说明

这个文档只有3章，所有章节请在百度云下载：http://pan.baidu.com/s/1ntOc6WL

第1页：Boost库简介

1）进一步学习C++

虽然C++是公认比较难学的编程语言；但是它能给投入精力的开发人员带来丰厚的效益。Boost是下一代的C++标准库，学习Boost是C++学习路线的中重要的一步。Boost库几乎包含了计算机学科的所有知识，学习Boost库将会增加学习者对计算机学科的本质认识。作为一个C++开发者，如果不掌握Boost，那么至少丧失了一半使用C++的好处，同时会耗费数倍的开发精力和时间(引用《Boost程序库完全开发指南-罗剑锋》)。

2）Boost官方网站

Boost库的官方网站是：www.boost.org。这个网站是全球能获取正版的源代码和boost使用手册的唯一网站；从官网获取文档和信息很考验读者的英语水平，英语阅读水平不好的读者慎入。本课程撰写的参考的第一手资料就是官方的说明文档。

3）Boost库内容

Boost库由一系列子库组成。列出开发者可能感兴趣的库如下：

string_algo
    字符串算法
smart_ptr
    智能指针
asio
    异步通信
thread
    多线程库
function
    函数对象
lexical_cast
    文本转换
regex
    正则表达式
signals2
    信号处理

4）学习建议

学习Boost库要求对C++基础知识和STL库要有相当程度的了解。在学习本库之前，你必须要掌握的课程如下：

一般要求：
    C语言程序设计基础
    数据结构
    C++程序设计基础
    C++STL库应用与开发
    操作系统
    计算机网络

```
部分内容要求：
    微积分、线代和离散数学
```

特别地，如果读者对于C语言指针、C++基础知识和STL相关知识点还很陌生，那么笔者强烈不推荐开始学习这个课程。学习本课程最好的情况是掌握好基础知识而且有过实际开发经验之后。如果读者发现自己阅读吃力，一定要及时翻阅以前的课程内容。

第2页：安装Boost库

1）说明

本课程的演示的环境运行在Linux系统下。操作系统为:Ubuntu 14.04LTS,并且使用g++编译器。Boost库的安装如下面所示。

2）获取拷贝

在官方网站www.boost.org获取最新的安装包：boost_1_XX.tar.bz2。版本不同，文件名的版本号也不同；安装包放在拥有系统权限的目录下。

3）解压安装包

获得安装包后，先解压出来，然后切换到解压目录。命令如下：

```
tar --bzip2 -xf /你的路径/boost_1_XX.tar.bz2
cd /你的路径/boost_1_XX
```

4）安装boost

输入以下命令执行安装：

```
sudo ./bootstrap.sh
sudo ./b2 install
```

安装进度根据机器配置，可能十几分钟到一个小时，请耐心等待。

4）或者，直接从软件中心安装boost

在Ubuntu14系统下，也可以从软件中心直接安装；建议使用这种安装方式，因为它是直接把文件复制到文件系统中，速度比较快。命令如下：

```
sudo apt-get update
sudo apt-get install libboost-dev
```

6）验证安装

安装完成后，使用编辑器编辑如下程序：

```
1  #include<iostream>
2  #include<boost/lexical_cast.hpp>
3  using namespace std;
4  using namespace boost;
5  int main(int argc, char **argv)
6  {
7      string str="123456";
8      int num=123456;
9      int a=lexical_cast<int>(str);
```

```
    string b=lexical_cast<string>(num);
    cout<<a<<endl<<b<<endl;
    return 0;
}
```

运行结果如下：

```
123456
123456
```

如果成功输出上面的结果，说明Boost库安装成功。上面的安装步骤在最新的Boost 1.57版本全部通过。

第1页：字符串算法库简介

1）简介

字符串算法库提供了一系列支持字符串运算的算法，这些算法作为对STL库的补充。

2）第一个例子

下面是一个大写转换和文本替换的例子：

```
#include<boost/algorithm/string.hpp>
#include<iostream>
using namespace std;
using namespace boost;
int main()
{
    string str1("    hello world! ");
    to_upper(str1);
    trim(str1);

    string str2=to_lower_copy(
         ireplace_first_copy(
             str1,"world!",",how are you?"));

    cout<<str1<<"\n"<<str2;
}
```

运行结果如下：

```
HELLO WORLD!
hello ,how are you?
```

3）说明

学习好这些算法有助于降低C++文本处理程序的难度;本章节将会使用大幅内容来讲解这个库。

第2页：大小写转换

1）简介

STL库虽然有大小写转换的函数，但是很遗憾，它仅对于单个字符起作用。

2）算法介绍

大小写转换分为4个函数，分别为：to_upper_copy()、to_upper和to_lower()、to_lower_copy()。例子程序如下：

```
#include<boost/algorithm/string.hpp>
#include<iostream>
using namespace std;
using namespace boost;
int main()
{
    string str1="abcdefg";
    string str2=to_upper_copy(str1);

    string str3="HELLO";
    to_lower(str3);

    cout<<str2<<"\n"<<str3;
}
```

运行结果如下：

```
ABCDEFG
hello
```

3）带"copy"与不带"copy"

在字符串算法库中，带"copy"的函数会返回计算完毕后的值，不会对参与运算的字符串起作用；不带"copy"的函数将会改变参与运算的值。使用哪一种方法视情况而定。

第3页：截断算法

1）简介

截断算法用来去掉字符串首尾不符合条件的字符；最经典是去掉字符串的首尾空格函数。

2）函数列表

从尾部截断：

```
trim_left_copy_if()
trim_left_if()
trim_left_copy()
trim_left();
```

从开头截断：

```
trim_right_copy_if()
trim_right_if()
trim_right_copy()
trim_right();
```

从两边截断：

```
trim_copy_if()
trim_if()
trim_copy()
trim();
```

3）使用例子：

```
#include<boost/algorithm/string.hpp>
#include<iostream>
using namespace std;
using namespace boost;
int main()
{
    string str1="      abc     ";
    string str2="ABcdefGHI";

    trim(str1);  //去掉首尾空格
    trim_if(str2,is_upper());//去掉首尾的大写字母

    cout<<str1<<"\n"<<str1;
}
```

运行结果如下：

```
abc
cdef
```

3）带"if"与不带"if"

带if的截断函数的第二个参数需要传入一个“谓词函数”；谓词函数可以是字符串算法库自带的谓词函数，这些函数后面将会讲解。谓词函数用来控制要截断的字符。如果不带if，那么默认截断空格符。

第4页：断言算法

1）断言

断言算法用来判断两个字符串之间是否满足特定条件；它返回的值显然是逻辑型的。

2）函数列表

包括：

```
starts_with()
istarts_with()
   断言是否以特定字符串开始
ends_with()
iends_with()
   断言是否以特定字符串结束
contains()
icontains()
   断言是否包括某个子串
equals()
iequals()
   断言两个字符串是否相等
lexicographical_compare()
ilexicographical_compare()
   断言第一个参数在字典顺序中靠前
all()
   断言是否符合给定的谓词函数
```

3）使用例子：

```
#include <boost/algorithm/string.hpp>
#include<iostream>
```

```
using namespace std;
using namespace boost;
int main()
{
    cout<<starts_with("abcd","a")<<endl;    //判断是否以a开始
    cout<<contains("abcdefg","cde")<<endl; //判断是否包含子串
    cout<<lexicographical_compare("abcd","abce")<<endl;//按字典判断字符串大小
    cout<<equals("abcd","abcd")<<endl;//判断相等
    cout<<all("ABCDEF",is_upper())<<endl;  //判断是否都符合特定条件（都是大写）
}
```

运行结果如下：

```
1
1
1
1
1
```

3）带"i"与不带"i"

在字符串算法库中，带i开头的函数，就是对大小写不敏感；否则就是对大小写敏感。有了这些带i开头的函数，解决的多年以来不少C++程序员的心病，因为以前的算法库中的断言算法都是不区分大小写的。

第5页：谓词函数

1）谓词函数

谓词函数不能单独存发挥作用，它必须配合带有“if”的函数或者all()函数一起使用；它的本质是函数对象。谓词函数拥有一种模式：它可以检测一个字符是否符合给定的条件并反馈给调用的函数；这些反馈信息将会影响到调用它的函数的行为。谓词函数用“is”开头。

2）使用谓词函数的例子

```
#include <boost/algorithm/string.hpp>
#include<iostream>
using namespace std;
using namespace boost;
int main()
{
    string str1="am12389e";
    string str2="abcdefgh";
    //在trim_if函数中使用谓词
    trim_if(str1,is_alpha());    //去掉str1两头的字母
    cout<<str1<<endl;
    //在all函数中使用谓词
    cout<<all(str2,is_from_range('a','z'))<<endl;    //断言str2是否所有的字符都在a-z之
}
```

运行结果如下：

```
12389
1
```

3）谓词函数列表

包括：

is_space()
识别空格符
is_alnum()
识别字母和数字字符
is_alpha()
识别字母字符
is_cntrl
识别控制字符
is_digit()
识别数字字符
is_graph()
识别图形字符
is_lower()
识别小写
is_upper()
识别大写
is_print()
识别可打印字符
is_punct()
识别标点符号字符
is_xdigit()
识别十六进制字符
is_any_of()
识别传入的字符
is_from_range()
识别在范围内的字符

说明：is_any_of()谓词函数可以传入一个字符串，那么将会识别字符串中的每一个字符。谓词函数可以通过不同的组合实现复杂的功能，比如:is_from_range('a','z') || is_digit() 。

3）自定义谓词函数

如果系统自带的谓词函数不够用了，那么需要实现自己符合需求的谓词函数。自定义一个谓词函数在本章的第9页有讲解。

第6页：替换与删除

1）替换与删除

替换和函数非常多，下面将会逐一讲解。

2）删除与替换第一次出现的字符串

下面的函数只会对第一次出现的目标字符串起作用。包括的函数如下：

replace_first()
替换第一个
replace_first_copy()
替换第一个并拷贝
ireplace_first()
忽视大小写替换第一个
ireplace_first_copy()
忽视大小写替换第一个并拷贝
erase_first()
删除第一个

erase_first_copy()
　删除第一个并拷贝
ierase_first()
　忽视大小写删除第一个
ierase_first_copy()
　忽视大小写删除第一个并拷贝

```
#include<boost/algorithm/string.hpp>
#include<iostream>
using namespace std;
using namespace boost;
int main()
{
    string str1="bcbcbc";
    string str2="BCBCBC";
    //删除第一个
    erase_first(str1,"bc");
    //替换第一个
    replace_first(str2,"BC","MM");
    cout<<str1<<"\n"<<str2<<endl;
}
```

运行结果如下：

```
bcbc
MMBCBC
```

3）删除与替换最后一次出现的字符串

下面的函数只会对最后一次出现的目标字符串起作用。包括的函数如下：

replace_last()
　替换最后一个
replace_last_copy()
　替换最后一个并拷贝
ireplace_last()
　忽视大小写替换最后一个
ireplace_last_copy()
　忽视大小写替换最后一个并拷贝
erase_last()
　删除最后一个
erase_last_copy()
　删除最后一个并拷贝
ierase_last()
　忽视大小写删除最后一个
ierase_last_copy()
　忽视大小写删除最后一个并拷贝

```
#include<boost/algorithm/string.hpp>
#include<iostream>
using namespace std;
using namespace boost;
int main()
{
    string str1="abcabcabc";
    string str2="BCBCBC";
    //删除最后一个
    erase_last(str1,"bc");
    //替换最后一个
    replace_last(str2,"BC","MM");
    cout<<str1<<"\n"<<str2<<endl;
```

```
14  }
```

运行结果如下：

```
abcabca
BCBCMM
```

4）删除与替换第N次出现的字符串

下面的函数只会对第N(从0开始)次出现的目标字符串起作用,这些函数的第三个参数需要传入一个N值。包括的函数如下：

```
replace_nth()
    替换最后一个
replace_nth_copy()
    替换最后一个并拷贝
ireplace_nth()
    忽视大小写替换最后一个
ireplace_nth_copy()
    忽视大小写替换最后一个并拷贝
erase_nth()
    删除最后一个
erase_nth_copy()
    删除最后一个并拷贝
ierase_nth()
    忽视大小写删除最后一个
ierase_nth_copy()
    忽视大小写删除最后一个并拷贝
```

```
1  #include<boost/algorithm/string.hpp>
2  #include<iostream>
3  using namespace std;
4  using namespace boost;
5  int main()
6  {
7      string str1="abcabcabc";
8      string str2="BCBCBC";
9      //删除第2个bc
10     erase_nth(str1,"bc",1);
11     //替换第2个bc为MM
12     replace_nth(str2,"BC",1,"MM");
13     cout<<str1<<"\n"<<str2<<endl;
14 }
```

运行结果如下：

```
abcaabc
BCMMBC
```

5）删除与替换所有出现的字符串

下面的函数会对出现的所有目标字符串起作用。包括的函数如下：

```
replace_all()
    替换所有
replace_all_copy()
    替换所有并拷贝
ireplace_all()
    忽视大小写替换所有
```

ireplace_all_copy()
忽视大小写替换所有并拷贝
erase_all()
删除所有
erase_all_copy()
删除所有并拷贝
ierase_all()
忽视大小写删除所有
ierase_all_copy()
忽视大小写删除所有并拷贝

```
#include<boost/algorithm/string.hpp>
#include<iostream>
using namespace std;
using namespace boost;
int main()
{
    string str1="abcabcabc";
    string str2="BCBCBC";
    //删除所有的bc
    erase_all(str1,"bc");
    //替换所有的bc为MM
    replace_all(str2,"BC","MM");
    cout<<str1<<"\n"<<str2<<endl;
}
```

运行结果如下：

```
aaa
MMMMMM
```

6）删除与替换开头N个字符

下面这些函数的作用是删除和替换字符串开头的N个字符

replace_head()
替换开头N个字符
replace_head_copy()
替换开头N个字符并拷贝
erase_head()
删除开头N个字符
erase_copy()
删除开头N个字符并拷贝

```
#include<boost/algorithm/string.hpp>
#include<iostream>
using namespace std;
using namespace boost;
int main()
{
    string str1="abcabcabc";
    string str2="abcdefg";
    //删除前3个字符
    erase_head(str1,3);
    //替换前3个字符为ABC
    replace_head(str2,3,"ABC");
    cout<<str1<<"\n"<<str2<<endl;
}
```

运行结果如下：

```
abcabc
ABCdefg
```

7）删除与替换结尾的N个字符

下面这些函数的作用是删除和替换字符串结尾的N个字符；使用规则请参照上面一个算法。

```
replace_tail()
    替换结尾的N个字符
replace_tail_copy()
    替换结尾的N个字符并拷贝
erase_tail()
    删除结尾的N个字符
erase_copy()
    删除结尾的N个字符并拷贝
```

8）使用正则表达式删除和替换

正则表达式是编写字符串处理程序的有力工具；后面将会单独讲解正则表达式在字符串算法库中的使用。

第7页：查找和分割算法

1）查找子串

包括：

```
find_all()
    查找所有子串
ifind_all()
    忽略大小写查找所有子串
```

使用例子如下：

```
#include<boost/algorithm/string.hpp>
#include<iostream>
using namespace std;
using namespace boost;
int main()
{
    string str1="今天的天气真的很好，小明的心情也非常的好";
    //声明查找结果
    vector<string>vs;
    //查找字符 的
    find_all(vs,str1,"的");
    cout<<"找到了"<<vs.size()<<"处\n";
    //显示找到的结果
    for(unsigned int i=0;i<=vs.size()-1;i++)
    {
        cout<<vs[i]<<endl;
    }
}
```

运行结果如下：

```
找到了4处
的
的
的
的
```

保存查找函数产生的结果可以是如下类型：

```
容纳字符串的容器：
    std::vector<std::string>
容纳iterator_range<string::iterator>的容器：
    std::list<boost::iterator_range<std::string::iterator>>
```

2）字符串分割

字符串分割函数包括：

```
split()
    用给定的谓词函数分割字符串
```

使用例子如下：

```
#include <boost/algorithm/string.hpp>
#include<iostream>
using namespace std;
using namespace boost;
int main()
{
   string str1="hello,my name is Colo;l am fine .how old are you?";
      //声明查找结果
      vector<string>vs;
      //按标点符号分割字符串
      split(vs,str1,is_punct(),token_compress_off);
      cout<<"分割了"<<vs.size()<<"处\n";
      for(unsigned int i=0;i<=vs.size()-1;i++)
      {
          cout<<vs[i]<<endl;
      }
}
```

运行结果如下：

```
分割了5处
hello
my name is Colo
l am fine
how old are you
```

说明：split函数有4个参数，第一个参数为容纳结果的容器；第二个为被分割的字符串；第三个参数为一个谓词函数对象，用来指明如何分割。第四个参数为一个常量：token_compress_off或者token_compress_on;当常量值为off的时候，会分割相邻的两个谓词函数成立的字符串为空字符串；也就是说，产生的结果中会存在空字符串。当常量值为on的时候，就会把多个符合分割条件的字符串视为一个，从而不会产生空字符串。

容纳分割函数产生的结果可以是如下类型：

```
容纳字符串的容器：
    std::vector<std::string>
容纳iterator_range<string::iterator>的容器：
    std::list<boost::iterator_range<std::string::iterator> >
```

3)使用iterator_range<string::iterator>数据类型

它就是一个数据类型而已，并没有什么特殊性。range指的是boost.range,它是一种数据结构，用来描述一组线性的数据，比如字符串、vector这样的数据结构都属于range的概念。因此，string_algo库的设计的主要目的是为range数据类型服务的，但我们一般把它于string。也就是说，字符串算法库是工作在

range数据类型上的，任何符合range样式的数据类型都能使用这个库，当然包括string。range数据类型在其它章节另外讲解。iterator_range是range数据结构的迭代器。下面是一个例子：

```
#include<boost/algorithm/string.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main(int argc, char** argv)
{
    string str1 = "Long Long ago,the story was start on an dark night.";
    //声明容纳查询结果的容器
    list<boost::iterator_range<string::iterator> >lrs;
    //声明查询结果的迭代器
    list<boost::iterator_range<string::iterator> >::iterator li;
   //声明查询结果
    iterator_range<string::iterator>ir;
    //在str1中查询"on"
    lrs = find_all(lrs,str1,"on");
    //输出查询到的个数
    cout<<"查询到"<<lrs.size()<<"个on"<<endl;
    //遍历查询结果，输出查询到的下标
    int i=0;
    for(li=lrs.begin();li!=lrs.end();li++)
    {
        ir=*li;
        i++;
        string::iterator si=ir.begin();
        cout<<"第"<<i<<"个on的下标："<<(si-str1.begin())<<endl;
    }
    return 0;
}
```

运行结果：

```
查询到3个on
第1个on的下标：1
第2个on的下标：6
第3个on的下标：34
```

第8页：字符串合并

1）合并

和并运算是分割运算的逆运算；它指的是把分布在泛型容器中的字符串通过谓词函数给出的规则合并成一个新的字符串。这些函数包括：

```
join()
    合并所有子串
join_if()
    按给定的谓词函数规则合并
```

使用例子如下：

```
#include<boost/algorithm/string.hpp>
#include<iostream>
using namespace std;
using namespace boost;
//定义谓词类
class  predicate
{
public:
```

```
    //定义要实现的函数对象
     struct is_short_than5
    {
        bool operator() (string s) const
        {
            return s.length()<=5;
        }
    };
};
int main()
{
    vecto<std::string>vs;
    vs.push_back("Hello");
    vs.push_back("l am");
    vs.push_back("fine");
    vs.push_back("l am fine too");
    //合并所有的字符串，用"|"连接
    string str1=join(vs,"|");
    //合并长度不大于5的字符串，用"|"连接
    string str2=join_if(vs,"|",predicate::is_short_than5());
    cout<<str1<<endl<<str2;
}
```

运行结果如下：

```
Hello|l am|fine|l am fine too
Hello|l am|fine
```

说明:上面的程序实现了一个自定义的谓词函数，并在join_if()函数中使用了这个谓词，虽然这段程序简单，但是里面的含义却不简单；那么谓词函数的行为是怎样控制的呢？

第9页：子串查找算法

1）查找算法返回的类型

这些查找函数全部返回一个iterator_range<string::iterator>数据类型，它的本质是一个容纳string::iterator的boost::range对象的迭代器，通过迭代器我们可以方便地对原字符串修改和遍历访问。

2）查找第一个和最后一个子串

包括如下函数：

```
find_first()
ifind_first()
   查找第一个子串
last_first()
ilast_first()
   查找最后一个子串
```

使用例子如下：

```
#include<boost/algorithm/string.hpp>
#include<iostream>
using namespace std;
using namespace boost;
int main()
{
     string str1="123456";
    //下面是一个容纳string迭代器的range迭代器
```

```
    //反正有点绕，看不懂的请自行补充stl的课程
    iterator_range<string::iterator>r;
    r=find_first(str1,"23");
    //输出下标
    cout<<"第一个\"23\"开始的下标："<<r.begin()-str1.begin()<<endl;
    //通过string::iterator修改str1
    *r.begin()='w';
    cout<<"修改str1后,str1="<<str1<<endl;
}
```

运行结果如下：

```
第一个"23"开始的下标：1
修改str1后,str1=1w3456
```

3）查找第N个子串

包括如下函数：

```
find_nth()
ifind_nth()
    查找第N个子串
```

使用例子如下：

```
#include<boost/algorithm/string.hpp>
#include<iostream>
using namespace std;
using namespace boost;
int main()
{
    string str1="1234235623923";
    //查找第2个“23”
    iterator_range<string::iterator>r=find_nth(str1,"23",2);
    //输出下标
    cout<<"第2个\"23\"开始的下标："<r.begin()-str1.begin()<<endl;
    //通过string::iterator修改str1
    *r.begin()='w';
    cout<<"修改str1后,str1="<<str1<<endl;
}
```

运行结果如下：

```
第2个"23"开始的下标：8
修改str1后,str1=12342356w3923
```

说明:N从0开始。

3）查找开头和结尾N个子串

下面函数返回字符串的开头或结尾的N个子串的迭代器对象，函数包括：

```
find_head()
    得到前N个字符的迭代器对象
find_tail()
    得到结尾N个字符的迭代器对象
```

使用例子如下：

```
#include<boost/algorithm/string.hpp>
#include<iostream>
using namespace std;
using namespace boost;
```

```
int main()
{
     string str1="1234567";
    //返回前3个字符的迭代器对象
    iterator_range<string::iterator>r=find_head(str1,3);
    //遍历访问并修改之
    for(string::iterator si=r.begin();si!=r.end();si++)
    {
        *si='*';
    }
    cout<<str1<<endl;
}
```

运行结果如下：

```
***4567
```

第10页：通用查找与查找器

1）通用查找算法与查找器

查找就是子串定位；当所有的查找算法都失败了之后，就要考虑使用通用查找算法了，通用查找算法引用了"查找器"的概念；查找器的本质与谓词一样属于函数对象。查找器定义了如何在给定的字符串中查找的模式，然后使用通用查找算法来查找。通用查找算法是：

```
find()
   通过给定的查找器查找子串
```

使用例子如下：

```
#include<boost/algorithm/string.hpp>
#include<iostream>
using namespace std;
using namespace boost;
int main()
{
    string str1="11*aB*Ab*aB*AB**56";
    //使用查找器(finder)查找第一个ab,忽视大小写
    iterator_range<string::iterator>isi=find(str1,first_finder("ab",is_iequal()));
    //通过string::iterator修改str1
    *(isi.begin())='O';
    cout<<"第一个ab的下标:"<<isi.begin()-str1.begin()<<endl;
    cout<<"修改后str1="<<str1<<endl;
}
```

运行结果如下：

```
第一个ab的下标:3
修改后str1=11*OB*Ab*aB*AB**56
```

2）算法库内部定义的查找器

我们可以随意使用这些查找器，列表如下：

```
first_finder()
last_finder()
nth_finder()
   匹配第一个、最后和第N个的查找器
head_finder
tail_finder()
```

匹配首尾N个字符的查找器
token_finder
匹配记号的查找器
regex_finder()
匹配正则表达式的查找器

说明：上面的记号和正则表达式查找器后面章节讲解。

第11页：查找与分割迭代器

1）查找与分割迭代器

一个查找或分割迭代器代表着一个查询到的子串。查找和分割迭代器由make_find_iterator()和make_split_iterator()函数通过给定的查找器返回。用于std::string的查找与分割迭代器的外观如下：

find_iterator<string::iterator>
表示一个查找结果
split_iterator<string::iterator>
表示一个分割结果

使用方法如下：

```
#include<boost/algorithm/string.hpp>
#include<iostream>
using namespace std;
using namespace boost;
int main()
{
     string str1="11*aB*Ab*aB*AB**56";
   //定义查找和分割迭代器
   find_iterator<string::iterator> fsi;
   split_iterator<string::iterator> ssi;

   //使用查找器查找第一个23
   fsi=make_find_iterator(str1,first_finder("Ab"));
   //使用查找器分割字符串
   ssi=make_split_iterator(str1,first_finder("*"));
   //取得迭代器指向的字符串,然后输出
   string s1= copy_range<std::string>(*fsi) ;
   string s2=copy_range<std::string>(*ssi) ;
   cout<<s1<<endl<<s2<<endl;
}
```

运行结果如下：

```
第一个ab的下标:3
修改后str1=11*OB*Ab*aB*AB**56
```

2）不断遍历迭代器对象获取结果

查找与分割迭代器可以像stl中的迭代器那样，不断使用++操作符获取下一个迭代器对象，直到获取不到为止。例子程序如下：

```
#include<boost/algorithm/string.hpp>
#include<iostream>
using namespace std;
using namespace boost;
int main()
{
    string str1="123423323";
```

```
8       string str2="a*b*c";
9       //定义查找和分割迭代器
10      find_iterator<string::iterator> fsi;
11      split_iterator<string::iterator> ssi;
12      //演示迭代查找算法
13      cout<<"发现23的下标:";
14      for(fsi=make_find_iterator(str1,first_finder("23"));!fsi.eof();fsi++)
15      {
16          //获取下标和并修改str1
17          int index=fsi->begin()-str1.begin();
18          cout<<index<<" ";
19          *(fsi->begin())='@';
20      }
21      cout<<"\n修改后str1="<<str1<<endl;
22      //演示迭代分割算法
23      for(ssi=make_split_iterator(str2,first_finder("*"));!ssi.eof();ssi++)
24      {
25          //简单输出分割的结果
26          string r=copy_range<std::string>(*ssi) ;
27          cout<<"新分割:"<<r<<endl;
28      }
29  }
```

运行结果如下：

```
发现23的下标:1 4 7
修改后str1=1@34@33@3
新分割:a
新分割:b
新分割:c
```

说明：查找与分割迭代器访问std::iterator成员必须用“->”操作符，想想是为什么呢?作者一定知道，就是不告诉你。

下面是字符串算法库的所有函数与数据结构
有星号代表本章节没有讲解，请查询官方网站。
大小写转换算法：

- to_upper()
- to_lower()
- to_upper_copy()
- to_lower_copy()

截断算法：

- trim()
- trim_right()
- trim_left()
- trim_copy()
- trim_left_copy()
- trim_right_copy()
- trim_if()
- trim_left_copy_if()
- trim_left_if()
- trim_right_copy_if()
- trim_right_if()
- trim_copy_if()

断言算法：

- starts_with()
- istarts_with()
- ends_with()

iends_with()
contains()
icontains()
equals()
iequals()
lexicographical()
ilexicographical_comp()
all()

查找算法：

find_first()
ifind_first()
find_last()
ifind_last()
find_nth()
ifind_nth()
find_head()
find_tail()
*find_token()
*find_regex()
find()

消除和替换算法：

replace_first()
replace_first_copy()
ireplace_first()
ireplace_first_copy()
erase_first()
erase_first_copy()
ierase_first()
ierase_first_copy()
replace_last()
replace_last_copy()
ireplace_last()
ireplace_last_copy()
erase_last()
erase_last_copy()
ierase_last()
ierase_last_copy()
replace_nth()
replace_nth_copy()
ireplace_nth()
ireplace_nth_copy()
erase_nth()
erase_nth_copy()
ierase_nth()
ierase_nth_copy()
replace_all()
replace_all_copy()
ireplace_all()
ireplace_all_copy()
erase_all()
erase_all_copy()

ierase_all()
ierase_all_copy()
replace_head()
replace_head_copy()
erase_head()
erase_head_copy()
replace_tail()
replace_tail_copy()
erase_tail()
erase_tail_copy()
replace_regex()
replace_regex_copy()
erase_regex()
erase_regex_copy()
replace_all_regex()
replace_all_regex_copy()
erase_all_regex()
erase_all_regex_copy()
*find_format()
*find_format_copy()
*find_format_all()
*find_format_all_copy()

分割算法：

find_all()
ifind_all()
*find_all_regex()
split()
split_regex()
*iter_find()
*iter_split()

合并算法：

join()
join_if()

查找器：

first_finder()
last_finder()
nth_finder()
head_finder()
tail_finder()
*token_finder()
*range_finder()
*regex_finder()

格式器：

*const_formatter()
*identity_formatter()
*empty_formatter()
*regex_formatter()

迭代器：

find_iterator
split_iterator

谓词（器）：

is_space()
is_alnum()
is_alpha()
is_cntrl（）
is_digit()
is_graph()
is_lower()
is_upper()
is_print()
is_punct()
is_xdigit()
is_any_of()
is_from_range()
*is_classified()

第1页：设计动机

1）设计动机

文本转换库(lexical_cast)是一个支持字面常量和字符串类型之间相互转换的库。很多时候，我们需要把int类型转换到std::string,后者需要相反的转换。文本转换库就是为了实现这些需求而设计的。使用lexical_cast最经典的例子就是应用程序读取和加载配置文件中的设定参数。

2）标准库的局限性

标准的C和C++库中提供了一系列的转换函数，比如:atoi()；但是它们只能从一个方向转换；而且转换对象只能是int、long和double这些基本数据类型。最重要的是，标准库缺乏对自定义数据类型的转换规则。由于传统的转换方式存在这么多的缺陷，那么lexical_cast库的存在试图解决这个缺陷。

3）lexical_cast的优势

lexical_cast库的设计充分地利用了C++语言的模板编程，它支持任意数据类型到string的转换，或者相反。使用lexical_cast很简单，只需要开发者在表达式级别上的学习就能掌握这个库。lexical_cast在执行转换的过程中，转换效率和性能也说的过去。那么我们接下来开始学习lexical_cast吧。

第2页：字符串与数值间的转换

1）数值转换到字符串

转换例子程序如下：

```
#include <iostream>
#include<boost/lexical_cast.hpp>
using namespace boost;
using namespace std;
int main(int argc, char **argv)
{
    int a=123456;
    long b=12345678999;
    //把int和long转换到string
    string stra=lexical_cast<std::string>(a);
```

```
    string strb=lexical_cast<std::string>(b);
    cout<<stra<<std::endl<<strb;
    return 0;
}
```

运行结果：

```
123456
12345678999
```

2）字符串转换到数值

转换例子程序如下：

```
#include <iostream>
#include<boost/lexical_cast.hpp>
using namespace boost;
using namespace std;
int main(int argc, char **argv)
{
    //字符串转换到int
    int a=lexical_cast<int>("123");
    cout<<(a+1)<<endl;
    return 0;
}
```

运行结果：

```
124
```

3）异常的捕获

当转换过程发生失败的时候，会抛出异常，下面演示异常的捕捉：

```
#include <iostream>
#include<boost/lexical_cast.hpp>
using namespace boost;
using namespace std;
int main(int argc, char **argv)
{
    try
    {
        //一行很失败的代码
        short a=lexical_cast<short>("abcd");
    }
    catch(const bad_lexical_cast &e)
    {
        cout<<e.what()<<endl;
    }
    return 0;
}
```

运行结果：

```
bad lexical cast: source type value could not be interpreted as target
```

第3页：转换原理

1）lexical_cast实现原理

lexical_cast实质上是一个模板函数；它的基石是stringstream，利用标准库的流操作来完成转换。整个转换过程先利用"<<"读入stringstream，然后再利用">>"输出到目的地就可以了。这个过程跟cout和cin的使用是一个原理。

2）先理解cout与cin拥有类似转换功能

我们知道，cin的类型是istream，定义为标准输入流；cout的类型是ostream，定义为标准输出流。其实我们天天用的cout与cin就拥有完成string到数值相互转换的功能，只是我们忽视了而已。

例子程序如下：

```
#include<iostream>
using namespace std;
int main(int argc, char **argv)
{
    int d;
    cin>>d;
    cout<<d<<endl;
}
```

在上面的例子程序中发生了两个转换：

```
1:cin从标准输入设备读取一行字符串，转换为int后存入d
2:把d转换为字符串后，利用cout输出
```

2）使用stringstream进行转换

与istream和ostream一样，stringstream也是一种流类型。使用stringstream把int转换成string的例子如下：

例子程序如下：

```
#include <string>
#include <sstream>
#include <iostream>
int main()
{
    //演示int到string的转换
    std::stringstream stream;
    std::string result;
    int i = 1000;
    stream << i;
    stream >> result;
    std::cout << result << std::endl;
    return 0;
}
```

运行结果：

```
1000
```

3）对转换对象的要求

从上面的例子可以看出，一个对象是否可以利用stringstream实现与string的相互转换，那么这个对象必须符合如下3点要求：

```
1:可以重载<<运算符
2:可以重载>>运算符
3:拥有公有的拷贝构造函数
```

知道了lexical_cast库的工作原理后，有兴趣的读者可以阅读实现这个库的源代码；动手能力较强的读者也可以试着设计一个这样的库。

第1页：日期处理简介

1）设计动机

日期处理的范围是年、月和日，不包括时、分和秒。日期处理功能几乎是所有开发项目的基础设施；然而，C++在日期处理上没有一个标准的接口。是的，我们很多优秀的开发者不得不借用C语言中的tm结构体去完成各种日期处理业务；恐怕这种情况已经发生了无数次了。正是这样，这个库的设计试图弥补C++在日期处理上的缺陷。如果你在工程中使用这个库，你使用了几次，那么你就受益了几次。

2）格里高利日期表示系统与date_time库

本章讲解格利高里日期系统，它属于date_time库的内容。由于date_time库非常庞大，所以分为若干个章节讲解。

4）基本概念

在学习格利高利日期系统之前，先掌握3个概念：

时间点Point
　表示时刻
时间长度Duration
　表示时间持续长度
时间区间Period
　表示开始与结束之间所有时间点的集合

上面三个概念对应着某种数据类型，这三种数据类型间又定义了各种运算。

5）基本运算

如果P表示Point,D表示Duration,Pe表示Period。那么存在如下运算：

P-P=D
　两个P相减的类型为D
D±D=D
　D之间的运算结果仍为D类型
P±D=P
　P与D右结合类型为P
P∈Pe
　P属于I

说明：理解基本概念与基本运算是学习格利高利日期系统的基础。

第2页：格里高利日期系统

1）简介

格里高利日期时间系统建立在格里高利日历的基础上；格里高利日历就是我们平时说的"阳历"。格里高利日历是1582年由罗马教皇推广，目前格里高利日期表示是全球最通用的日期表示方式。表示格里高利日期系统的所有数据类型都归纳在boost::gregorian命名空间内。

2）boost::gregorian::date数据类型

date数据类型用来表示格里高利日期。它的表示范围在“1400-1-1”到"9999-12-1”之间。date数据类型是日期编程的基础类；date数据类型也可以从其它库的日期时间类型构造，比如标准的C语言日期结构体tm。

3）date数据类型的关注焦点

date其实是一个32bit大小的数据类型，在底层用一个long类型表示。date数据类型没有采用虚函数技术。这种设计技术带来的好处是处理效率大大提高。date数据类型关注：

```
1:从本地时钟获取日期
2:使用日期迭代器
3:定义日期算法与数据结构
4:定义日期生成器
5:表示无效(空)时间，包括正无穷和负无穷时间
```

格里高利日期系统就是围绕着上面的焦点进行定义的的，那么我们开始学习格里高利日期系统吧。

第3页：由给定的参数生成date

1）从“年月日”参数生成

这种生成方式是调用构造方法，使用year、moth和day数据类型生成date。

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
   //定义年、月、日数据类型
    gregorian::greg_year year(1993);
    gregorian::greg_month month(3);
    gregorian::greg_day day(25);
    //使用年、月、日数据类型生成date
    boost::gregorian::date Brithday(year,month,day);
    cout<<"我的生日是:"<<Brithday<<endl;
}
```

运行结果是：

```
我的生日是:1993-Mar-25
```

“月”数据类型也可以使用常量生成，这些常量的名字为英文月份的前三个单词的缩写，如下程序：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
    gregorian::greg_year year(1993);
    //使用常量生成"月"数据类型
    gregorian::greg_month month(gregorian::Mar);
    gregorian::greg_day day(25);
    boost::gregorian::date Brithday(year,month,day);
    cout<<"我的生日是:"<<Brithday<<endl;
}
```

运行结果是：

```
我的生日是:1993-Mar-25
```

2）从数值参数生成

通过直接给出"年月日"的数字代码可以直接构造一个date。如下程序：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
    boost::gregorian::date Brithday(1993,3,25);
    //月份也能使用常量
    boost::gregorian::date d(1993,gregorian::Mar,25);
    cout<<"我的生日是:"<<Brithday<<endl;
}
```

运行结果是：

```
我的生日是:1993-Mar-25
```

3）从字符串参数生成

可以通过给出的字符串数据类型构造date，如下程序：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
    gregorian::date d1(gregorian::from_undelimited_string("20130101"));
    gregorian::date d2(boost::gregorian::from_string("2013/1/1"));
    cout<<d1<<endl<<d2;
}
```

运行结果是：

```
2014-Jan-10
2014-Jan-10
```

说明：由于每个国家的习惯不同，声明的年月日顺序不同；因此，相同的字符串在不同的国家可能代表着不同的日期，所以并不推荐使用从字符串构造。

第4页：由拷贝构造函数和系统时钟生成date

1）从拷贝构造函数生成

拷贝构造函数用来复制一个对象。如下程序：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
    gregorian::date date1(2014,gregorian::Mar,25);
```

```
    //利用拷贝构造函数生成
    gregorian::date date2(date1);
    gregorian::date date3=date2;
    cout<<date3<<endl;
    return 0;
}
```

运行结果是：

```
1993-Mar-25
```

2）从系统时钟生成

利用系统时钟可以获取当前系统的日期。获取的时间分为本地时间或UTC时间；本地时间依赖于计算机设置的时区：在一台设置了北京时区的电脑上，本地时间就是表示UTC+8的时间。使用例子如下：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
    //获取本地日期，北京为UTC+8
    gregorian::date date1=gregorian::day_clock::local_day();
    //获取UTC日期，相当于格林威治日期
    gregorian::date date2=gregorian::day_clock::universal_day();
    cout<<"北京日期："<<date1<<endl<<"格林威治日期："<<date2<<endl;
    return 0;
}
```

运行结果是：

```
北京日期：2014-Dec-28
格林威治日期：2014-Dec-29
```

第5页：特殊日期的生成

1）生成正无穷和负无穷日期

定义正无穷和负无穷日期是非常有意义的，比如诺贝尔基金的到期时间就是正无穷；表示宇宙大爆炸发生的日期为负无穷。例子程序如下：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
    //负无穷日期
    gregorian::date date1(gregorian::neg_infin);
    //正无穷日期
    gregorian::date date2(gregorian::pos_infin);
    cout<<date1<<endl<<date2<<endl;
    return 0;
}
```

运行结果是：

```
-infinity
+infinity
```

2）生成无效日期

无效日期可以理解为还没有初始化的日期，也叫空值日期；无效日期不能理解为"没有日期"。比如，某太太的年龄属于她的私人秘密，在程序中，它的生日就可以设置为无效日期。

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
    //声明无效日期
    gregorian::date date1(gregorian::not_a_date_time);
    cout<<date1<<endl;
    return 0;
}
```

运行结果是：

```
not_a_date_time
```

第6页：从date数据类型提取年月日

1）从date提取数据

gregorian::greg_day、gregorian::gre_month和gregorian::gre_year这3种数据类型用来表示日、月和年；它们兼有时间点和时间长度的功能，并且可以与int相互转化。下面这个例子表示从系统时钟获取当前日期，提取当前是几号，并且转化为int的例子：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
   //获取当天日期,并提取天数到int
   gregorian::date now=gregorian::day_clock::local_day();
   gregorian::greg_day d=now.day();
   int day_i=d.as_number();
   cout<<"今天是"<<day_i<<"号"<<endl;
}
```

运行结果：

```
今天是2号
```

从date数据类型获取月份和年份的例子与上面的步骤一样。

2）从date提取date::ymd_type结构体

date::ymd_type结构体包装了gregorian::greg_day、gregorian::gre_month和gregorian::gre_year这3种数据类型。下面是从date获取ymd_type结构体的例子：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
```

```
using namespace boost;
using namespace std;
int main()
{
   gregorian::date today=gregorian::day_clock::local_day();
   //获取ymd_type结构体
   gregorian::date::ymd_type ymd=today.year_month_day();
   cout<<"今天是"<<ymd.month.as_number()<<"月";
   cout<<ymd.day.as_number()<<"号"<<endl;
}
```

运行结果：

```
今天是1月2号
```

第7页：日期定位

1：定位日期是当年的第几天

日期定位指的是获取某个日期是当年、或当月的第几天。下面演示了获取今天是今年的第几天的例子：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
   //获取今天是今年的第几天
   gregorian::date today=gregorian::day_clock::local_day();
   int i=today.day_of_year();
   cout<<"今天是"<<(int )today.year()<<"年的";
   cout<<"第"<<i<<"天" <<endl;
}
```

运行结果：

```
今天是2015年的第2天
```

2：定位日期是当年的第几周

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
    gregorian::date today=gregorian::day_clock::local_day();
   //获取日期是当年的几周
   int i=today.week_number();
   cout<<"今天是"<<(int) today.year()<<"年的";
   cout<<"第"<<i<<"周" <<endl;
}
```

运行结果：

```
今天是2015年的第1周
```

3：定位日期是星期几

```
#include<boost/date_time/gregorian/gregorian.hpp>
```

```
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
    gregorian::date today=gregorian::day_clock::local_day();
    //获取星期数
    int i=today.day_of_week();
    cout<<"今天是星期"<<i<<endl;
}
```

运行结果：

```
今天是星期5
```

第8页：获取当月最后一天的日期

1：获取当月最后一天的日期

有个date对象的成员方法很有用，它用来获取当月最后一天的日期。有了这个方法，很多事情的解决就很明了，比如计算当月剩下的天数，计算月利息等。这个方法有必要独立设置一页来讲解。

```
date.end_of_month()
   获取当月最后一天的日期
```

使用例子如下：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
   gregorian::date today=gregorian::day_clock::local_day();
   //获取本月最后一天
   gregorian::date monthEnd=today.end_of_month();
   cout<<today<<endl;
   cout<<monthEnd;

}
```

运行结果：

```
2015-Jun-3
2015-Jun-31
```

第8页：特殊日期的判断

1：特殊日期的判断

特殊日期指的是无效日期、正无穷日期和负无穷日期。date数据类型的特殊日期的判断方法就是用来判断其是否为特殊日期。它们的返回值为逻辑型，包括：

```
date.is_special()
   判断是否为特殊日期
date.is_neg_infinity()
   判断是否为负无穷日期
date.is_pos_infinity()
```

```
    判断是否为正无穷日期
date.is_not_a_date()
    判断是否为无效日期
date.is_infinity()
    判断是否为正、负无穷日期
```

使用例子如下：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
   gregorian::date day(gregorian::not_a_date_time);
   bool i=day.is_not_a_date();
   cout<<i<<endl;

}
```

运行结果：

```
true
```

第10页：日期比较

1：日期比较

date数据类型重载了一整套比较运算符，用来支持日期的比较运算。比较运算的运算结果都是逻辑型的。列表如下：

```
date.operator!=(const date& )
    不等于比较
date.operator>=(const date& )
    大于等于比较
date.operator<=(const date& )
    小于等于比较
date.operator==(const date& )
    等于比较
date.operator>(const date& )
    大于比较
date.operator<(const date& )
    小于比较
```

使用例子如下：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
   //日期比较
   gregorian::date day1(2013,1,1);
   gregorian::date day2(2014,1,1);
   bool b=(day<day2);
   cout<<b<<endl;
}
```

运行结果：

```
true
```

第11页：输入与输出date

1：输入与输出date

date数据类型重载了"<<"和">>"运算符，它用来支持流式输入和输出；是的，正是因为date数据类型重载了"<<",我们才可以利用cout对象把date输出到屏幕。date对象重载了这两个运算之后，立即赋予它灵活的输入输出功能;下面演示这两个运算符与stringstream和ostream的配合使用：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
    //声明一个空日期
    gregorian::date day(gregorian::not_a_date_time);
    stringstream ss("2002-Jan-01");
    //从流对象中获取日期
    ss >> day;
    //利用ostream输出到屏幕
    cou t<< day << endl;

}
```

运行结果：

```
2002-Jan-01
```

第12页：日期的运算

1：两个date数据类型不能定义运算

虽然对于日期来说，定义减法运算是很必要的，但是两个date数据类型是不能直接运算；这个很好理解，比如，用你的生日加上或减去另外一个人的生日不可能得到一个日期；这是为什么呢?因为date数据类型属于"时间点"的概念，两个"时间点"是不能定义运算的。

2：引入date_duration 数据类型

由于date属于"时间点"的概念，从而导致两个date数据类型之间不能参与运算;因此我们必须使用一种叫做date_duration的数据类型，date_duration数据类型属于"时间长度"的概念。日期加法和减法运算规则的外观如下：

```
operator+(date_duration dd)
    日期加上日期长度
operator-(date_duration dd)
    日期减去日期长度
```

下面例子表示求出一个日期加上100天的时间长度之后的例子：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
```

```
    gregorian::date date1(2013,3,4);
    //声明一个100天的日期长度
    gregorian::date_duration d_dur(100);
    //做日期加法
    gregorian::date date2=date1+d_dur;
    cout<<date1<<加上100天后是<<date2<<endl;

}
```

运行结果：

```
2002-Mar-01加上100天后是2013-Jun-12
```

3:计算两个日期之间的日期长度

date数据类型的减法运算是有意义的，2个date相减，得到的类型是date_duraiton。date数据类型重载了“-”，它的外观如下：

```
date_duration operator-(date)
   日期减去日期
```

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
    gregorian::date date1(2013,3,4);
    gregorian::date date2(2013,3,14);
    //两个日期相减
    gregorian::date_duration d_dur=date2-date1;
    cout<<"date2和date1相差"<<d_dur.days()<<"天"<<endl;

}
```

运行结果：

```
date2与date1相差10天
```

4：说明

date_duraiton数据类型后面将会详细讲解；这一页只是为了讲解日期运算而引入这个数据类型。

第13页：把date转换到string

1：把date转换到string

gregorian名字空间内存在3个独立的方法，这些方法拥有把date数据类型转化到string的功能。它们的外观如下：

```
to_simple_string(date d)
   转换到YYYY-mmm-DD形式的字符串,mmm为3个月份的英文前缀
to_iso_string(date d)
   转换到YYYYMMDD形式的字符
to_iso_extended_string(date d)
   转换到YYYY-MM-DD形式的字符串
```

使用例子如下：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
    gregorian::date today=gregorian::day_clock::locak_day();
    string a=gregorian::to_simple_string(today);
    string b=gregorian::iso_extended_string(today);
    string c=gregorian::to_iso_string(today);
    cout<<a<<endl<<b<<endl<<c;
}
```

运行结果：

```
2015-Jan-05
2015-01-05
201501105
```

第14页：date和tm结构体的转换

1：date和tm结构体的转换

熟悉C/C++的开发者一定知道tm结构体是标准库里关于日期时间处理的结构体。date数据类型当然可以跟tm结构体相互转化，这个任务由gregorian名字空间内的2个函数完成，它们的外观如下：

```
tm to_tm(date d)
    转换到tm结构体
date date_from_tm(tm tm_)
    转换到date类型
```

这两个转换函数为使用传统的日期时间开发和boost开发提供了桥梁。桥梁的这边，date数据类型本章已经进行了全面讲解；桥梁的那边tm结构体编程的内容请查看其它课程的文档。这里就不过多讲解了。

第16页：日期长度数据类型

1：gregorian::date_duration数据类型

date数据类型用来表示日期点，而date_duration数据类型用来表示一个日期长度。date_duration的计量单位为"天"。在gregorian名字空间内，date_duration数据类型被typedef包装成days；所以在以后的内容中，days和date_duration其实是一个类型。

2：days的构造方法

days数据类型有两个构造方法可以使用，列表如下：

```
days(long n)
    生成一个表示n天的日期长度
days(special_values sv)
    生成一个表示特殊的日期长度
```

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
```

```
6  {
7      //声明一个为100天的日期长度，可以传入负数
8      gregorian::days d1(100);
9      //下面分别声明：负无穷、正无穷、空日期长度、最大日期长度和最小日期长度
10     gregorian::days dd1(gregorian::neg_infin);
11     gregorian::days dd2(gregorian::pos_infin);
12     gregorian::days dd3(gregorian::not_a_date_time);
13     gregorian::days dd4(gregorian::max_date_time);
14     gregorian::days dd5(gregorian::min_date_time);
15 }
```

3：days的运算符

days数据类型跟date数据类型一样，重载了一整套算数运算符和输出输出运算符，用来支持days的算数运算和输入输出。列表如下：

operator<<
　重载的输出运算符
operator>>
　重载的输入运算符
operator>=,!=,>,<,<=,==
　重载的一整套比较运算符
date_duration operator+(date_duration) const
　日期长度加法
date_duration operator-(date_duration) const
　日期长度减法

4：days的成员方法

days的成员方法列表如下：

long days.days()
　得到表示的天数
bool days.is_negative()
　判断是否为负日期长度
bool days.is_special()
　判断是否为特殊日期长度

5：其它日期长度数据类型

除了days之外，还有months,years,和weeks数据类型用来表示，月、年和周日期长度;它们的使用方法与days一样。所以，在gregorian名字空间内，一共有四种表示日期长度的数据类型，一般来说，这四种日期长度类型数据要与date配合起来完成对日期的业务处理。

第17页：日期与日期长度间的运算

1：日期与日期长度间的运算

这一页讨论date与days、month、years和weeks间相互运算的细节问题。

2：date与days相加和相减

date加上或减去days,相当于在日历上往前或往后移动对应的天数。

```
1  #include<boost/date_time/gregorian/gregorian.hpp>
2  #include<iostream>
```

```
using namespace boost;
using namespace std;
int main()
{
    gregorian::date d1(2014,1,1);
    gregorian::days ds(100);
    //date加上days
    gregorian::date d2=d1+ds;
    cout<<"2014年元旦过后100天是"<<d2<<endl;
}
```

```
2014年元旦过后100天是2014-Apr-11
```

3：date与weeks相加和相减

date加上或减去weeks,相当于在日历上向前或向后移动7*weeks表示的星期数。比如，日期a加上weeks(2),等价于a加上days(14)。

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
    gregorian::date d1(2014,1,1);
    gregorian::weeks wks(2);
    gregorian::date d2=d1+wks;
    //date加上weeks
    cout<<"2014年元旦过后2个星期是"<<d2<<endl;
}
```

```
2014年元旦过后2个星期是2014-Jan-15
```

4：date与months相加和相减

date与months相加或相减，相当于日期的月份相加或相加对应的月数。

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
    gregorian::date d1(2014,3,15);
    gregorian::months mons(1);
    //date加上months
    gregorian::date d2=d1+mons;
    cout<<"2014年3月15日过一个月是"<<d2<<endl;
}
```

```
2014年3月15日过一个月是2014-Apr-15
```

5：date与years相加和相减

date与years相加和相减,相当于年份加上或减去years表示的年数。

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
```

```
7       gregorian::date d1(2008,2,29);
8       gregorian::months mons(1);
9       gregorian::years yrs(1);
10      gregorian::date d2=d1+yrs;
11      cout<<"2008年2月21日一年后日期是"<<d2<<endl;
12  }
```

```
2008年2月21日一年后日期是2009-Feb-21
```

6：说明

日期与日期长度的运算，如果在运算的日期在月份的中间，那么运算结果日期的天数不会改变；如果运算日期在月份月末，那么还要考虑平年和闰年以及30和31号的情况，这个问题就变得复杂起来了。希望读者自己动手做实验弄懂格利高利日期系统对其处理的规律。

第17页：3种基本数据类型

1）3种数据类型

点、长度和区间是格里高利日期系统最重要的3重基本数据类型，在gregorian名字空间内，所有的数据结构和算法就是围绕着这3个数据类型展开的。

2）表示时间点

表示时间点的数据类型是date。

3）表示时间长度

表示时间长度的数据类型是date_duration。weeks、months、years和days本质就是date_duration，只不过被重定义而已。

4）表示时间区间

表示时间区间的数据类型是gregorian::period。

5）区分这3种数据类型

这3种数据类型描述不同的物理量，它们的意义不同。比如在"奥林匹克运动会于2008年8月8日举行，持续16天，到8月24日结束"这句话中，"8月8日"属于时间点，"16天"属于日期长度，"8月8日到8月24日"属于日期区间。后面的内容，我们重点讲解gregorian::period数据类型。

第18页：日期区间数据类型

1：日期区间数据类型

gregorian::date_period数据类型用来表示一个左闭右开的日期区间（结束日期不算）。这个数据类型一般很少单独使用，要是它与date和date_duriaon配合起来一起使用，就可以完成各种复杂的日期处理程序。

2：构造方法

date_period可以从两个日期构造，也可以从一个日期和一个日期长度构造。构造函数列表如下：

```
date_period(date min, date max)
    从2个date构造
date_period(date d, days d_dur)
    从一个date和一个date_duraiotn构造
date_period(date_period)
    拷贝构造函数
```

下面是生成date_period的例子：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
    gregorian::date min(2008,2,21);
    gregorian::date max(2009,2,21);
    gregorian::date_duration dur(100);
    //由2个date生成
    gregorian::date_period dp1(min,max);
    //由1个date和1个date_duraiton生成
    gregorian::date_period dp2(min,dur);
    cout<<dp1<<endl<<dp2;
}
```

因为date_period数据类型重载了<<运算符，所以可以利用cout对象输出到屏幕。运行结果：

```
[2008-Feb-21/2009-Feb-20]
[2008-Feb-21/2008-May-30]
```

3：区间的滑动与伸缩

date_period拥有2个成员函数，用来伸缩和滑动它表示的日期区间。注意，因为days是可以表示一个为负数的日期长度，因而这个两个函数的行为大大地灵活起来。

```
date_period.shift(days)
    端点向同一个方向移动days表示的天数，形象描述为"滑动"
date_period.expand(days)
    端点向外或向内移动days表示的天数，形象描述为"伸缩"
```

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
    gregorian::date min(2008,2,21);
    gregorian::date max(2009,2,21);
    gregorian::date_period dp1(min,max);
    gregorian::date_period dp2(min,max);
    gregorian::days ds(3);
    //两端同时扩张
    dp1.expand(ds);
    //两端同时向右边滑动
    dp2.shift(ds);
    cout<<dp1<<endl<<dp2;
}
```

运行结果：

```
[2008-Feb-18/2009-Feb-23]
[2008-Feb-24/2009-Feb-23]
```

4：访问边界日期和日期长度

date_period表示的范围是一个左闭右开的区间，下面3个函数用于获取一个date_period的边界日期。列表如下：

```
date date_period.begin()
    获取日期区间开始的日期
date date_period.last()
    获取日期区间最后一个日期
date date_period.end()
    获取日期区间结束后，开始的第一天日期。
```

date_period有一个成员方法，用来获取这个日期区间持续的长度。列表如下：

```
days date_period.length()
    获取日期区间持续的长度
```

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace std;
int main()
{
    gregorian::date min(2008,2,21);
    gregorian::date max(2009,2,21);
    gregorian::date_period dp1(min,max);
    //得到持续长度
    gregorian::days ds=dp1.length();
    //得到开始日期
    gregorian::date date1=dp1.begin();
    //得到最后一天的日期
    gregorian::date date2=dp1.last();
    //得到区间之后第一天的日期
    gregorian::date date3=dp1.end();
    cout<<ds.days()<<"天"<<endl;
    cout<<date1<<endl<<date2<<endl<<date3<<endl;
}
```

```
366天
2008-Feb-21
2009-Feb-20
2009-Feb-21
```

第19页:日期区间判断

1：日期区间判断

日期区间数据类型存在一些判断算法，它们的返回值都是bool类型的。它们用于判断日期、日期的长度和日期区间之间的关系。

2：判断日期区间之间的关系

下面这些方法用来判断日期区间之间的关系。列表如下：

```
bool date_period.is_null()
    判断日期区间是否为无效日期区间
```

```
bool date_period.contains(date_period)
    判断2个日期区间是否包含
bool date_period.intersects(date_period)
    判断2个日期区间是否交叉
bool date_period.is_adjacent(date_period)
    判断2个日期区间是否相邻
```

例子程序如下：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace boost::gregorian;
using namespace std;
int main()
{
    date_period dp1(date(1993,3,1),date(1993,2,25));
    date_period dp2(date(1993,3,3),date(1993,3,10));
    date_period dp3(date(1993,3,4),date(1993,3,6));
    date_period dp4(date(1993,2,25),date(1993,3,6));
    //判断dp1是否有意义
    cout<<dp1.is_null()<<endl;
    //判断dp2是否包含dp3
    cout<<dp2.contains(dp3)<<endl;
    //判断dp2和dp4是否交叉
    cout<<dp2.intersects(dp4)<<endl;
}
```

运行结果：

```
null1
1
1
```

3：判断日期区间与日期的关系

下面这些方法用来判断日期点和日期区间的关系。列表如下：

```
bool date_period.is_after(date)
    判断日期是否在区间的前面
bool date_period.is_before(date)
    判断日期是否在区间的后面
bool date_period.contains(date)
    判断日期是否在区间的中间
```

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace boost::gregorian;
using namespace std;
int main()
{
    date_period dp1(date(1993,3,3),date(1993,3,10));
    cout<<dp1.is_before(date(1993,3,20))<<endl;
}
```

运行结果如下：

```
1
```

第20页:日期区间的交集和并集运算

1：日期区间交、并集运算

intersection方法是取得两个日期区间的交集，即重合的部分。外观如下：

```
date_period date_period.intersection(date_period dp)
    求与dp之间的交集
```

merge方法是取得两个日期区间的的并集，要求是两个区间必须相邻或有交集。外观如下：

```
date_period date_period.intersection(date_period dp)
    求与dp之间的并集
```

span方法用于返回一个日期区间Pd，其中Pd.begin=min(pd1.begin,pd2.bengin)、Pd.end=max(pd1.end,pd2.end)，如下描述：

```
date_period date_period.span(date_period dp)
    取自己与dp最小的开始日期和最大的结束日期来生成新的日期区间
```

例子程序如下：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace boost::gregorian;
using namespace std;
int main()
{
    date_period dp1(date(1993,3,3),date(1993,3,10));
    date_period dp2(date(1993,3,4),date(1993,3,25));
    cout<<dp1.intersection(dp2)<<endl;
    cout<<dp1.merge(dp2)<<endl;
    cout<<dp1.span(dp2)<<endl;
}
```

运行结果：

```
[1993-Mar-04/1993-Mar-09]
[1993-Mar-03/1993-Mar-24]
[1993-Mar-03/1993-Mar-24]
```

第21页:日期区间的比较运算

1：日期区间的比较运算

date_period数据类型与date和date_duration数据类型一样，重载了一整套比较运算符。date_period数据类型的比较运算定义如下：

```
bool date_period.operator<(date_period dp)
    成立条件是：dp1().end<dp2.end()
bool date_period.operator<=(date_period dp)
    成立条件是：dp1().end<=dp2.end()
bool date_period.operator>(date_period dp)
    成立条件是：dp1().end>dp2.end()
bool date_period.operator>=(date_period dp)
    成立条件是：dp1().end>=dp2.end()
bool date_period.operator==(date_period dp)
    成立条件是：两个date_period的开始与结束日期都相同
bool date_period.operator!=(date_period dp)
    取值与"=="相反
```

第22页:日期迭代器

1：日期迭代器

日期迭代器可以说是这个库中最精彩的内容；有了它，我们可以像使用vector::iterator一样方便地对日期进行遍历操作。 gergorian一共定义了4种迭代器，分别为天数、周、月数和年数迭代器。 迭代器每调用一次"++"运算符，它所表示的日期就更改相应的天数。

2：天数迭代器

天数迭代器的外观如下：

```
day_iterator(date,int count=1)    天数迭代器
```

天数迭代器的第二个参数默认为1。

使用例子如下:

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace boost::gregorian;
using namespace std;
int main()
{
    day_iterator di(date(2015,1,1),2);
    //迭代器指针移动一次
    ++di;
    cout<<*di<<endl;
}
```

运行结果：

```
2015-Jan-03
```

下面是利用迭代器的广泛例子，作用是遍历1月1号到10号的每一天：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace boost::gregorian;
using namespace std;
int main()
{
   for(day_iterator di(date(2015,1,1),1);di!=date(2015,1,10);++di)
   {
       //你的业务代码
   }
}
```

3:另外3种迭代器

除此之外，还有3种迭代器。列表如下：

```
week_iterator(date,int count=1)
   周迭代器
month_iterator(date,int count=1)
   月份迭代器
```

```
year_iterator(date,int count=1)
    年份迭代器
```

所有的迭代器第二个参数不给出，那么默认值就是1。这3个迭代器的使用与天数迭代器的使用方法是一样的。要注意在使用迭代器的时候，需要考虑平年和闰年以及月份的最后一天的取舍。

第23页:星期N算法

1：星期N算法

这一页讲解求某天到星期N的日期和日期长度的算法。算法分为两组，第一组是求出某天到上（下）一个星期N的日期长度；第二组是求出日期。

2：求某天到下(上)一个星期N的日期长度

对于这种问题：求2015年1月20日到下（上）一个星期5的天数。如果使用常规方式，至少需要3个步骤，使用内置算法只需要调用一行函数就行了，它们的外观如下：

```
days days_until_weekday（date, greg_weekday)
    求date到下一个星期N的日期长度
days days_before_weekday(date, greg_weekday)
    求date到上一个星期N的日期长度
```

使用例子如下：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace boost::gregorian;
using namespace std;
int main()
{
   //获取(2015,1,20)之后第一个星期五的日期长度
    days d_dur=days_until_weekday(date(2015,1,20),greg_weekday(Friday));
    cout<<"2015年1月20日再过"<<d_dur.days()<<"天是星期五"<<endl;

    //下面5行是上面2行的等价形式，便于理解。
    gregorian::days d_dur1;
    gregorian::date date1(2015,1,20);
    gregorian::greg_weekday gw(gregorian::Friday);
    d_dur1=gregorian::days_until_weekday(date1,gw);
    cout<<"20151年月20日再过"<<d_dur1.days()<<"天是星期五"<<endl;
}
```

运行结果：

```
2015年1月20日再过3天是星期五
2015年1月20日再过3天是星期五
```

3:求某天到下（上）一个星期N的日期

实现这个需求的算法外观如下：

```
date next_weekday(date, greg_weekday)
    求date下一个星期N的日期
date previous_weekday(date, greg_weekday)
    求date上一个星期N的日期
```

算法的功能是求出某天到下（上）一个星期N的日期。使用例子如下：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace boost::gregorian;
using namespace std;
int main()
{
    //获取(2015,1,20)之后第一个星期五的日期
    date d=next_weekday(date(2015,1,20),greg_weekday(Friday));
    cout<<"2015年1月20号下一个星期五是"<<d.day().as_number()<<"号">>endl;
}
```

运行结果：

```
2015年1月20号下一个星期五是23号
```

date_time库与其它的日期库的优越性就是体现在这些细节上，虽然算法并不复杂，但是很实用。

第24页:date数据类型总结

1：头文件与名字空间

```
头文件: #include "boost/date_time/gregorian/gregorian.hpp"
命名空间： using namespace boost::gregorian
```

2:构造函数

```
年月日构造：    date(greg_year, greg_month, greg_day)
拷贝构造函数：date(date d)
特殊日期构造：date(special_values sv)
默认构造函数：date()
```

3:生成函数

```
从字符串生成：gergorian::from_string(std::string)
从字符串生成：gergorian::from_undelimited_string(std::string)
从系统时钟生成：gergorian::day_clock::local_day()
获取UTC日期：gergorian::day_clock::universal_day()
```

4:转化到string

```
std::string gergorian::to_simple_string(date d)
std::string gergorian::to_iso_string(date d)
std::string gergorian::to_iso_extended_string(date d)
```

5:运算符重载

```
输入输出运算符：<<,>>
一整套比较运算符:==,>=,<=,!=,<,>
日期相减：date_duration operator-(date)
日期减去日期长度：date operator-(date_duration
日期加上日期长度：date operator+(date_duration)
```

6:tm结构体转化函数

转换到tm:tm to_tm(date)
从tm构造:date date_from_tm(tm datetm)

7:成员函数

获取年类型：greg_year year()
获取月类型：greg_month month()
获取天类型：greg_day day()
获取年月日结构体：greg_ymd year_month_day()
获取日期是星期几：greg_day_of_week day_of_week()
获取日期是当年的第几天：greg_day_of_year day_of_year()
获取当月最后一天的日期：date end_of_month()
判断特殊日期：is_infinity()，is_neg_infinity()，is_pos_infinity()，is_not_a_date()， is_special()
*获取modjulian天数：long modjulian_day()
*获取julian天数：long julian_day()
获取日期是当年的第几周：int week_number()

第25页:date_duration数据类型总结

1:头文件与名字空间

头文件: #include "boost/date_time/gregorian/gregorian.hpp"
命名空间： using namespace boost::gregorian

2:构造函数

生成n天的日期长度：date_duration(long n)
生成特殊日期长度：days(special_values sv)

3:成员函数

获取日期长度：long days()
判断是否为负长度： bool is_negative()
获取date_druaiton(1)对象： date_duration unit()
判断是否为特殊日期长度： bool is_special()

4:运算符重载

输入输出运算符：<<,>>
一整套比较运算符:==,>=,<=,!=,<,>
日期长度加减：date_duration operator±(date_duration)

5:额外的日期长度类型

表示一个月的长度：months
表示一年的长度：years
表示一周的长度：weeks

第27页：date_period数据类型总结

1:头文件与名字空间

头文件: #include "boost/date_time/gregorian/gregorian.hpp"
命名空间： using namespace boost::gregorian

2：构造函数

日期构造：date_period(date, date)
日期与长度构造：date_period(date, days)
拷贝构造：date_period(date_period)

3:成员函数：

区间滑动：date_period shift(days)
区间伸缩：date_period expand(days)
获取开始日期：date begin()
获取结束日期：date last()
获取结束后第一天的日期：date end()
取日期长度：days length()
判断是否有意义：bool is_null()
判断包含某日期：bool contains(date)
判断日期在区间的前面：bool is_after(date)
判断日期在区间的后面：bool is_before(date)
判断包含某日期区间：bool contains(date_period)
判断日期区间是否有公共部分：bool intersects(date_period)
求并集：date_period intersection(date_period)
判断相邻：date_period is_adjacent(date_period)
日期区间合并:date_period merge(date_period)
日期区间扩展:date_period span(date_period)

4:转化到string

std::string to_simple_string(date_period dp)

5:一整套比较运算符

<、>、!、<>、<=、>=

第28页：迭代器与算法

1：头文件与名字空间

头文件: #include "boost/date_time/gregorian/gregorian.hpp"
命名空间： using namespace boost::gregorian

2：迭代器

天数迭代器：day_iterator(date start_date, int day_count=1)
周数迭代器：week_iterator(date start_date, int week_count=1)
月份迭代器：month_iterator(date start_date, int month_count=1)
年数迭代器：year_iterator(date start_date, int year_count=1)

3：星期N算法

求到下一个星期的长度：Ndays days_until_weekday date, greg_weekday)
求到上一个星期的长度：days days_before_weekday(date, greg_weekday)
求下一个星期N的日期：date next_weekday(date, greg_weekday)
求上一个星期N的日期：date previous_weekday(date, greg_weekday)

第29页:例子程序

1：说明

格利高利日期系统完全符合工业要求。下面给出一个例子程序。

2：求某人在这颗星球上存在的天数

这个程序接受了用户输入的生日，然后计算用户的生日到当前日期的天数。程序运行的预期如下：

```
你出生在哪年?year=1993
出生的月份?month=4
几号出生？day=5
你在这个星球存在了7959天
```

3：实现

这个程序并不难，实现如下：

```
#include<boost/date_time/gregorian/gregorian.hpp>
#include<iostream>
using namespace boost;
using namespace boost::gregorian;
using namespace std;
int main()
{
    //输入数据并构造date
    int year,month,day;
    int live;
    cout<<"你出生在哪年?year=";cin>>year;
    cout<<"出生的月份?month=";cin>>month;
    cout<<"几号出生？day=";cin>>day;
    date birth=date(year,month,day);
    //获取当前时间并求差值
    date today=day_clock::local_day();
    days d=today-birth;
    //输出
    cout<<"你在这个星球存在了"<<d.days()<<"天\n";
}
```

4：结语

本章讲解了格里高利日期系统中最常用的内容，掌握好这些内容，至少可以应对90%的开发；还有一小部分没有讲解，希望读者自己阅读官网上给出的资料。

第1页：BOOST_ASSERT基本用法

1：BOOST_ASSERT库简介

软件开发者很容易陷入与bug之间的搏斗中，程序调试是一个极其枯燥和无聊的过程；那么，BOOST_ASSERT的诞生至少可以为开发者减小这个痛苦。BOOST_ASSERT跟标准库"cassert.h"中的assert宏非常相似，但是比它更强大。BOOST_ASSERT工具是对付bug的有力武器。

2：运行时诊断

如果代码中存在语法错误，编译器就根本不让编译，编译器向程序员指明了详细的语法错误规则，比如少了分号；所以语法错误相对来说是非常好处理的。一旦程序运行起来后，由于非用户原因导致的逻辑错误

却是最难找到的。我们期望实现一个能在运行时候进行代码诊断的工具，那么BOOST.ASSERT库就可以胜任这个要求。诊断功能是面向程序员，它让程序员更好地完成代码设计。注意，这个库并不是要实现错误处理功能，而是一个调试工具。

3：基本用法

BOOST_ASSERT需要传入一个被诊断的参数，它的类型必须是bool型的；如果为真，程序继续往下运行，如果为假，程序立即停止，并向控制台输出文件名和出错的行号。使用例子如下：

```
#include<boost/assert.hpp>
using namespace std;
int main()
{
    //诊断成功
    BOOST_ASSERT(1);
    //诊断失败
    BOOST_ASSERT(0);
}
```

运行结果：

```
test: test.cpp:9: int main(): Assertion '0' failed.
```

输出的结果表示在test文件main函数中，第9行诊断失败。

4：输出调试文本

使用"&&"运算符把诊断的值和文本进行运算，当诊断失败的时候，就能输出这条文本；如下程序：

```
#define BOOST_DISABLE_ASSERTS
#include<boost/assert.hpp>
using namespace std;
int main()
{
    //诊断成功
    BOOST_ASSERT(1);
    //诊断失败
    BOOST_ASSERT(0 && "something gose wrong");
}
```

运行结果：

```
test: test.cpp:9: int main(): Assertion '0 && something gose wrong' failed.
```

在上面的运行结果中，我们可以清楚地看到输出的自定义诊断文本。

5：取消诊断

当程序发布的时候，诊断代码可能有很多，但是，我们只需要加入一条指令就能取消所有的诊断功能；只需要加入"#define BOOST_DISABLE_ASSERTS"一行就可以了。程序如下：

```
#define BOOST_DISABLE_ASSERTS
#include<boost/assert.hpp>
using namespace std;
int main()
{
    BOOST_ASSERT(1);
    BOOST_ASSERT(0 && "something gose wrong");
}
```

运行结果：

```
Press any key to exit...
```

第2页：实现诊断处理功能

1：诊断处理

在进行程序调试时候，我们期望当诊断失败的时候进行某些处理功能，比如生成诊断报告；所以必须要实现一个"信号"的机制：当诊断失败的时候，通知诊断处理函数去实现特定的功能。那么，BOOST_ENABLE_ASSERT_HANDLER宏定义就可以完成这个需求。

2：基本用法

当启用宏BOOST_ENABLE_ASSERT_HANDLER之后，BOOST__ASSERT就会增加诊断处理功能；这个时候，需要自己手动实现BOOST命名空间中的一个诊断处理函数，这个函数包装了诊断错误的表达式，函数名，文件名和行数，当诊断失败的时候，会自动调用这个函数。它的外观如下：

```
namespace boost
{
    void assertion_failed
    (
        //表达式
        char const*expr,
        //函数名
        char const *function,
        //文件名
        char const *file,
        //诊断行数
        long line
    )
    {
        //处理逻辑
    }
}
```

3：使用例子

下面例子是，当诊断失败后，立即转入诊断处理程序。

```
#define BOOST_ENABLE_ASSERT_HANDLER
#include<boost/assert.hpp>
#include<iostream>
using namespace std;
int main()
{
    //诊断失败
    //然后由内部机制调用boost::assertion_failed()方法
    BOOST_ASSERT(0 && "something gose wrong");
}
namespace boost
{
      void assertion_failed(char const*expr,char const *function,char const *file,
      {
          cout<<"调用诊断失败方法："<<endl;
          cout<<<<expr<<endl;
          cout<<function<<endl;
          cout<<file<<endl;
          cout<<line<<endl;
```

```
        }
}
```

运行结果：

```
调用诊断失败方法:
0 && "something gose wrong"
int main()
test.cpp
7
```

上面的例子只是简单地输出诊断失败之后的调试信息，实际上方法里面应该由程序员来完成日志处理或者断点分析。

4：注意的问题

如果实现了诊断处理功能，诊断函数执行完后并不会终止程序，而是向下继续运行；如果有必要，需要自己在诊断处理函数中手动调用std::abort()终止程序。

第3页：自定义诊断日志

1：诊断日志

当程序达到万行以上的级别后，诊断日志就有必要了。日志的功能是保留以前的消息，这个功能是cout做不到的。当程序发生失败的时候，第一件事情就应该打开诊断日期文件，然后进行分析和处理；这样做起来，查找bug和解决bug才能做到有根有据而且目的性强。

2：例子程序

在下面的例子程序中，getArea()方法是由需求分析到得的用户业务处理函数，然而我们利用诊断功能给这个业务函数加上诊断处理机制，当它发生失败的时候会把诊断信息输出到日志中;这里程序员监测的重点是传入的值是否在合理的范围内。

```
#define BOOST_ENABLE_ASSERT_HANDLER
#include<boost/assert.hpp>
#include<boost/date_time/posix_time/posix_time.hpp>
#include<fstream>
#include<iostream>
using namespace std;
//用户的业务实现方法
float getArea(float r)
{
      BOOST_ASSERT(r!=0 && "计算圆面积的半径参数r不能为负数");
      return 3.1415926*r*r;
}
int main()
{
    //调用业务方法
    float area=getArea(-10);
}
namespace boost
{
      void assertion_failed(char const*expr,char const *function,
                            char const *file,long line)
      {
         ofstream DebugFile("DebugFile",ios::app);
         DebugFile<<"诊断时间:"
```

```
                <<boost::posix_time::second_clock::local_time()
                <<endl;
            DebugFile<<"表达式："<<expr<<endl;
            DebugFile<<"函数："<<function<<endl;
            DebugFile<<"文件："<<file<<endl;
            DebugFile<<"行数："<<line<<"\n______\n";
            DebugFile.close();
            cout<<"程序发生了失败的诊断，详细信息请查看Debug文件。";
            std::abort();
        }
}
```

运行结果：

```
程序发生了失败的诊断，详细信息请查看Debug文件
```

生成的日志文件如下：

```
DebugFile.txt:
诊断时间:2015-Jan-23 09:28:21
表达式：0 && "计算圆面积的半径参数r不能为负数"
函数：float getArea(float)
文件：main.cpp
行数：10
--------------
```

上面程序往日志输出了时间和诊断信息，重要的是这些信息会随着时间一直保存下来，以便错误分析排除。

3：使用建议

几十行的程序对于诊断处理来说，看起来好像没有用；但是在大型程序开发中，如果在关键函数的入口和出口都利用日志做诊断检查，意义是非常重要的。实现诊断日志是提高软件质量的一个好的办法，上面的例子已经完成了一个诊断框架，读者直接拷贝然后添加到自己程序中合理的位置就能使用。

第4页：实现原理

1：本质

BOOST_ASSERT库看起来很高端，其实不然，它的本质就是宏定义。下面例子是一个定义符号常量“PI”的宏，诊断库的实现跟下面宏定义的原理是一样的：

```
#define PI 3.1415926
```

接下来，我们自己手动实现一个MY_ASSERT宏，它的行为跟BOOST_ASSERT一样。

2：知识点

要实现自定义的MY_ASSERT，需要温习如下知识点：

```
1:多行宏定义用“\”结尾
2：利用"__***___"样式的宏可以获取运行时参数
3:可以定义带参数的宏
```

第一点：，定义的宏太长了，可以用“\”结尾，并重新书写一行。第二点：我们可以通过标准库自带的宏定义"__FILE__"和“__LINE__”来获取程序当前所在的文件和代码行数。 第三点：宏是可带参数的，比如#define f(x) (x)+1 这个带参数宏的作用是获取参数x加上1的值。

3：实现MY_ASSERT

只要熟悉#define，并弄懂原理之后，实现一个MY_ASSERT其实并不困难，实现如下：

```
#include <cstdlib>
//定义MY_ASSERT宏
#define MY_ASSERT(expr) \
if(!(expr))\
{\
    cout<<"诊断失败";\
    cout<<"\n行数:"<<__LINE__;\
    cout<<"\n文件:"<<__FILE__;\
    cout<<"\n函数:"<<__func__;\
    cout<<"\n程序已经退出";\
    std::abort();\
}
//结束定义
#include<iostream>
using namespace std;
int main()
{
   int a=3,b=5;
   //诊断
   MY_ASSERT(a==b);
   cout<<a;
}
```

运行结果：

```
诊断失败
行数:16
文件:test.cpp
函数：int main(void)
程序已经退出
```

说明：上面程序的原理是这样的：在对#define MY_ASSERT的定义中，先对参数进行了判断，如果判断为假，就输出文件名和行号，然后退出程序。根据运行结果来看，我们自己实现的MY_ASSERT的行为跟BOOST_ASSERT一样。

诊断库中的其它功能就是在这个基础上扩展的，在知道这个库实现原理之后，读者可以继续阅读"boost/assert.h"中的源代码进行深入研究。

第5页：诊断库总结

1：BOOST_ASSERT宏的详细行为

默认地，BOOST_ASSERT(expr)是对标准库中的assert(expr)的扩展。如果BOOST_DISABLE_ASSERTS宏在之前被定义，BOOST_ASSERT(expr)就会失效。 标准库中的assert(expr)与BOOST_ASSERT(expr)的行为是独立的，不会发生相互冲突的情况。

如果BOOST_ENABLE_ASSERT_HANDLER宏在之前被定义，那么BOOST_ASSERT(expr)就扩展为拥有诊断处理机制功能的宏，当诊断失败的时候，会调用BOOST::assertion_failed()函数，这个函数必须由用户自己实现；如果BOOST_ENABLE_ASSERT_NDEBUG_HANDLER宏被定义，BOOST_ASSERT(expr)就会失去诊断处理功能。

2：诊断库中的其它宏定义

下面这些宏定义与BOOST_ASSERT行为差别不大，一般很少使用。

BOOST_ASSERT_MSG
　　这个宏定义把输出的诊断文本设置成一个独立的参数
BOOST_VERIFY
　　表达式一定会被求值
BOOST_VERIFY_MSG
　　把输出的诊断文本设置成一个独立的参数，同时表达式会被求值

第1页：传统的内存管理

1：C++指针与内存管理

C++是通过new与delete来进行指针操作的。new代表着申请资源，delete代表着释放资源；在调用delete之前，申请指针和它指向的数据结构一直占用内存。 只要一个指针被声明，程序员要负责管理它的整个生命周期。下面是C++使用指针的例子：

```
#include<iostream>
using namespace std;
class Book
{
public:
    int Id;
    string Name;
    float Price;
    Book(int id,string name,float price)
    {
        Id=id;
        Name=name;
        Price=price;
    }
};
void doLogical()
{
    //申请指针并初始化
    Book *b=new Book(1,"童话世界",15.5);
    //使用指针
    cout<<b->Name<<endl;
    //释放
    delete b;
}
int main()
{
    //调用业务方法
    doLogical();
}
```

运行结果：

```
童话世界
```

2：指针的生命周期

指针完整的生命周期是“声明-初始化-使用-释放”这4个过程,下面的程序表现了一个Book类型指针的完整的生命周期。

```
#include<iostream>
using namespace std;
class Book
{
```

```
    public:
    Book()
    {
        cout<<"申请资源\n";
    }
    void doLogical()
    {
        cout<<"调用资源\n";
    }
    ~Book()
    {
        cout<<"释放资源\n";
    }
};
int main()
{
      //一个指针的生命周期
       Book *b1=new Book();
       b1->doLogical();
       delete b1;
}
```

运行结果：

```
申请资源
调用资源
释放资源
```

一个指针的行为无论多诡异，也不会跳出这4个阶段。在复杂的指针程序中，如果指针出了问题，对其进行生命周期分析往往是最管用的。

3：调用delete的时机

关于如何调用delete，只要记住一点就可以了：由于一个指针可以被多个对象持有（访问），只有当没有对象持有这个指针的时候才能delete之。事实情况是，每一个程序员都会使用new运算符去申请指针，但并不是每一个程序员都能准确地使用delete去释放资源。在拥有多个深层次嵌套，类之间的耦合关系在10个以上的情况中,经验不足的程序员很可能写不出正确的delete语句。

4：传统方式管理指针的落后性

如果存在大量指针要管理，一定会很累,因为只要一个指针被声明，程序员要负责管理它的整个生命周期。

第2页：使用指针常见的错误

1:传统方式的局限性

采用传统方法管理指针对程序员的水平要求非常高，不仅累，而且容易出错。最有经验的程序员也会犯如下2种常见错误：

```
内存泄漏：
   分配指针并使用完毕后，没有释放指针
无效指针：
   释放指针后，仍试图访问这个指针
```

2:内存泄漏

内存泄漏是C++指针编程中常见的错误。内存泄漏指的是当指针初始化后，没有调用delete删除之，导致

指针失效后，造成内存永久丢失。通俗理解来说，就是你声明了一个指针，当完成操作后，并没有删除这个垃圾指针，导致内存永久被占用；当应用程序运行之后，垃圾指针越来越多，内存占用量就会越来与大。严重情况下可以导致程序崩溃。内存泄漏的宏观表现是，应用程序的内存会随着处理数据数量的增大而增大，当数据量减小之后，却不见得内存占用量减小。

用下面的例子程序来讲解内存泄漏是如何发生的；这个例子是一个C/S结构的服务器的模型。在例子中，主进程(while(1))负责监听和分发连接，当客户端连入的时候，就会新建一个Service指针，并启动一个服务线程对客户进行服务。但是服务器启动后，内存占用量却不随着用户数据降低而减小，从而导致每隔一段时间服务器由于内存吃紧而不得不重启。

```
#include<iostream>
using namespace std;
int main()
{
    //开启服务端口
    //....
    //进入工作循环
    while(1)
    {
        //等待客户端连入
        if(User u=WaitConnet())
        {
            //初始化一个服务
            Service *S=new Service(u);
            //服务线程启动，主线程继续等待
            S->Start();
        }

    }
}
```

最后由开发者分析，这个问题是由于Service指针在用户退出的时候并没有释放导致的；一旦每一个用户连入，就会产生一个新的*S,由于它不会随着用户退出而释放，大量的*S永久地占用了内存，从而导致内存发生了严重的泄漏。解决这个问题只需要在Service线程中加入合适的delete语句就可以了。

3：无效指针

无效指针也叫做"野指针"。它指的是当某个指针被释放后，仍然有对象试图去访问这个指针。当发生无效指针错误的时候，程序可能会立即崩溃；在win32应用程序中，有很多程序崩溃之后提示的"内存不能为读"这种错误基本上都是无效指针在作怪。下面演示无效指针：

```
#include<iostream>
using namespace std;
class Book
{
    int id;
};
void fun(Book *b)
{
   delete b;
}
int main()
{
    Book *b1=new Book;
    Book *b2=b1;
    fun(b1)
    //b1已经无效，所以根本不能通过b2去访问id
    cout<<b2->id;
}
```

在上面程序中，因为fun()方法释放了b1,导致b2的引用失效，所以b2就成为了"野指针"。特别是在多线程程序中，基本上没有办法判断一个指针是不是无效指针;因此，无效指针的问题几乎困扰了一代C++开发者。

4：总结

由于指针很容易出问题，所以C++程序员无比向往Java语言的垃圾回收机制。其实C++要比Java优越地多，不要忘记Java虚拟机底层是用C设计的。那么，我们就自己动手设计一个简易版的智能指针，借此抛砖引玉来讲解Boost库中的智能指针。这个简易版本的智能指针用来管理Book对象的内存分配与回收问题。

第3页：局部智能指针的实现

1：说明

本页要讲解局部智能指针的实现,内容按照循循渐进的方式导入。

2：局部指针和共享指针

指针按照共享对象的数目不同，可以分为自持有指针和共享指针，其中自持有指针也叫做局部指针。分类如下表：

```
局部指针：
    不可拷贝，离开作用域后被销毁
共享指针：
    多个对象公有，没有对象共享的时候被销毁针
```

局部指针的生命周期在它声明的段内，它指向的数据结构的类型大多数为"服务";共享指针的生命周期很广，它指向的数据结构的类型大多数为"资源"。看下面例子：

```
使用局部指针例子：
    作业
    客户
    线程指针
    子窗口句柄
使用共享指针例子：
    数据库连接
    文件句柄
    网络连接
```

3：局部智能指针的行为

局部指针是"自持有"的，它有两个鲜明的特点：第一，它不能被拷贝;第二，离开作用域的时候自动被销毁。

4：实现局部智能指针的理论基础

要实现局部智能指针，我们必须知道编译器是如何管理内存的，编译器把内存分为这两个部分：

```
栈区：
    编译器自动维护
堆区：
    程序员负责维护
```

栈区的资源由编译器来管理，用来维护非指针数据类型;栈区的对象在离开作用域后会自动销毁。堆区的资源利用new和delete来管理，必须由程序员手动释放。 既然栈区的对象可以在离开作用域的时候释放，我们就可以实现一个"代理"类，让这个类的内部设置一个Book指针，我们通过这个代理类去操作Book指针，就像使用普通的指针一样；当代理类离开作用域前，在析构函数内调用delete删除Book指针就可以了。由于堆区和栈区的工作方式的特点，它为我们实现智能指针提供了理论基础。

5：实现

在下面的程序中，Book是一个简单的类，为了关于观察资源的释放情况，在它的构造和析构函数中加入了输出。My_Scoped_Ptr是我们要实现的代理类，它的构造函数接受一个Book指针，而在析构函数中删除Book指针；这样做可以使它所表示的Book指针的生命周期跟自己同步，这意味着当My_Scoped_Ptr离开作用域的同时，Book指针一起被销毁。为了使自己实现的代理类操作起来像指针，还重载了"->"运算符，这样使My_Scoped_Ptr几乎成为了一个真正的指针。

实现如下：

```
#include<iostream>
using namespace std;
//一个最普通的book类
class Book
{
public:
    int Id;
    string Name;
    float Price;
    Book(int id,string name,float price)
    {
        Id=id;
        Name=name;
        Price=price;
        cout<<"申请资源\n";
    }
    ~Book()
    {
        cout<<"释放资源\n";
    }
};
//管理Book指针的代理类
class My_Scoped_Ptr
{
public:
    My_Scoped_Ptr(Book *b)
    {
        book=b;
    }
    //const修饰返回值用来防止用户更改
    Book* operator->() const
    {
        return book;
    }
    ~My_Scoped_Ptr()
    {
        delete book;
    }
private:
    Book *book;
    My_Scoped_Ptr(){};
    //拷贝构造设置为私有，提供“自持有“性质
    My_Scoped_Ptr(My_Scoped_Ptr &Msp){};
};
//业务方法
void doLogical()
```

```
{
    My_Scoped_Ptr BookPtr(new Book(1,"童话世界",15.5));
    //看起来像访问指针一样
    cout<<BookPtr->Name<<endl;
}
//对比方法
void doLogical2()
{
    Book *book=new Book(1,"童话世界",15.5);
}
int main()
{

    doLogical();
    cout<<"----------\n";
    doLogical2();
}
```

在上面程序中，doLogical()使用代理指针管理Book，doLogical2()使用了传统的指针，从运行结果中可以清楚地看到doLogical2()方法发生了内存泄漏，而doLogical()没有。

运行结果：

```
申请资源
童话世界
释放资源
---------
申请资源
```

6：使局部智能指针泛型化

上面程序中的My_Scoped_Ptr类实现了代理"Book *"的功能，我们很容易利用泛型编程把它扩展为能代理任何对象。使用模板修改My_Scoped_Ptr如下：

```
#include<iostream>
using namespace std;
class Book{};
class User{};
template <class T>
class My_Scoped_Ptr
{
public:
    My_Scoped_Ptr(T *t)
    {
        Ptr=t;
    }
    T* operator->() const
    {
        return Ptr;
    }
    ~My_Scoped_Ptr()
    {
        delete Ptr;
    }
private:
    T *Ptr;
    My_Scoped_Ptr(){};
    My_Scoped_Ptr(My_Scoped_Ptr &Msp){};
};
int main()
{
    //自己实现的局部智能指针可以代理任意数据类型
    My_Scoped_Ptr<Book> BookPtr(new Book());
```

```
30      My_Scoped_Ptr<User> UserPtr(new User());
31  }
```

上面的程序中，由于My_Scoped_Ptr使用了模板，这样使它的作用大大增强了。我们实现的这个代理类My_Scoped_Ptr就叫做局部智能指针。

7:说明

boost库中的局部智能指针实现的原理跟本页讲解的原理是一模一样的，只不过boost库中的智能指针更加高效。

第4页：共享智能指针的实现

1:共享智能指针

共享智能指针用来代理拥有"共享"性质的指针；共享指针往往被多个对象持有，比如在一个程序中多个模块共享同一个数据库连接指针。 共享指针的特点有两点，第一：可以拷贝；第二：在最后一个持有者销毁的时候被释放。

2：单一版本的共享指针实现

共享指针也是一个代理类，由于多个对象要持有同一个指针，所以它要设置一个计数器，初始化为0，当增加一个持有者，计数器加1；减少一个持有者计数器减1，当计数器为0的时候，销毁它代理的指针。由于计数器是多个同类型对象共有的，所以计数器采用static修饰。下面的My_Shared_Ptr类用来代理Book*指针：

```
1   #include<iostream>
2   using namespace std;
3   class Book
4   {
5       public:
6       Book()
7       {
8           cout<<"Book()"<<endl;
9       }
10      ~Book()
11      {
12          cout<<"~Book()"<<endl;
13      }
14  };
15  class My_Shared_Ptr
16  {
17  public:
18      My_Shared_Ptr(Book *b)
19      {
20          book=b;
21      }
22      //拷贝构造
23      My_Shared_Ptr(My_Shared_Ptr &m)
24      {
25          book=m.book;
26          Count++;
27      }
28      //重载的->运算符，使它拥有指针的行为
29      Book* operator->() const
30      {
31          return book;
32      }
```

```
33      ~My_Shared_Ptr()
34      {
35          Count--;
36          if(Count==0)
37          {
38              delete book;
39          }
40      }
41
42  private:
43      Book *book;
44      //计数器，表示当前有几个对象持有这个指针
45      static int Count;
46  };
47  //计数器初始化
48  int My_Shared_Ptr::Count=0;
49  int main()
50  {
51      //申请一个指针并代理
52      My_Shared_Ptr book1(new Book());
53      //发生共享，计数器一次加1
54      My_Shared_Ptr  book2=book1;
55      My_Shared_Ptr  book3=book2;
56      //离开作用域，计数器依次减1
57      //当计数器为0时，释放资源
58  }
```

运行结果：

```
Book()
~Book()
```

从运行结果看来， book1对象在构造和被book2和boo3拷贝的时候，计数器依次加1；当这些对象依次离开作用域的时候，计数器依次减1；最后计数器为0的时候，删除了它代理的Book指针。

My_Shared_Ptr类如果利用模板设计，就可以代理任意类型的指针。

3：更复杂的计数器

上面的My_Shared_Ptr只能代理单个的指针，对多个Book*就无法代理了；如果把计数器设计成一个链式结构，那么它就可以代理多个指针，只不过这种计数器实现起来比较复杂；好在Boost库为我们实现了一个，接下来的内容将会过渡到Boost库的智能指针库。

欢迎来到智能指针的世界，理解这一章的内容，你的编程水平会有质的飞跃。

第5页：Boost智能指针库简介

1：智能指针库设计目的

使用智能指针可以避免“内存泄漏”、“无效指针”这些错误。实现垃圾回收机制。

2：智能指针的原理

智能指针库的底层原理是：用栈内存中的对象去代理堆内存中的指针数据。

3：智能指针定义

智能指针定义为一种代理对象，它的内部持有它代理的指针，智能指针会在合适的时候自动销毁它代理的指针。

4：使用智能指针的效果

使用智能指针库，程序员就彻底告别了delete操作。有了它，我们只管放心地使用new操作，因为智能指针库巧妙地帮你调用了delete操作，所以再也不会担 心资源不会被释放。

这个文档只有3章，所有章节请在百度云下载：http://pan.baidu.com/s/1ntOc6WL