3.3 scoped_array

scoped_array 很像 scoped_ptr，它包装了 new[]操作符（不是单纯的 new）在堆上分配的动态数组，为动态数组提供了一个代理，保证可以正确地释放内存。

scoped_array 相当于 C++11 标准中管理数组对象用法的 unique_ptr。

3.3.1 类摘要

scoped_array 的类摘要如下：

```
template<class T>
class scoped_array                                          //noncopyable
{
public:
   explicit   scoped_array(T * p = 0);                      //构造函数
   ~scoped_array();                                         //析构函数

   void       reset(T * p = 0);                             //重置智能指针
   T &        operator[](std::ptrdiff_t i) const;           //重载 operator[]
   T *        get() const;                                  //获得原始指针

   explicit   operator bool() const;                        //显式 bool 值转型
   void       swap(scoped_array & b);                       //交换指针
};
```


scoped_array 的接口和功能几乎与 scoped_ptr 是相同的（甚至还要少一些），主要特点是：

- 构造函数接受的指针 p 必须是 new[]的结果，而不能是 new 表达式的结果；
- 没有*、->操作符重载，因为 scoped_array 持有的不是一个普通指针；
- 析构函数使用 delete[]释放资源，而不是 delete；
- 提供 operator[]操作符重载，可以像普通数组一样用下标访问元素；
- 没有 begin()、end()等类似容器的迭代器操作函数。

3.3.2 用法

scoped_array 与 scoped_ptr 源于相同的设计思想，故而用法非常相似：它只能在被声明的作用域内使用，不能拷贝、赋值。唯一不同的是，scoped_array 包装的是 new[]产生的指针，并在析构时调用 delete[]——因为它管理的是动态数组，而不是单个动态对象。

通常 scoped_array 的创建方式是这样的：

```
scoped_array<int> sa(new int[100]);          //包装动态数组
```

scoped_array 重载了 operator[]，因此它用起来就像是一个普通的数组，但因为它不提供指针运算，所以不能用“数组首地址+N”的方式访问数组元素：

```
sa[0] = 10;                                  //正确用法,使用 operator[]
*(sa + 1) = 20;                              //错误用法,不能通过编译!
```

在使用重载的 operator[]时要小心，scoped_array 不提供数组索引的范围检查，如果使用了超过动态数组大小的索引或者是负数索引将引发未定义行为。

下面的代码进一步示范了 scoped_array 的用法：

```
#include <boost/smart_ptr.hpp>
using namespace boost;

int main()
{
  int *arr = new int[100];                    //一个整数的动态数组
  scoped_array<int> sa(arr);                  //scoped_array 对象代理原始动态数组

  fill_n(&sa[0],100, 5);                      //可以使用标准库算法赋值数据
  sa[10] = sa[20] + sa[30];                   //用起来就像是个普通数组
}                                             //这里 scoped_array 被自动析构，
                                                  //释放动态数组资源
```

3.3.3 对比 unique_ptr

C++11 标准中的 unique_ptr 使用模板特化技术提供了对数组对象的支持，类代码摘要如下：

```
template <class T, class D>
class unique_ptr<T[], D> {                          //注意：对数组形式特化
public:
  typedef some_define    pointer;                   //内部类型定义
  typedef T              element_type;

  constexpr  unique_ptr() noexcept;                 //构造函数
  explicit   unique_ptr(pointer p) noexcept;

  ~unique_ptr();                                    //析构函数

  T&         operator[](size_t i) const;            //重载 operator[]
  pointer    get() const noexcept;                  //获得原始指针
  explicit   operator bool() const noexcept;        //显式 bool 值转型

  pointer    release() noexcept;                    //释放指针的管理权
  void       reset(pointer p) noexcept;             //重置智能指针

  void       swap(unique_ptr& u) noexcept;          //交换指针

  unique_ptr(const unique_ptr&) = delete;           //使用 delete 禁用拷贝
  unique_ptr& operator=(const unique_ptr&) = delete;
};
```

unique_ptr 的数组对象用法与 scoped_array 基本相同，但模板参数中需要声明为数组类型：

```
unique_ptr<int[]> up(new int[10]);              //注意模板参数和 new[]

assert(up);                                     //同样支持 bool 转换
up[0] = 10;                                     //使用 operator[]
cout << up[0] << endl;

up.reset();                                     //释放 new 出的数组内存空间
assert(!up);                                    //此时不管理任何指针
```

因为 unique_ptr 的数组用法仅是模板特化，所以它同样具有 unique_ptr 的其他功能，如比较运算，定制删除器等，功能要比 scoped_array 更多。

同样的，头文件<boost/smart_ptr/make_unique.hpp>里提供了创建数组对象

unique_ptr 的工厂函数 make_unique()：

```
auto a = boost::make_unique<int[]>(5);      //5 个元素的动态数组
a[0] = 100;                                 //operator[]访问第一个元素
a[4] = 500;                                 //operator[]访问最后一个元素
a[5] = 1000;                                //数组越界，未定义行为!
```

3.3.4 使用建议

scoped_array 没有给程序增加额外的负担，用起来很方便轻巧。它的速度与原始数组同样快，很适合那些习惯于用 new 操作符在堆上分配内存的程序员。但 scoped_array 的功能很有限，不能动态增长，没有边界检查，也没有迭代器支持，不能搭配 STL 算法，仅有一个纯粹的“裸”数组接口。而且，我们应当尽量避免使用 new[]操作符，它比 new 更可怕，是许多错误的来源，因为

```
int *p = new int[10];                       //创建动态数组
delete p;                                   //错误地使用 delete 删除数值指针!
```

这样的代码完全可以通过编译，无论是编译器还是程序员都很难区分出 new[]和 new 分配的空间，而错误地运用 delete 将导致资源异常。

在需要动态数组的情况下我们应该使用 std::vector，它比 scoped_array 提供了更多的灵活性，而只付出了很小的代价。由于 vector 有丰富的成员函数来操纵数据，能够使代码更加简单明了，易于维护。

除非对性能有非常苛刻的要求，或者编译器不支持标准库，否则本书不推荐使用 scoped_array，它只是为了与老式 C 风格代码兼容而使用的类，它的出现往往意味着你的代码中存在着隐患。

3.5 shared_array

shared_array 类似 shared_ptr，它包装了 new[]操作符在堆上分配的动态数组，同样使用引用计数机制为动态数组提供了一个代理，可以在程序的生命周期里长期存在，直到没有任何引用后才释放内存。

3.5.1 类摘要

shared_array 的类摘要如下：

```
template<class T>
class shared_array {
public:
  explicit           shared_array(T * p = 0);          //构造函数
  template<class D>  shared_array(T * p, D d);
  ~shared_array();                                     //析构函数

  shared_array(shared_array const & r);                //拷贝构造函数
  shared_array & operator=(shared_array const & r);    //拷贝赋值

  void                    reset(T * p = 0);            //重置指针
  template<class D> void reset(T * p, D d);

  T & operator[](std::ptrdiff_t i) const() const;      //重载操作符[]
  T * get() const;                                     //获得原始指针

  bool   unique() const;                               //是否唯一
  long   use_count() const;                            //引用计数

  void   swap(shared_array<T> & b);                    //交换指针
};
```


shared_array 的接口与功能几乎是与 shared_ptr 是相同的，主要区别是：

- 构造函数接受的指针 p 必须是 new[]的结果，而不能是 new 表达式的结果；
- 提供 operator[]操作符重载，可以像普通数组一样用下标访问元素；
- 没有*、->操作符重载，因为 shared_array 持有的不是一个普通指针；
- 析构函数使用 delete[]释放资源，而不是 delete。

3.5.2 用法

shared_array 就像是 shared_ptr 和 scoped_array 的结合体——既具有 shared_ptr 的优点，也具有 scoped_array 的缺点。有关 shared_ptr 和 scoped_array 的讨论大都适合它，因此这里不再详细讲解，仅给出一个小例子说明：

```
#include <boost/smart_ptr.hpp>
using namespace boost;

int main()
{
  int *p = new int[100];              //一个动态数组

  shared_array<int> sa(p);            //shared_array 代理动态数组
  assert(sa.unique());                //唯一持有指针
```

```
    shared_array<int> sa2 = sa;          //共享数组，引用计数增加
    assert(sa2.use_count() == 2);        //引用计数增加

    sa[0] = 10;                          //可以使用 operator[]访问元素
    assert(sa2[0] == 10);
}                                        //离开作用域,自动删除动态数组
```

同样的，在使用 shared_array 重载的 operator[]时要小心，shared_array 不提供数组索引的范围检查，如果使用了超过动态数组大小的索引或者是负数索引将引发可怕的未定义行为。

shared_array 能力有限，多数情况下它可以用 shared_ptr<std::vector>或者 std::vector<shared_ptr>来代替，这两个方案具有更好的安全性和更多的灵活性，而所付出的代价几乎可以忽略不计。

3.6.4 enable_shared_from_raw

smart_ptr 库在未文档化的头文件<boost/smart_ptr/enable_shared_from_raw.hpp>里提供另外一个与 enable_shared_from_this 类似的辅助类 enable_shared_from_raw，它不要求对象必须被一个 shared_ptr 管理，可以直接从一个原始指针创建出 shared_ptr。

enable_shared_from_raw 的类摘要如下：

```
class enable_shared_from_raw
{
protected:
    enable_shared_from_raw();
    enable_shared_from_raw( enable_shared_from_raw const & );
    enable_shared_from_raw & operator=( enable_shared_from_raw const & );

    ~enable_shared_from_raw()
private:
    template<class Y> friend class shared_ptr;

    template<typename T>
    friend boost::shared_ptr<T>  shared_from_raw(T *);

    template<typename T>
    friend boost::weak_ptr<T>    weak_from_raw(T *);
};
```

enable_shared_from_raw 利用了 shared_ptr 的别名构造函数(3.4.9 节)特性，内部持有一个 void*的空指针 shared_ptr 作为引用计数的观察者，从而达到管理原始指针的目的。

enable_shared_from_raw 同样需要继承使用，但它不是模板类，所以不需要指定模板参数，比 enable_shared_from_this 写法上要简单一些。它不提供成员函数 shared_from_this()，而是用两个 friend 函数 shared_from_raw()和 weak_from_raw()完成创建智能指针的工作。

但请注意:在调用 shared_from_raw()后,由于存在 shared_ptr 成员变量的原因，对象内部会有一个 shared_ptr 的强引用，所以即使其他的 shared_ptr 都析构了原始指针也不会被自动删除（因为 use_count()>=1）——这使得 enable_shared_from_raw 用法略微不同于 enable_shared_from_this，它可以安全地从一个普通对象而非指针创建出 shared_ptr。

示范 enable_shared_from_raw 用法的代码如下：

```
#include <boost/smart_ptr/enable_shared_from_raw.hpp>

class raw_shared: public boost::enable_shared_from_raw
{
public:
    raw_shared()
    {   cout << "raw_shared ctor" << endl;    }
    ~raw_shared()
    {   cout << "raw_shared dtor" << endl;    }
};


int main()
{
```

```
    raw_shared x;                                   //一个普通对象
    assert(!weak_from_raw(&x).use_count());         //此时无引用，注意要用&取地址

    auto px = shared_from_raw(&x);                  //获取 shared_ptr
    assert(px.use_count() == 2);                    //引用计数为 2!
}                                                   //对象自动删除
```

把 enable_shared_from_raw 应用于原始指针要当心，使用不当有可能造成内存泄漏：

```
int main()
{
    auto p = new raw_shared;                        //创建一个原始指针

    auto wp = weak_from_raw(p);                     //获取 weak_ptr
    assert(wp.use_count() == 0);                    //此时无引用

    auto sp = shared_from_raw(p);                   //获取 shared_ptr
    assert(sp.use_count() == 2);                    //引用计数为 2!

    auto sp2 = sp;                                  //拷贝一个 shared_ptr
    auto wp2 = weak_from_raw(p);                    //获取 weak_ptr
    assert(wp2.use_count() == 3);                   //引用计数为 3
}                                                   //对象没有被删除，内存泄漏!
```

如果在代码里的某个时刻使用 shared_ptr 来管理原始指针——而不是调用 shared_from_raw()，那么指针的管理权就会转移到 shared_ptr，从而可以正确地自动销毁，例如：


```
int main()
{
    auto p = new raw_shared;                        //创建一个原始指针

    decltype(shared_from_raw(p)) spx(p);            //使用 shared_ptr 管理指针
    ...                                             //其他操作
}                                                   //对象被自动删除
```

enable_shared_from_raw 的用法比较特殊，实际应用的场景较少，也许这就是它没有在 Boost 库里被文档化的原因。

4.5 swap

boost::swap 是对标准库里的 std::swap 的增强和泛化，为交换两个变量（可以是 int 等内置数据类型，或者是类实例、容器）的值提供了便捷的方法。

为了使用 boost::swap，需要包含头文件<boost/swap.hpp>[1]，即

```
#include <boost/swap.hpp>
```

4.5.1 原理

先让我们来复习一下 C++98 标准中的 std::swap()：

```
// C++98 里 std::swap 的典型实现，具体代码依版本而不同
template<typename T>
void swap(T& a, T& b)
{
      T tmp(a);
      a = b;
      b = tmp;
}
```

从代码中可以看出，std::swap()要求交换的对象必须是可拷贝构造和可拷贝赋值的，它提供的是最通用同时也是效率最低的方法，需要进行一次复制构造和两次赋值操作，如果交换的对象很大，那么运行代价会相当昂贵。

C++11 标准中使用转移语义对 std::swap()进行了优化，避免了拷贝的代价(C++11.20.2.2)：

```
template<typename T>
void swap(T& a, T& b)                    //C++11 标准里的 std::swap 典型实现
{
      T tmp = std::move(a);              //move 语义，把 a“偷”到 tmp
      a = std::move(b);                  //move 语义，把 b“偷”到 a
      b = std::move(tmp);                //move 语义，把 tmp“偷”到 b
}
```

但不是所有的类都实现了自己的转移构造和赋值函数的，而且编译器的支持也是个必须考虑的问题，所以对于我们自己写的类，最好能够实现优化的 swap()来提高效率。

解决方案有两种：第一种方案是直接利用函数重载，编写一个同名的 swap 函数，这个 swap()再调用类内部的高效成员交换函数，这样编译器在编译时就不会使用 std::swap()。第二种方案是使用 ADL[2]查找模板特化的 std::swap。这两种方案就是 boost::swap 的工作原理。

boost::swap 查找有无针对类型 T 的 std::swap()的特化或者通过 ADL 查找模板特化的 swap()，如果有则调用，如果两种查找都失败时则退化为 std::swap()。此外，boost::swap 还增加了一个很实用的功能——对 C++内建数组交换的支持（已经被收入 C++11 标准）。

---

[1] 真正的实现源代码在<boost/core/swap.hpp>。

[2] argument dependent lookup，参数依赖查找，又称 Koening Lookup。

boost::swap()函数的声明是：

```
template<class T1, class T2>
void swap(T1& left, T2& right);
```

由于 boost::swap()与 std::swap()同名，所以我们不能使用 using 语句打开 boost 名字空间，应该总以 boost 名字空间限定的方式调用它。

4.5.2 交换数组

boost::swap 可以直接交换两个数组的内容，但要求参与交换的两个数组必须具有相同的长度。下面的代码使用标准库算法 fill_n 将两个数组分别赋值为 5 和 20，然后调用 boost::swap()交换：

```
int a1[10];                                  //两个数组
int a2[10];

std::fill_n(a1, 10, 5);                      //fill_n 赋初始值
std::fill_n(a2, 10, 20);

boost::swap(a1, a2);                         //交换两个数组的内容
```

boost::swap 交换数组内容的实现很简单，它使用了一个 for 循环，对数组中的每个元素调用单个元素版的 boost::swap 完成整个数组内容的交换。在上面的代码执行后 a1 中元素的值将为 20，而 a2 中元素的值将为 5。

如果企图用 boost::swap 交换两个长度不相同的数组，那么将无法通过编译：

```
int a1[10], a2[12];                          //两个长度不相同的数组
boost::swap(a1, a2);                         //发生编译错误
```


4.5.3 特化 std::swap

接下来我们用一个简单的三维空间的点 point 作为例子，示范模板特化的方法使用 boost::swap。它实现了内部高效的交换函数：

```
class point
{
  int x, y, z;
public:
  explicit point(int a=0, int b=0, int c=0):x(a),y(b),z(c){}

  void print()const
  {       cout << x <<","<< y <<","<< z << endl;    }

  void swap(point &p)                        //内置高效交换函数
  {
      std::swap(x, p.x);
      std::swap(y, p.y);
      std::swap(z, p.z);
      cout << "inner swap" << endl;
  }
};
```

特化 std::swap()的方法需要向 std 名字空间添加自定义函数：

```
namespace std
{
  template<>
  void swap(point &x, point &y)              //模板特化 swap 函数
```

```
    {   x.swap(y);}
}

int main()
{
  point a(1,2,3), b(4,5,6);

  cout << "std::swap" << endl;
  std::swap(a,b);                              //调用 std::swap

  cout << "boost::swap" << endl;
  boost::swap(a, b);                           //调用 boost::swap
}
```

程序运行结果如下：

```
std::swap
inner swap
boost::swap
inner swap
```

由于我们在名字空间特化了 std::swap，因此，boost::swap 与 std::swap 的效果相同，都使用了特化后的 swap 函数。

4.5.4 特化 ADL 可找到的 swap

依然使用刚才的 point 类，但这次我们不变动 std 名字空间，而是在全局域实现 swap 函数：


```
void swap(point &x, point &y)                  //全局域的 swap 函数
{ x.swap(y);}

int main()
{
  point a(1,2,3), b(4,5,6);

  cout << "std::swap" << endl;
  std::swap(a,b);                              //调用 std::swap


  cout << "boost::swap" << endl;
  boost::swap(a, b);                           //调用 boost::swap
}
```

程序运行结果如下：

```
std::swap
boost::swap
inner swap
```

这段代码的运行结果与之前的特化 std::swap 有明显不同，std::swap 使用了标准的交换操作，而 boost::swap 通过 ADL 规则找到了全局名字空间的特化交换函数，实现了高效的交换。

如果读者担心在全局名字空间编写自由函数 swap 会造成名字“污染”，则可以把特化的 swap 加入到 boost 名字空间，或者其他 ADL 可以找到的名字空间。

4.5.5 使用建议

我们应该尽量使用 boost::swap，它提供了比 std::swap 更好的优化策略。如果你

自己写的类实现了高效的交换（应该总这样），或者想交换两个数组的内容，那么就使用 boost::swap，它可以保证总能够找到最恰当的交换方法。

变量值交换是一个基础但很重要的操作，几乎所有 Boost 库组件都实现了自己的 swap 成员函数，并且用 boost::swap 来提高交换的效率，可以在很多代码中找到 swap 的实现范例。

在附录 A 推荐书目[6]的条款 25 有关于 swap 的详细论述，读者可以参考。

4.6 singleton

singleton 即单件模式，实现这种模式的类在程序生命周期里只能有且仅有一个实例。单件模式是一个很有用的创建型模式，有许多实际的应用，并被广泛且深入地研究。如果读者还不熟悉这个模式，则最好先阅读附录 A 推荐书目[1]，它提供了对 singleton 模式完整的表述。

虽然单件模式非常重要，但很遗憾目前 Boost 中并没有专门的单件库，而仅是在 serialization 库中有一个并不十分完善的实现，它位于名字空间 boost::serialization。

为了使用 singleton，需要包含头文件<boost/serialization/singleton.hpp>，即：


```
#include <boost/serialization/singleton.hpp>
using boost::serialization::singleton;
```

4.6.1 类摘要

singleton 的类摘要声明如下：

```
template <typename T>
class singleton : public boost::noncopyable
{
public:
    static const T & get_const_instance();
    static       T & get_mutable_instance();
};
```

singleton 把模板参数 T 实现为一个单件类，对类型 T 的要求是有缺省构造函数，而且在构造和析构时不能抛出异常，因为单件在 main()前后构造和析构，如果发生异常则会无法捕获。

singleton 提供两个静态成员函数访问单件实例，可分别获得常对象和可变对象。这种区分是出于线程安全的考虑，常对象单件总是线程安全的，因为它不会改变内部状态，而可变对象单件则不是线程安全的，可能会发生线程竞争问题。

4.6.2 用法

在 4.5 节我们定义了一个三维点 point，它具有缺省构造函数，因此可以应用于 singleton，实现一个唯一的原点实例。

为了演示 singleton 的创建与销毁，下面的代码增加了构造函数与析构函数：

```
class point
{
public:
  point(int a=0, int b=0, int c=0):x(a),y(b),z(c)      //构造函数
  {   cout << "point ctor" << endl;}

  ~point()                                              //析构函数
  {   cout << "point dtor" << endl;}
...
};
```

singleton 的用法非常简单，只需要把想成为单件的类作为它的模板参数就可以了，接下来的工作由 singleton 自动完成。可能唯一的一点不方便之处是过长的类型名，但可以用 typedef 来简化：

```
#include <boost/serialization/singleton.hpp>
using boost::serialization::singleton;

int main()
{
  cout << "main() start" << endl;
  typedef singleton<point> origin;                      //单件类定义

  origin::get_const_instance().print();                 //常对象
  origin::get_mutable_instance().print();               //可变对象

  cout << "main() finish" << endl;
}
```


代码中需要注意的是类名 singleton，还有就是访问实例的函数不是常用的 instance()，而是 get_const_instance()和 get_mutable_instance()，程序运行结果是：

```
point ctor
main() start
0,0,0
0,0,0
main() finish
point dtor
```

除了这种模板方式， singleton 还可以通过继承的方式来使用，把 singleton 作为单件类的基类，并把单件类作为 singleton 的模板参数：

```
#include <boost/serialization/singleton.hpp>
using boost::serialization::singleton;

class point : public singleton<point>                   //注意这里
{...}


int main()
{
```

```
    cout << "main() start" << endl;

    point::get_const_instance().print();
    point::get_mutable_instance().print();

    cout << "main() finish" << endl;
}
```

这段代码与模板参数的用法仅有很小的不同，最重要的变化在于 point 类的基类声明处 public 继承 singleton<point>（如果读者不熟悉泛型编程，则可能会对这种父类中使用子类的形式有些困惑），因此 point 继承了 singleton 的所有能力，包括不可拷贝和单件。

与模板参数方式相比，继承方式实现单件模式更为彻底一些（很像 noncopyable 的用法），使被单件化的类成为了一个真正的单件类，而模板参数方式则更“温和”一些，它包装（wrap）了被单件化的类，对原始类没有任何影响。

5.5 tokenizer

tokenizer 库是一个专门用于分词（token）的字符串处理库，可以使用简单易用的方法把一个字符串分解成若干个单词。它与 string_algo 库的分割算法很类似，但有更多的变化。

tokenizer 位于名字空间 boost，为了使用 tokenzier 组件，需要包含头文件<boost/ tokenizer.hpp>，即：

```
#include <boost/tokenizer.hpp>
using namespace boost;
```

5.5.1 类摘要

tokenizer 类是 tokenizer 库的核心，它以容器的外观提供分词的序列，类摘要如下：

```
template <
    typename TokenizerFunc   = char_delimiters_separator<char>,
    typename Iterator        = std::string::const_iterator,
    typename Type            = std::string
  >
class tokenizer
{
    tokenizer(Iterator first, Iterator last, const TokenizerFunc& f);
    tokenizer(const Container& c, const TokenizerFunc& f);

    void     assign(Iterator first, Iterator last);
    void     assign(const Container& c);
    void     assign(const Container& c, const TokenizerFunc& f)

    iterator  begin() const ;
    iterator  end()   const;
}
```


tokenizer 接受三个模板类型参数，分别是：

- TokenizerFunc：tokenizer 库专用的分词函数对象，默认是使用空格和标点分词；
- Iterator：字符序列的迭代器类型；
- Type：保存分词结果的类型。

这三个模板类型都提供了默认值，但通常只有前两个模板参数可以变化，第三个类型通常只能选择 std::string 或者 std::wstring,这也是它位于模板参数列表最后的原因。

tokenizer 的构造函数接受要进行分词的字符串，可以以迭代器的区间形式给出，也可以是一个有 begin()和 end()成员函数的容器。

assign()函数可以重新指定要分词的字符串，用于再利用 tokenizer。

tokenizer 具有类似标准容器的接口，begin()函数使 tokenizer 开始执行分词功能，返回第一个分词的迭代器，end()函数表明迭代器已经到达分词序列的末尾，分词结束。

5.5.2 用法

tokenizer 的用法很像 string_algo 的分割迭代器，但要简单一些。我们可以像使

用一个容器那样使用它，向 tokenizer 传入一个欲分词的字符串构造，然后用 begin()获得迭代器反复迭代，就可以轻松完成分词功能。

由于 tokenizer 具有类似容器的接口，所以可以使用 auto 来搭配新式 for 或者 foreach 库（8.1 节）来遍历分词结果。

示范 tokenizer 用法的代码如下：

```
#include <boost/tokenizer.hpp>
using namespace boost;
int main()
{
  string str("Link raise the master-sword.");

  tokenizer<> tok(str);              //使用缺省模板参数创建分词对象

  //可以像遍历一个容器一样使用 for 循环获得分词结果
  for(auto& x: tok)                  //或者是 BOOST_FOREACH(auto& x, tok)
  {   cout << "["<< x << "]";    }
}
```

程序运行结果如下：

```
[Link][raise][the][master][sword]
```

注意一下这个分词结果，tokenizer 默认把所有的空格和标点符号作为分隔符，因此分割出的只是单词，这与 string_algo::split 的算法含义（分割字符串）有所差别。

5.5.3 分词函数对象

tokenizer 作为分词的容器本身的用法很简单，它的真正威力在于第一个模板类型参数 TokenizerFunc。TokenizerFunc 是一个函数对象，它决定如何进行分词处理。TokenizerFunc 同时也是一个规范，只要具有合适的 operator()和 reset()语义的函数对象都可以用于 tokenizer 分词。

tokenizer 库提供预定义好的四个分词对象，它们是：

- char_delimiters_separator：使用标点符号分词，是 tokenizer 默认使用的分词函数对象。但它已经被声明废弃，应当尽量不使用它；
- char_separator：它支持一个字符集合作为分隔符，默认的行为与 char_delimiters_separator 类似；
- escaped_list_separator：用于 CSV 格式（逗号分隔）的分词；
- offset_separator：使用偏移量来分词，在分解平文件格式的字符串时很有用。

下面我们分别来介绍后三个分词函数对象。为了简化程序输出代码，接下来的代码使用了一个辅助模板函数 print()，它遍历输出分词的结果：

```
template<typename T>
void print(T &tok)
{
  for (auto& x : tok)
  {   cout << "[" << x << "]";   }
  cout << endl;
}
```

5.5.4 char_separator

char_separator 使用一个字符集合作为分词依据，行为很类似 split 算法（参见 5.4.10 节），它的构造函数声明是：

```
char_separator(const Char* dropped_delims,
               const Char* kept_delims = 0,
               empty_token_policy empty_tokens = drop_empty_tokens);
```

构造函数中的参数含义如下：

- 第一个参数 dropped_delims 是分隔符集合，这个集合中的字符不会作为分词的结果出现；
- 第二个参数 kept_delims 也是分隔符集合，但其中的字符会保留在分词结果中；
- 第三个参数 empty_tokens 类似 split 算法的 eCompress 参数，处理两个连续出现的分隔符。如为 keep_empty_tokens 则表示连续出现的分隔符标识了一个空字符串，相当于 split 算法的 token_compress_off 值；如为 drop_empty_tokens，则空白单词不会作为分词的结果。

如果使用默认的构造函数，不传入任何参数，则其行为等同于 char_separator(" ", 标点符号字符，drop_empty_tokens)，以空格和标点符号分词，保留标点符号，不输出空白单词。

示范 char_separator 用法的代码如下：

```
char str[] = "Link ;; <master-sword> zelda";

char_separator<char> sep;                    //一个 char_separator 对象

//传入 char_separator 构造分词对象分词并输出
tokenizer<char_separator<char>, char*> tok(str, str +strlen(str), sep);
print(tok);

tok.assign(str, str +strlen(str),            //重新分词
  char_separator<char>(" ;-", "<>"));
print(tok);

tok.assign(str, str +strlen(str),            //重新分词
  char_separator<char>(" ;-<>", "", keep_empty_tokens));
print(tok);
```


这段代码我们对 tokenizer 的模板参数稍微做了一下改变，第二个参数改为 char*，这使得 tokenizer 可以分析 C 风格的字符串数组。同时构造函数也必须变为传入字符串的首末位置，不能仅传递一个字符串首地址，因为字符串数组不符合容器的概念。

第一次分词我们使用 char_separator 的缺省构造，以空格和标点分词，保留标点作为单词的一部分，并抛弃空白单词；第二次分词我们使用" ;-"和"<>"共 5 个字符分词，保留<>作为单词的一部分，同样抛弃空白单词；最后一次分词我们同样使用" ;-<>"分词，但它们都不作为单词的一部分，并且我们保留空白单词。

程序运行结果如下：

```
[Link][;][;][<][master][-][sword][>][zelda]
[Link][<][master][sword][>][zelda]
[Link][][][][][master][sword][][zelda]
```

5.5.5 escaped_list_separator

escaped_list_separator 是专门处理 CSV 格式（Comma Separated Values，逗号分隔值）的分词对象，它的构造函数声明是：

```
escaped_list_separator(Char e = '\\', Char c = ',',Char q = '\"')
```

escaped_list_separator 的构造函数参数一般都取默认值，含义如下：

- 第一个参数 e 指定了字符串中的转义字符，默认是斜杠（\）；
- 第二个参数是分隔符，默认是逗号（,）；
- 第三个参数是引号字符，默认是引号（"）；

示范 escaped_list_separator 用法的代码如下：

```
string str = "id,100,name,\"mario\"";          //CSV 格式的字符串

escaped_list_separator<char> sep;              //分词对象
tokenizer<escaped_list_separator<char> > tok(str, sep);
print(tok);
```

程序运行结果如下：

```
[id][100][name][mario]
```

5.5.6 offset_separator

offset_separator 与前两种分词函数对象不同，它分词的功能不基于查找分隔符，而是使用偏移量的概念，在处理某些不使用分隔符而使用固定字段宽度的文本时很有用。它的构造函数声明如下：


```
template <typename Iter>
offset_separator(Iter begin, Iter end,
                 bool wrap_offsets = true,
                 bool return_partial_last = true);
```

offset_separator 的构造函数接收两个迭代器参数（也可以是数组指针）begin 和 end，指定分词用的整数偏移量序列，整数序列的每个元素是分词字段的宽度。

bool 参数 wrap_offsets 决定是否在偏移量用完后继续分词，bool 参数 return_partial_last 决定在偏移量序列最后是否返回分词不足的部分。这两个附加参数的默认值都是 true。

示范 offset_separator 用法的代码如下：

```
string str = "2233344445";
int offsets[] = {2,3,4};

offset_separator sep(offsets, offsets + 3, true, false);
tokenizer<offset_separator> tok(str, sep);
print(tok);

tok.assign(str, offset_separator(offsets, offsets + 3, false));
print(tok);

str += "56667";
tok.assign(str, offset_separator(offsets, offsets + 3, true, false));
print(tok);
```

代码中我们用一个数组 offsets 指定了三个偏移量，要求分割出三个长度分别为 2、3、4 的单词。程序运行结果如下：

```
[22][333][4444][5]
[22][333][4444]
[22][333][4444][55][666]
```

请读者注意输出的第一行和最后一行，两者的分词都设置了 return_partial_last 为 false，但输出却略有不同。

这是因为 return_partial_last 只影响偏移量序列的最后一个元素（即代码中的 offsets[2]=4），对于序列中的其他元素则不起作用，无论是否分词不足均输出。所以输出中第一行的[5]因为它对应偏移量序列的第一个元素，所以总能输出，而最后一行字符串末尾原本应该有输出[7]，但因为 return_partial_last 为 false 且它对应偏移量序列的最后一个元素，所以不被输出。

5.5.7 tokenizer 库的缺陷

tokenizer 库可以很容易地执行分词操作，但它存在一些固有的缺陷。

缺陷之一是它只支持使用单个字符进行分词，如果要分解如“||”等多个字符组成的分隔符则无能为力，只能自己定义分词函数对象，或者使用 string_algo、正则表达式等其他字符串功能库。

它的另一个小的（但很麻烦的）缺陷是对 wstring（unicode）缺乏完善的考虑，也没有像 string_algo 那样使用 std::locale()，不方便使用。

例如，如果我们使用 wstring，string_algo 库可以很简单地分词：

```
wstring str(L"Link mario samus");           //注意宽字符的使用方式

auto p =                                    //分割迭代器
      make_split_iterator(str, first_finder(L" ", is_iequal()));
decltype(p) endp;

for (;p != endp; ++p)
{     wcout << L"[" << *p << L"]" ; }
```

为了使用 wstring，我们需要把字符串类型改为 wstring，字符串常量用 L 标记是宽字符，再使用 wcout 输出。由于使用了 auto 自动推导类型，处理字符串的 string_algo 算法完全不需要变动。

而使用 tokenizer，除了以上的操作，我们必须完整无误地写出它的全部模板参数，像这样：

```
char_separator<wchar_t>                          sep(L" ");
tokenizer<char_separator<wchar_t>,
          wstring::const_iterator, wstring >     tok(str, sep);

for(auto& x : tok)
{     wcout << L"["<< x << L"]";}
```

使用 escaped_list_separator 等其他分词函数对象也是如此，如：

```
escaped_list_separator<wchar_t>                  sep;
tokenizer<escaped_list_separator<wchar_t>,
          wstring::const_iterator, wstring >     tok(str, sep);
```

这显得过于烦琐。如果我们将来某一天必须要支持 wstring，对这些代码的修改和维护将会是个不小的工作量。

针对这个问题，本书提供了一个包装类（元函数），它能够部分解决这个问题：

```
template<typename Func, typename  String = std::string>
struct tokenizer_wrapper
{
  //typedef typename Func::string_type String;
  typedef tokenizer<Func, typename String::const_iterator, String > type;
};
```

tokenizer_wrapper 有两个模板参数：第一个 Func 是分词函数对象，第二个是分词所使用的字符串类型，内部用这两个模板类型 typedef 简化了 tokenizer 的模板声明，以::type 返回真正的 tokenizer 类型。

使用这个包装类，上面的声明可以简化为：

```
tokenizer_wrapper<
      char_separator<wchar_t>,wstring>::type       tok(str, sep);
tokenizer_wrapper<
      escaped_list_separator<wchar_t>, wstring>::type  tok(str, sep);
```

这样看起来就好多了。

请读者注意 tokenizer_wrapper 类内部的那行注释，很遗憾 tokenizer 库的分词函数对象均把字符串类型作为它内部的 typedef，不能被外界使用，并且 offset_separator 不提供这个 typedef，否则包装类可以节省一个模板参数，像这样：


```
template<typename Func >
struct tokenizer_wrapper
{
  //不能通过编译！string_type 是私有 typedef
  typedef typename Func::string_type String;
  typedef tokenizer<Func, typename String::const_iterator, String > type;
};
```

但现实不允许我们这么做，tokenizer 库设计之初就没有对这些问题做很好的考虑，除非改动源代码。但本书不建议修改，因为正则表达式和 string_algo 通常都比 tokenizer 工作得更好。

6.4.9 函数执行监视器

test 库在 UTF 框架底层提供一个函数执行监视器类 execution_monitor，它被 UTF 用于单元测试，但也可以被用于生产代码。execution_monitor 可以监控某个函数的执行，即使函数发生预想以外的异常，也能够保证程序不受影响地正常运行，异常将会以一致的方式被 execution_monitor 处理。

execution_monitor 位于名字空间 boost，在需要使用 execution_monitor 的地方应加入如下声明：

```
#include <boost/test/execution_monitor.hpp>
using namespace boost;
```

基本用法

execution_monitor 目前可以监控返回值为 int 或者可转换为 int 的函数，之后成员函数 execute()就可以监控执行被包装的函数。

execution_monitor 的监控执行语句需要嵌入在一个 try-catch 块里。如果一切正常，那么被 execution_monitor 监控执行的函数就像未被监控一样运行并返回。否则，如果发生了未捕获的异常、软硬件 signal/trap 以及 assert 断言，那么 execution_monitor 就会捕获这个异常，重新抛出一个 execution_exception 异常，它存储了异常相关的信息。

execution_exception 不是标准库异常 std::exception 的子类，必须在 catch 块明确地写出它的类型，否则 execution_monitor 抛出的异常不会被捕获。

示范 execution_monitor 用法的代码如下：

```
#include <boost/test/execution_monitor.hpp>
using namespace boost;

int f()                                  //一个简单的测试函数,必须是无参返回值为 int
{
  cout << "f execute." << endl;
  throw "a error accoured";              //抛出一个未捕获的异常
  return 10;
}

int main()
{
  execution_monitor em;                                    //声明一个监视器对象
  try                                                      //开始 try-catch 块
  {
      em.execute(f);                                       //监控执行 f
  }
  catch (execution_exception& e)                           //捕获异常
  {
      cout << "execution_exception" << endl;
      cout << e.what().begin()<< endl;                     //输出异常信息
  }
}
```

请读者注意在上面的代码中对 execution_exception 异常的使用，在输出它存储的信息时我们使用了 e.what().begin()的方式。这是因为 execution_monitor 被设计为在很少或者没有内存的情况下也可以使用，故它没有使用 std::string 来表示字符串，

而是使用了一个内部类 const_string。const_string 默认不支持流输出，如果要使用流输出功能，需要包含头文件<boost/test/utils/basic_cstring/io.hpp>。

程序运行结果如下：

```
f execute.
execution_exception
C string: a error accoured
```

其他用法

除了基本的监控函数执行，execution_monitor 还有其他的用法。它提供了 p_timeout、p_auto_start_dbg 等读写属性用来设置监控器的行为，或者检测内存泄露，但这些功能不是完全可移植的，有的功能仅限于某些编译器或操作系统。

execution_monitor 也可以用于统一处理程序的异常，使用户不必自己编写错误处理代码。在这种情况下，程序抛出的异常类型必须是 C 字符串、std::string 或者 std::exception 三者之一，才能够被 execution_exception 所处理，通常这能够适合大多数应用。

如果因为某些原因必须使用自定义的异常类，而又想利用执行监控器的监控功能，那么可以定义异常类的翻译函数，并注册到 execution_monitor 中。例如：

```
struct my_error                                          //一个自定义异常类
{
  int err_code;                                          //错误代码
  my_error(int ec):err_code(ec){}                        //构造函数
};
void translate_my_err(const my_error& e)                 //翻译函数
{
  cout << "my err = " << e.err_code << endl;             //输出到 cout
}
int g()                                                  //被监控函数
{
  cout << "g execute." << endl;
  throw my_error(100);                                   //抛出自定义异常
  return 0;
}
int main()
{
  execution_monitor em;

  //注册异常翻译函数
  em.register_exception_translator<my_error>(&translate_my_err);
  try                                                    //开始 try-catch 块，监控函数的执行
  {
      em.execute(g);
  }
  catch (const execution_exception& e)
  {
      cout << "execution_exception" << endl;
      cout << e.what().begin();
  }
}
```

程序的运行结果如下：

```
g execute.
my err = 100
```

6.4.10 程序执行监视器

test 库在函数执行监视器 execution_monitor 的基础上提供程序执行监视器，它的目的与 execution_monitor 相似，监控整个程序的执行，把程序中的异常统一转换为标准的操作系统可用的错误返回码。

程序执行监视器的用法类似 minimal test，只需要包含一个头文件，并实现与 main() 具有相同签名的 cpp_main()：

```
#include <boost/test/included/prg_exec_monitor.hpp>
int cpp_main( int argc, char* argv[] ){...}
```

注意：cpp_main()必须要返回一个整数，它不同于 main()，不具有默认返回 0 值的能力。

程序执行监视器使用一个函数对象包装了 cpp_main()，将它转换成一个返回 int 的无参函数对象，然后使用 execution_monitor 监控执行。因此它的行为基本上与 execution_ monitor 相同，当 cpp_main()发生异常或者返回非 0 值时就会被捕获，并把异常信息输出到屏幕上。