Effective STL

- ## 条款1：慎重选择容器类型
  - **知识点1：STL容器的类型**
    - 标准STL序列容器：vector string deque list forward-list（单项列表） array
    - 标准STL关联容器：set mutiset map mutimap
    - 非标准序列容器：rope，本质上是一个重型string
    - 非标准关联容器：hash-set，hash-mutiset ，hash-map，hash-mutimap
    - vector<char>作为string的替代
    - vector作为标准关联容器的替代。
    - 几种标准的非STL容器
  - **知识点2：关于容器选择的考虑点**
    - 是否需要在容器的任意位置插入元素？是：选序列容器而不是关联容器
    - 是否关系容器的排序？是：考虑哈希容器
    - 容器是否必须是C++标准库的一部分？是：排除哈希容器，rope等非标准库容器
    - 需要何种类型的迭代器？
      - 随机：从vector string deque中选择
      - 双向：list
    - 当发生插入或者删除的操作的时候，是否需要避免连续内存的容器？是：避免他
    - 是否介意容器中的引用计数技术？是：避免使用string
    - 对于插入和删除，是否需要回滚的能力吗？是：使用基于节点的容器
    - 是否需要让迭代器和指针以及引用变为无效的次数最少吗？是：使用基于节点的容器
    - 对于使用swap让迭代器指针或引用变为无效，是否重要？是：避免使用string
    - 容器的数据布局是否需要和C兼容？是：只能用vector
    - 元素查找的时候速度是否是关键因素？是：考虑哈希容器
    - 如果序列容器的迭代器是随机访问类型，而且只要没有删除操作发生，且插入操作只发生在容器的末尾，则指向数据的指针和引用就不会变为无效，这样的容器是否对你有帮助？是：考虑deque（deque在重复插入后方元素的时候，迭代器会失效，但是指针和引用不会，deque是唯一的，迭代器会失效但是指针和引用不会失效的容器
- ## 条款2：不要试图编写独立于容器类型的代码
  - **知识点1：容器是泛化的，但是不能泛化容器**
    - STL容器是以泛化为基础的，每一个容器都有着对应独特的数据结构与算法
    - 我们不能尝试泛化容器，创造出一个可以任意使用的容器，那种容器不好用，因为他只能使用所有容器的交集部分，这可提供的操作就已经很少了
  - **知识点2：封装容器，让容器的类型转换变得容易**
    - 前言：很多时候我们可能会发现一种其他的容器类型比现在用的容器类型好很多，但是更改所有的容器名称非常麻烦，这个时候我们需要通过封装让容器的转换变得容易
    - 我们可以使用类型定义进行封装（typedef）
      - 通过类型定义这样的封装，虽然仅仅是名字上的，但是如果改变容器，那么后续操作起来会非常方便

```
class Widget{…};
typedef vector<Widget> WidgetContainer;
WidgetContainer cw;
Widget bestWidget;
…
WidgetContainer::iterator i = find(cw.begin(), cw.end(), bestWidget);
```

```
class Widget{…};
template<typename T>
SpecialAllocator{…};                        //为什么要声明成模板？见第10条

typedef vector<Widget, SpecialAllocator<Widget> > WidgetContainer;
WidgetContainer cw;                         //仍可工作
Widget bestWidget;
…
WidgetContainer::iterator i=
 find(cw.begin(), cw.end(), bestWidget);    //仍可工作
```

  - **知识点3：再进一步封装容器**
    - 如果要想减少在替换容器类型时所需要修改的代码，那么就可以考虑把容器隐藏到一个类中去，并尽量减少那些通过类接口（而使外部）可见的，与容器相关的信息。
    - 这么做的好处就是，将容器隐藏于类中并且开放一些接口，当在更换容器的时候，只需要更改该类开放的接口就行，不需要代码全局搜索去修改直接使用该容器及其成员函数的地方，工作量会减少很多
- ## 条款3：确保容器中的对象拷贝正确且高效
  - **知识点1：拷贝对象时STL的工作方式**
    - 首先我们可以认为，STL的工作方式为拷贝（但是在C++11中，这种工作方式需要再加上一个，移动）
    - 在C++11中，对于一个左值，我们依然会调用拷贝构造函数进行创建，但是对于一个临时创建出来的变量或者纯右值就会调用移动构造函数

```cpp
class ex12
{
private:
    char a;
    char b;
public:
    ex12(/* args */)=default;
    ex12(char a,char b):a(a),b(b){};
    ex12(const ex12& x){
        a=x.a;
        b=x.b;
    }
    ex12(const ex12&& x){
        a=x.a;
        b=x.b;
    }
};

int main()
{
    ex12 c;
    vector<ex12> a;
    a.push_back(ex12('a','b'));//调用移动构造函数
    a.push_back(c);//调用拷贝构造函数
}
```

- 知识点2：了解基类的容器不能承载继承类
  - 因为上面的原因可以看的出来，基类的容器是无法装在继承类的，如果装载了继承类那么原来继承类的数据就面临丢失的风险。因为调用的是基类的拷贝构造函数。
  - 如果希望插入后的对象依然表现的像派生类对象一样那么可以考虑使用指针或者智能指针，这样有两个好处
    - 拷贝指针的速度非常快：可以快速的通过指针去排序，而不用调用一大堆的拷贝构造函数等等拖慢速度
    - 保留对象的“继承性”通过动态绑定，指针可以让其调用virtual函数，从而完整的保留派生类及其特性

条款4：调用empty而不是检查size是否为0

- 原因：empty在大多数标准容器都是常数时间操作，而对于一些list的实现，size会耗费线性时间（主要是list）

条款5：区间成员函数优先于与之对应的单元素成员函数

- **知识点1：优先考虑区间成员函数**
  - 区间成员函数的定义：使用两个迭代器作为参数传入的函数
  - 使用区间成员函数的好处
    - 使得代码更加容易被理解，copy函数总比使用for循环好理解的多
    - 使用合适的区间成员函数可以大幅减少完成工作的时间。提升效率
    - 使用区间成员函数的效率，最差为和for循环一样，比如说对istream使用的时候2者相同，但大多数比这个好
- **知识点2：使用区间成员函数的时候，优先考虑非泛型函数**
  - 所有的标准容器都提供了以下几个构造函数
    - 区间创建函数
    - 区间插入函数insert
    - 区间删除函数erase
    - 区间赋值函数assign
  - 可以考虑优先选用stl自带的函数，毕竟大多数时候自己人更懂自己人，效率更高速度更快

条款6：当心C++编译器最烦人的分析机制

- **情景1：糊涂的C++分析机制**
  - 首先，声明一个变量有几种奇怪的方法

```cpp
int f(double d);
int f(double (d));//和上面的形式一样的。但是d两边的括号会被忽略
int f(double);//等于上面的第二个，其实参数名也被一起忽略了
```

  - 同样，声明一个指针也是这样

```cpp
int g(double(*pf)());//正常的指针形式，传入无参数，返回一个double的指针
int g(double pf());//隐式指针形式
int g(double())//把参数名字也省去了
```

  - 那么，对于这种类型，C++解析出来的意思就很奇怪

```
ifstream dataFile("ints.dat");
list<int> data(istream_iterator<int>(dataFile), //小心！结果不会是
               istream_iterator<int>());        //你所想象的那样
```

- 你的意思：传入两个迭代器，拷贝ints.dat文件里的东西
- 编译器的意思：
  - 第一个是一个int类型的迭代器，但是datafile会被忽略（多余的，考虑第一个图片第二条f）
  - 第二个是一个不带参数的指针，（考虑第二个图片第三条g）

- **知识点1：确保你传入的东西不会被C++的编译机制干扰**
  - 如情景1所示，我们在直接传入迭代器的时候，就会发生离谱的事情，这就是因为C++的编译机制。直接传入两个迭代器不可行，因为他可能会被编译为上面图片两种形式
  - 解决方法很简单，单独定义再传入，这样做的好处就是所有编译器都能过，且通过合适的命名让人们更加容易理解

```
ifstream dataFile("ints.dat");
istream_iterator<int> dataBegin(dataFile);
istream_iterator<int> dataEnd;
list<int> data(dataBegin, dataEnd);
```

- **条款7：如果容器中包含了通过new操作创建的指针，切记在容器对象析构前将指针delete掉**
  - **知识点1：delete指针**
    - 虽然说容器在销毁的时候会对每一个对象进行delete，但是指针不是所谓的“对象”没有析构函数，所以不会做任何事情。如果不对指针进行delete就会导致资源泄露
    - 解决方法1：在销毁容器之前，使用for循环挨个delete指针
    - 解决方法2：编写一个template函数，用来挨个删除，这种方法是类型安全的而不是异常安全的，当有资源泄露的时候还是会出现错误

```
struct DeleteObject{                    //从这里去掉了模板化和基类
    template<typename T>                //在这里加入模板化
    void operator()(const T* ptr) const
    {
        delete ptr;
    }
};
void doSomething()
{
    deque<SpecialString*> dssp;
    for_each(dssp.begin(), dssp.end(),
             DeleteObject());           //哈！确定的行为
}
```

    - 解决办法3：使用智能指针，确保资源不泄露
- **条款8：切勿创建包含auto-ptr的对象（对于C++11过时）**
- **条款9：慎重选择删除元素的方法**
  - **知识点1：删除容器对象选择的方法**
    - 如果容器是vector，string，deque，使用erase（容器自带）-remove（algorithm库自带）的习惯用法

```
c.erase(remove(c.begin(), c.end(), 1963),   //当 c 是 vector、string 或 deque
        c.end());                           //时，erase-remove 习惯用法是删除
                                            //特定值的元素的最好办法
```

    - 如果容器是list，则使用list：：remove

```
c.remove(1963);                             //当 c 是 list 时，remove 成员函数
                                            //是删除特定值的元素的最好办法
```

    - 如果是关联容器，那么就调用erase成员函数

```
c.erase(1963);                              //当 c 是标准关联容器时，erase 成员
                                            //函数是删除特定值元素的最好办法
```

  - **知识点2：要删除容器中满足特定判别式（条件）的所有对象**

```
bool badValue(int );                        //返回 x 是否为“坏值”
```

    - 如果容器是vector，string，deque，使用erase（容器自带）-remove-if（algorithm库自带）的习惯用法

```
c.erase(remove_if(c.begin(), c.end(),       //当 c 是 vector、string 或 deque
        badValue),c.end());                 //时，这是删除使 badValue 返回 true
                                            //的对象的最好办法
```

    - 如果容器是list，使用list：：remove-if

```
c.remove_if(badValue);                      //当 c 是 list 时,这是删除使
                                            //badValue 返回 true 的对象
                                            //的最好办法
```

    - 如果容器是一个标准关联容器
      - 可以使用remove-copy-if，然后利用swap来获得所需要的元素（效率稍低且浪费空间）

```
AssocContainer<int> c;                  //c 现在是一个标准关联容器
...
AssocContainer<int> goodValues;         //保存不被删除的值的临时容器
remove_copy_if(c.begin(), c.end(),      //把不被删除的值从 c 复制到
               inserter(goodValues,     //goodValues 中
                        goodValues.end()),
               badValue);
c.swap(goodValues);                     //交换 c 和 goodValues 的内容
```

- 写一个循环遍历容器中的元素，在把迭代器传给erase时，要对他进行后缀递增

```
AssocContainer<int> c;
...
for (AssocContainer<int>::iterator i = c.begin();     //for 循环的第三部分是
     i != c.end();                                    //空的，i 在下面递增
     /*什么也不做*/){
  if (badValue(*i)) c.erase(i++);                     //对坏值，把当前的 i 传给
  else ++i;                                           //erase，递增 i 是副作用
}                                                     //对好值，则简单地递增 i
```

- **知识点3：要在循环内部做某些（除了删除对象之外）的操作**
  - 如果容器是一个标准序列容器，则写一个循环来遍历容器中的元素，记住每次调用erase时，要用它的返回值更新迭代器

```
for (SeqContainer<int>::iterator i = c.begin();
     i != c.end();){
  if (badValue(*i)){
     logFile << "Erasing " << *i << '\n';
     i = c.erase(i);                    //把 erase 的返回值赋给 i
  }                                     //使 i 的值保持有效
  else ++i;
}
```

  - 如果容器是一个关联容器，则写一个循环来遍历容器中的元素，当把迭代器传给erase时，要对迭代器的后缀做++

```
ofstream logFile;                               //要写入的日志文件
AssocContainer<int> c;
...
for (AssocContainer<int>::iterator i = c.begin();    //循环条件同上
     i != c.end();){
  if (badValue(*i)){
     logFiel << "Erasing " << *i << '\n';          //写日志文件
     c.erase(i++);                                //删除元素
  }
  else ++i;
}
```

条款10：了解分配子allocator的约定和限制

- **回忆：static的用法**
  - 对于面向过程：
    - static对应的函数一般都是被放在全域区，他们在程序开始时候就获得，结束的时候销毁
  - 对于面向对象
    - 初始化
      - static放入类中，如果不加const需要在类外面初始化

```
class ex13
{
public:
    static int x;

};
int ex13::x=3;
```

      - 加了const可以直接在类内赋值初始化
    - 对于对象
      - static命名的变量属于是一次定义，全类共享，所有类共享一个公有的static。只是一个“包含关系”，所以他不能被构造函数构造。
      - 对于静态成员函数，类可以通过点运算符调用静态成员函数，但是成员函数本身只能直接调用类内的静态类型的变量或者是成员函数，对于非静态成员或函数是调用不了的
    - 静态类
      - 和变量一样，静态类的生命周期直到程序的结束。在main结束后才会调用静态类的析构函数。
- **知识点1：确保你的分配子是一个模板，模板参数T代表你为他分配内存的对象的类型**
- **知识点2：提供类型定义pointer和reference，但是始终让pointer为T*，reference为T&，这是标准库所规定的**
- **知识点3：千万不要让你的分配子拥有随对象而不同的状态，通常分配子不应该有非静态类型的数据成员，因为STL实现假定同一类型的分配子是等价的**
- **知识点4：传给allocator成员函数的是那些要求内存对象的个数，而不是所需要的字节数，同时要记住，这些函数返回T*指针（通过pointer类型定义，即使尚未有T对象被构造出来**
- **知识点5：通过嵌套的rebind模板，因为标准容器依靠模板取得类的成员的类型来分配内存**

条款11：理解自定义分配子的合理用法

- **知识点1：自定义分配子的合适场景**
  - 在alloctor<T>是线程安全的，你想使用一个单线程的alloctor，那么就可以考虑定制
  - 想建立一个与共享内存相对应的特殊的堆，然后在这块内存中放一个或者多个容器，让其他的进程可以共享这些容器

条款12：切勿对STL容器的线程安全性有不切实际的依赖

- 为了速度STL不支持完整的线程安全性。
- **知识点1：STL的线程安全的要求**
  - 对于一个STL，多个线程的读是安全的。
  - 对于一个STL，多个线程对不同容器做写入操作是安全的
- **知识点2：实现STL线程安全的可能行为**
  - 对容器的成员函数的每次调用，都锁住容器直到调用结束，且具有异常安全

```
vector<int> v;
...
getMutexFor(v);
vector<int>::iterator first5(find(v.begin(), v.end(), 5));
if (first5 != v.end()){
   *first5 = 0;
}
releaseMutexFor(v);
```

- 在容器所返回的每个迭代器的生存期结束前，都锁住容器
- 对于作用于容器的每个算法，都锁住该容器，直到算法结束

条款13：vector和string优先于动态分配的数组

- **知识点1：当你使用new进行内存分配的时候，必须承担的责任：**
  - 确保以后会有人用delete来删除所有分配的内存，如果没有后面的delete，那么你的new将会导致一个资源泄露
  - 确保使用正确的delete形式，对于一个使用delete，对于多个使用delete【】
  - 确保只能delete一次
- **知识点2：代替数组**
  - 一般就是用vector代替常见数组
  - 如果你的string是引用计数的，那么可以考虑使用vector<char>代替string，因为vector不允许引用计数

条款14：使用reserve来避免不必要的重新分配

- **知识点1：vector和string自动加大的原理和作用**
  - 原理：
    - 分配一块大小为当前容量的某个倍数的新内存。每次需要增长的时候，容量就加倍一次
    - 把容器中所有元素从旧的内存中复制到新的内存中
    - 析构掉旧内存中的对象
    - 释放旧内存
  - 作用
    - 增大内存的操作会使迭代器，引用，指针失效，可能内存的操作就是插入操作，push类，insert类等
- **知识点2：明确vector 和string都提供以下几个函数**
  - size（）：告诉你容器中有多少个元素，但是不会告诉你为了包含这些数据所分配的内存
  - capacity（）：告诉你容器中利用已经分配的内存可以容纳多少个元素，这个是容器能容纳的容器元素总数，要知道还有多少未使用，需要用capacity-size
  - resize（）强迫容器改变到包含n个元素的状态，
    - 如果n大于容器中元素总数，
      - 小于目前的capacity：
        - 通过构造函数将新的元素添加到容器末尾
      - n大于capacity
        - 重新分配内存
    - n小于容器中元素的总数
      - 将容器为部的元素析构
  - reserve():强迫容器把他的容量变到至少是n，n不小于当前的大小，这通常会导致重新分配
- 知识点3：如何避免vector，string被频繁重复的分配
  - 尽可能不要一个一个insert进去，那样会导致vector被频繁的重新分配内存
  - 在已知大小的情况下，调用reserve（）重新提前分配好
  - 先预留足够大的空间，然后加入数据，加入完成后去除多余容量，方法见17条

条款15：注意string实现的多样性

- 前言：在STL标准库中，string的大小是多样性的，如果使用sizeof（string）他的结果可能和指针一样为1，或者是7，这和STL的实现有关系
- **知识点1：string的基本特征**
  - 每个string都包括以下信息
    - size，包含字符的个数
    - capacity，string的容量
    - value，string的值
  - 可能还包含
    - string分配子的一份拷贝
    - string对值的引用计数（这个只是在建立在引用计数基础上的string才有的）
- **知识点2：string的重要特性**
  - string的值可能会被引用计数，也可能不会，如果一旦引用，那么他的值的大小会扩大。
  - sizeof（string）的大小可以是sizeof（char*）的1到7倍
  - 创建一个新的字符串值可能需要0次，1次，或两次的动态分配内存

- string可能支持，也可能不支持针对单个对象的分配子
- 不同的实现对字符内存的最小分配单位有着不同的策略

条款16：了解如何把vector和string数据传递给旧的API

条款17：使用swap技巧去除多余的变量

- 备注：虽然C++11中的vector有shrink-to-fit成员函数，但是呢这个函数只是一个请求，随着STL实现的方式不同，使用后capicity可能还是大于size，swap则是一种相对强硬的手段，用来去除多余的容量，但是也有可能失败
- **知识点1：使用swap去除多余的容量**
  - 首先，使用swap的时候，一般会调用vector的拷贝构造函数，这种情况下vector只会分配和之前容量大小相等的元素，如果再分配后进行swap操作，那么储存的元素就不会有多余的容量了

```
vector<int> s{1,2,3,4,5,6,7};
vector<int>(s).swap(s);
```

  - 同样的，swap操作对于string同样适用

```
string a("hello");
string(a).swap(a);
}
```

  - 有意思的是，虽然交换会带来指针，迭代器的错位，但是因为交换的是自身，所以完全就没影响
  - 如果你需要清空一个容器，也可以使用swap的小方法

```
vector<Contestant> v;
string s;
...                                     //使用 v 和 s
vector<Contestant>().swap(v);           //清除 v 并把它的容量变为最小
string().swap(s);                       //清除 s 并把它的容量变为最小
```

条款18：避免是使用vector<bool>

- 作为一个STL容器，vector<bool>只有两点不对：他不是一个STL容器，其次他不能储存bool
- 如果需要使用类似于其的结构，使用deque<bool>或者bitset是更好的选择

条款19：理解相等和等价之间的区别

- **知识点1：相等与等价的字面区别**
  - 相等，一般是调用operator==判断，代表两者完全相同，
  - 等价，在一定的规则下，两者所内涵的区别可以忽略不计，代表等价，一般都会采用operator<去组织
- **知识点2：相等和等价使用时候的区别**
  - 比较典型的例子就是find和关联容器的find，对于泛型算法的find，一般都是以相等去组织，但是关联容器的find可能会用等价去组织，因此对于这种情况下，一般使用成员函数会更好，否则可能导致查找失败

条款20：为包含指针的关联容器指定比较类型

- **知识点1：明确关联容器<>里的内容**
  - 一般来讲，有序关联容器的<>的内容包括三点

```
set<T,less<T>,allocator<T> >
```

- **知识点2：保存在指针里面的关联容器**
  - 对于指针包含的关联容器，其为T*，less<T*>,allocator<T*>,也就是按照指针保存的对于的类型，按照指针本身确定的排序大小，以及allocator分配器
  - 因为是按照指针本身确定的排序大小，所以我们如果想要按照指针对应的内容做排序，需要自定义一个类，重载operator（），进行排序

```
struct StringPtrLess:
   pulic binary_function<const string*, //采用该基类的原因见第 40 条
                         const string*,
                         bool> {
   bool operator()(const string *ps1, const string *ps2) const
   {
      return *ps1 < *ps2;
   }
};

   typedef set<string*, StringPtrLess> StringPtrSet;
   StringPtrSet ssp;                          //创建一个包含字符串的集合，并把它
```

  - 如果用迭代器通过循环输出，就需要解两次引用

```
for (StringPtrSet::const_iterator i = ssp.begin();
     i != ssp.end();                    //打印出的是"Anteater", "Lemur",
     ++i)                               //"Penguin"和"Wombat"
   cout << **i <<endl;
```

  - 如果用算法输出，就需要重新写一个模板进行第二次解引用的操作

```
void print(const string *ps)            //把 ps 指向的对象打印到 cout
{
   cout << *ps <<endl;
}
for_each(ssp.begin(), ssp.end(), print);//对 ssp 中的每个对象调用 print
```

```
//当向该类型的函数子传入T*时，它们返回const T&
struct Dereference {
   template<typename T>
   const T& operator()(const T *ptr) const
   {
      return *ptr;
   }
};
transform(ssp.begin(), ssp.end(),              //通过解除指针引用，“转
          ostream_iterator<string>(cout, "\n"), //换”ssp中的每个元素
          Dereference());                      //并把结果写到cout
```

- **条款21：总是让比较函数在等值情况下返回false**
  - **知识点1：严格的弱序化**
    - 在关联容器中，我们需要一个严格的弱序化的比较，及我们应该主动用<或>而不是<=或>=，如果使用<=或>=我们就会破坏关联容器，包括其对应可使用的算法也一样会失效
- **条款22：切勿直接修改set或multiset中的键**
  - **知识点1：对于map和mutimap**
    - 对于map和mutimap，这两个的元素类型是pair<const K,V>所以他们的键是不可以修改的
  - **知识点2：对于set和mutiset**
    - 对于set和mutiset，这两个键对于某些STL标准是可以被修改的，也就意味着代码可能具有不可移植性，
    - 所以如果要有可移植性可以考虑这样两种方法
      - 使用const-cast（而不是static-cast，其转换的为副本）做一次去除const的类型转换，然后修改key
      - 擦去再添加方法：
        - 找到合适的元素
        - 对元素做一份拷贝
        - 修改该拷贝
        - 删除该元素
        - 重新插入元素
- **条款23：考虑用排序的vector替代关联容器**
  - **知识点1：程序使用数据结构的三个阶段**
    - 设置阶段：创建一个新的数据结构，插入大量元素，基本都是删除或者插入操作
    - 查找阶段：查询该数据结构以找到特定的信息，这个阶段基本都是查找操作，没有或很少有插入或者删除操作
    - 重组阶段：改变该数据结构的内容，或许是删除所有的当前数据，再插入新的数据
  - 知识点2：什么时候考虑用排序的vector代替关联容器
    - 查找操作几乎从不跟插入和删除操作混在一起的时候，在考虑使用排序vector
- **条款24：当效率至关重要的时候，请在map：：operator【】与map：：insert之间做出选择**
  - **知识点1：**
    - 对于map，当你在插入一个数据的时候，选择insert，因为这样的话可以减少一个临时对象的构造节约时间
    - 对于map，当你在更新一个数据的时候，选择opeartor【】，因为这样可以直接修改数据，减少查找所消耗的时间
- **条款25：熟悉非标准的散列容器**
  - 注：C++11中已经提供了unorder系列作为替代，其实现和原先的hash基本一致

C++标准库	民间高手造的轮子
unordered_set	hash_set
unordered_map	hash_map
multiset	hash_multiset
multimap	hash_multimap

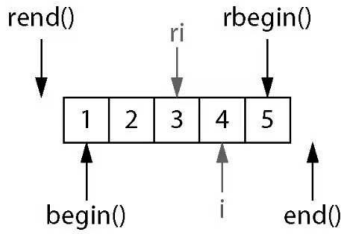
- **条款27：使用distance和advance将容器的const-iterator转换为iterator**
  - **知识点1：图示转换代码**

```
typedef deque<int> IntDeque;            //同前
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator ConstIter;

IntDeque d;
```

```
ConstIter ci;
...                                     //使ci指向d
Iter i(d.begin());                      //使i指向d的起始位置

   advance(i, distance(i, ci));                 //移动i，使它指向ci所指的位置
```

- **条款28：正确理解由reverse-iterator的base（）成员函数所产生的iterator的用法**

- **知识点1：reverse-iterator的base函数获得的迭代器的位置关系**
  - 存在向右偏移一格的距离



- **知识点2：insert操作**
  - 如果要在一个reverse-iterator ri指定位置进行插入新的元素，那么只需要在ri。base（）=i位置处直接插入，对于插入而言是等价的
- **知识点3：erase操作**
  - 如果要在reverse-iterator ri指定一个位置删除要给元素，那么就是需要在++ri后在进行base转换最后删除

```
...                                          //同上
v.erase((++ri).base());                      //删除 ri 所指的元素；这下编译没问题了！
```

条款29：对于逐个字符的输入请考虑使用istreambuf-iterator

- **知识点1：为什么不使用istream-iterator?**
  - istream-iterator比较慢，因为在读取单个字符的时候往往要做很多的操作，包括判断是否是一个空格，是否是发生错误，是否需要抛出异常，但是对于单个字符来说就比较多余

```
ifstream inputFile("interestingData.txt");
string fileData((istream_iterator<char>(inputFile)),      //将 inputFile 读入
                istream_iterator<char>());                //fileData；为什么
                                                          //这是不正确的，请看
                                                          //下文解释；关于该语
                                                          //句的一条警告，请参
                                                          //阅第 6 条

ifstream inputFile("interestingData.txt");
inputFile.unsetf(ios::skipws);          //禁止忽略 inputFile 中的空格
string fileData((istream_iterator<char>(inputFile)),
                istream_iterator<char>());
```

  - 所以可以考虑待用istreambuf-iterator输入，这样的话速度会快很多

```
ifstream inputFile("interestingData.txt");
string fileData((istreambuf_iterator<char>(inputFile)),
                istreambuf_iterator<char>());
```

条款30：确定目标区间足够大

- **知识点1：必须在目标区间内有足够大的，已经“实例化”的“空对象”来存放数值**
  - 无论如何，如果使用的算法需要指定一个目标区间，那么必须确保这个目标区间足够大，或者随着算法的运行而增大。
  - 要在算法执行的过程中增大目标区间，需要使用插入型迭代器。inserter等
- **情景1：使用transform算法移动元素，这个算法必须保证result容器是有已经分配的空间，否则就会出错**

```
int transmogrify(int x);                //该函数根据 x 生成一个新的值

vector<int> values;
...                                     //在 values 中存入一些值
vector<int> results;                    //将 transmogrify 作用在 values
transform(values.begin(), values.end(), //的每个对象上，并把返回值追加在
          results.end(),                //results 的末尾
          transmogrify);                //这段代码有一个错误！
```

```
vector<int> results;                        //将 transmogrify 作用在 values
transform(values.begin(), values.end(),     //的每个对象上，并将返回值插入到
          back_inserter(results),           //results 的末尾
          transmogrify);
```

条款31：了解各种与排序有关的选择

- **知识点1：sort算法和list：：sort**
  - sort算法安照一定要求对元素进行排序，要求排序的类满足严格弱序列
  - list：：sort是一个稳定的算法（相当于链表版stable-sort）
- **知识点2：partial—sort算法**
  - 这是一个对部分元素进行排序的功能，将所有元素中最符合要求的部分按照排序放在最前面

```
bool qualityCompare(const Widget& lhs, const Widget& rhs)
{
   //返回 lhs 的质量是否好于 rhs 的质量的结果
}
...
partial_sort(widgets.begin(),           //将质量最好的 20 个元素
             widgets.begin() + 20,      //顺序放在 widgets 的前
             widgets.end(),             //20 个位置上
             qualityCompare);
...                                     //使用 widgets
```

- **知识点3：nth-element算法**
  - 它可以对部分的元素进行排列，把符合要求的元素放在前面，但是不进行排序

```
nth_element(widgets.begin(),              //将最好的 20 个元素放在
            widgets.begin() + 19,         //widgets 的前部，但并不
            widgets.end(),                //关心它们的具体排列顺序
            qualityCompare);
```

- 它也可以用来找一个区间的中间值，或者某个特定百分比的值

```
//下面的代码找到区间中具有 75%质量的元素
vector<Widget>::size_type goalOffset =      //找出如果全排序的话，待查找
  0.25 * widgets.size();                    //的 Widget 离起始处有多远

nth_element(begin, begin + goalOffset, end, //找到 75%处的质量值
            qualityCompare);
...                                         //现在 begin + goalOffset
                                            //所指的元素具有 75%的质量

vector<Widget>::iterator begin(widgets.begin()); //两个便捷变量，分别代表
vector<Widget>::iterator end(widgets.end());     //widgets 的 begin 和 end
                                                 //迭代器
vector<Widget>::iterator goalPosition;           //用于定位感兴趣的元素

//下面的代码找到具有中间质量级别的 Widget
goalPostion = begin + widgets.size() / 2;        //如果全排序的话，待查找
                                                 //的 Widget 应该位于中间
nth_element(begin, goalPosition, end,            //找到 widgets 的中间质量值
            qualityCompare);
...                                              //现在 goalPosition 所指
                                                 //的元素具有中间质量
```

- **知识点4:partition算法**
  - 它可以简单的进行交换，将好的放在起始迭代器和自定义迭代器中间，不好的放在自定义迭代器和结束迭代器之间

```
bool hasAcceptableQuality(const Widget& w)
{
    //判断 w 的质量值是否为 2 或者更好
}

vector<Widget>::iterator goodEnd =      //将满足 hasAcceptableQuality
   partition(widgets.begin(),           //的所有元素移到前部，然后返回一个
             widgets.end(),             //迭代器，指向第一个不满足条件
             hasAcceptableQuality);     //的 Widget
```

  - 与sort、stable_sort、partial_sort和nth_element不同的是，partition和stable_partition只要求双向迭代器就能完成工作。所以，对于所有的标准序列容器，你都可以使用partition或者stable_partition。
- **知识点5：stable类算法**
  - 可以有stable形式的算法有：sort，partition等
  - stable算法和非stable算法的主要区别是，对于两个相同的变量，可以保证他们的相对位置不改变，但是非stable算法就不一定了
- **知识点6：算法选择**
  - 如果需要对vector、string、deque或者数组中的元素执行一次完全排序，那么可以使用sort或者stable_sort。
  - 如果有一个vector、string、deque或者数组，并且只需要对等价性最前面的n个元素进行排序，那么可以使用partial_sort。
  - 如果有一个vector、string、deque或者数组，并且需要找到第n个位置上的元素，或者，需要找到等价性最前面的n个元素但又不必对这n个元素进行排序，那么，nth_element正是你所需要的函数。
  - 如果需要将一个标准序列容器中的元素按照是否满足某个特定的条件区分开来，那么，partition和stable_partition可能正是你所需要的。
  - 如果你的数据在一个list中，那么你仍然可以直接调用partition，stable_partition算法；可以用list::sort来替代sort和stable_sort算法。但是，如果你需要获得partial_sort或nth_element算法的效果，那么，正如前面我所提到的那样，你可以有一些间接的途径来完成这项任务。
  - 除此以外，你可以通过使用标准的关联容器来保证容器中的元素始终保持特定的顺序，也可以考虑使用标准的非STL容器priority_queue，它总是保持其元素的顺序关系
- **知识点7：算法时间长短**
  - 首先，算法的选择更多的应该基于你需要完成的功能，而不是算法的性能，最好使用能恰好完成你的功能的算法
  - 1.partition
    2.stable_partition
    3.nth_element
    4.partial_sort
    5.sort
    6.stable_sort

条款32：如果确实需要删除元素，需要在remove这一类算法之后调用erase

- **知识点1：对于绝大部分容器remove不是真正意义上的删除，他做不到**
  - remove不是真正意义上的删除，他做不到，虽然他确实移除了元素，但是容器的大小是没有改变的。
  - 如果按照我们希望的，在删除的时候改变容器的大小，需要配合着erase一起调用

```
vector<int> v;                                  //同前
…
v.erase(remove(v.begin(), v.end(), 99), v.end()); //真正删除所有值
                                                  //等于 99 的元素
cout << v.size();                               //现在返回 7
```

- 知识点2：list：：remove是真正意义上的删除
  - list的remove是一个名为remove且确实删除了容器中元素的函数，他也更高效

条款33：对包含指针的容器使用remove这一类算法时候要特别小心

- **知识点1：remove算法对于指针使用可能会导致资源泄露**
  - 对于一个储存了容器的指针来讲，remove算法remove是容器里面指针本身，而不是指针指向的资源，如果直接调用remove就会导致资源泄露
  - 解决的方法是使用share-ptr，share-ptr会正常的释放资源，保证不泄露

- **条款34：了解那些算法要求使用排序区间作为参数**
  - **知识点1：需要排序区间的算法**

    - 所谓需要排序区间的算法，就是指的需要进行操作的区间对应必须是有序的（有operator<）的，并且有排序标准，否则效率大大降低
    - 所有的算法一般需要随机访问迭代器，如果是低级的迭代器可能效率大大降低或者无法运行
    - 注：unique和unique-copy的操作和remove差不多，所以如果用来排序指针需要谨慎小心
- **条款35：通过mismatch或lexicographical-compare实现简单的忽略大小写的字符串比较**
  - 对于实现字符串的比较，有两种常见的实现方式
  - **知识点1：mismatch**
    - mismatch算法接受两个迭代器，然后返回第一个字符不相同的位置
  - **知识点2：lexicographical-compare**
    - 这个算法是一个缩写版本stl算法，他可以与任何类型的值的区间一起工作
- **条款36：理解copy-if算法的实现**
  - 在C++11中，copy-if算法在algorithm库里可以找到了
- **条款37：使用accumulate或者for-each进行区间统计**
  - **知识点1：accumulate的使用**
    - accumulate定义在numeric中，它可以用来计算两个区间中的和

```
list<double> ld;                                          //创建一个 list，并加
...                                                       //入一些 double 值
double sum = accumulate( ld.begin(), ld.end(), 0.0); //计算它们的和，初始
                                                          //值为 0.0
```

    - 或者是传入一个大小和一个字符串，对其做加法也是可以的

```
string::size_type                                  //关于 string::size_type，
stringLengthSum(string::size_type sumSoFar,        //请见下文
                const string& s)
{
   return sumSoFar + s.size();
}

set<string> ss;                                    //创建一个存放 string 的
...                                                //容器，然后加入字符串
string::size_type lengthSum =                      //对 ss 中的每个元素调用 string-
   accumulate(ss.begin(), ss.end(),                //LengthSum，然后把结果赋给
      static_cast<string::size_type>(0),           //lengthSum。0 作为初始值
      stringLengthSum);
```

  - **知识点2：for-each的使用**
    - for_each也带两个参数：一个是区间，另一个是函数（通常是函数对象）——对区间中的每个元素都要调用这个函数，但是，传给for_each的这个函数只接收一个实参（即当前的区间元素）；for_each执行完毕后会返回它的函数（实际上，它返回的是这个函数的一份拷贝，

```
struct Point {...};                    //同前
class PointAverage:
   public unary_function<Point, void> { //见第 40 条
public:
   PointAverage():xSum(0), ySum(0), numPoints(0) {}
   void operator()(const Point& p)
   {
      ++numPoints;
      xSum += p.x;
      ySum += p.y;
   }
   Point result() const
   {
      return Point(xSum/numPoints, ySum/numPoints);
   }
private:
   size_t numPoints;
   double xSum;
   double ySum;
};
list<Point> lp;
...
Point avg = for_each(lp.begin(), lp.end(), PointAverage()).result();
```

- **条款38：遵循按值传递的原则来设计函数子类**
  - **知识点1：传递函数子**
    - 无论是C还是C++，都不允许将一个函数作为参数传递给另一个函数，当然也大概率不会把一个函数的引用传递给其他函数，我们一般都是使用函数的指针去传递
    - 为了能让函数对象很大或者大概率的保留多态性，也能按照按值传递的习惯，我们也就可以将所需的数据和虚函数从函数子类中分离出来，放到一个新的类中；然后在函数子类中包含一个指针，指向这个新类的对象。例如，如果你希望创建一个包含大量数据并且使用了多态性的函数子类

```
template<typename T>                       //针对修改后的 BPFC
class BPFCImpl:                            //的新的实现类
   public unary_function<T, void> {
private:
   Widget w;                               //原来 BPFC 中所有数据
```

```
    int x;                              //现在都放在这里
    ...
    virtual ~BPFCImpl();                //多态类需要虚析构函数
    virtual void operator()(const T& val) const;
friend class BPFC<T>;                   //允许 BPFC 访问内部数据
};

template<typename T>
class BPFC:                             //新的 BPFC 类：短小、单态
  public unary_function<T, void> {
private:
  BPFCImpl<T> *pImpl;                   //BPFC 唯一的数据成员
public:
  void operator()(const T& val) const  //现在这是一个非虚函数
  {                                     //将调用转到 BPFCImpl 中
    pImpl->operator()(val);
  }
  ...
};
```

- **条款39：确保判别式是纯函数**
  - **知识点1：理解题目对应的定义**
    - 判别式：一个判别式是一个返回值位bool类型的/可以隐式转换为bool类型的函数。
    - 纯函数：一个纯函数是指返回值仅仅依赖其参数的函数，只有当函数值发生变化的时候，对应的返回值参能发生变化
    - 对于纯函数，其参数必须是const，对应的函数也需要是const的
    - 判别式类：判别式类是一个函数子类，他的operator（）函数是一个判别式，STL能接受判别式的地方，也能接收判别式类
- **条款40：若一个类是函数子，则应使它可配接（C++11已经废弃）**
- **条款41：理解ptr-fun，mem-fun和mem-fun-ref的来由**
  - 知识点1：理解他们的又来
    - 这几个完全是用来掩盖一个C++在调用一个函数的时候语法不一致的问题的

```
f(x);                               //语法#1：f 为一个非成员函数
x.f();                              //语法#2：f 是成员函数，并且 x
                                    //是一个对象或对象的引用
p->f();                             //语法#3：f 是成员函数，并且 p
                                    //是一个指向对象 x 的指针
```

    - 一般而言，我们的STL都接受第一种形式的调用，为了满足下面两种形式，可以使用mem-fun或者mem-fun-ref，这两个是针对无参函数的调用的（即f必须是没有参数的）
  - 知识点2：C++11的形式
    - C++11中，mem-fun和mem-fun-ref变成了历史包袱，更好的是用mem-fn。
    - 但是bind他不香嘛为啥不用他。。有着相同的功效还没有参数的限制。
- **条款42：确保less<T>与operator<具有相同的意思**
  - 知识点1：less与<的关系
    - operator<不仅仅是less的默认实现方式，也是程序员期望less做的事情，让less不调用operator<而去做别的事情是不好的
    - 对于multiset类型的可以考虑定义一个函数类，直接替换默认比较方式less，而不是使用模板特例化去实现

```
struct MaxSpeedCompare:
   public binary_function<Widget, Widget, bool> {
   bool operator()(const Widget& lhs, const Widget& rhs) const
   {
      return lhs.maxSpeed() < rhs.maxSpeed();
   }
};
```

```
multiset<Widget, MaxSpeedCompare> widgets;
```

- **条款43：算法的调用优先于手写的循环**
  - 知识点1：算法的调用优先于手写循环的原因
    - 效率：算法通常比程序员自己写的循环效率更高
    - 正确性：自己写的算法更加容易出错（比如迭代器这块）
    - 可维护性：使用算法的代码通常比手写循环的代码更加的简洁明了
- **条款44：容器的成员函数优先于同名的算法**
  - 知识点1：原因
    - 首先，成员函数的行为与关联容器的其他成员函数能够保持更好的一致性。
    - 相等性和等价性的区别，算法一般关注于相等性，而容器函数一般是等价性
    - 成员函数的性能更加卓越，毕竟自己人才懂自己人
- **条款45：正确区分count，find等算法**
  - 知识点1：算法

想知道什么	使用算法		使用成员函数	
	对未排序的区间	对排序的区间	对 set 或 map	对 multiset 或 multimap
特定的值存在吗	find	binary_search	count	find
特定的值存在吗？如果存在，那第一个有该值的对象在哪里	find	equal_range	find	find 或 lower_bound（见下文）

想知道什么	使用算法		使用成员函数	
	对未排序的区间	对排序的区间	对 set 或 map	对 multiset 或 multimap
第一个不超过特定值的对象在哪里	find_if	lower_bound	lower_bound	lower_bound
第一个在某个特定值之后的对象在哪里	find_if	upper_bound	upper_bound	upper_bound
具有特定值的对象有多少个	count	equal_range（然后 distance）	count	count
具有特定值的对象都在哪里	find（反复调用）	equal_range	equal_range	equal_range

- **条款46：考虑使用函数对象而不是函数作为STL算法的参数**
  - 知识点1：为什么要考虑使用函数对象而不是函数本身作为STL算法的参数
    - 原因1：函数对象与内联性
      - 定义在class内部的函数是会被优化成inline的，然而在外部的函数使用inline之后编译器只是针对性的，不一定会去优化。
    - 原因2：调用函数名字作为函数背后的故事
      - 每次我们调用一个函数名字作为函数，背后都是传递一个函数的指针，这个指针去调用函数，相当于一次间接的调用，这种调用很难进行内联优化
    - 原因3：不一定正确的算法调用
      - 直接的去调用函数的名字，可能会导致因为无法进行隐式转换/类型不匹配等原因导致jj
  - 所以我们应该多使用函数对象而不是函数本身作为STL算法的参数，使用函数对象更加安全且快速
- **条款47：避免产生“直写型”代码**
  - 知识点1：不要嵌套太多代码，可能会导致很难理解
    - 不要嵌套过于多的函数，这样可能导致难理解的同时还会难以维护
    - 代码的可读性是非常重要的，有了可读性的代码才能维护
    - 对于嵌套较多且难理解的代码可以选择加上一些注释
- **条款48：总是#include正确的头文件**
  - 几乎所有的标准STL容器都被声明在与之同名的头文件中，比如vector被声明在<vector>中，list被声明在<list>中，等等。但是<set>和<map>是个例外，<set>中声明了set和multiset，<map>中声明了map和multimap。
  - 除了4个STL算法以外，其他所有的算法都被声明在<algorithm>中，这4个算法是accumulate（参见第37条）、inner_product、adjacent_difference和partial_sum，它们被声明在头文件<numeric>中。
  - 特殊类型的迭代器，包括istream_iterator和istreambuf_iterator（见第29条），被声明在<iterator>中。
  - 标准的函数子（比如less<T>）和函数子配接器（比如not、bind）被声明在头文件<functional>中。
- **条款49：学会分析STL相关编译器的诊断信息**
  - 如果你看见basic-string，可以把他换成string
  - 对于std::_xxx这种类型为stl内部模板，可以统一的把<>里面的内容替换为something
  - 如果你错误的使用了iterator，那么编译器诊断类型中往往包括指针类型
  - 如果诊断信息中提到了back-insert/front-insert/insert iterator，那么你一定是错误的调用了他们
  - 如果诊断信息中提到了bind，那么就是错误的调用了它
  - 输出迭代器以及由哪些inserter函数返回的迭代器在使用的时候犯了错误，那么就是可以看到与赋值操作符有关的错误内容
  - 如果错误消息来自于某一个STL算法的实现，那么大概率就是在调用算法的时候使用了错误的类型
- **条款50：熟悉与STL相关的web站点**