素之前不存在的元素的迭代器，不能解引用； `lst.cbefore_begin()` 是对应的const版本

7) 访问元素

- **访问首尾**
  - **引用**
    - `c.front()`
    - `c.back()` ：返回最后一个元素。forward_list不支持
  - **迭代器**
    - `c.begin()`
    - `c.end()` ：最后一个元素下一个位置的迭代器（尾后迭代器）。可以递减后解引用访问末尾元素，由于forward_list不支持迭代器递减，所以也不能通过这种方式访问尾元素
- **访问随机**：返回引用。不适用用list和forward_list
  - `c[]` ：不会对下标进行检查
  - `c.at()` ： `at` 会进行下标越界检查，越界时抛出 `out_of_range` 异常。因此更安全

2.3 迭代器失效

> 迭代器在容器大小发生变化时才可能失效，因此注意容器元素的添加与删除操作。使用失效的迭代器，指针和引用是严重的运行时错误

- **添加元素**
  - **vector或string**
    - 存储空间重新分配
      - 都会失效
    - 存储空间未重新分配
      - 指向插入位置前的不失效

        - 指向插入位置后的失效
    - **deque**
        - 首尾位置之外插入则都失效
        - 首尾位置插入则迭代器会失效，指针和引用不失效
    - **list和forward_list**
        - 都有效
- **删除元素**
    - **vector或string**
        - 指向被删元素之前的都有效：因此，尾后迭代器总失效
    - **deque**
        - 首尾位置之外删除都失效
        - 删除末尾除了尾后迭代器失效，其它都有效
        - 删除首元素不失效
    - **list和forward_list**
        - 指向被删元素外的都有效

3.关联容器

> multi：允许关键字重复；unordered：无序

3.1 有序关联容器

- **map**
- **set**
- **multimap**
- **multiset**

关键字必须定义 < 比较方法。默认情况下，标准库使用关键字类型的 < 运算符比较两个关键字

```
//compareIsbn是自定义的<运算符
muliset<Sales_data,decltype(compareIsbn)> bookstore(compareIsbn);
```

3.2 无序关联容器

- **unordered_map**
- **unordered_set**
- **unordered_multimap**
- **unordered_multiset**

元素无序组织，输出(通常)会与使用有序容器不同。无序关联容器在存储上组织为一组桶，每个桶保存0个或多个元素，使用哈希函数将元素映射到桶。根据hash函数，具有不同关键字的元素可能会映射到相同的桶。如果允许重复关键字，所有具有相同关键字的元素也会映射到同一个桶中。因此，无序容器的性能依赖于哈希函数的质量和桶的数量及大小

关键字类型需要定义 == 比较方法以及哈希值计算函数。默认情况下，标准库使用关键字类型的 == 运算符来比较元素。标准库为内置类型(包括指针)提供了hash模板，还为string和智能指针等一些标准类型定义了hash，因此可以直接定义关键字是这些类型的无序关联容器

```
//42是桶大小
//hasher是自定义hash函数名，作为参数会自动转换为函数指针；
//eqop是自定义==运算函数名；
unordered_multiset<Sales_data,decltype(hasher)*,decltype(eqop)*> bookstore(42,hasher,eqop);
```

3.3 pair

`pair<T1,T2>`，头文件： `<utility>`

1）初始化与创建

```
pair<T1,T2> p //数据成员值初始化
pair<T1,T2> p = p2
pair<T1,T2> p(v1,v2)
pair<T1,T2> p{v1,v2}
//与上面等价
makepair(v1,v2)
```

2）比较

`(p1.T1 < p2.T1)` 或 `(p1.T1 == p2.T1 && p1.T2 < p2.T2)` 时 `p1 < p2`

3）数据成员

以下两个成员都为 `public` ：

- `firsr`
- `second`

3.4 操作

与顺序容器不同，通常不对关联容器使用泛型算法。关联容器只能用于读取元素的算法，尽管能使用但是性能不好

- 排序：关联容器不能通过关键字进行(快速)查找
- find：泛型find会顺序搜索，关联容器定义的find快得多

实际情况中，在泛型算法中使用关联容器，通常都是作为一个源序列或目的序列，如copy。多数时候关联容器的算法都使用成员函数

1）类型别名

- `key_type` ：关键字类型
- `mapped_type` (仅map)：关键字关联的类型（即值的类型）
- `value_type`
  - map： `pair<const k,v>` ，关键字为const意味着不能修改
  - set：与 `key_type` 相同

2）初始化

> 不接受一个元素值，一个数量值类型的初始化

- 默认初始化
- 拷贝初始化 `=`
- 列表初始化 `{}`
- `(b,e)` ：对set来说， `b` 、 `e` 不一定是关联容器的迭代器，可以是vector的迭代器

3）迭代器

> 都是双向的

- `map` ：迭代器指向的元素first成员无法修改（const）
- `set` ：iterator与const_iterator都不能修改指向的成员

4）添加元素

- **有序关联容器**
  - **添加一个**（除此之外，对于map和unordered_map，如果不需要查看插入状态，还可以使用下标操作来插入）
    - `c.emplace(args)`
      - 对于set和map：返回一个pair，second成员为bool值，true表示插入成功，false表示插入失败。first成员为包含关键字元素的迭代器
      - 对于multiset和multimap：因为同一关键字可以重复，所以总是插入成功，因此返回包含关键字元素的迭代器
    - `c.insert(v)`
      - 对于set和map：返回一个pair，second成员为bool值，true表示插入成功，false表示插入失败。first成员为包含关键字元素的迭代器
      - 对于multiset和multimap：因为同一关键字可以重复，所以总是插入成功，因此返回包含关键字元素的迭代器
    - `c.insert(p,v)`
      - 迭代器p作为一个提示，指示从哪里开始搜索新元素应该存储的位置。返回具有给定关键字的元素的迭代器
    - `c.emplace(p,args)` ：迭代器p作为一个提示，指示从哪里开始搜索新元素应该存储的位置。返回具有给定关键字的元素的迭代器
  - **添加多个**：返回 `void`
    - `c.insert(b,e)`
      - 如果 `c` 为set， `b` 和 `e` 不一定要是关联容器的迭代器，可以用vector的迭代器
    - `c.insert({...})`
      - map时， `{}` 中通常还有 `{}` ，表示pair

5）删除元素

- **删除一个**
  - `c.erase(k)` ：从 `c` 中删除每个关键字为 `k` 的元素，返回表示删除数量的 `size_type` 类型的值。对于不允许关键字重复的关联容器，始终返回0或1。返回0表示关联容器中没有关键字为 `k` 的元素
  - `c.erase(p)` ：从 `c` 中删除迭代器 `p` 指定的元素。 `p` 必须指向 `c` 中一个真实元素，不能等于 `c.end()` 。返回一个指向 `p` 之后元素的迭代器

- **删除多个**
  - `c.erase(b,e)` ：删除 `[b,e)` 范围的元素，返回 `e` ；

6）下标操作

- **set(4种)不支持下标**：set中有与关键字关联的"值"，元素本身就是关键字。因此"获取与一个关键字相关联的值"的操作就没有意义了
- **multi类型的不支持下标**：multi类型允许关键字重复，所以一个关键字与多个"值"关联
- **只有map和unordered_map支持**：如果关键字不在map中，会为它创建一个元素并插入map中。新建元素的关联值进行"值初始化"。 因为下标运算可能修改map，所以只可以对非const的map使用下标操作。下标运算会返回一个关联值类型的左值；
  - 下标运算符
  - `at()`

7）访问元素

下标操作当关键字不存在时会插入，如果只希望查找，不想插入，则可以使用这部分的操作； 除此之外，这些操作支持所有有序关联容器；

- `c.find(k)` ：查找第一个关键字为k的元素，返回指向这个元素的迭代器，不存在则返回尾后迭代器；
- `c.cout(k)` ：查找关键字为 `k` 的元素，返回这些元素的数量。对于不允许关键字重复的关联容器，永远返回0或1
- `c.lower_bound(k)` ：返回关键字 `>=k` 的第一个元素的迭代器。无序关联容器不支持
- `c.upper_bound(k)` ：返回关键字 `>k` 的第一个元素的迭代器。无序关联容器不支持
- `c.equal_range(k)` ：返回一个迭代器pair，指示关键字为 `k` 的元素的范围。 `first` 成员是第一个关键字为 `k` 的元素的迭代器， `second` 成员是第二个关键字为 `k` 的元素的迭代器

8）无序关联容器的桶操作

- **桶接口**
  - `c.bucket_count()` ：正在使用的桶数目
  - `c.max_bucket_count()` ：容器能容纳的最多桶数

    - `c.bucket_size(n)` ：第 `n` 个桶中有多少个元素
    - `c.bucket(k)` ：关键字为 `k` 的元素在哪个桶；
- **桶迭代**
    - `local_iterator` ：可以用来访问桶中元素的迭代器类型；
    - `const_local_iterator` ：桶迭代器的const版本
    - `c.begin(n)` 、 `c.end(n)` ：桶 `n` 的首元素迭代器和尾元素迭代器
    - `c.cbegin(n)` ， `c.cend(n)` ：上面的const版本
- **哈希策略**
    - `c.load_factor()` ：每个桶的平均元素数量，float类型
    - `c.max_load_factor()` ：最大桶平均元素数量，float类型。当 `load_factor` 过大时，会增加新桶以使 `load_factor <= max_load_factor`
    - `c.rehash(n)` ：重组存储，使得 `bucket_count >= n` 且 `bucket_count > size/max_load_factor`
    - `c.reserve(n)` ：重组存储，使得 `c` 可以保存 `n` 个元素且不必rehash；

容器适配器

适配器：一种机制，可以使某种事物的行为看起来像另外一种事物一样。如stack适配器接受一个顺序容器(除array或forward_list外)，并使其操作起来像一个stack一样

容器，迭代器，函数都有适配器

1.通用的容器适配器操作

- 类型别名
    - size_type

    - value_type：元素类型
    - container_type：实现适配器的底层容器类型
- 构造函数
    - `A a;` ：构建一个空适配器，根据适配器类型选择其默认底层容器类型来实现
    - `A a(c)` ：接受一个容器c来构造适配器a
- 关系运算符：每个适配器都支持所有关系运算符，运算符返回底层容器的比较结果
    - `==` 、 `!=` 、 `<` 、 `<=` 、 `>` 、 `>=`
- 大小
    - `a.empty()`
    - `a.size()`
- 交换： `a` 、 `b` 必须具有相同类型
    - `swap(a,b)`
    - `a.swap(b)`

2.三个顺序容器适配器

- 都能用deque作为底层容器
- 都要求具有添加删除元素的能力，所以底层容器都不能是array。此外，也不能是string
- 每个适配器都基于底层容器类型的操作定义了自己的特殊操作，只可以使用适配器操作而不能使用底层容器的操作

2.1 stack

> 默认底层容器：deque

需要的操作：

- `push_back()`
- `pop_back()`

- `back()`

可用的底层容器：

- 元素的添加删除需求决定了不能用array
- back操作使得不能使用forward_list
- 可以用：vector、list、deque

stack操作：

- 添加元素
  - `s.push(item)`
  - `s.emplace(args)`
- 删除元素
  - `s.pop()` ：弹出(删除)，但不会返回
- 返回元素
  - `s.top()` ：返回栈顶元素，但不弹出(删除)

2.2 queue

> 默认底层容器：deque

需要的操作：

- `push_back()`
- `push_front()`
- `front()`
- `back()`

可用底层容器：

- 元素的添加删除需求决定了不能用array

- back操作使得不能使用forward_list
- 首部添加元素使得不能使用vector
- 可以用：list，deque

queue操作：

- 添加元素
  - `q.push(item)`
  - `q.emplace(args)`
- 返回元素：都不会删除元素
  - `q.pop()` ：返回首元素
  - `q.front()` ：返回首元素
  - `q.back()` ：返回尾元素

2.3 priority_queue

> 默认底层容器：vector

需要的操作：

- `push_back()`
- `pop_back()`
- `front()`
- 随机访问能力

可用底层容器：

- 元素的添加删除需求决定了不能用array
- 删除末尾元素使得不能用forward_list
- 随机访问又不能使用list与forward_list
- 可用用：vector，deque

priority_queue操作：

- 添加元素
  - `q.push(item)`
  - `q.emplace(args)`
- 返回元素：都不会删除元素
  - `q.pop()` ：返回最高优先级元素
  - `q.top()` ：返回最高优先级元素
  - `q.front()` ：返回首元素

泛型算法

2个头文件： `<algorithm>` （大部分）、 `numeric`

主要包括以下几种形式：

- `alg(beg,end,other args)`
- `alg(beg,end,dest,other args)`
- `alg(beg,end,beg2,other args)`
- `alg(beg,end,beg2,end2,other args)`

1.常用算法

1.1 读容器(元素)算法

- **元素查找**

  - `find(b,e,v)` ：在序列中查找 `v` 。查找成功则返回指向它的迭代器，否则返回 `e`
  - `find_if(b,e,f)` ：在序列 `[b,e)` 中查找满足调用对象f的第一个元素，返回其迭代器，不存在则返回 `e`
- **序列累加**
  - `accumulate(b,e,v)` ：位于 `<numeric>` 头文件。第3个参数是和的初值，其类型决定了如何进行 `+` ，类型必须与序列中元素类型匹配。如果使用字符串字面值，由于类型为 `const char*` 没有 `+` 运算，所以不能用于 `string` 相加；
- **序列比较**
  - `equal(b1,e1,b2)` ：由于第二个序列只指定了开头位置，所以假定第二个序列至少和第一个序列一样长，否则会发生错误。可以用于比较不同类型的容器中的元素。元素类型也不必一样，能用 `==` 进行比较即可

1.2 写容器(元素)算法

- **序列赋值**（容易犯的错误是对空容器调用序列赋值函数，序列赋值隐含了大小指定，空容器元素为0。如果不使用插入迭代器，则结果会是未定义的）
  - `fill(b,e,v)` ：要求传入的序列有效
  - `fill_n(dest,n,val)` ：向 `dest` 指向位置开始的 `n` 个元素赋值为 `val`
- **序列拷贝**
  - `copy(b1,e1,b2)` ：将序列 `[b1,e1)` 的元素拷贝至 `b2` 开始的位置，如果b2不是插入迭代器，则需保证 `b2` 后有足够的空间。返回指向最后一个拷贝后一个元素的迭代器
- **元素替换**（前两个参数指定了序列范围， `ov` 是旧值， `nv` 是新值）
  - `replace(b,e,ov,nv)` ：这个版本直接在序列 `[b,e)` 中修改
  - `replace_copy(b1,e1,b2,ov,nv)` ：这个版本将 `[b1,e1)` 修改的结果保存到 `b2` 指向的位置，因此不会改变原序列
- **消除重复**
  - `unique(b,e)` ：将 `[b,e)` 范围中的重复元素移至末尾，返回第一个重复元素的迭代器。范围中的相同元素必须相邻，也就是说先要排好序。不删除元素，只是移到后面。可以根据返回的迭代器调用容器的 `erase()` 操作配合，删除重复元素；
- **元素排序**
  - `sort(b,e)`
  - `sort(b,e,up)` ： `up` 是谓词

- `stable_sort(b,e)` ： `stable_sort` 可以维持相等元素的有序性。如：谓词是比较string长度的函数，则 `stable_sort` 不会改变相同长度string的原始序列
- `stable_sort(b,e,up)`

1.3 for_each算法

`for_each(b,e,f)` ：算法遍历序列，对序列中的每一个元素调用f指定的操作

2.迭代器

头文件： `<iterator>` 。泛型算法中，通常需要传递几个迭代器来表示序列范围； 由于不是传递容器作为泛型算法的参数，使得泛型算法不依赖于具体容器

2.1 按类型划分

1) 插入迭代器

- 特点
  - 一种迭代器适配器，故类型包含容器类型
  - 赋值时会调用容器的操作添加元素
  - 如果泛型算法会添加元素，则应该使用插入迭代器
  - 只支持递增(但是不做任何事)，不支持递减
- `back_insert_iterator`
  - 使用 `push_back` (支持的容器才能使用)
  - `back_inserter(c)`
  - 始终在尾部插入
- `front_insert_iterator`
  - 使用 `push_front` (只有支持的容器才使用)

    - `front_inserter(c)`
    - 始终在首部插入
- `insert_iterator`
    - 使用 `insert`
    - `inserter(c,iter)`
    - 插入 `iter` 前，返回 `iter`

2）流迭代器

将对应的流当作一个特定的元素序列来处理

- 特点
    - 只支持递增不支持递减， `ostream_iterator` 递增不做任何事
    - 一次性访问
- `istream_iterator`
    - 绑定输入流，读取数据
    - `istream_iterator<T> it(cin)`
    - 尾后迭代器： `istream_iterator<T> eof` （空的 `istream_iterator` ）
    - 使用输入迭代器读取输入初始化vector： `vector<int> vec(it,eof)`
    - 懒惰求值：绑定到输入时不会理解从流数据，使用时才真正读取
- `ostream_iterator`
    - 绑定输出流，输出数据
    - `ostream_iterator<T> it(cout)` 或 `ostream_iterator<T> it(cout，c_str)` ：可以提供一个c风格字符串，每次输出一个元素都会打印这个字符串
    - 没有尾后迭代器（没有空的 `ostream_iterator` ）
    - 使用输出迭代器： `copy(vec.begin(),vec.end(),it)`

3）容器迭代器

- 普通迭代器
- 反向迭代器
  - `reverse_iterator`
  - `c.rbegin()` ， `c.rend()` ， `c.crbegin()` ， `c.crend()` 都会返回反向迭代器
  - 容器必须支持 `++` 和 `--` （除了forward_list外）
  - 注意反向迭代器的范围不对称
    - 输出时，结果反过来
    - 使用普通迭代器初始化(或赋值)时，增减序列相反

4）移动迭代器

- 特点
  - 一种迭代器适配器
  - 移动迭代器的解引用运算符生成一个右值：一般迭代器的解引用返回一个指向元素的左值
  - `make_move_iterator()` 函数将一个普通迭代器转换为一个移动迭代器：原迭代器的所有其它操作在移动迭代器中都照常工作，因此，可以将一对移动迭代器传递给算法。但是标准库并不保证哪些算法适用移动迭代器，哪些不适用；

2.2 按操作级别划分

泛型算法通常对迭代器操作级别有要求，必须使用大于等于要求级别的迭代器

高层类别的迭代器支持低层类别的迭代器，越往下越高(支持的操作越多)

- **输入迭代器**： `istream_iterator`
  - 只读，不写；单遍扫描，只能递增
- **输出迭代器**： `ostream_iterator`
  - 只写，不读；单遍扫描，只能递增
  - 与输入迭代器互为功能上的补集
- **前向迭代器**：forward_list的迭代器
  - 可读写，多遍扫描；只能递增

- **双向迭代器**：list的迭代器(普通与反向)
  - 可读写，多遍扫描；可递增递减
- **随机访问迭代器**：array，deque，string，vector的迭代器
  - 可读写，多遍扫描，支持全部迭代运算；
  - 随机说明可以迭代器支持 `+n`，`-n` ；

3.调用对象

3.1 谓词

谓词是一个可调用的表达式(如函数)，返回结果是一个能用作条件的值

- 一元谓词：只接受一个参数
- 二元谓词：只接受两个参数

3.2 lambda

> lambda是一种可调用对象，可理解为一种未命名内联函数。除此之外，可调用对象还有：函数、函数指针、函数对象、bind绑定的对象

头文件：`<functional>`

用途：

- 某些泛型算法只能接受某种谓词，谓词限制了参数个数
- 为了对序列执行某一操作，可能需要的参数大于谓词的限制

形式：`[capture list] (parameter list) -> return type {function body}`

- **捕获列表（必须有）**

    - 捕获列表就是用于向lambda传递我们原本想使用的参数。这些参数在函数体中会用到
    - 捕获列表中可以包含lambda所在函数的局部变量
    - lambda对静态局部变量和所在函数体外的名字可见。非静态局部变量传入捕获列表使得lambda可见
- **参数列表（可省略）**
    - 和函数的参数不同，lambda不能使用默认实参
- **箭头（可省略）**
    - 当函数体不只return一条语句时，默认返回 `void` ，如果想返回其它类型必须指明返回类型。当指明返回类型时，箭头不省略，即必须使用尾置返回类型
- **返回类型（可省略）**
    - 如果省略了返回类型，则通过下列方式判断返回类型
        - 1）函数体内的返回语句
        - 2）返回的表达式的类型
        - 3）如果lambda的函数体包含任何单一return语句之外的内容，且未指定返回类型，则返回void
- **函数体（必须有）**

捕获

- **显示捕获**
    - **值捕获**
        - 前提是变量可拷贝
        - 创建lambda时拷贝，而不是调用时
        - 随后修改不会影响lambda内对应的值
    - **引用捕获**（不推荐。引用可扩展开，包括指针，迭代器）
        - `&`局部变量名
        - lambda使用时实际是使用引用绑定的对象
        - 随后的修改会影响lambda内对应的值（如果引用的是一个const对象，则不能修改）
        - 捕获的是局部变量，必须确保引用的对象在lambda执行时存在（如果lambda可能在函数结束后执行，捕获的引用指向的局部变量已经消失）
        - 捕获某些对象必须使用引用捕获(I/O)

    - **修改捕获对象**
        - 通过值拷贝时，在参数列表后加mutable 可以改变其值。默认情况下，对于一个值被拷贝的变量，lambda不会改变其值。改变的只是lambda拷贝的对象，与原对象无关
- **隐式捕获**
    - 让编译器推断捕获列表
    - `&` ：告诉编译器采用引用捕获的方式
    - `=` ：告诉编译器采用值捕获的方式
- **显隐混合捕获**
    - 捕获列表第一个元素必须是 `&` 或 `=`
    - 显示捕获的变量必须使用与隐式捕获不同的方式

lambda对象可以作为返回值，如果函数返回一个lambda，则与函数不能返回一个局部变量的引用类似，此lambda也不能包含引用捕获

3.3 bind参数绑定

> 标准库中的一个函数。可以看作一个通用的函数适配器

头文件： `<functional>`

用途：

- 与lambda相同
- 解决谓词参数限制的另一种途径

形式： `auto newCallable = bind(callable,arg_list)`

- `callable` 为参数受限（受泛型算法限制）的谓词（谓词为可调用表达式，如函数）

当调用 `newCallable` 时， `newCallable` 会将 `arg_list` 传递给 `callable` ，调用 `callable` （ `callable` 函数的参数顺序与 `arg_list` 一致）

参数列表（ `arg_list` ）

- **占位符**：占位符定义在名为 `placeholders` 的命名空间，此命名空间又定义在 `std` 命名空间，所以使用时两个命名空间都要写上： `using namespace std::placeholders`； 通常是泛型算法对参数的限制来决定有多少个占位符
  - `_n` ， `n` 为整数，表示调用 `newCallable` 时的第 `n` 个参数。可以无序，即_2在_1前
- **变量**
  - 有些变量不能传值(I/O)，所以需要绑定引用
    - ref(os)：返回绑定对象的引用
    - cref(os)：返回绑定对象的const引用

4.链表的算法

通用泛型算法的问题：

- 通用sort要求随机访问，链表不支持
- 其它通用算法对链表来说代价太高
  - 链表指针修改更快；
  - 因此链表优先选择成员函数版本的算法
- **与泛型算法对应的算法**
  - `ls.merge(ls2)`
  - `ls.merge(ls2,comp)`
  - `ls.remove(val)`
  - `ls.remove_if(pred)`
  - `ls.reverse()`
  - `ls.sort()`

    - `ls.sort(comp)`
    - `ls.unique()`
    - `ls.unique(pred)`

- **链表独有的算法**

    - `ls.splice(args)`
        - `args` ：（代码）
            - `(p,ls2)` ：将链表 `ls2` 插入 `ls` 中 `p` 指向的元素前
            - `((p,ls2,p2)` ：将 `ls2` 中 `p2` 指向的元素插入 `ls` 中 `p` 指向的元素前
            - `((p,ls2,b,e)` ：将 `ls2` 中 `[b,e)` 范围的元素插入 `ls` 中 `p` 指向的元素前

模板与泛型编程

1.模板函数

```
template <typename T>
int compare (const T &v1,const T &v2)
{
    if(v1 < v2) return -1;
    if(v2 < v1) return 1;
    return 0;
}
```

当调用一个函数模板时，编译器(通常)用函数实参来为我们推断模板实参。编译器用推断出的模板参数为我们实例化一个特定版本的函数，这些编译器生成的版本通常被称为模板的实例

上面的模板函数说明了编写泛型代码的两个重要原则：

1. 模板中的函数参数是const的引用（保证了函数可以用于不能拷贝的类型。同时，如果compare用于处理大对象，这种设计策略还能使函数运行得更快）
2. 函数体中的条件判断仅使用<比较运算（如果编写代码时只使用<运算符，就降低了compare函数对要处理的类型的要求。这种类型必须支持<，但不必支持>。实际上，如果真的关系类型无关和可移植性，应该用less，因为<无法比较指针，但是less可以）

> 函数模板可以声明为inline或constexpr的，如同非模板函数一样。inline或constexpr说明符放在模板参数列表之后，返回类型之前

1.1 模板参数

- 在模板定义中，模板参数列表不能为空
- 模板参数的名字没有什么内在含义，通常将类型参数命名为T，但实际上可以使用任何名字
- 一个模板参数名的可用范围是在其声明之后，至模板声明或定义结束之前。模板参数会隐藏外层作用域中声明的相同名字，模板内不能重用模板参数名
- 与函数参数相同，声明中的模板参数的名字不必与定义中相同；
- typename和class并没有什么不同，typename可能更直观，因为class可能会让人觉得能使用的类型必须是类类型

1）模板类型参数

用来指定返回类型或函数类型，以及在函数体内用于变量声明或类型转换

```
//T用作了返回类型、参数类型、变量类型
template <typename T> T foo (T* p)
{
    T tmp = *p;
    //...
    return tmp;
}
```

2）非类型模板参数

```
template<unsigned N,unsigned M>
int compare(const char (&p1)[N], const char (&p2) [M])
{
    return strcmp(p1,p2);
}
```

- 第一个非类型模板参数表示第一个数组的长度
- 第二个非类型模板参数表示第二个数组的长度

当调用这个模板时， `compare("hi","mom");` 编译器会使用字面常量的大小来代替 `N` 和 `M` ，从而实例化模板

非类型模板参数包括：

- **整形**：绑定到非类型整形参数的实参必须是一个常量表达式
- **指针或引用**：绑定到指针或引用非类型参数的实参必须具有静态的生存期，不能用一个普通局部变量或动态对象作为指针或引用非类型模板参数的实参。指针也可以用nullptr或一个值为0的常量表达式来实例化

1.2 函数形参

模板函数的**形参**中可以含有正常类型。即，不一定全必须是模板类型：

```
template <typename T> ostream &print(ostream &os,const T &obj)
{
    return os << obj;
}
```

1.3 成员模板

1）普通类的成员模板

```
class DebugDelete {
public:
    DebugDelete(std::ostream &s = std::cerr) : os(s) { }
    template <typename T> void operator( ) (T *p) const
    {os << "deleting unique_ptr" << std::endl;delete p;}
private:
    std::ostream &os ;
};
```

2) 类模板的成员模板

类和成员各自有自己的独立的模板参数

```
template <typename T> class Blob {
    template <typename It> Blob(It b,It e);
}
```

当在类外定义成员模板时，必须同时为类模板和成员模板提供模板参数：

```
template <typename T>
template <typename It>
Blob<T>::Blob(It b,It e) : data(...) {...}
```

实例化成员模板：

```
int ia[ ] = {0,1,2,3,4,5,6,7,8,9};
vector<long> vi = {0,1,2,3,...};
list<const char*> w = {"now","is","the"};
Blob<int> a1(begin(ia),end(ia));
Blob<int> a2(vi.begin( ),vi.end( ));
Blob<string> a3(w.begin( ),w.end( ));
```

2.类模板

```
template <typename T> class Blob {
    //typename告诉编译器size_type是一个类型而不是一个对象
    typedef typename std::vector<T>::size_type size_type
    //...
};
```

一个类模板的每个实例都形成一个独立的类：

```
Blob<string> names;
Blob<double> prices;
```

2.1 与模板函数的区别

- **编译器不能为类模板推断模板参数类型**
- 使用时必须在模板名后的尖括号中提供额外信息

2.2 模板类名的使用

1）类内使用不需要指明

```
BlobPtr& operator++( );
```

当处于一个类模板的作用域中时，编译器处理模板自身引用时就好像我们已经提供了与模板参数匹配的实参一样

2）类外使用需要指明

```
template <typename T>
BlobPtr<T> BlobPtr<T>::operator++(int)
{
 //...
}
```

由于位于类作用域外，必须指出返回类型是一个实例化的BlobPtr，它所用类型与类实例化所用类型一致

2.3 类模板的成员函数

类外定义成员函数时要加 `template<typename T>` 。类模板的成员函数具有和模板相同的模板参数。因此，定义在类模板之外的成员函数就必须以关键字 `template` 开始，后接类模板参数列表：

```
template <typename T>
ret-type Blob<T>::member-name(parm-list)
```

对于一个实例化了的类模板，其成员函数只有当程序用到它时才进行实例化

```
//实例化Blob<int>和接受initializer_list<int>的构造函数
Blob<int> squares = {0,1,2,3,4,5,6,7,8,9};
```

如果一个成员函数没有被使用，则它不会被实例化，成员函数只有在被用到时才会进行实例化，这一特性使得即使某种类型不能完全符合模板操作的要求，我们仍然能用该类型实例化类

2.4 类型成员

假定T是一个模板类型参数，当编译器遇到类似T::mem这样的代码时，它不会知道mem是一个类型成员还是一个static数据成员，直至初始化时才会知道。但是，为了处理模板，编译器必须知道名字是否表示一个类型。例如，假定T是一个类型参数的名字，当编译器遇到如下形式的语句时：

```
T::size_type *p;
```

它需要知道我们是整在定义一个名为 p 的变量还是一个名为 size_type 的static数据成员与名为 p 的变量相乘

默认情况下，C++假定通过作用域运算符访问的名字不是类型。因此，如果我们希望使用一个模板类型参数的类型成员，就必须显示告诉编译器该名字是一个类型。通过关键字 typename 来实现这一点

2.5 类模板和友元

1）普通类中将另一模板类声明为友元

```
template <typename T> class Pal;

class C {
    //用类C实例化的Pal是C的一个友元
    friend class Pal<C>;
    //Pal2的所有实例都是C的友元
    template <typename T> friend class Pal2;
};
```

2）模板类中将另一模板类声明为友元

```
template <typename T> class Pal;

template <typename T> class C2 {
    //C2的每个实例将相同实例化的Pal声明为友元
    friend class Pal<T>;
    //Pal2的所有实例都是C2的每个实例的友元
    template <typename X> friend class Pal2;
};
```

为了让所有实例成为友元，友元声明中必须使用与类模板本身不同的模板参数（上面的X）

3）令模板自己的类型参数成为友元

```
template <typename T> class Bar{
    //将访问权限授予用来实例化Bar的类型
    friend T;
};
```

对于某个类型Foo，Foo将成为Bar的友元...

2.6 模板类型别名

类模板的一个实例化定义了一个类类型，可以定义一个typedef来引用实例化的类：

```
typedef Blob<string> StrBlob;
```

由于模板不是一个类型，所以不能定义一个typedef引用一个模板。即，无法定义一个typedef引用 `Blob<T>`

但是，**新标准**允许我们为类模板定义一个类型别名：

```
template <typename T> using twin = pair<T,T>;
twin<string> authors; //authors是一个pair<string,string>；
```

定义一个模板类型别名时，可以固定一个或多个模板参数；

```
template <typename T> using partNo = pair<T,unsigned>；
partNo<string> books; //pair<string,unsigned>；
```

2.7 类模板的static成员

- **static属于每个实例化的类类型，而不是类模板。即，每个实例化的类都有一个自己对应的static成员**
- **模板类的每个static成员必须有且仅有一个定义。但是，类模板的每个实例都有一个独有的static对象**

```
template <typename T>
size_t Foo<T>::ctr = 0;
```

可通过类类型对象或作用域运算符访问：

```
Foo<int> f1;
auto ct = Foo<int>::count( );
ct = f1.count( );
```

只有使用时才会实例化

3.模板编译

- 遇到模板时不生成代码，实例化时生成代码
- 函数模板和类模板成员函数的定义通常放在头文件中
- 实例化冗余：当模板被使用时才会进行实例化这一特性意味着，相同的实例可能出现在多个对象文件中。当两个或多个独立编译的源文件使用了相同的模板，并提供了相同的模板参数时，每个文件中就都会有该模板的一个实例

3.1 实例化声明

形式： `extern template declaration`

```
extern template class Blob<string>;
extern template int compare(const int&,const int&);
```

当遇到extern模板声明时，不会在本文件中生成实例化代码。将一个实例化声明为extern就表示承诺在程序其他位置有该实例化的一个定义。对于一个给定的实例化版本，可能有多个extern声明，但必须只有一个定义

- 实例化声明可以有多个：即多个源文件可能含有相同声明
- 实例化声明必须出现在任何使用此实例化版本的代码之前。因为编译器在使用一个模板时会自动对其实例化

3.2 实例化定义

```
template declaration
template int compare(const int &,const int&);
template class Blob<string>;
```

- **类模板的实例化定义会实例化该模板的所有成员**
- **所用类型必须能用于模板的所有成员**：与处理类模板的普通实例化不同，编译器会实例化该类的所有成员。即使我们不使用某个成员，它也会被实例化。因此，我们用来显式实例化一个类模板的类型，必须能用于模板的所有成员

4.模板参数

4.1 默认模板实参

为模板提供默认类型

1）模板函数

```
template <typename T,typename F = less<T>>
int compare(const T &v1,const T &v2,F f = F( ))
{
    if(f(v1,v2)) return -1;
    if(f(v2,v1)) return 1;
```

```
    return 0;
}
```

和函数默认实参一样，所有提供了默认实参的形参右边的形参都需要提供默认实参

2）类模板

```
template <class T = int> class Numbers {
public:
    Numbers(T v = 0) : val(v) { }
    private:
    T val;
};

Numbers<long double> lots_of_precision;
Numbers<> average_precision;            //空<>表示希望使用默认类型；
```

4.2 模板实参推断

1）函数模板的参数转换

- **模板类型参数的类型转换**：将实参传递给带模板类型的函数形参时，能够自动应用的类型转换只有**const转换**及**数组或函数到指针的转换**
  - **const的转换**
    - 可以将一个const对象传递给一个非const的非引用形参

```
template <typename T> fobj(T,T);
string s1("a value");
const string s2("another value");
fobj(s1,s2); //正确；
```

fobj调用中，传递了一个string和一个const string。虽然这些类型不严格匹配，但两个调用都是合法的，由于实参被拷贝，因此原对象是否是const没有关系；

- 可以将一个非const对象的引用(或指针)传递给一个const的引用(或指针)形参

```
template <typename T> fref(const T&,const T&);
string s1("a value");
const string s2("another value");
fref(s1,s2); //正确；
```

在fref调用中，参数类型是const的引用。对于一个引用参数来说，转换为const是允许的，因此合法；

- **非引用类型形参可以对数组或函数指针应用正常的指针转换**

```
template <typename T> fobj(T,T);
template <typename T> fref(const T&,const T&);
int a[10],b[42];
fobj(a,b); //调用fobj(int*,int*)
fref(a,b); //错误，数组类型不匹配；
```

在fobj调用中，数组大小不同无关紧要，两个数组都被转换为指针。fobj中的模板类型为Int*；但是，fref调用是不合法的，如果形参是一个引用，则数组不会转换为指针。a和b的类型不匹配

- **普通类型参数的类型转换**：模板函数可以有用普通类型定义的参数，即，不涉及模板类型参数的类型。这种函数实参不进行特殊处理，这些实参执行正常类型的转换

2）显示实参

为什么需要显示实参？编译器无法推断出模板实参的类型。假设定义如下模板:

```
template <typename T>
T sum(T,T);
```

则调用sum时，必须要求传入相同类型的参数，否则会报错。因此可以按这种方式定义模板：

```
template <typename T1,typename T2,typename T3>
T1 sum(T2,T3);
```

但是，这种情况下，无论传入什么函数实参，都无法推断T1的类型。因此，每次调用sum时，调用者必须为T1提供一个显示实参：

```
auto val3 = sum<long long>(i,lng);
```

这个调用显示指定了T1的类型，而T2和T3的类型则由编译器从i和lng的类型判断出来

显示实参配对顺序：由左至右。只有尾部参数的显示模板实参可以忽略，但必须能推断出来

因此，如果按找这种形式定义模板：

```
template <typename T1,typename T2,typename T3>
T3 sum(T2,T1);
```

则总是必须为所有三个形参指定参数。希望控制模板实例化

对于sum模板，如果保留原有的设计： `template T sum(T,T)` 则当函数调用传入不同类型的参数时，我们必须放弃参数类型推断，采取控制模板实例化的方式来调用： `sum<int>(long,1024);` 这种情况下，会实例化一个 `int sum(int,int)` 的函数，传入的参数都会按照内置类型的转换规则转换为int

3）尾置返回类型与traits

当我们希望用户确定返回类型时，用显示模板实参表示模板函数的返回类型是很有效的。在其他情况下，要求显示指定模板实参会给用户增添额外负担，而且不会带来什么好处：

```
template <typename It>
??? &fcn(It beg,It end)
{
```

```
    //处理序列
    return *beg;
}
```

在这个例子中，并不知道返回结果的准确类型，但知道所需类型是所处理的序列的元素类型;我们知道函数应该返回 `*beg` ，可以使用 `decltype(*beg)` 来获取此表达式的类型。但是在编译器遇到函数的参数列表之前， `beg` 是不存在的。所以必须使用尾置类型：

```
template <typename It>
auto fcn(It beg,It end) -> decltype(*beg)
{
    //处理序列
    return *beg;//返回序列中一个元素的引用
}
```

也可以使用标准库的类型转换模板。可以使用 `remove_reference` 来获得元素类型。这个模板有一个模板类型参数和一个名为type的成员。如果用一个引用类型实例化这个模板，则type将表示被引用的类型。如果实例化 `remove_reference<int&>` ，则type成员将是int。因此，可以通过下列模板满足需求：

```
template <typename It>
auto fcn2(It beg,It end) ->
typename remove_reference<decltype(*beg)>::type
{
     //处理序列
    return *beg;
}
```

4）函数指针和实参推断

用一个函数模板初始化一个函数指针或为一个函数指针赋值时，编译器使用指针的类型来推断模板实参

```
template <typename T> int compare(const T&,const T&);
int (*pf1)(const int&,const int&) = compare;
```

pf1中参数的类型决定了T的模板实参的类型。如果不能从函数指针类型确定模板实参，则产生错误：

```
void func(int(*)(const string&,const string&));
void func(int(*)(const int&,const int&));
func(compare); //错误，使用那个实例?
```

对于这种情况，可以使用显示模板实参：

```
func(compare<int>);
```

5）引用与实参推断

非常重要的是记住两个规则：

1. 编译器会应用正常的引用绑定规则；
2. const是底层的，不是顶层的；

当一个函数的参数是模板类型参数的一个普通(左值)引用时，绑定规则告诉我们，只能传递给它一个左值：

```
template <typename T> void f1(T&);
f1(i);  //i是int，T推断为int；
f1(ci); //ci是const int，T推断为const int；
f1(5);  //错误
```

如果参数类型是const T&，正常的绑定规则告诉我们可以传递给它任何类型的实参：一个对象，临时对象或字面值常量：

```
template <typename T> void f2(const T&);
f2(i); //i是int，T推断为int；
f2(ci); //ci是const int，但T推断为int；
f2(5); //T推断为int；
```

当参数是一个右值引用时，正常绑定规则告诉我们可以传递给它一个右值：

```
template <typename T> void f3(T&&);
f3(42); //实参是int型的右值，T推断为int；
```

引用折叠：

1. 如果将一个左值传递给函数的右值引用参数，且此右值引用指向模板类型参数(如:T&&)时，编译器推断模板的类型参数为左值引用类型 2. 如果因为1.间接的创建了一个引用的引用，则引用形参了“折叠”、则： * 右值引用的右值引用会被折叠成右值引用 * 其它情况下都折叠成左值引用

因此，对于前面的f3：

```
f3(i); //i是左值，T推断为int&，T&&被折叠成int &；
f3(ci); //ci是左值，T是const int&；
```

因此，如果模板参数类型为右值引用，可以传递给它任意类型的实参

右值引用的问题：因为可以传递任意实参，引用折叠会导致T被推断为引用或非引用类型，所以函数内使用这个类型在传入不同参数时可能产生不同结果，此时，编写正确的代码就变得异常困难；

右值引用的使用场景：因为上述问题，所以右值引用主要应用于两个场景

1. **模板转发其实参**：当使用右值引用作为模板参数时，如果T被推断成普通类型(即非引用)，可以通过std::forward保持其右值属性，会返回一个T&&。如果被推断成一个(左值)引用，通过引用折叠，最终也还是会返回T&；因此，当用于一个指向模板参数类型的右值引用函数参数(T&&)时，forward会保持实参类型的所有细节

2. **模板被重载**

5.重载与模板

包含模板的函数匹配规则：

1. **候选函数包括所有模板实参推断成功的函数模板实例**

```
template <typename T> string debug_rep(const T &t) {...}
template <typename T> string debug_rep(T *p) {...}
string s("hi");
//第二个模板实参推断失败，所以调用第一个模板；
cout << debug_rep(s) << endl;
```

2. **可行函数按类型转换来排序**
3. **如果恰好有一个比其他提供更好的匹配则使用该函数**

```
template <typename T> string debug_rep(const T &t) {...}
template <typename T> string debug_rep(T *p) {...}
string s("hi");
//两个模板都能匹配：
//第一个模板实例化debug_rep(const string*&)，T被绑定到string*；
//第二个模板实例化debug_rep(string*)，T被绑定到string；
//但由于第一个实例化版本需要进行普通指针到const指针的转换，所以第二个更匹配；
cout << debug_rep(&s) << endl;
```

4. **如果有多个函数提供"同样好的"匹配**
   - 同样好的函数中只有一个是非模板函数，则选择此函数

```
template <typename T> string debug_rep(const T &t) {...}
template <typename T> string debug_rep(T *p) {...}
string debug_rep(const string &s) {...}
```

```
string s("hi");
//以下调用有两个同样好的可行函数：
//第一个模板实例化debug_rep<string>(const string &)，T被绑定到string；
//非模板版本debug_rep(const string &s)；
//编译器会选择非模板版本，因为最特例化；
cout << debug_rep(s) << endl;
```

- 同样好的函数中全是模板函数，选择更“特例化的模板”

```
template <typename T> string debug_rep(const T &t) {...}
template <typename T> string debug_rep(T *p) {...}
string s("hi");
const string *sp = &s;
//以下调用两个模板实例化的版本都能精确匹配：
//第一个模板实例化debug_rep(const string *&)，T被绑定到string*；
//第二个模板实例化debug_rep(const string *)，T被绑定到const string；
//我们可能觉得这个调用是有歧义的。但是，根据重载函数模板的特殊规则，调用被解析为debug_rep(T*)，即更特例化的版本
//如果不这样设计，将无法对一个const的指针调用指针版本的debug_rep。
//问题在于模板debug_rep(const T&)本质上可以用于任何类型，包括指针类型。此模板比debug_rep(T*)更通用，后者只能
cout << debug_rep(sp) << endl;
```

- 否则，调用有歧义

6.可变参数模板

参数包：

- **模板参数包**： `template<typename T,typename... Args>` Args为模板参数包，class...或typename...指出接下来的参数表示零个或多个类型的列表，一个类型名后面跟一个省略号表示零个或多个给定类型的非类型参数的列表；
- **函数参数包**

```
template <typename T,typename... Args>
void foo(const T &t,const Args& ... rest);
```

rest为函数参数包

使用参数包：

- `sizeof...` 获取参数包大小。可以使用 `sizeof...` 运算符获取包中元素的数目
- **扩展包**：扩展一个包就是将包分解为构成的元素，对每个元素应用模式，获得扩展后的列表，通过在模式右边放一个省略号来触发扩展操作：

```
template <typename T,typename... Args>
ostream& print(ostream &os,const T &t,const Args&... rest) //扩展Args
{
    os << t << ", ";
    return print(os,rest...); //扩展rest
}
```

扩展中的模式会独立地应用于包中的每个元素：

```
debug_res(rest)... 是对包rest的每一个元素调用debug_res；
debug_res(rest...) 是调用一个参数数目和类型与rest中元素匹配的debug_rest；
```

转发包参数：

新标准下，可以组合使用可变参数模板与forward机制来编写函数，实现将其参数不变地传递给其他函数：

```
template <typename... Args>
void fun(Args&&... args) //将Args扩展为一个右值引用的列表
{
    //work的实参既扩展Args又扩展args
```

```
    work(std::forward<Args>(args)...);
}
```

7.模板特例化

编写单一模板，使之对任何可能的模板实参都是最合适的，都能实例化，这并不总是能办到。当我们不能（或不希望）使用模板版本时，可以定义类或函数模板的一个特例化版本

一个特例化版本本质上是一个实例，而非函数名的一个重载版本。因此，特例化不影响函数匹配；

- **函数模板特例化**

```
template <typename T> int compare(const T&,const T&);
//compare函数模板的通用定义不适合字符指针的情况，
//我们希望compare通过strcmp比较两个字符指针而非比较指针值；
template <>
int compare(const char* const &p1,const char* const &p2)
{
    return strcmp(p1,p2);
}
```

当定义一个特例化版本时，函数参数类型必须与一个先前声明的模板中对应的类型匹配。这个特例化版本中，`T`为`const char*`

- **类模板特例化**

```
template <>
struct 模板类名<Sales_data>
{
    ...
}
```

定义了某个模板能处理Sales_data的特例化版本

- **类模板（偏特化）部分特例化**：与函数模板不同，类模板的特例化不必为所有模板参数提供实参。可以只提供一部分而非所有模板参数，或是参数的一部分而非全部特性。部分特例化本身是一个模板，**部分特例化版本的模板参数列表是原始模板的参数列表的一个子集或者是一个特例化版本**

```
//原始的，最通用的版本
template <class T> struct remove_reference
 { typedef T type; };
//部分特例化版本，将用于左值引用和右值引用
template <class T> struct remove_reference<T&>
{ typedef T type; };
template <class T> struct remove_reference<T&&>
{ typedef T type; };
//用例
int i;
remove_reference<decltype(42)>::type a;           //decltype(42)为int，使用原始模板；
remove_reference<decltype(i)>::type b;            //decltype(i)为int&，使用第一个部分特例化版本；
remove_reference<decltype(std::move(i))>::type c; //decltype(std::move(i))为int&&，使用第二个部分特例化版本；
```

- **特例化成员而非类**

```
template <>
void Foo<int>::Bar( )
{
    //进行应用于int的特例化处理；
}
Foo<string> fs; //实例化Foo<string>::Foo( );
fs.Bar( );      //实例化Foo<string>::Bar( );
Foo<int> fi;    //实例化Foo<int>::Foo( );
fi.Bar( );      //使用特例化版本的Foo<int>::Bar( );
```

内存管理

1.new和delete

1.1 new

1) 动态分配单个对象

初始化：

```
int *pi1 = new int;          //默认初始化
int *pi2 = new int();        //值初始化
int *pi2 = new int(1024);    //直接初始化

string *ps = new string(10,'9');

//若obj是一个int，则p1是int*；
//不能用{...}代替(obj)包含多个对象；
auto p1 = new auto(obj);
```

动态分配const对象：

- 必须进行初始化
- 不能修改指向的对象，但是能delete(销毁)这个动态分配的const对象

```
const int *pci = new const int(1024);
const string *pcs = new const string;   //隐式初始化
```

内存耗尽：

- 内存不足时，new会失败
- 抛出类型为bad_alloc的异常
- `new (nothrow) T` 可以阻止抛出异常（定位new）

2）动态分配多个对象

使用注意：

- 大多数应用应该使用标准库容器而不是动态分配的数组
- 动态分配数组的类必须定义自己版本的拷贝，复制，销毁对象的操作

理解：

- 通常称 `new T[]` 分配的内存为"动态数组"某种程度上有些误导
- 返回的并不是一个"数组类型"的对象，而是一个"数组元素类型"的指针
- 即使使用类型别名也不会分配一个数组类型的对象

不能创建大小为0的动态数组，但当 `[n]` 中 `n` 为0时，是合法的。此时new返回一个合法的非空指针，次指针保证与new返回的其它任何指针都不同，就像尾后指针一样，可以进行比较操作，加0，减0，不能解引用

初始化：

```
int *pia = new int[get_size()];      //维度不必是常量，但是必须是整形
int *p1 = new int[42];               //未初始化
//以下为上一行的等价调用
typedef int arrT[42]；
int *p = new arrT;

int *p2 = new int[42]();             //值初始化

//初始值列表中没有给定初始值的元素进行"值初始化"，如果初始值列表中元素超出，new会失败
int *p3 = new int[5]{1,2,3,4,5};
```

1.2 delete

- delete单个对象： `delete p;`
- delete动态数组： `delete [] pa;`
  - 不管分配时有没有用类型别名，delete时都要加上 `[]`
  - 逆序销毁
  - `[]` 指示编译器指针指向的是一个数组的首元素

注意：

- 不要delete非new分配的对象
- 不要重复delete
- 可以delete空指针
- 可以delete动态分配的const对象

> 通常情况下，编译器不能分辨一个指针指向的是静态还是动态分配的对象。类似的，编译器也不能分辨一个指针所指向的内存是否已经被释放了。对于这些delete表达式，大多数编译器能通过，尽管它们是错误的。这些错误delete的结果是未定义的

空悬指针：指向原本存在数据现在已经无效的内存的指针

- 当delete一个动态分配的对象后，原本指向这个对象的指针就变成了空悬指针
- 防止使用空悬指针（只能保证这个指针不会再访问无效内存，但是可能也还有其它指针也指向这块动态分配的内存，它们在delete后也可能会访问）
  - 在即将离开指针作用域时delete：这样之后，当离开作用域后这个指针就销毁了，而在delete前，指针指向的内存是有效的
  - delete后赋值为空指针nullptr

2.智能指针

2.1 通用操作

> 以下操作支持shared_ptr和unique_ptr

- 创建

```
//默认初始化，保存一个空指针
shared_ptr<T> sp;
unique_prt<T> up;
```

- 作为条件： `p`
- 访问指向的对象： `*p`
- 获取保存指针： `p.get()`
  - 不要delete get()返回的指针，假设delete没问题，在引用计数为0时，智能指针会重复delete
  - 如果p是shared_ptr，不要用get()返回的指针初始化另一个shared_ptr，这样不会递增引用计数，当新建智能指针销毁后，这个动态对象就被释放了
- 交换

```
swap(p,q);
p.swap(q);
```

2.2 shared_ptr

1）创建：

- 调用函数make_shared
  - `make_shared<T>(args)` ：推荐使用这种方式。args用于初始化指向的对象，不传参数时”值初始化“

```
shared_ptr<int> p1 = make_shared<int>(42); //动态对象初始化为42
shared_ptr<string> p2 = make_shared<string>(10,'9');  //动态对象初始化为"9999999999"
shared_ptr<int> p3 = make_shared<int>(); //动态对象值初始化，0
```

- 使用构造函数
  - `shared_ptr<T> p(q)`
    - **q为shared_ptr时，会递增q的引用计数**
    - **构造函数为explicit，如果q不是一个智能指针，必须直接初始化，此时q必须能转换为T***，如 `shared_ptr<int> p(new int(1024))`
    - **如果q不是一个指向动态内存的指针，须自定义释放操作**（shared_ptr默认使用delete释放所指动态对象，如果指针不指向动态内存，不能delete）
    - **q不是智能指针时，这种方式构建临时shared_ptr很危险**（比如一个函数参数为shared_ptr，由于explicit，因此不能隐式转换。如果q是new int创建的内置类型指针，则可能通过这个构造函数创建一个临时shared_ptr来满足调用要求，这样的话当函数返回时，两个shared_ptr(形参与实参)都被销毁，所以函数外部原本指针指向的动态对象会被释放掉，在函数调用之后再使用就是空悬指针，因此，最好使用make_shared来创建智能指针）
  - `shared_ptr<T> p(q,d)` ：d是可调用对象，用于代替delete执行释放操作，在这里q可以不指向动态内存
  - `shared_ptr<T> p(p2,d)` ：p是shared_ptr p2的拷贝，但是使用可调用对象d代替delete执行释放操作
  - `shared_ptr<T> p(u)` ：从unique_ptr u那里接管了对象的所以权，将u置为空

2）赋值

```
p = q;  //递增q引用计数，递减p引用计数
```

3）重置

```
// 1）若p是唯一指向其对象的shared_ptr，则释放对象；
// 2）将p置为空；
p.reset();
```

```
// 1）若p是唯一指向其对象的shared_ptr，则释放对象；
// 2）p = q；
p.reset(q);

// 1）若p是唯一指向其对象的shared_ptr，则释放对象；
// 2）p = q；
// 3）d代替delete执行释放操作；
p.reset(q,d);
```

4）状态

```
//返回与p共享对象的智能指针数量；可能很慢，主要用于调试
p.use_count();

//若use_count()为1则返回true，否则返回false
p.unique();
```

2.3 unique_ptr

1）初始化

```
unique_ptr<T> u1;           //创建一个空的unique_ptr
unique_ptr<T D> u2;         //D为自定义释放操作的类型
//D为自定义释放操作的类型，d为自定义释放操作的指针。这里没有传入指针参数，是一个空unique_ptr
unique_ptr<T,D> u(d);
unique_ptr<T,D> u(T*,d);
```

2）赋值与拷贝

只有在unique_ptr即将销毁时才能赋值或拷贝。如：当函数返回一个局部unique_ptr时

3）交出控制权

```
//返回指针，放弃对指针的控制权，并将u置为空
//不会释放，主要目的在于切断与原来管理对象的联系，将其交由其它unique_ptr来管理
u.release()

p.release()             //内存泄露
auto pp = p.release()  //要记得delete pp
```

4）释放

```
u = nullptr;        //释放u指向的对象，将u置为空；
u.reset();          //释放u指向的对象，并将u置为空；
u.reset(q);         //释放u指向的对象，转为控制指针p指向的对象
u.reset(nullptr);   //释放u指向的对象，并将u置为空；
```

5）管理动态数组

shared_ptr不直接管理动态数组，如果要用shared_ptr来管，须提供自定义的删除操作，因为默认情况下shared_ptr使用delete销毁所指对象。但即使如此，也不能用下标访问每个元素，需要用get()函数。unique_ptr可以用下标访问

```
unique_ptr<int[]> up(new int[10]); //创建

up.release();    //放弃对指针的控制权，并将u置为空（不会释放。测试如此，和书本不同）

up[i];           //返回位置i处的对象，左值；
```

2.4 weak_ptr

1）初始化

```
//空weak_ptr，可以指向类型为T的对象
weak_ptr<T> w;
```

```
//与shared_ptr sp指向相同对象的weak_ptr，T必须能转换为sp指向的类型
weak_ptr<T> w(sp);
```

2）赋值

```
w = p;  //p是shared_ptr或weak_ptr，赋值后w与p共享对象
```

3）重置

```
w.reset();      //将w置为空（不会释放对象）
```

4）状态

```
//返回与w共享对象的“shared_ptr”的数量
w.use_count();

//如果共享对象的"shared_ptr"为0(没有共享对象的shared_ptr)，则返回true，否则返回false
w.expired();
```

5）访问

```
//获取shared_ptr
//  如果没有共享对象的shared_ptr，则返回一个空的shared_ptr；
//  否则返回一个指向共享对象的shared_ptr；
//这种访问方式提供了对动态对象的安全访问；
w.lock();
```

输入输出

1.I/O流

> 定义了用于读写流（普通流）的基本类型，处理控制台

头文件：

- `istream`
- `ostream`
- `iostream`

2.文件流

> 定义了读写命名文件（文件流）的类型，处理文件

头文件：`<fstream>`

- `ifstream`
- `ofstream`
- `fstream`

2.1 文件模式

创建或打开文件流时可以指定文件模式：`ofstream::mode`

- `in` ：以读方式打开，只能对 `ifstream` 和 `fstream`
- `out` ：以写方式打开，只能对 `ofstream` 和 `fstream`
- `app` ：每次写操作前定位到文件末尾，只要没指定 `trunc` 就能指定。只以 `out` 打开时，唯一保存数据的办法就是指定 `app`
- `ate` ：打开文件后立即定位到文件末尾
- `trunc` ：截断文件。只有指定了 `out` 才能指定。默认情况下以 `out` 模式打开，即使不指定 `trunc` 也会截断 (同时以 `in` 和 `out` 打开则不会)
- `binary` ：以二进制方式进行I/O

2.2 创建文件流

```
fstream fstrm;        //1.创建一个未绑定的文件流，可以随后调用open将其与文件关联
fstream fstrm(s);     //2.创建一个文件流，并打开文件s，s可以是string或字符数组
fstream fstrm(s,mod); //3.与前一个相同，同时指定了模式
```

都是explict声明的构造函数，限制了隐式转换

2.3 打开文件流

```
fstrm.open(s); //打开文件s，并将fstrm与文件绑定
```

如果 `open` 失败， `failbit` 会被置位。对一个已打开文件流调用 `open` 会失败

`is_open()` 可以查看流是否打开：

```
fstrm. is_open(); //返回一个bool，指出 与fstrm关联的文件是否成功打开且尚未关闭
```

2.4 关闭文件流

```
fstrm.close(); //关闭与fstrm绑定的文件。返回void
```

当一个 `fstream` 对象被销毁时， `close` 会自动被调用

3.字符串流

> 定义了读写(内存)string对象(string流)的类型，处理内存

头文件： `<sstream>`

- `istringstream`
- `ostringstream`
- `stringstream`

3.1 创建string流

```
sstream strm; //未绑定
sstream strm(s); //保存了s拷贝的string流
```

都是explict声明的构造函数，限制了隐式转换

3.2 返回string流

```
strm.str(); //返回string流中的string
```

3.3 将string拷贝到string流

```
strm.str(s); //将s拷贝到string流中
```

4.四个常用I/O对象

- **输入**(`istream`)
  - `cin`
- **输出**(`ostream`)
  - `cout`
  - `cerr` ：输出警告和错误信息
  - `clog` ：输出程序运行时的一般信息

5.流状态

5.1 条件状态

> badbit，eofbit，failbit任意一个被置位时，检测流状态的条件都会失败

状态类型： `strm::iostate`

4个iostate类型的constexpr值：

- `strm::goodbit`
- `strm::badbit` ：流已崩溃，系统级错误，不可恢复。通常情况下，一旦被置位，流就无法再使用了
- `strm::eofbit` ：流到达结尾
- `strm::failbit` ：I/O操作失败

1）查询状态

- `s.good()` ： `badbit` ， `eofbit` ， `failbit` 都未置位时返回 `true`
- `s.eof()` ： `eofbit` 被置位时，返回 `true`
- `s.fail()` ： `failbit` 或 `badbit` 被置位时返回 `true`
- `s.bad()` ： `badbit` 被置位时返回 `true`

2）管理状态

- `s.rdstate()` ：返回流对象的当前状态
- `s.clear()` ：复位所有位
- `s.clear(flags)` ：复位flags位
- `s.setstate(flags)` ：设置flags位

5.2 格式状态

操纵符：操纵符可以修改流的格式状态，当操纵符改变流的格式状态时，通常改变后的状态对所有后续I/O都生效（endl不是）

- 不接收参数的操纵符（头文件： `<iostream>` ）
- 接收参数的操纵符（头文件： `<iomanip>` ）

1）输出控制

> cout << 操纵符;

- **数值输出控制**
  - **整形**
    - **bool值的格式**：默认情况下bool值输出为0或1，可以改为输出“false”或“true”
      - **设置**： `boolalpha`
      - **恢复**： `noboolalpha`

        - **整形进制**
            - **八进制**： `oct`
            - **十进制**： `dec`
            - **十六进制**： `hex`
        - **显示进制信息**
            - **设置**： `showbase`
            - **恢复**： `noshowbase`
    - **浮点型**
        - **指定精度**：默认情况下，精度指数字的总数，包括整数部分和小数部分
            - **设置**： `cout.precision(精度值)` 或 `setprecision(精度值)`
            - **获取**： `cout.precision()`
        - **十六进制、定点十进制或科学记数法**
            - **科学计数法**： `scientific`
            - **定点十进制**： `fixed`
            - **十六进制**： `hexfloat`
            - **恢复成默认**： `defaultfloat`
        - **打印小数点**：默认情况下，当一个浮点值的小数部分为0时，不显示小数点
            - **设置**： `showpoint`
            - **恢复**： `noshowpoint`
- **补白的数量和位置**
    - `setw` (最小宽度)：设置下一数字或字符串的"最小"宽度。注意是"最小"，如果设置过小，并不会限制输出。同时，`setw` 与 `endl` 类似，它不改变流的内部状态。 `setw` 只影响下一个输出
    - `left` ：输出左对齐
    - `right` (默认)：输出右对齐
    - `internal` ：负号左对齐，值右对齐，中间填充空格
    - `setfill` (字符)：使用指定字符代替空格来补白输出

2）输入控制

跳过空格：默认情况下会输入会跳过空格，制表符和换行符

- **跳过**： `skipws`
- **不跳过**： `noskipws`

6.流操作

6.1 关联输入输出流

> 每个流最多同时关联一个输出流

`istream.tie()` ：返回 `istream` 关联的输出流的指针，如果没有返回空指针 `istream.tie(&ostream)` ：将 `istream` 关联到 `ostream` ，返回一个指向 `ostream` 的指针。 由于每个流最多关联一个输出流，所以可以传入 `nullptr` 来解除关联

6.2 未格式化I/O操作

> 将流当作一个无解释的字节序列来处理

1）单字节操作

- **(从输入流)读取字符**
  - `is.get(ch)` ：从 `is` 读取下一个字节存入字符 `ch` ，返回 `is`
  - `is.get()` ：从 `is` 读取下一个字符作为 `int` 返回
- **(向输出流)写入字符**
  - `os.put(ch)` ：将字符 `ch` 输出到 `os` ，返回 `os`
- **退回字符(到输入流)**：可以退回最多一个值
  - `is.putback(ch)` ：将字符 `ch` 放回 `is` ，返回 `is` 。 `ch` 必须与最后读取的字符相同
  - `is.unget()` ：使输入流向后移动，从而最后读取的值又回到流中
  - `is.peek()` ：返回输入流中下一次字符的副本，不会将字符从流中删除，返回的值仍留在流中

2）多字节操作

- **(从输入流)读取多个字符**
  - `is.get(sink,size,delim)` ：从输入流中读取多个字符，保存到 `sink` 指向的字符数组中 - 结束条件： * 1）遇到 `delim` （ `delim` 字符不会从输入流中读取出来，不会存入数组中） * 2）读取完 `size` 个字符 * 3）文件末尾
  - `is.getline(sink,size,delim)` ：从输入流中读取多个字符，保存到 `sink` 指向的字符数组中 - 结束条件： * 1）遇到 `delim` （ `delim` 字符会从输入流中读取出来并丢弃，不会存入数组中） * 2）读取完 `size` 个字符 * 3）文件末尾
  - `is.read(sink,size)` ：读取最多 `size` 个字节，存入 `sink` 。返回 `is`
  - `is.ignore(size,delim)` ：读取并忽略最多 `size` 个字符，包括 `delim` 。 `size` 默认为1， `delim` 默认为文件尾
- **(向输出流)写入多个字符**
  - `os.write(source,size)` ：将字符数组 `source` 中的 `size` 个字节写入 `os` ，返回 `os` ；
- **返回上次读取的字节数**
  - `is.gcount()`

3）随机访问

随机I/O本质上依赖于系统，为了理解如何使用这些特性，必须查询系统文档。由于 `istream` 和 `ostream` 类型通常不支持随机访问，所以这部分介绍的随机访问操作只适用于 `fstream` 和 `sstream`

- **标记**
  - **通过维护一个标记来支持随机访问**：标记记录了下一个读写操作进行的位置
  - **不存在独立的读标记和写标记**：也就是说，在一个读写流中，只有一个标记。并不存在分离的读标记和写标记
- **操作**
  - **获取标记**
    - `tellg()` ：获取输入流中的标记当前位置
    - `tellp()` ：获取输出流中的标记当前位置
  - **设置标记**
    - `seekg(pos)` ：设置输入流标记的位置为 `pos` 。 `pos` 类型为 `pos_type` （例： `stringstream::pos_type` ）
    - `seekp(pos)` ：设置输出流标记的位置为 `pos` 。 `pos` 类型为 `pos_type` （例： `stringstream::pos_type` ）

- `seekg(off,from)` ：设置输入流标记为从 `from` 开始，偏移量为 `off` 的位置， `off` 可以是负值 * `from` 可以是： * 1） `beg` ：流开始位置（例： `fstream::beg` ） * 2） `cur` ：当前位置； * 3） `end` ：流结尾位置；
- `seekp(off,from)` ：设置输出流标记为从 `from` 开始，偏移量为 `off` 的位置， `off` 可以是负值 * `from` 可以是： * 1） `beg` ：流开始位置（例： `fstream::beg` ） * 2） `cur` ：当前位置； * 3） `end` ：流结尾位置；

7.缓冲区管理

每个输出流都管理一个缓冲区，用来保存程序输出的数据。有了缓冲区，操作系统就能将程序多个输出操作组合成单一的系统级写操作，由于写操作耗时，所以可以带来很大性能提升

7.1 刷新缓冲区

> 程序异常崩溃时，缓冲区不会被刷新

以下情况会刷新缓冲区：

1. **程序正常结束**
2. **缓冲区满时**
3. **操作符刷新缓冲区**
   - endl：插入一个换行符，然后刷新
   - flush：不插入任何额外字符，只刷新
   - ends：插入一个空格字符，然后刷新
4. **unitbuf设置流的内部状态清空缓冲区**
   - `cerr` 设置了 `unitbuf` ，所以写到 `cerr` 的内容都是立即刷新的
   - `unitbuf` 操作符会告诉流在接下来的每次写操作之后都进行一次 `flush` 操作
   - `nounitbuf` 操作符可以重置：
     - 设置： `cout << unitbuf;`
     - 重置： `cout << nounitbuf;`

5. **“试图”从输入流读取数据时会刷新与其关联输出流的缓冲区**
    - 标准库将 `cout` 和 `cin` 关联在一起，故下列操作会刷新 `cout` 的缓冲区： `cin >> ival;`
    - 交互式系统通常应该关联输入流与输出流，这意味着所有输出，包括用户提示信息，都会在读操作之前被打印出来