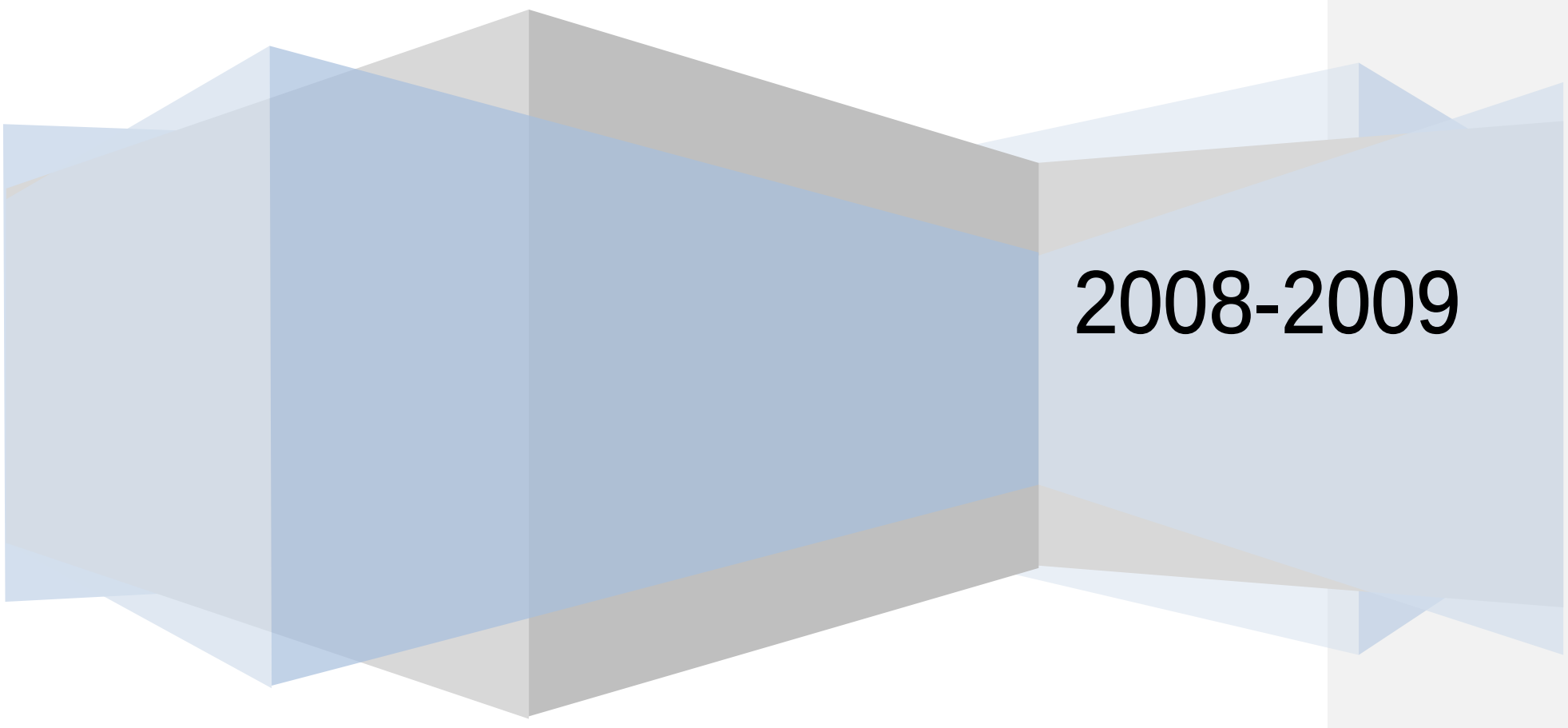
# C++ string 深入详解

董波

QQ: 84638372

Email: dbdongbo@vip.qq.com

Blog: http://84638372.qzone.qq.com



2008-2009

# 目录

# 正文

**简介：**

字符串处理是程序设计语言的一个重要的组成部分。有的字符串是内置的(VB 等)，有的是模拟的(C 字符串为字符集合，算法等以库的形式提供)，而 C++所提供的字符串 std::string 是以库的形式提供的。

学习 string 相关的操作是学习 C++语言很重要的一个内容，灵活的使用 string 对我们的程序设计是非常有帮助的，熟练使用 string 以及其相关的算法也是一项基本技能。

本文将对 C++的 string 体系做一些讨论，主要内容包括：

1.string 的使用

2.boost.algorithm.string 的使用以及其部分源码分析。

3.C 风格字符串的操作和使用。

声明：

本文中选择的源码为 MSSTL 和 boost1.37，例子为作者亲手所写，如果代码来自网络或者其它地方我会尽量声明出处。人难免会有疏漏，所以可能会有照顾不周的地方，我也无意侵害您的权益，如果您发现了这样的情况在本文档中出现，请您通知我，我保证马上更正。本文中凡是作者原创的内容，您可以随意修改与使用，欢迎您传播。

另外如果您发现文档中有疏漏的地方请您及时通知我，我会及时修改，免得误导更多的朋友，谢谢！

批注 [董波1]: 准确的说，应该是 std::basic_string。

为了方便，以后提到的字符串时通常我都直接使用 string。

## 一、 C++的 string 的使用

### 1.1 C++ string 简介

C++兼容 C 对字符串的处理方式，与此同时还在标准库(STL)中提供了 string 容器，我们可以很容易的使用 string 来进行字符串处理。而且 string 还能很好的与标准库中的泛型算法结合起来使用，非常的方便。虽然在 MFC 等框架中也提供了诸如 CString 这样的字符串处理类，但是个人认为 STL 的 string 依然是最棒的，使用标准库提供的 string 可以轻松的与原来的 C API 兼容，也可以很好的与系统底层的 API 兼容。

批注 [董波2]: 通过使用 char 和 wchar_t 对 basic_string 进行特化可以分别得到 string 和 wstring。为了方便我都直接说 string。

批注 [董波3]: 当然，std::string 并不是完美无暇的，实际上它是 STL 当中受人非议最多的，但是这并不妨碍我对它的喜爱。
我也希望大家都能对此有客观的认识。

### 1.2 string 的成员

#### 1.2.1 append

在尾部添加字符或者字符串

append 共有 8 种重载，分别如下：

```
basic_string<CharType, Traits, Allocator>& append(
   const value_type* _Ptr
   );    // 添加一个C风格的字符串_Ptr
basic_string<CharType, Traits, Allocator>& append(
   const value_type* _Ptr,
   size_type _Count
   );   // 添加 C风格的字符串 _Ptr中的_Count个字符
basic_string<CharType, Traits, Allocator>& append(
   const basic_string<CharType, Traits, Allocator>& _Str,
   size_type _Off,
   size_type _Count
   );  // 添加_Str从第_Off个开始的_Count个字符串
basic_string<CharType, Traits, Allocator>& append(
   const basic_string<CharType, Traits, Allocator>& _Str
   );  // 添加一个_Str
basic_string<CharType, Traits, Allocator>& append(
   size_type _Count,
   value_type _Ch
   ); // 添加_Count个_Ch
template<class InputIterator>
basic_string<CharType, Traits, Allocator>& append(
   InputIterator _First,
   InputIterator _Last
   ); // 添加迭代器指定的范围内的字符【可以来自其它容器】
basic_string<CharType, Traits, Allocator>& append(
   const_pointer _First,
   const_pointer _Last
   ); // 来自const_pointer
basic_string<CharType, Traits, Allocator>& append(
   const_iterator _First,
   const_iterator _Last
```

批注 [董波4]: 比如说 string 所提供的 c_str() const 成员函数将返回内部的 const 指针。

提供了得到长度的两个成员函数 size() const 和 length() const，实际上，这两个接口的接口是相同的。声明一个 length 完全是为了照顾以前 C 程序员的编程习惯。

批注 [董波5]:
下面的内容来自 MSDN。此处作为一个示例，因此我提供全部详细的情况，为了节省篇幅，以后我将不会提供全部的声明，具体的情况还请查看 MSDN。
其本身的含义也很简单，以后也不会多说，除非真的值得一说。

批注 [董波6]: 在 STL 中，const_pointer 几乎总是 const value_type * ，详细的内容将在源码分析部分讲述。

```
        ); // 来自const_iterator
```

例子：

MSDN

批注 [董波7]: 凡是在 MSDN 能够找到例子的我都不另外提供例子。请自行查阅。
除非我觉得有必要自己书写一个例子。

### 1.2.2 assign

为字符串重新赋予新的内容。
你可以将它看做先把字符串清空，然后再 append。因此 assign 也有 8 种重载。
例子：

```
#include <iostream>
#include <iterator>
#include <string>

using namespace std;

int main()
{
   string str( "Hello World!" );
   cout<< str << endl;

   str.assign( istream_iterator<char>(cin), istream_iterator<char>() );

   cout<< str << endl;

   return 0;
}
```

输出实例：

```
Hello World!
Dongbo! JiaYou!
^Z
Dongbo!JiaYou!
请按任意键继续. . .
```

批注 [董波8]: 这个是用 ctrl+Z 结束输入时所产生的。
可以看到这里忽略了空格，如果要接受空格的且要用迭代器的话有两个方法：
1. 设置标志位
2. 使用 istreambuf_iterator

### 1.2.3 at

类似于[]取值，根据调用情况分别返回 reference 或者 const_reference。
与[]的差别：

```
reference __CLR_OR_THIS_CALL at(size_type _Off)
   {  // subscript mutable sequence with checking
   if (_Mysize <= _Off)
      _String_base::_Xran();  // _Off off end
   return (_Myptr()[_Off]);
   }

const_reference __CLR_OR_THIS_CALL at(size_type _Off) const
   {  // subscript nonmutable sequence with checking
   if (_Mysize <= _Off)
      _String_base::_Xran();  // _Off off end
   return (_Myptr()[_Off]);
   }
reference __CLR_OR_THIS_CALL operator[](size_type _Off)
      {  // subscript mutable sequence

#if _HAS_ITERATOR_DEBUGGING
      // skip debug checks if the container is initizialed with _IGNORE_MYITERLIST
      if (this->_Myfirstiter != _IGNORE_MYITERLIST)
         {
         if (_Mysize < _Off)
            {
            _DEBUG_ERROR("string subscript out of range");
            _SCL_SECURE_OUT_OF_RANGE;
            }
         }
#else
      _SCL_SECURE_VALIDATE_RANGE(_Off <= _Mysize);
```

批注 [董波9]:

```
class _CRTIMP2_PURE
_String_base
    : public
_Container_base_secure
{// ultimate base class
for basic_string to hold
error reporters
public:
    _MRTIMP2_NPURE_NCEEPURE
static void
__CLRCALL_PURE_OR_CDECL
_Xlen();    // report a
length_error

    _MRTIMP2_NPURE_NCEEPURE
static void
__CLRCALL_PURE_OR_CDECL
_Xran();    // report an
out_of_range error
报告一个out_of_range错误
    _MRTIMP2_NPURE_NCEEPURE
static void
__CLRCALL_PURE_OR_CDECL
_Xinvarg();
};
```

```
#endif /* _HAS_ITERATOR_DEBUGGING */

     return (_Myptr()[_Off]);
     }

  const_reference __CLR_OR_THIS_CALL operator[](size_type _Off) const
     {  // subscript nonmutable sequence

#if _HAS_ITERATOR_DEBUGGING
     // skip debug checks if the container is initizialed with _IGNORE_MYITERLIST
     if (this->_Myfirstiter != _IGNORE_MYITERLIST)
        {
        if (_Mysize < _Off)  // sic
           {
           _DEBUG_ERROR("string subscript out of range");
           _SCL_SECURE_OUT_OF_RANGE;
           }
        }
#else
     _SCL_SECURE_VALIDATE_RANGE(_Off <= _Mysize);
#endif /* _HAS_ITERATOR_DEBUGGING */

     return (_Myptr()[_Off]);
     }
```

因此我们可以指导，at 遇到越界的时候是抛出异常，而重载[]是_DEBUG_ERROR!

### 1.2.4 begin

返回指向 string 首部的 iterator 或者 const_iterator。

### 1.2.5 c_str

返回 C 风格的字符串。可以与以前的 API 进行兼容。

关于“判等”的效率：

当 string 对象 str 与 const _Elem * _Ptr 比较之时有人这么做：

```
0 == strcmp( str.c_str(), _Ptr );
```

实际上这和 str == _Ptr 效果一样，效率也是一样。因为：

```
template _CRTIMP2_PURE bool __CLRCALL_OR_CDECL operator==(
   const basic_string<char, char_traits<char>, allocator<char> >&,
            const char *);
```

因为库编写者也考虑到了这种情况。

### 1.2.6 capacity

返回如果不再进行内存分配的话共能放多少字符。

例子：

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
   string str( "Hello World!" );
   cout<< str.size() << endl;
   cout<< str.capacity() << endl;
   cout<< str.max_size() << endl;

   cout<< str.append( " Haha!" ) << endl;

   cout<< str.size() << endl;
   cout<< str.capacity() << endl;

   return 0;
}
```

输出：

```
12
15
```

```
4294967294
Hello World! Haha!
18
31
请按任意键继续. . .
```

### 1.2.7 clear

删除所有字符。

### 1.2.8 compare

以字典顺序进行比较。>为正，=为零，<为负。

### 1.2.9 copy

拷贝，为内部调用。

### 1.2.10 _Copy_s

非标准，MSSTL only!为 copy 所调用。

### 1.2.11 data

与 c_str()相同。

### 1.2.12 empty

返回字符串是否为空。

批注 [董波10]: empty 的实现如下：

```
bool __CLR_OR_THIS_CALL empty() const
{
return (_Mysize == 0);
}
```

因此，使用 empty 的效率和 size() == 0;的效率相同。

这说明 string 和 list 等在 empty 实现上有本质的区别。

### 1.2.13 end

返回指向末尾的 iterator 或者 const_iterator。不能对其返回值提领!

### 1.2.14 erase

删除字符串。

```
iterator erase(
 iterator _First,
 iterator _Last
 );  // 删除指定范围 返回的是被删除位置的后一个元素的新位置
iterator erase(
 iterator _It
 );  // 删除指定位置
basic_string<CharType, Traits, Allocator>& erase(
 size_type _Pos = 0,
 size_type _Count = npos
 );  // 删除从_Pos开始的_Count个，当_Count为npos时，则为删除_Pos以及其后的部分!
```

### 1.2.15 find

批注 [董波11]: 查找算法经常用到，所以这一系列的多说一些。

在字符串中查找字符或者字符串。

```
 size_type find(
       value_type _Ch,
       size_type _Off = 0
       ) const;  // 从_Off开始查找_Ch，返回index。
   size_type find(
      const value_type* _Ptr,
      size_type _Off = 0
      ) const;  // 从_Off开始查找_Ptr
   size_type find(
      const value_type* _Ptr,
      size_type _Off,
      size_type _Count
      ) const;  // 从_Off开始查找_Ptr的前_Count个子串，有意义吗?
   size_type find(
      const basic_string<CharType, Traits, Allocator>& _Str,
      size_type _Off = 0
               ) const;  // 从_Off 开始查找_Str
```

例子:

```
#include <iostream>
#include <string>
```

```
using namespace std;

#define Print( exp )\
{\
   string::size_type pos = exp; \
   if( pos != string::npos ) \
      cout<< #exp <<"="<< pos << endl;\
   else\
      cout<< #exp <<"= 找不到" << endl;\
}

int main( )
{
   string ss( "Hello World!_Hello World!" );

   cout<< ss <<endl;

   Print( ss.find('o') );        // 直接查找o
   Print( ss.find('o', 5 ) );    // 从开始查找o
   Print( ss.find("He") );
   Print( ss.find("He",10) );
   Print( ss.find("Hello_") );
   Print( ss.find("Hello_",0,5 ) ); // 注意这个和上面的区别
   Print( ss.find("Hello_",1,5 ) );

   cout<<endl;

   string res("World");
   cout<< res <<endl;
   Print( ss.find( res,3 ) );

   return 0;
}
```

输出：

```
Hello World!_Hello World!
ss.find('o')=4
ss.find('o', 5 )=7
ss.find("He")=0
ss.find("He",10)=13
ss.find("Hello_")= 找不到
ss.find("Hello_",0,5 )=0
ss.find("Hello_",1,5 )=13

World
ss.find( res,3 )=6
请按任意键继续. . .
```

### 1.2.16 find_first_not_of

查找第一个不是指定字符或者不是指定字符串中的任意一个元素的位置并返回。

例子：

```
int main( )
{
   string ss( "Hello World!_Hello World!" );

   cout<< ss <<endl;

   Print( ss.find_first_not_of(' ') );
   Print( ss.find_first_not_of("Hel ") );

   return 0;
}
```

输出：

```
Hello World!_Hello World!
ss.find_first_not_of(' ')=0
ss.find_first_not_of("Hel ")=4
```

批注 [董波12]: 注意，以下四个类似的函数都有这样的一个或者类似声明：

```
size_type
find_first_not_of(
   const value_type* _Ptr,
   size_type _Off,
   size_type _Count
) const;
```

其中的_Count 都是对_Ptr 的限制，当然你可以选择不用这个限制。

比如例子中我们可以添加这样的一个语句：

```
Print( ss.find_first_not_of("Hel") );
Print( ss.find_first_not_of("Hel",0,2) );
```

这样第一个返回的是 4，而第二个是 2，因为虽然输入字符串是 Hel，但是却只取了 He，所以返回的是第一个不是 H 或者 e 的位置！

```
请按任意键继续. . .
```

### 1.2.17 find_first_of

查找第一个是指定字符或者是指定字符串中的任意一个元素的位置并返回！

### 1.2.18 find_last_not_of

查找最后一个不是指定字符或者不是指定字符串中的任意一个字符的位置并返回！

### 1.2.19 find_last_of

查找最后一个是指定字符或者是指定字符串中的任意一个元素的位置并返回！

### 1.2.20 get_allocator

返回我们所使用的 allocator，通常我们都不需要使用这个函数。

### 1.2.21 insert

插入一个字符、插入指定个数的字符、插入一个指定范围的字符串的元素。
这个函数拥有大量的重载，我们可以选择使用 index 或者是 iterator 来操作，非常的方便。
详细的例子可以参考 MSDN，其它没有什么值得说的。

### 1.2.22 length

返回字符串长度。

### 1.2.23 max_size

返回一个 string 所能容纳的最大字符个数。通常我们都不需要用到这个函数。

### 1.2.24 push_back

在尾部插入一个字符。

### 1.2.25 rbegin

返回反向迭代器，指向反向的第一个元素。

### 1.2.26 rend

返回反向迭代器，指向最后一个元素。绝对不能对它进行提领操作。

### 1.2.27 replace

字符串替换。

```
basic_string<CharType, Traits, Allocator>& replace(
   size_type _Pos1,
   size_type _Num1,
   const value_type* _Ptr
   );  // 把从_Pos1开始的_Num1个字符替换成_Ptr。
basic_string<CharType, Traits, Allocator>& replace(
   size_type _Pos1,
   size_type _Num1,
   const basic_string<CharType, Traits, Allocator>& _Str
   );  // 同上
basic_string<CharType, Traits, Allocator>& replace(
   size_type _Pos1,
   size_type _Num1,
   const value_type* _Ptr,
   size_type _Num2
   );  // 跟上面类似，不过这次只替换)_Ptr中的前_Num2个。如果_Num2>_Ptr的长度则会发生很奇怪的事情!
basic_string<CharType, Traits, Allocator>& replace(
   size_type _Pos1,
   size_type _Num1,
   const basic_string<CharType, Traits, Allocator>& _Str,
   size_type _Pos2,
   size_type _Num2
   );  // 这个就是替换从_Pos1开始的_Num1个字符为_Str从_Pos2开始的_Num2个。
       // 如果_Num2>_Str.size()，则取_Str.size()为_Num2
basic_string<CharType, Traits, Allocator>& replace(
   size_type _Pos1,
   size_type _Num1,
   size_type _Count,
   value_type _Ch
```

批注 [董波13]: 注意，系统相关。例子请参见 capacity

批注 [董波14]: 在我记忆中 VC6 中好像都没有吧？不过我们可以用 insert 来代替，或者是使用 char sz[2] = {0}; sz[0]=_Ch， str += sz; 这样也可以做到。

批注 [董波15]: 貌似未定义。
总之例子中执行这样代码的时候：
Print( ss.replace(0,4,"ABC",4 ) );
ss 将会变成"ABC"。后面的输出为什么会是那样的我亦不知为何。

```
    ); // 把从_Pos1开始的_Num1个字符替换为_Count个_Ch。
basic_string<CharType, Traits, Allocator>& replace(
    iterator _First0,
    iterator _Last0,
    const value_type* _Ptr
    );
basic_string<CharType, Traits, Allocator>& replace(
    iterator _First0,
    iterator _Last0,
    const basic_string<CharType, Traits, Allocator>& _Str
    );
basic_string<CharType, Traits, Allocator>& replace(
    iterator _First0,
    iterator _Last0,
    const value_type* _Ptr,
    size_type _Num2
    );
basic_string<CharType, Traits, Allocator>& replace(
    iterator _First0,
    iterator _Last0,
    size_type _Num2,
    value_type _Ch
    );
template<class InputIterator>
basic_string<CharType, Traits, Allocator>& replace(
    iterator _First0,
    iterator _Last0,
    InputIterator _First,
    InputIterator _Last
    );
basic_string<CharType, Traits, Allocator>& replace(
    iterator _First0,
    iterator _Last0,
    const_pointer _First,
    const_pointer _Last
    );
basic_string<CharType, Traits, Allocator>& replace(
    iterator _First0,
    iterator _Last0,
    const_iterator _First,
    const_iterator _Last
   );
```

批注 [董波16]: 接下来的部分跟前面的类似，不过是用 iterator 形成的 Rangle 来指定范围罢了。所以就不多说了。

例子:

```
#include <iostream>
#include <string>

using namespace std;

#define Print( exp )\
{  \
  exp;\
  cout<< ss << endl;\
}

#define Reset \
  ss.assign( "Hello World!_Hello World!" );\
  cout<<"Reset"<<endl<< ss << endl

int main( )
{
  string ss( "Hello World!_Hello World!" );

  cout<< ss <<endl;
  Print( ss.replace(0,2,"AA") );
  Print( ss.replace(0,4,"ABC") );
```

```
    Reset;
    Print( ss.replace(0,4,"ABC",2 ) );  // 特别注意这句和上面不同的效果
    Reset;
    Print( ss.replace(0,4,"ABC",4 ) );  // 多了的话就会发生很奇怪的事情，所以大家以后不要这么用

    Reset;
    Print( ss.replace(0,4,string("ABC"),1,2 ) );
    Reset;
    Print( ss.replace(0,4,string("ABC"),1,3 ) );

    Reset;
    Print( ss.replace(0,3,4,'H') );

    return 0;
}
```

输出：

```
Hello World!_Hello World!
AAllo World!_Hello World!
ABCo World!_Hello World!
Reset
Hello World!_Hello World!
ABo World!_Hello World!
Reset
Hello World!_Hello World!
ABC o World!_Hello World!
Reset
Hello World!_Hello World!
BCo World!_Hello World!
Reset
Hello World!_Hello World!
BCo World!_Hello World!
Reset
Hello World!_Hello World!
HHHHlo World!_Hello World!
请按任意键继续. . .
```

### 1.2.28 reserve

批注 [董波17]: 请特别注意 reserve 和 resize 的区别。

重新设置容器的容量至少要大于或者等于_Count

```
    void reserve(
         size_type _Count = 0
    );
```

例子：

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
   string str( "Hello World!" );
   cout<< str << endl;
   cout<< str.size() << endl;
   cout<< str.capacity() << endl;

   str.append( "Heh" );

   cout<< str << endl;
   cout<< str.size() << endl;
   cout<< str.capacity() << endl;

   str.push_back( 'a' );

   cout<< str << endl;
   cout<< str.size() << endl;
```

```
    cout<< str.capacity() << endl;

    str.reserve( 100 );

    cout<< str << endl;
    cout<< str.size() << endl;
    cout<< str.capacity() << endl;

    return 0;
}
```

这个简单所代表的意义在哪里？或者说reserve( 100 )的意义在哪里？

因为我们第一次输出的时候知道，我们一共能放15个字符，如果再需要插入一个新的字符必将导致内存重新分配。如果有的时候我们可以预见到新的内存分配，那么，我们就可以提前分配内存，这样当再需要push_back或者append的时候，我们可以得到常数时间的消耗。

批注 [董波18]: 指时间复杂度
因为你的内存已经分配好了，再次 push_back 只需要直接读'\0'为_Ch，然后将下一个设置为'\0'就可以了。

输出：

```
Hello World!
12
15
Hello World!Heh
15
15
Hello World!Heha
16
31
Hello World!Heha
16
111
请按任意键继续. . .
```

批注 [董波19]: 这里 push_back 了一个'a'就发生了新的内存分配。
而且新的内存分配不是线性增加的，而是倍数增加，这是 STL 内存分配的一个特色，是经过实践考验的。

### 1.2.29 resize

设置一个新的大小，可能会导致扩展或者删除字符串。

```
void resize(
         size_type _Count,
         );
void resize(
         size_type _Count,
         _Elem _Ch
          );
```

### 1.2.30 rfind

跟 find 类似，不过是反过来搜索。特别注意，返回值是 index，没有 rindex 的概念。

### 1.2.31 size

返回长度。跟 length()一样。

### 1.2.32 substr

返回字串

```
basic_string<CharType, Traits, Allocator> substr(
 size_type _Off = 0,
 size_type _Count = npos
          ) const;
```

返回从_Off 开始的_Count 个字符组成的字串，当_Count 使用默认值时，返回的是从_Off 到末尾。

### 1.2.33 swap

交换

批注 [董波20]: 你可能会用这种方法来构造一个字符串:

```
string
ss( istreambuf_iterator<char>(cin),
istreambuf_iterator<char>( ) );
```

你会发现编译器把它视为了函数声明，怎么解决这个问题呢？

1.把 istreambuf_iterator(cin)这种方式变换为变量定义的方式，然后将其对象传入定义即可。

```
2.string
ss( (istreambuf_iterator<char>(cin)),
istreambuf_iterator<char>( ) );
```

加个括号就好了。

## 1.3 string 的构造

string 有很多构造方式:

1. 默认的，也可以说是空的，大小为 0。
2. 从 const value_type * pStr 构造，长度是 pStr 的长度。
3. 从用 string::iterator 指定的范围构造。
4. 从 string 拷贝构造。
5. 使用别的 allocator 构造，大小也是 0，空的。
6. 还可以从指定 string 的某个索引开始的_Count 个字符来构造。
7. 使用_Count 个某特定字符_Ch 来构造。

## 1.4 string 的重载运算符

通常重载了+=、+、=，以及[]。含义都很明显，不多说了。
为什么不说<、==、>等？因为他们在 MSSTL 中不是以成员函数的方式提供的。不过它们的含义也很明显，这个不是我所强调的重点。

## 1.5 string 与 algorithm 相结合的使用

string 是 C++以库形式提供的字符串，但是首先，它是一个通用的容器，因此用它和 C++的泛型算法结合使用完全没有问题。

### 1.5.1 string 与 remove

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;

int main()
{
     string ss( "HelloWorld!_HelloWorld!" );
     cout<< ss <<endl;
     ss.erase( remove( ss.begin(), ss.end(), 'H' ),ss.end() );
     cout<< ss <<endl;

     ss = "HelloWorld!_HelloWorld!";

     ss.erase( remove_if( ss.begin(),ss.end(), islower ),ss.end() );

     cout<< ss << endl;

     return 0;
}
```

批注 [董波21]: 我喜欢用代码说话，所以以后我都直接上代码。这样可以不用说太多废话。

批注 [董波22]: 这叫 erase_remove 惯用法，我们必须要这样做，因为外部泛型算法只能调整内部结构并不能真正的删除。

批注 [董波23]: 删除所有的小写字母。

### 1.5.2 string 与 unique、sort

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;
int main()
{
     string ss( "HelloWorld!_HelloWorld!" );
     cout<< ss <<endl;
     sort( ss.begin(),ss.end() );
     cout<< ss <<endl;
     ss.erase( unique( ss.begin(),ss.end() ),ss.end() );
     cout<< ss << endl;

     return 0;
}
```

批注 [董波24]: 我不知道这是否应该叫 erase_unique 惯用法，这样使用的原因和 remove 是一样的。
可能这种用法给然的感觉很怪异，然而实际上就是如此。
或许有一天你能用到它也说不定。

### 1.5.3 string 与 search

```
#include <iostream>
#include <string>
#include <functional>
#include <algorithm>

using namespace std;

struct iCmp : public binary_function< bool, char , char >
{
  bool operator()( char left, char right )
  {
     return toupper( left ) == toupper( right );
  }
};
```

```
int main()
{
  string str( "Hello World!" );
  cout<< str << endl;

  string target( "Wo" );
  string::iterator pos = search( str.begin(), str.end(), target.begin(),target.end() );

  if( pos != str.end() )
  {
     cout<< *pos << endl;
     cout<< str.substr( distance( str.begin(),pos ), target.size() ) << endl;
     // ////////////////
     str.insert( distance( str.begin(), pos ),"wo" );

     pos = search( str.begin(),str.end(),target.begin(),target.end(), iCmp() );

     if( pos != str.end() )
     {
        cout<< str.substr( distance( str.begin(), pos ),target.size() ) << endl;
     }

  }

  return 0;
}
```

在 STL 中，我们可以选择这种方式来查找子串。

### 1.5.4 string 和 find、find_if

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;

int main()
{
   string str( "Shut Down!" );

   string::difference_type index = str.find( 'D' );

   string::iterator pos = str.begin();

   if( index != string::npos )
   {
      advance( pos, index );
      cout<< "advance : " << *pos << endl;
   }

   pos = find( str.begin(), str.end(), 'D' );

   if( pos != str.end() )
   {
      cout<< "find : " << *pos << endl;
   }

   return 0;
}
```

因为有成员函数版本的find，也许我们完全没必要用find，但是或许你也想像我一样得到迭代器呢？另外还可以与find_if结合来使用。

### 1.5.5 string 与 copy、copy_if

```
#include <iostream>
#include <string>
#include <algorithm>
```

批注 [董波25]: 注意，pos 必须初始化，而且必须是 index 相对的迭代器位置，否则 advance 不能得到正确的结果。

批注 [董波26]: template<class _InIt,
class _Diff> inline
void __CLRCALL_OR_CDECL
_Advance(_InIt& _Where,
_Diff _Off,
input_iterator_tag)
{// increment iterator by offset, input iterators

for (; 0 < _Off; --_Off)
++_Where;
}
注意看其源码。也就是说，我们的 _Off 是不能是负的。要达到那样的效果必须要用反向迭代器，而这一转换是相当费事费时的。

批注 [董波27]: 注意，标准中并没有 copy_if 这个算法，但是我们可以简单的实现一个，请看代码。

```
using namespace std;

template < typename InputIterator, \
         typename OutputIterator, \
         class Predicate >
OutputIterator copy_if( InputIterator begin , \
                  InputIterator end, \
                  OutputIterator out, \
                  Predicate pre )
{
   while( begin != end )
   {
      if( pre( *begin ) )
      {
         *out++ = *begin;
      }
      ++begin; // 这决定了它至少是一个前向迭代器!
   }

   return out;
};


int main()
{
   string str( "Shut DownO!" );

   copy( str.begin(),str.end(),ostream_iterator< char >( cout,"\n" ) );

   cout<<"-----------分割线----------"<<endl;

   // 只输出大写字母
   copy_if( str.begin(),str.end(),ostream_iterator< char >( cout,"\n"),isupper );

   return 0;
}
```

这里我的目的迭代器是输出迭代器，这里也可以是其它目的地。

### 1.5.6 string 与 count、count_if

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;

int main()
{
   string str( "Shut DownO!" );

   string::size_type uSize = count( str.begin(),str.end(), 'o' );

   cout<< uSize << endl;

   uSize = count_if( str.begin(),str.end(),isalnum ); // 空格和感叹号被忽略
   cout<< uSize << endl;

   return 0;
}
```

## 1.6 string 与 wstring

### 1.6.1 简介

basic_string 可以实例化任何字符，可以不是 char 或者 wchar_t。string 和 wstring 已经被设计为内置的分别支持传统字符和宽字符的字符串，它们的 char_type 分别是 char 和 wchar_t。

批注 [董波28]: 这里特指这两个：

```
typedef basic_string<char,
char_traits<char>,
allocator<char> >
string;
typedef
basic_string<wchar_t,
char_traits<wchar_t>,
allocator<wchar_t> >
wstring;
```

批注 [董波29]: 我们通常不需要这么做，但是如果你想要这么做的话，你最好是特化自己的 allocator 和 char_traits，这通常能够保证效率和可移植性。

### 1.6.2 wstring 实例

wstring 和 string 的方法差不多，只不过内部使用的是 wchar_t。这可以让我们更好的在 UNICODE 环境下使用 STL 提供的字符串。

```
typedef std::vector< wstring > FILEVEC;
FILEVEC GetFies()
{
    FILEVEC vec;

    TCHAR szOpenFileNames[80*MAX_PATH];

    // 初始化该结构体
    OPENFILENAME ofn;
    ZeroMemory( &ofn, sizeof(ofn) );

    ofn.Flags         = OFN_EXPLORER | OFN_ALLOWMULTISELECT;
    ofn.lStructSize    = sizeof(ofn);
    ofn.lpstrFile     = szOpenFileNames;
    ofn.nMaxFile      = sizeof(szOpenFileNames);
    ofn.lpstrFile[0]   = L'\0';
    ofn.lpstrFilter    = L"音乐文件(*.mp3)\0*.mp3\0";  // 初始化过滤器

    if( GetOpenFileName( &ofn ) )
    {
        wstring strPath( szOpenFileNames,szOpenFileNames + ofn.nFileOffset-1 );
        if( *strPath.rbegin() != L'\\' )
        {
            strPath.push_back( L'\\'  );
        }

        wchar_t * p = szOpenFileNames + ofn.nFileOffset; //把指针移到第一个文件
        while( *p )
        {
            vec.push_back( strPath + wstring(p) );
            p += wcslen(p) + 1 ;    //移至下一个文件
        }
    }

    return vec;
}
```

比如这个函数，它的作用是从“打开”对话框中获取到所选择的所有 mp3 文件的文件名。在这种情况下，我们需要使用 wstring 来和基于 wchar_t 的 API 结合使用。

### 1.6.3wstring 与控制台

批注 [董波30]: 准确点应该是 STL IO。

如果我们有一个 wstring 对象，那么如果要使用 STL 输出到文件的话，需要使用 wofstream；如果要输出到控制台需要 wcout。使用 ofstream 和 cout 是无法做到的。

```
#include <iostream>
#include <string>
#include <fstream>
#include <vector>

using namespace std;

int main()
{
    wstring wss( L"祖国万岁！" );
    wcout.imbue( std::locale( "chs" ) );

    wcout<< wss << endl;

    wofstream wofs( L"output.txt" );
    if( wofs.fail() )
    {
        wcout<<"错误"<< endl;
        return -1;
```

批注 [董波31]: 这是一个必要的步骤，至少在我的机器上。否则你无法输出 wstring。

```
    }

    wofs.imbue( std::locale( "chs" ) );

    wofs.write( wss.c_str(), wss.size() );

    typedef vector< wstring > WSVEC;
    WSVEC wsv;
    wsv.push_back( wss );
    wsv.push_back( L" 万岁！" );

    copy( wsv.begin(),wsv.end(),ostream_iterator<
wstring,wstring::value_type >( wcout,L"" ) );
    copy( wsv.begin(),wsv.end(),ostream_iterator<
wstring,wstring::value_type >( wofs,L"" ) );

    wofs.close();

    return 0;
}
```

批注 [董波32]: 这里为什么需要明确指定 wstring::value_type?

```
template<class _Ty,
→class _Elem = char,
→class _Traits =
char_traits<_Elem> >
→class ostream_iterator
→→: public _Outit
```

这是 ostream_iterator 的声明，因此它的_Elem 默认是 char，所以我们必须显示的指定这个模板参数。

### 1.6.4string 与 wstring 的相互转换

我们有很多种方式实现这种转换，我选择的是 WINDOWS API 版本的，如果您有更好的方式不妨告知与我。

```
#include <iostream>
#include <string>
#include <Windows.h>

using namespace std;

class BaseStr2Wstr
{
private:
    wchar_t    * m_szBuf;
public:
    BaseStr2Wstr() : m_szBuf(0)
    {
    }
    ~BaseStr2Wstr()
    {
        if( m_szBuf )
        {
            delete [] m_szBuf;
        }
    }
    std::wstring operator ()( const char* lpStr )
    {
        if( !lpStr )  // 为什么需要这一个判断？因为strlen(NULL)会崩溃
        {
            return wstring();
        }
        // 首先得到我们需要的缓冲区大小
        unsigned uSize = strlen( lpStr );
        int iLength = ::MultiByteToWideChar( CP_ACP, 0, \
            lpStr, uSize ,NULL,0 );
        wchar_t * m_szBuf = new wchar_t[ iLength + 1 ];
        if( !m_szBuf )
        {
            return wstring();
        }
        ::MultiByteToWideChar( CP_ACP, 0, \
            lpStr,uSize, m_szBuf,iLength );
        m_szBuf[ iLength ] = L'\0';

        return m_szBuf;
```

```
    }
    std::wstring operator () ( const std::string & str )
    {
        return operator()( str.c_str() );
    }
};

// 这样做的目的是为了提供像函数一样的外观。
#define Str2Wstr( str ) BaseStr2Wstr( )( str )

class BaseWstr2Str
{
private:
    char * m_szBuf;
public:
    BaseWstr2Str():m_szBuf(0){}
    ~BaseWstr2Str()
    {
        if( m_szBuf )
        {
            delete [] m_szBuf;
        }
    }
    std::string operator () ( const wchar_t * lpStr )
    {
        if( !lpStr )
        {
            return std::string();
        }
        unsigned uSize = wcslen( lpStr );
        int iLength = ::WideCharToMultiByte( 0,0, lpStr,uSize,NULL,0,NULL,NULL );

        m_szBuf = new char[ iLength + 1 ];

        ::WideCharToMultiByte( 0,0,lpStr, uSize, m_szBuf,iLength,NULL,NULL );
        m_szBuf[ iLength ] = '\0';

        return m_szBuf;
    }
    std::string operator () ( const std::wstring & wstr )
    {
        return operator () ( wstr.c_str() );
    }
};

#define Wstr2Str( str ) BaseWstr2Str()(str)

#define SafeDeleteArray( p ) \
    if( (p) ) \
{\
    delete [] p;\
}

// 这是一个函数版本的实现
static std::wstring DBConvertString2Wstring( const std::string& str )
{
    unsigned uLength = ::MultiByteToWideChar( CP_ACP, 0, \
        str.c_str(),static_cast<int>(str.size()),NULL,0 );
    wchar_t * pwBuffer = new wchar_t[ uLength + 1];
    ::MultiByteToWideChar( CP_ACP, 0, \

    str.c_str(),static_cast<int>( str.size() ),pwBuffer,static_cast<int>( uLength ) );
    pwBuffer[uLength] = L'\0';
```

```
    std::wstring tmp( pwBuffer );

    SafeDeleteArray( pwBuffer );

    return tmp;
}
static std::string DBConvertWstring2String( const std::wstring & wstr )
{
    unsigned uLength = ::WideCharToMultiByte( 0,0, wstr.c_str(), \
        wstr.size() , NULL,0,NULL,NULL );
    char * lpBuffer = new char[ uLength +1 ];

    ::WideCharToMultiByte( 0,0,wstr.c_str(),wstr.size(), lpBuffer, uLength ,NULL,NULL );
    lpBuffer[uLength] = '\0';

    std::string tmp( lpBuffer );
    SafeDeleteArray( lpBuffer );
    return tmp;
}

int main()
{
    string ss( "C++ string学习与研究__董波" );

    wcout.imbue( locale("chs") );

    wstring wss( Str2Wstr( ss.c_str() ) );

    wcout<< wss << endl;

    cout<< Wstr2Str( wss ) << endl;

    return 0;
}

#ifndef _DB_STRING_CAST_H_
#define _DB_STRING_CAST_H_

#include <string>
#include <cstdlib>
#include "Buffer.h"

namespace db
{
    // 几乎从来不用，用于扩展!
    template < typename _Ty >
class string_cast
{
private:
    string_cast();
    template < typename _Dummy >
    string_cast& operator = ( const string_cast<_Dummy>& );
    template < typename _Dummy >
    string_cast( const string_cast<_Dummy>& );
};

// 到wchar_t字符串的转换
template <>
class string_cast< wchar_t >
{
private:
    // 避免产生一些奇怪的语法!
    string_cast();
    template < typename _Dummy >
```

批注 [董波33]: 抛弃这种方式，原因就是涉及到了一些平台相关的东西，因为我找到了用 C 库实现的方法。

```
        string_cast& operator = ( const string_cast<_Dummy>& );
        template < typename _Dummy >
        string_cast( const string_cast<_Dummy>& );
    public:
        template < typename _TCHAR >
        string_cast( const _TCHAR* str )
        {
            if( NULL == str )
            {
                throw std::bad_cast( "目标串为空" );
            }
            m_strBuf = str;
        }

        template <>
        string_cast( const char* str )
        {
            // 检查指针状态
            if( NULL == str )
            {
                throw std::bad_cast( "目标串为空" );
            }

            // 获取长度以创建缓冲区
            unsigned iLength = strlen( str ) + 1;

            Buffer<wchar_t> buffer( iLength );

            // 修改现场以支持中文
            setlocale( LC_CTYPE, "chs" );

            // 转换
            size_t iSize = 0;
#if _MSC_VER > 1310
            mbstowcs_s( &iSize, buffer.GetBufPtr(), iLength, str, iLength );
#else
            mbstowcs( buffer.GetBufPtr(), str, iLength );
#endif

            // 还原现场
            setlocale( LC_CTYPE, "" );

            // 基本错误检查
            if( (iSize<<1) < iLength )
            {
                throw std::bad_cast( "转换未完成" );
            }

            // 拷贝到字符串中
            m_strBuf.assign( buffer.GetBufPtr() );
        }

        // 获取结果!
        operator std::wstring() const
        {
            return m_strBuf;
        }

    public:
        const std::wstring& ToWstr() const
        {
            return m_strBuf;
        }
    protected:
        std::wstring m_strBuf;
```

```
};

// 向string的转换
template<>
class string_cast< char >
{
private:
   string_cast();
   template < typename _Dummy >
   string_cast& operator = ( const string_cast<_Dummy>& );
   template < typename _Dummy >
   string_cast( const string_cast<_Dummy>& );
public:
   template < typename _TCHAR >
   string_cast( const _TCHAR* str )
   {
      if( NULL == str )
      {
         throw std::bad_cast( "目标串为空" );
      }
      m_strBuf = str;
   }

   template <>
   string_cast( const wchar_t* str )
   {
      if( NULL == str )
      {
         throw std::bad_cast( "目标串为空" );
      }

      unsigned iLength = ( wcslen( str ) + 1 )<<1;

      CharBuffer buffer( iLength );

      // 修改现场以支持中文
      setlocale( LC_CTYPE, "chs" );

      size_t iSize = 0;
#if _MSC_VER > 1310
      wcstombs_s( &iSize, buffer.GetBufPtr(), iLength, str, iLength );
#else
      wcstombs( buffer.GetBufPtr(), str, iLength );
#endif

      setlocale( LC_CTYPE, "" );
      if( (iSize<<1) < iLength )
      {
         throw std::bad_cast( "转换未完成" );
      }

      m_strBuf.assign( buffer.GetBufPtr() );
   }

   operator std::string() const
   {
      return m_strBuf;
   }
public:
   const std::string& ToStr() const
   {
      return m_strBuf;
   }
protected:
   std::string m_strBuf;
```

```
    };
}#endif // #ifndef _DB_STRING_CAST_H_
```

简单的一个测试程序:

```
#include <iostream>
#include "string_cast.h"

using namespace std;

int main()
{
    cout<< (string)( db::string_cast<char>( L"haha_董波" ) ) << endl;

    wcout.imbue( locale( "chs" ) );

    wcout<< (db::string_cast<wchar_t>( L"haha_董波"  )).ToWstr() << endl;

    wcout<< (wstring)( db::string_cast< wchar_t >( "哈哈_董波" ) ) << endl;

    cout<< ( db::string_cast<char>("呵呵") ).ToStr() << endl;
}
```

## 1.7 string 与 C++流

### 1.7.1 C++流简介

C++的流也是一个很广泛的内容，有很多东西值得我们去学习，比如说控制台的 IO 流，文件流，字符串流等等，Boost 中 Asio 也提供了网络数据流这样的东西。C++的流用起来是很方便的，也很好使用，以至于我现在都不知道 fopen 和 fread 等这些函数怎么用了，呵呵。

为什么说流用起来很简单呢？举个简单的例子，我们要把文件 1.txt 中的数据输出到 2.txt 中去，我们可以写这样的代码：

```
#include <iostream>
#include <fstream>
#include <cassert>
using namespace std;

int main()
{
    ifstream ifs( "1.txt" );
    ofstream ofs( "2.txt" );

    assert( ifs.is_open() && ofs.is_open() );

    ofs<< ifs.rdbuf();

    ifs.close();
    ofs.close();

    return 0;
}
```

是不是很简单呢？这里不多说了，直接进入下一主题吧。

### 1.7.2 string 与 iostream、fstream

String 在其实现中都对 iostream 的相关内容做了重载，所以用起来还是蛮方便的，比如说：

```
String str;
Cin>> str; cout<< str;
```

像这样的代码。

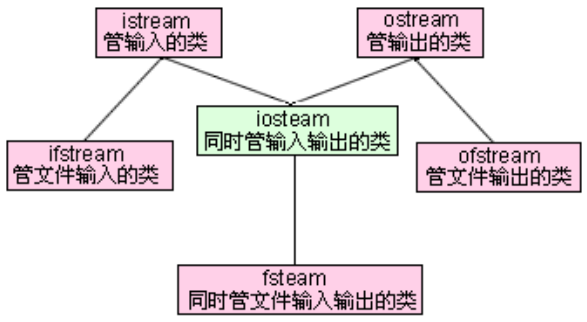
由于 string 本质是容器的特性，使得我们还可以有很多使用 string 的技巧，比如说我们要把控制台流中的数据用来构造一个 string，怎么做呢？

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;

int main()
{
   // 下面这个会忽略空格等
```

```
        //string str( (istream_iterator<char>(cin)), istream_iterator<char>() );

        // 下面这个你只要不输入Ctrl+Z，其它的都会吃进去
        string str( (istreambuf_iterator<char>(cin)), istreambuf_iterator<char>() );
        cout<< str << endl;

        return 0;
    }
```

是的，就这么简单。

由于 fstream 与 iostream 之间的关系，所以，iostream 能用的操作基本上 fstream 都是可以的。想知道为什么就看下图：




这不就是典型的 is-a 关系么？

## 1.8 格式化字符串

### 1.8.1 简单常用的 C 方法

比较常用的方法是自定义缓冲区来格式化之后再构造，比如：

```
#include <iostream>
#include <string>
#include <cstdlib>

using namespace std;

int main()
{
    char szBuf[100];

    sprintf_s( szBuf, "%s 能得%d分", "C++", 90 );


    string str(szBuf);

    cout<< str << endl;

    return 0;
}
```

好处就是方便撒，坏处就是 szBuf 如果不够大怎么办？不能很好的支持格式化字符串是 C++的阵痛点。

### 1.8.2 boost 的方法

人们的欲望永远都是无穷的，boost 恰好提供了这样的一个解决方案，它的名字叫 boost.Format。先给大家一个简单的例子：

```
#include <iostream>
#include <boost/format.hpp>

using namespace std;
using boost::format;
using boost::io::group;

int main()
{
    cout << format("(x,y) = (%+5d,%+5d) \n") % -23 % 35;
    cout << format("(x,y) = (%|+5|,%|+5|) \n") % -23 % 35;
    cout << format("(x,y) = (%1$+5d,%2$+5d) \n") % -23 % 35;
    cout << format("(x,y) = (%|1$+5|,%|2$+5|) \n") % -23 % 35;

    return 0;
}
```

这个例子采集自Boost的文档，这里我不打算详细的讲解这个库，只是想告诉大家有这样的一个东西可以做到这个事情。我在附录中提供Format的中文版文档。

批注 [董波34]: 不说的原因就是因为如果要详细讲解的话我就需要向大家引入 Boost.iostreams 体系，这带来的就不是一点点的内容了。
另外的一个原因就是这个库的效率并不是很好，所以没有必要作太多的解释。

## 1.9 string 与 CString

提出这个主题的目的就是说一些大家可能会遇到的问题。我们不能保证在MFC下面我们就一定不会使用string。其实有很简单的方法可以在它们两个之间进行转换，这个桥梁就是const char*，也就是Windows里边定义的LPCSTR。

批注 [董波35]: 对于 wchar_t 和 wstring 我们使用 LPCWSTR 就可以了。当然我们 MFC 中的宏 CString 将被替换为 CStringW。

比如：

```
CString strMFC = "C++";
string strIo( (LPCSTR)(strMFC) );
```

你没有看错，确实，CString提供一个到LPCSTR的强制转换！反过来就更容易了：

```
CString strMFC( strIO.c_str() );
```

就这么简单！

大家都指导CString有Format方法，这也提供了一个格式化string的机会，不过在使用之前请大家考虑下效率。呵呵。

# 二、 boost 字符串算法库

## 2.1 boost 字符串算法库导论

### 2.1.1 boost.algorithm.string 是什么？

boost.algorithm.string 是 boost 的算法库，它是对 STL 的 string 相关算法以及 STL 的 algorithm 的扩充，内容涵盖修剪、大小写转换、判断式、替换，以及与正则表达式结合等等。boost 字符串算法库的作者是 Pavol Droba。

批注 [董波36]: 伟人啊!! 呵呵!

### 2.1.2 相关

boost 字符串算法库实现为模板，因此我们要使用它们的话不需要编译 Boost，它们几乎全部位于 boost/algorithm/string 下面。

下面为 boost/algorithm/string.hpp 的内容，它包含了常使用的功能：

```
#include <boost/algorithm/string/std_containers_traits.hpp>
#include <boost/algorithm/string/trim.hpp>
#include <boost/algorithm/string/case_conv.hpp>
#include <boost/algorithm/string/predicate.hpp>
#include <boost/algorithm/string/find.hpp>
#include <boost/algorithm/string/split.hpp>
#include <boost/algorithm/string/join.hpp>
#include <boost/algorithm/string/replace.hpp>
#include <boost/algorithm/string/erase.hpp>
#include <boost/algorithm/string/classification.hpp>
#include <boost/algorithm/string/find_iterator.hpp>
```

从这些头文件我们就应该可以看出它们所具备的功能了：修剪、断言、转换、查找、分割、连接、替换、删除等等。然而这些头文件并不是真正的实现这些算法的地方，它们真正的实现位于 detail 文件夹下面的同名文件内。

### 2.1.3 boost.range 导论

批注 [董波37]: 因为字符串算法库中会大量使用到这个概念，所以不能不提前简单说说。

boost.range 贯穿着整个 boost 的算法体系，与字符串算法库也是如此。就像它的名字一样，range，范围(或者说是区间)，简单的说，它封装了两个迭代器，使之能够形成一个范围，这样使用起来比使用如 STL 一般的两个迭代器的方式要简单得多，而且也更优雅，还能够支持 boost 的新的迭代器定义体系。

批注 [董波38]: 它的作者是 Thorsten Ottosen。

range 支持下面这些类型：

批注 [董波39]: 指 boost.range，以后都这些了，除非有必要写完整的。

内置容器

pair< iterator , iterator >

固定数组

以_Elem()结尾的字符串。如 const char * p = “Hello World!”。

我们可以通过包含 boost/range.hpp 来使用它的功能。可以通过传入一个容器或者是两个迭代器来构造一个 range，还可以使用外部函数 make_iterator_range 来构造一个 range。虽然它的功能类似 pair< iterator , iterator >，但是 range 绝对比这个 pair 好用得多。

通常我们都使用 iterator_range 或者是 sub_range 就能够满足我们的需要了。

### 2.1.4 boost.regex 导论

批注 [董波40]: 因为字符串算法库中有与正在表达式相关的内容，而且还不少。

正则表达式是 boost 一个很重要的功能，它已经被 tr1 所接受将出现在新的 STL 当中。我们经常会用到的是 regex 对象和 wregex 对象，它们分别是 basic_regex 针对 char 和 wchar_t 的特化版本。通过传递一个合法的正则表达式字符串就可以构建一个 regex。通过 regex_march 我们可以检测当前正则表达式是否与目标字符串匹配，通过 regex_search 可以查找到能够匹配的子串，通过 march_results 可以得到相关的结果信息，使用 regex_replace 还可以进行替换。

必须要注意的是正则表达式默认的贪婪策略和在重复搜索中可能会出现的很多问题，这些可以通过在构造正则表达式的时候进行 case 配置来解决。

另外还提供了 regex_iterator 来方便我们使用，这个迭代器的出现使得我们的代码将更加的优雅。我们还可以使用 regex_token_iterator 来分割字符串。当用于构造的正则表达式不合法时，系统将会抛出 regex_error 异常，通过检测这

个异常我们可以得到更多的信息。

更多内容可以参阅 boost 相关文档或者是《C++标准库扩展权威指南》这本书。

### 2.1.5 boost.algorithm.string 的 DNA

boost 算法库中的很多模板函数的命名都非常的有规律，因此整个库用起来是非常容易的，比如说抓换为小写字母：

```
template<typename OutputIteratorT, typename RangeT>
inline OutputIteratorT
to_lower_copy(
    OutputIteratorT Output,
    const RangeT& Input,
    const std::locale& Loc=std::locale())
{
    return ::boost::algorithm::detail::transform_range_copy(
      Output,
      as_literal(Input),
      ::boost::algorithm::detail::to_lowerF<
          typename range_value<RangeT>::type >(Loc));
}
template<typename SequenceT>
inline SequenceT to_lower_copy(
    const SequenceT& Input,
    const std::locale& Loc=std::locale())
{
    return ::boost::algorithm::detail::transform_range_copy<SequenceT>(
        Input,
        ::boost::algorithm::detail::to_lowerF<
            typename range_value<SequenceT>::type >(Loc));
}
template<typename WritableRangeT>
inline void to_lower(
    WritableRangeT& Input,
    const std::locale& Loc=std::locale())
{
    ::boost::algorithm::detail::transform_range(
        as_literal(Input),
        ::boost::algorithm::detail::to_lowerF<
            typename range_value<WritableRangeT>::type >(Loc));
}
```

虽然红色部分非常值得一看，但是我们这里关注的应该是绿色的部分。

to_lower_copy 和 to_lower，有_copy 的都有返回值，而没有的则为 void，其中前者的输入仅仅是作为输入，而后则的输入还作为了输出。

另外还有修剪系列中的算法名：

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

凡是带有_if 的我们都可以传递自定义的断言。

批注 [董波41]: 可能它会有别的称谓，但是我比较喜欢断言。它的英文名是：Predicate。

在断言式当中还有这样的组合：

```
starts_with()
istarts_with()
```

其中 i 打头的意味着这个断言式是大小写不敏感的，也就是比较的时候会根据本地(locale)设置来忽略大小写。

上面只是从命名上做得讨论，而从实现上来说，其大部分实现都位于 detail 下面，命名空间亦为 detail。

## 2.2 boost 字符串算法解密

### 2.2.1 修剪（trim.hpp）

trim 算法库中是用于修剪字符串的：

```
trim_left_copy_if()
trim_left_if()
trim_left_copy()
trim_left()
trim_right_copy_if()
trim_right_if()
trim_right_copy()
trim_right()
```

trim_copy_if()
trim_if()
trim_copy()
trim()

第一组绿色表示修剪字符串左端的空格（含_if 的代表断言为真的字符）。
第二组黄色的表示修剪字符串右端的空格( 含_if 的代表断言为真的字符)。
第三组紫色的表示删除（修剪）两端。(_if 含义同上)
一个简单的例子：

批注 [董波42]: 别告诉我你是色盲，那我会很郁闷滴。

```
#pragma warning( disable : 4819 )

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

using namespace std;

// 输出一个\才好看到字符串结尾的地方
#define PrintStr( str ) cout<< #str <<"="<< str <<"\\"<< endl

int main()
{
    string str( " Hello World! " );
    PrintStr( str );

    string str1 = boost::trim_left_copy( str );
    PrintStr( str );
    PrintStr( str1 );

    str1 = str;
    PrintStr( str1 );
    boost::trim_left( str1 );
    PrintStr( str1 );

    boost::trim_left_if( str1,boost::algorithm::is_upper() );
    PrintStr( str1 );
    str1 = str;
    boost::trim_left_if( str1,boost::algorithm::is_upper() );
    PrintStr( str1 );

    str1 = boost::trim_copy( str );
    PrintStr( str );
    PrintStr( str1 );

    boost::trim( str );
    boost::trim_if( str, boost::algorithm::is_lower() );
    PrintStr( str );
    boost::trim_if( str, boost::algorithm::is_alpha() );
    PrintStr( str );

    return 0;
}
```

批注 [董波43]: boost 为了防止大家误用，没有_copy 的都是没有返回值的。

上面所有的_copy 版本的函数都是基于这种模板的：

```
template<typename SequenceT, typename PredicateT>
SequenceT trim_left_copy_if(const SequenceT & Input, PredicateT IsSpace);
```

这种重载能够为 trim 提供强安全保证，下面还有一种重载，这在 trim 算法集中只有_copy_if 版本才有：

```
template<typename OutputIteratorT, typename RangeT, typename PredicateT>
OutputIteratorT
trim_left_copy_if(OutputIteratorT Output, const RangeT & Input,
                  PredicateT IsSpace);
```

下面是这样的一个例子：

```
int main()
{
    string str( " Hello World! " );
    PrintStr( str );
```

```
string ss;
back_insert_iterator<string> it = \
    boost::algorithm::trim_left_copy_if( back_insert_iterator<string>(ss) ,\
    boost::make_iterator_range( str ),boost::algorithm::is_space() );
PrintStr( str );
PrintStr( ss );
*it = 'H';
PrintStr( ss );

ss.assign( str.begin(),str.end() );

PrintStr( ss );
string::iterator pos = \
    boost::algorithm::trim_copy_if( ss.begin(), boost::make_iterator_range( str ),\
    boost::algorithm::is_space() );
PrintStr( ss );

cout<< *pos << endl;

cout<< distance( ss.begin(),pos )<< endl;

    return 0;
}
```

输出：

```
str= Hello World! \
str= Hello World! \
ss=Hello World! \
ss=Hello World! H\
ss= Hello World! \
ss=Hello World!! \
H
0
请按任意键继续. . .
```

虽然加上命名空间限制，每个函数看起来都是那么长，但是用起来实际上还是非常简单的。

### 2.2.2 转换(case_conv.hpp)

这里涉及到大小写转换相关的模板函数。
它们都是这些：

```
template<typename OutputIteratorT, typename RangeT>
 inline OutputIteratorT
 to_lower_copy(
    OutputIteratorT Output,
    const RangeT& Input,
    const std::locale& Loc=std::locale())

 template<typename SequenceT>
 inline SequenceT to_lower_copy(
    const SequenceT& Input,
    const std::locale& Loc=std::locale())

template<typename WritableRangeT>
inline void to_lower(
    WritableRangeT& Input,
    const std::locale& Loc=std::locale())

template<typename OutputIteratorT, typename RangeT>
inline OutputIteratorT
to_upper_copy(
    OutputIteratorT Output,
    const RangeT& Input,
    const std::locale& Loc=std::locale())

template<typename SequenceT>
inline SequenceT to_upper_copy(
    const SequenceT& Input,
```

**批注 [董波44]**: 不要奇怪为什么上面一个例子只写 boost::也可以。因为在 trim.hpp 下面有这样的几行东西：

```
    } // namespace algorithm

    // pull names to the boost namespace
 using algorithm::trim_left;
    using algorithm::trim_left_if;
    略…
} // namespace boost
```

即在 boost 命名空间下声明使用 boost::algorithm 命名空间下面的模板函数。

**批注 [董波45]**: trim_left_copy_if，像这样的函数的返回值可能是两种：
1.如 back_insert_iterator 这样的迭代器将返回最后一个插入的_Elem 之后的输出迭代器。
2.如 string::iterator 这样返回一个输入迭代器的一个 Copy。

**批注 [董波46]**: 这里其实直接写 str 也一样。这里我这么写只是为了更明显罢了。因为我们可以有类型到对应 range 的隐式转换。

**批注 [董波47]**: 注意，这里只是输出，因此对原来的字符串的长度是不发生影响的。而且这里还必须假定 ss 的空间必须足够大，否则会出问题，因为它可能会对 ss.end()后面的内存区域解引用（提领）。这和 std::transform 的要求类似。

**批注 [董波48]**: 如果你经常使用 boost 的算法库的话，这会是一张熟脸，在这个文档中我不打算讨论与本地化相关的内容。

```
    const std::locale& Loc=std::locale())

template<typename WritableRangeT>
inline void to_upper(
    WritableRangeT& Input,
    const std::locale& Loc=std::locale())
```

例子：

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

using namespace std;

#define PrintStr( str ) cout<< #str <<"="<< str <<"\\"<< endl

int main()
{
    string str( "Hello World!" );

    PrintStr( str );
    string ss = boost::algorithm::to_lower_copy( str );
    PrintStr( ss );

    boost::algorithm::to_upper( ss );
    PrintStr( ss );

    return 0;
}
```

### 2.2.3 判断式、断言函数（predicate.hpp）【Predicates】

函数列表：

starts_with()
istarts_with()

检查一个字符串是否是另一个的前缀。

ends_with()
iends_with()

后缀检查。

contains()
icontains()

检查是否包含。

equals()
iequals()

检查是否相等。

lexicographical_compare()
ilexicographical_compare()

字典顺序比较。

all()

检查字符串中的每一个元素是否都满足某一判断式或者断言式。

在这一系列的模板函数当中，都是以 bool 来作为返回值的，也就是说，它们只管告诉你是与不是，其它的它都不管。而且除 all 和 i 系以外的函数都有两种形式，其中一种可以接受一个断言式作为其参数，如：

```
template<typename Range1T, typename Range2T, typename PredicateT>
bool starts_with(const Range1T & Input, const Range2T & Test,
                 PredicateT Comp);
template<typename Range1T, typename Range2T>
bool starts_with(const Range1T & Input, const Range2T & Test);
```

例子：

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

using namespace std;

#define PrintStr( str ) cout<< #str <<"="<< str <<"\\"<< endl
```

批注 [董波49]: template<typename RangeT, typename PredicateT> bool all(const RangeT & Input, PredicateT Pred);

批注 [董波50]: 指以 i 开头的函数。

批注 [董波51]: 类似的东西就不测试了。

```
#define PrintTest( exp ) cout<< #exp <<"="<< exp << endl
int main()
{
    string str( "Hello World!" );

    PrintStr( str );
    string ss = boost::algorithm::to_lower_copy( str );
    PrintStr( ss );

    string head = str.substr( 0, 5 );
    PrintStr( head );

    boolalpha( cout );
    PrintTest( boost::algorithm::starts_with( str , head ) );
    boost::to_lower( head );
    PrintTest( boost::algorithm::starts_with( str,head ) );
    PrintTest( boost::algorithm::istarts_with( str,head ) );
    PrintTest( boost::algorithm::equals( str,ss ) );
    PrintTest( boost::algorithm::iequals( str,ss ) );

    PrintTest( boost::algorithm::all( str, boost::algorithm::is_lower() ) );
    PrintTest( boost::algorithm::all( head, boost::is_lower() ) );

    return 0;
}
```

输出：

```
str=Hello World!\
ss=hello world!\
head=Hello\
boost::algorithm::starts_with( str , head )=true
boost::algorithm::starts_with( str,head )=false
boost::algorithm::istarts_with( str,head )=true
boost::algorithm::equals( str,ss )=false
boost::algorithm::iequals( str,ss )=true
boost::algorithm::all( str, boost::algorithm::is_lower() )=false
boost::algorithm::all( head, boost::is_lower() )=true
请按任意键继续. . .
```

### 2.2.4 查找

一些泛型算法的实现，大量的使用了 boost.range:

```
find_first()    // 查找第一次出现
ifind_first()
find_last()  // 查找最后一次出现
ifind_last()
find_nth()   // 查找第 N 次出现，从 0 次开始
ifind_nth()
find_head()  // head 和 tail 跟 substr 类似
find_tail()
find_token() // 查找满足条件的字符位置
find_regex() // 查找匹配正则表达式字串
find()  // 普通版
```

例子：

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

using namespace std;

#define Print( exp ) cout<< #exp <<"="<< exp << endl

using namespace boost;

int main()
```

```
{
    string str( "Hello World!" );
    Print( str );
    iterator_range< string::iterator > range = \
       algorithm::find_last( str, "or" );
    algorithm::to_upper( range );
    Print( str );

    copy( range.begin(), range.end(),ostream_iterator<char>( cout,"\n" ) );

    algorithm::to_lower( range );

    str += str + str;

    range = algorithm::find_nth( str, "Wor" ,2 );

    Print( str );
    cout<< str.size() << endl;
    cout<< distance( str.begin(),range.begin() )<< endl;

    range = algorithm::find_tail( str, 3 );

    Print( string( range.begin(),range.end() ) );

    range = algorithm::find_token( str, algorithm::is_punct() );

    cout<< range.size() << endl;

    Print( string( range.begin(),range.end() ) );

    cout<< distance( str.begin(), range.begin() ) << endl;

    return 0;
}
```

输出:

```
str=Hello World!
str=Hello WORld!
O
R
str=Hello World!Hello World!Hello World!
36
30
string( range.begin(),range.end() )=ld!
1
string( range.begin(),range.end() )=!
11
请按任意键继续. . .
```

## 2.2.5 删除和替换

| 算法名称 | 说明 | 函数 |
| --- | --- | --- |
| replace/erase_first | 替换/删除一个 string 在输入中的第一次出现 | replace_first()<br>replace_first_copy()<br>ireplace_first()<br>ireplace_first_copy()<br>erase_first()<br>erase_first_copy()<br>ierase_first()<br>ierase_first_copy() |
| replace/erase_last | 替换/删除一个 string 在输入中的最后一次出现 | replace_last()<br>replace_last_copy()<br>ireplace_last()<br>ireplace_last_copy()<br>erase_last() |

批注 [董波52]: 以下内容全部来自 Boost 的文档

| 算法名称 | 说明 | 函数 |
| --- | --- | --- |
| replace/erase_nth | 替换/删除一个 string 在输入中的第 n 次（从 0 开始索引）出现 | erase_last_copy()<br>ierase_last()<br>ierase_last_copy()<br><br>replace_nth()<br>replace_nth_copy()<br>ireplace_nth()<br>ireplace_nth_copy()<br>erase_nth()<br>erase_nth_copy()<br>ierase_nth()<br>ierase_nth_copy() |
| replace/erase_all | 替换/删除一个 string 在输入中的所有出现 | replace_all()<br>replace_all_copy()<br>ireplace_all()<br>ireplace_all_copy()<br>erase_all()<br>erase_all_copy()<br>ierase_all()<br>ierase_all_copy() |
| replace/erase_head | 替换/删除输入的开头 | replace_head()<br>replace_head_copy()<br>erase_head()<br>erase_head_copy() |
| replace/erase_tail | 替换/删除输入的末尾 | replace_tail()<br>replace_tail_copy()<br>erase_tail()<br>erase_tail_copy() |
| replace/erase_regex | 替换/删除与给定正则表达式匹配的一个 substring | replace_regex()<br>replace_regex_copy()<br>erase_regex()<br>erase_regex_copy() |
| replace/erase_regex_all | 替换/删除与给定正则表达式匹配的全部 substring | replace_all_regex()<br>replace_all_regex_copy()<br>erase_all_regex()<br>erase_all_regex_copy() |
| find_format | 通用替换算法 | find_format()<br>find_format_copy()<br>find_format_all()<br>find_format_all_copy() |

批注 [董波52]: 以下内容全部来自 Boost 的文档

一个简单例子：

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

using namespace std;

int main()
{
   string res( "Hello World!_Hello World!" );
   string de( "Wor" );
   string rep( "1wor1" );

   // 把第一次出现的Wor替换成wor1!!
   boost::algorithm::replace_first( res,de,rep );
```

```
        cout<< res << endl;
        // 忽略大小写的删除所有的wor
        boost::algorithm::ierase_all( res,de );

        cout<< res << endl

        return 0;
    }
```

其它的都类似，这里就不多写了，由此可见，boost 下面的字符串算法库真的很好很强大！同时也可以看到这个部分的内容是非常多的。

### 2.2.6 分割和组合

```
template< typename SequenceSequenceT, typename RangeT, typename PredicateT >
inline SequenceSequenceT& split(
    SequenceSequenceT& Result,
    RangeT& Input,
    PredicateT Pred,
    token_compress_mode_type eCompress=token_compress_off )
```

把指定的字符串按照要求分开。需要自己提供断言。

```
template< typename SequenceSequenceT, typename Range1T>
inline typename range_value<SequenceSequenceT>::type
join(
    const SequenceSequenceT& Input,
    const Range1T& Separator)
 template< typename SequenceSequenceT, typename Range1T, typename PredicateT>
inline typename range_value<SequenceSequenceT>::type
join_if(
    const SequenceSequenceT& Input,
    const Range1T& Separator,
    PredicateT Pred)
```

结合字符串，后者组合的字符串必须通过断言认证才能被组合，否则会被忽略。

一个简单例子:

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

using namespace std;

bool isEvenFirst( const string & str )
{
   return str[0] == '2' || str[0] == '4' || str[0] == '6' || str[0] == '8';
}

int main()
{
string ss( "HelloWorld!He.lloWorld!he" );

vector<string> tmp;
// 以标点符号分开!
vector<string>& tt = boost::algorithm::split( tmp, ss, boost::algorithm::is_punct() );

assert( boost::addressof(tmp) == boost::addressof(tt) );
copy( tt.begin(),tt.end(),ostream_iterator< string >( cout,"\n" ) );

tmp.clear();
int i = 10;
char sz[3];
while( i )
{
   sprintf_s(sz,"%d",i-- );
   tmp.push_back( sz );
}
```

```
    cout<< boost::algorithm::join( tmp, "%x%" )<<endl;

    cout<< boost::algorithm::join_if( tmp, "%x%",isEvenFirst )<<endl;

    return 0;
}
```

输出：

```
HelloWorld
He
lloWorld
he
10%x%9%x%8%x%7%x%6%x%5%x%4%x%3%x%2%x%1
8%x%6%x%4%x%2
请按任意键继续. . .
```

### 2.2.7 其它

其它还有一些查找迭代子、分割迭代子等等内容，此处就不说了，因为这并不是非常重要，我感觉，呵呵。

下面列举一些 boost 提供的判断式：

| 判断式名称 | 说明 | 生成器 |
|---|---|---|
| is_classified | 基于分类的通用 ctype 掩码 | is_classified() |
| is_space | 识别空格 | is_space() |
| is_alnum | 识别字母和数字字符 | is_alnum() |
| is_alpha | 识别字母 | is_alpha() |
| is_cntrl | 识别控制字符 | is_cntrl() |
| is_digit | 识别十进制数字 | is_digit() |
| is_graph | 识别图形字符 | is_graph() |
| is_lower | 识别小写字符 | is_lower() |
| is_print | 识别可打印字符 | is_print() |
| is_punct | 识别标点符号字符 | is_punct() |
| is_upper | 识别大写字符 | is_upper() |
| is_xdigit | 识别十六进制数字 | is_xdigit() |

批注 [董波53]: 取自 boost 文档

## 三、 C 字符串

C++从设计之初就考虑到了要兼容 C，因此 C 的字符串处理函数 C++也完全可以用，而且在 C++中我们同样也可以采用类 C 的字符串处理方式，特别是为了兼容以前的代码的时候。因此，我们有必要学习一下这方面的内容。

### 3.1 C 字符串常用算法

批注 [董波54]: 注意，在这些早期的 C 字符串函数中，通常不会对传入的参数做有效性检查，或者只做简单的检查，因此我们在传递参数之前都应该自己做一次检查。比如说当我们要通过 strlen 得到字符串长度的时候，通常应该判断将要传递给 strlen 的参数是否等于 NULL。如果等于 NULL，就不应该执行这个动作。

#### 3.1.1 strcpy wcscpy

拷贝字符串到字符数组中。

```
char *strcpy( char *strDestination, const char *strSource );
wchar_t *wcscpy( wchar_t *strDestination, const wchar_t *strSource );
```

太经典了，就不举例子了。

#### 3.1.2 strcat wcscat

追加字符串到原字符数组的缓冲区的字符串的末端。

```
char *strcat(
      char *strDestination,
      const char *strSource
      );
wchar_t *wcscat(
      wchar_t *strDestination,
      const wchar_t *strSource
      );
```

批注 [董波55]: 部分函数还有 MBS 版本，比如 unsigned char *_mbsrev( unsigned char *str ); 由于不常用，故以后都不写。

#### 3.1.3 strchr wcschr

在字符串中查找某个字符出现的位置。如果找到了则返回该字符的指针，否则返回NULL。

```
#include <iostream>

using namespace std;

int main()
{
    const wchar_t* pszRes = L"Hello World!";
```

```
    const wchar_t* pPos = wcschr( pszRes, L'W' ); // 区分大小写

    if( NULL == pPos )
    {
        cout<<"没找到"<<endl;
    }
    else
    {
        wcout.imbue( locale("chs") );
        wcout<< pPos << endl;
    }

    return 0;
}
```

### 3.1.4 strcmp wcscmp

字符串比较，按照字典顺序。小于返回-1 等于返回 0，大于返回 1.

### 3.1.5 stricmp wcsicmp

除了忽略大小写之外，其它同上。

### 3.1.6 strlen wcslen

计算字符串长度。注意：如果传入 NULL 会导致崩溃。

### 3.1.7 strlwr/_strlwr wcslwr/_wcslwr

从 VC2005 开始，strlwr 这样的函数被 deprecated，所以当_MSC_VER >1310 的时候我们应该甚至必须选择_strlwr 等。这个函数的作用是将字符中的小写字母转换成大写。

批注 [董波56]: 以后出现类似的表达，即有个下划线打头的函数名，如果没有特殊说明，也是这个原因，即原版的被 VC2005 以及其后的 VS 所 deprecated!。

MSDN 的例子：

```
#include <string.h>
#include <stdio.h>

int main( void )
{
   char string[100] = "The String to End All Strings!";
   char * copy1 = _strdup( string ); // make two copies
   char * copy2 = _strdup( string );

   _strlwr( copy1 ); // C4996
   // Note: _strlwr is deprecated; consider using _strlwr_s instead
   _strupr( copy2 ); // C4996
   // Note: _strupr is deprecated; consider using _strupr_s instead

   printf( "Mixed: %s\n", string );
   printf( "Lower: %s\n", copy1 );
   printf( "Upper: %s\n", copy2 );

   free( copy1 );
   free( copy2 );
}
```

### 3.1.8 strncat wcsncat

```
char *strncat( char *strDest, const char *strSource, size_t count );
wchar_t *wcsncat( wchar_t *strDest, const wchar_t *strSource, size_t count );
```

追加 count 个字符。如果 count 大于 strSource 的长度，那么 count 将被替换成 strSource 的长度。

### 3.1.9 strcspn wcscspn

批注 [董波57]: 个人推荐使用 strpbrk/wcspbrk 来代替它。

```
size_t strcspn( const char *string, const char *strCharSet );
size_t wcscspn( const wchar_t *string, const wchar_t *strCharSet );
```

返回字符串中连续不含指定字符串内容的字符数。

简单的来说，这个函数就是对 strCharSet 中的每个字符执行 strchar/wcschr。

```
#include <iostream>

using namespace std;
```

```
int main()
{
   const char* pszPes = "Hello World!";

   // 遇到e或者是W就结束搜寻
   cout<< strcspn( pszPes, "eW" ) << endl;

   if( strlen(pszPes) == strcspn( pszPes, "xyz") )
   {
      cout<<"找不到"<<endl;
   }

   return 0;
}
```

### 3.1.10 strdup/_strdup wcsdup/_wcsdup

```
 char *_strdup( const char *strSource );
wchar_t *_wcsdup( const wchar_t *strSource );
```

_strdup 将在底层调用 malloc 为我们分配一个字符串内存，然后拷贝字符串到该缓冲区中。因此我们必须自己 free 这个内存，看 MSDN 的例子：

```
#include <string.h>
#include <stdio.h>

int main( void )
{
   char buffer[] = "This is the buffer text";
   char *newstring;
   printf( "Original: %s\n", buffer );
   newstring = _strdup( buffer );
   printf( "Copy:    %s\n", newstring );
   free( newstring );
}
```

个人认为我们不应该再继续使用这个函数。注意：在这里我们应该检查其返回值，它可能返回 NULL，表示内部内存分配失败。

### 3.1.11 strncpy wcsncpy

```
 char *strncpy( char *strDest, const char *strSource, size_t count );
wchar_t *wcsncpy( wchar_t *strDest, const wchar_t *strSource, size_t count );
```

拷贝 count 个 strSource 中的字符到 strDest 中去。

```
#include <iostream>

using namespace std;

int main()
{
   char szBuf[10];
   memset( szBuf, 1, 10 );

strncpy( szBuf, "haha", 3 );

cout.write( szBuf, 10 );

memset( szBuf, 1, 10 );

strncpy( szBuf, "haha", 6 );  // 注意，这里取决于C库的实现

cout<<endl;
cout.write( szBuf, 10 );

memset( szBuf, 1, 10 );

strcpy( szBuf, "haha" );
cout<<endl;
cout.write( szBuf, 10 );

return 0;
```

```
}
```

需要注意的就是当 count 比 strSource 长度大的时候的处理，Windows 平台下面，函数会用'\0'来填充。另外，如果源字符串和缓冲区内存区域有重叠的话，那么函数的行为是未定义的。

### 3.1.12 strpbrk wcspbrk

```
char *strpbrk( const char *string, const char *strCharSet );
wchar_t *wcspbrk( const wchar_t *string, const wchar_t *strCharSet );
```

功能上类似 strcpsn，不过它返回的是指针。如果查找不到则返回 NULL。

### 3.1.13 strrev/_strrev wcsrev/_wcsrev

```
char *_strrev( char *str );
wchar_t *_wcsrev( wchar_t *str );
```

就是反转字符串。

### 3.1.14 strset/_strset/_strset_l wcsset/_wcsset/_wcsset_l

```
char *_strset(
    char *str,
    int c
    );
char *_strset_l(
    char *str,
    int c,
    locale_t locale
    );
wchar_t *_wcsset(
    wchar_t *str,
    wchar_t c
    );
wchar_t *_wcsset_l(
    wchar_t *str,
    wchar_t c,
    locale_t locale
    );
unsigned char *_mbsset(
    unsigned char *str,
    unsigned int c
    );
unsigned char *_mbsset_l(
    unsigned char *str,
    unsigned int c,
    _locale_t locale
   );
```

即用指定字符填充字符缓冲区，不能是 const _Ch*，只能是可写的并且需要以'\0'结尾，否则会出错。

### 3.1.15 strstr/wcsstr

```
char *strstr(
    const char *str,
    const char *strSearch
    ); // C only
char *strstr(
    char *str,
    const char *strSearch
    ); // C++ only
const char *strstr(
    const char *str,
    const char *strSearch
    ); // C++ only
wchar_t *wcsstr(
    const wchar_t *str,
    const wchar_t *strSearch
    ); // C only
wchar_t *wcsstr(
    wchar_t *str,
    const wchar_t *strSearch
    ); // C++ only
```

批注 [董波58]: 后面有跟 l 的表示 locale，即和本地化有关，由于我们不常用，因为我们可以通过设置 locale 来代替这个操作，当然，在取决于个人习惯，本人推荐使用_l 版的函数，这样你不用显示的还原全局 locale。因此其它函数都没有着重说明。

```
const wchar_t *wcsstr(
   const wchar_t *str,
   const wchar_t *strSearch
      ); // C++ only
```

查找子串，这个千万别和 strcspn 和 strpbrk 搞混淆了。成功返回起始指针，失败返回 NULL。

### 3.1.16 strtok/wcstok

```
char *strtok(
   char *strToken,
   const char *strDelimit
   );
wchar_t *wcstok(
   wchar_t *strToken,
   const wchar_t *strDelimit
      );
```

分割字符串。大家可以注意到，因为函数会修改缓冲区，所以我们不能传递 const _Ch*给该参数。这个修改发生于当找到 Delimit 的时候，函数会将其修改为'\0'。看 MSDN 的例子：

```
#include <string.h>
#include <stdio.h>

char string[] = "A string\tof ,,tokens\nand some  more tokens";
char seps[]   = " ,\t\n";
char *token;

int main( void )
{
   printf( "Tokens:\n" );

   token = strtok( string, seps ); // C4996
   while( token != NULL )
   {
      printf( " %s\n", token );

      // Get next token:
      token = strtok( NULL, seps ); // C4996
   }

   return 0;
}
```

注意，我们的 Delimit 是一个字符串而不是字符，说明了它以是字符集，也就是像微软提供的这个例子一样，Delimit 是一个字符的集。

### 3.1.17 strupr/_strupr wcsupr/_wcsupr

```
char *_strupr(
   char *str
   );
wchar_t *_wcsupr(
   wchar_t *str
      );
```

这个函数和 strlwr 是兄弟函数，它就是将缓冲区中的字符串全部转换成大写。由于这个涉及到本地化的问题，它也有_l 版本的函数，这里被我忽略。

## 3.2 更安全的 C 字符串函数

批注 [董波59]: 本节内容必须在 Windows 环境下，同时必须是 VC2005 或者其更新版本的 VC。

### 3.2.1 简述

从 Visual C++ 2005 开始，微软就将上面我们所提到的函数 dprecated 了，如果您在 VC2005 或者是其后的版本中使用上面的字符串函数，通常会得到一个 4496 警告告诉你这个函数可能是不安全的。它推荐我们使用跟安全的版本，而这些新的安全版本通常都由_s 结尾(如果有 locale 的话以_s_l 结尾)。值得注意的是，通常只有在使用需要对缓冲区写操作的时候才会得到这个 warning，比如说我们依然可以继续使用 strlen、strcmp 等，由于我们不会对对象进行写操作，则不会出现缓冲区溢出的问题。由此我们可以了解到微软提供这样的一批函数的主要目的就是避免缓冲区溢出。

### 3.2.2 简单实例

下面我就以 strcpy_s 为例进行说明。

```
errno_t strcpy_s(
   char *strDestination,
```

```
        size_t numberOfElements,
        const char *strSource
        );
    errno_t wcscpy_s(
        wchar_t *strDestination,
        size_t numberOfElements,
        const wchar_t *strSource
        );
    errno_t _mbscpy_s(
        unsigned char *strDestination,
        size_t numberOfElements,
        const unsigned char *strSource
        );
    template <size_t size>
    errno_t strcpy_s(
        char (&strDestination)[size],
        const char *strSource
        ); // C++ only
    template <size_t size>
    errno_t wcscpy_s(
        wchar_t (&strDestination)[size],
        const wchar_t *strSource
        ); // C++ only
    template <size_t size>
    errno_t _mbscpy_s(
        unsigned char (&strDestination)[size],
        const unsigned char *strSource
        ); // C++ only
```

我们可以看到函数将不再返回字符指针了，取而代之的是一个 errno_t。这个返回值用来传递错误码。特别注意下面的一些 C++ Only 版本，因为在这些函数中使用模板，通过传递数组的引用，使得我们对于静态的位于栈上的数组不必重复输入其长度，实际上这恰好是将 _countof 宏的功能整合到了函数中。

```
    #if !defined(_countof)
    #if !defined(__cplusplus)
    #define _countof(_Array) (sizeof(_Array) / sizeof(_Array[0]))
    #else
    extern "C++"
    {
        template <typename _CountofType, size_t _SizeOfArray>
        char (*__countof_helper(UNALIGNED _CountofType (&_Array)[_SizeOfArray]))[_SizeOfArray];
    #define _countof(_Array) sizeof(*__countof_helper(_Array))
    }
```

因为通常我们都这样用：

```
  char szbuf[100];
  strcpy_s( szbuf, _countof(szbuf), "haha" );
```

由于这个 c++ Only 的模板的存在这样用也对的：

```
  char szbuf[100];
   strcpy_s( szbuf, "haha" );
```

因此可以知道当字符缓冲区只是一个原生指针的时候，我们必须显示的提供大小，同时绝对不可以对其使用_countof，也不能是 sizeof.

MSDN 的列表告诉了我们 strcpy_s 的返回值：

**Zero if successful; an error otherwise.**

| *strDestination* | *numberOfElements* | *strSource* | Return value | Contents of *strDestination* |
|---|---|---|---|---|
| NULL | any | any | EINVAL | not modified |
| any | any | NULL | EINVAL | *strDestination*[0] set to 0 |
| any | 0, or too small | any | ERANGE | *strDestination*[0] set to 0 |

我们可以通过检查这些返回值来进行错误处理，使得我们的程序容错性更强！MSDN 的例子：

```
    #include <string.h>
    #include <stdlib.h>
```

```
#include <stdio.h>
#include <errno.h>

int main( void )
{
   char string[80];
   // using template versions of strcpy_s and strcat_s:
   strcpy_s( string, "Hello world from " );
   strcat_s( string, "strcpy_s " );
   strcat_s( string, "and " );
   // of course we can supply the size explicitly if we want to:
   strcat_s( string, _countof(string), "strcat_s!" );

   printf( "String = %s\n", string );
}
```

### 3.2.3 定制

上一节说到了_s 系列的函数通常都有一个 errno_t 类型的返回值。errno_t 实际上就是一个 int 的 typedef:

```
typedef int errno_t;
```

关于更多的错误信息请查看 VC2005 或者其后版本的 VS 安装目录下的 errno.h 头文件。比如 VC2008 的：

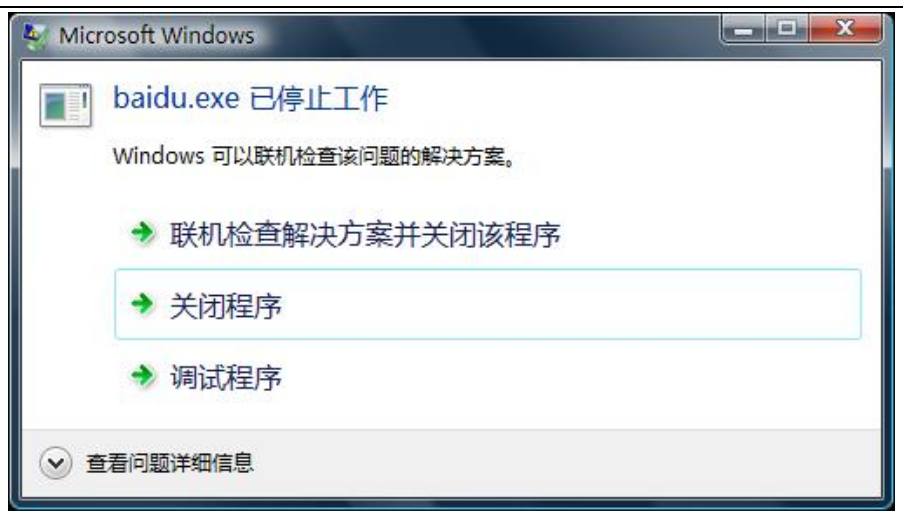
```
X:\Program Files\Microsoft Visual Studio 9.0\VC\include\errno.h
```

我们来制造一个错误：

```
#include <iostream>

using namespace std;

int main()
{
   char szbuf[2];

   errno_t et = strcpy_s( szbuf, "haha" );

   if( 0 != et )
   {
      cerr<<"格式化失败"<<endl;
   }

   return 0;
}
```

本来我们应该检查到这个错误的，但是很遗憾，如果是在 Debug 下面，你会看到：


如果实在 Release 下面会更惨一些：




当然，这是在 Vista 下面的截图。到 VC 目录下面找到 tcscpy_s.inl：

```
/***
*tcscpy_s.inl - general implementation of _tcscpy_s
*
*      Copyright (c) Microsoft Corporation. All rights reserved.
*
*Purpose:
*      This file contains the general algorithm for strcpy_s and its variants.
*
****/

_FUNC_PROLOGUE
errno_t __cdecl _FUNC_NAME(_CHAR *_DEST, size_t _SIZE, const _CHAR *_SRC)
{
    _CHAR *p;
    size_t available;

    /* validation section */
    _VALIDATE_STRING(_DEST, _SIZE);
    _VALIDATE_POINTER_RESET_STRING(_SRC, _DEST, _SIZE);

    p = _DEST;
    available = _SIZE;
    while ((*p++ = *_SRC++) != 0 && --available > 0)
    {
    }

    if (available == 0)
    {
        _RESET_STRING(_DEST, _SIZE);
        _RETURN_BUFFER_TOO_SMALL(_DEST, _SIZE);
    }
    _FILL_STRING(_DEST, _SIZE, _SIZE - available + 1);
        _RETURN_NO_ERROR;
    }
```

这就是 C 运行库的实现源码的部分，我们在 Debug 环境下面触发的就是这个 Assert。

```
  #define  _RETURN_BUFFER_TOO_SMALL(_String,  _Size)  _RETURN_BUFFER_TOO_SMALL_ERROR((_String),
(_Size), ERANGE)
```

现在，我们要设法使得安全版字符串函数更友好更人性化的处理程序中出现的错误，这就需要定制 CRT 检查。下面是我写的一个例子：

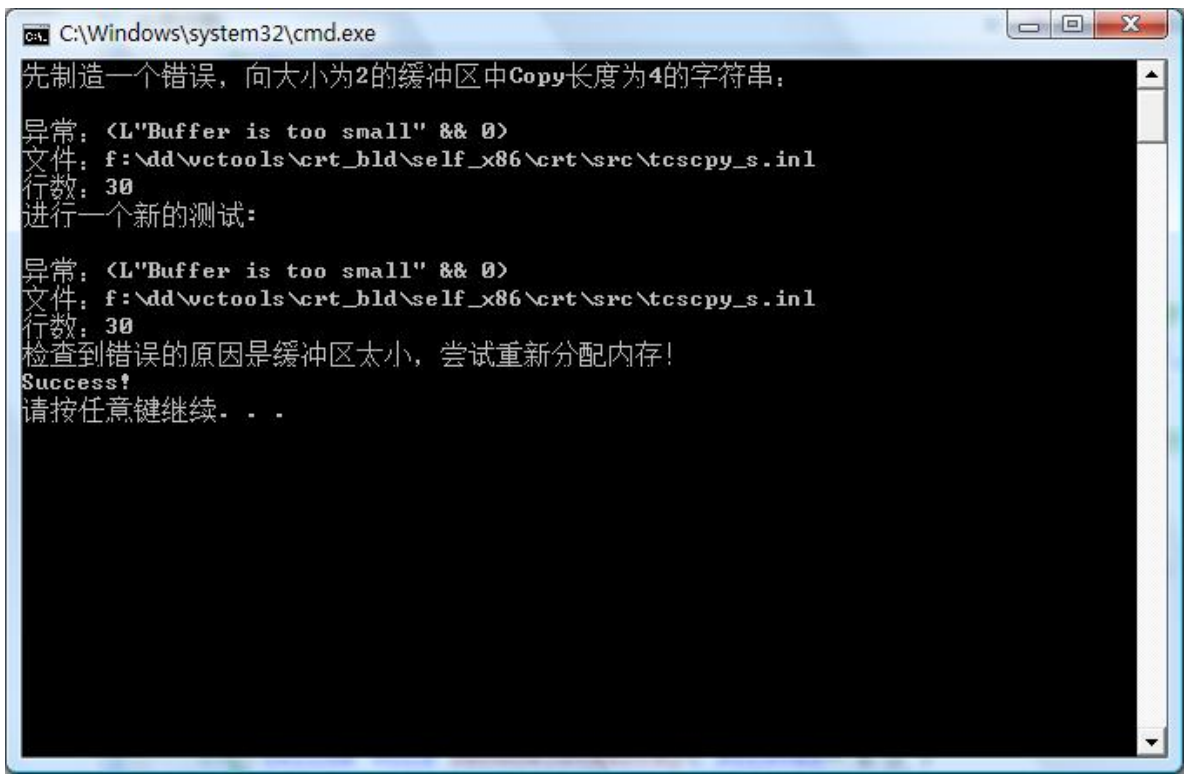
```
#include <iostream>
#include <cerrno>

using namespace std;

#include <tchar.h>


// 2. 写自己的处理函数// 注意，这个handle只能接受wchar_t字符串
void InvalidHandle( const wchar_t* expression, const wchar_t* function,
                const wchar_t* file, unsigned int line, uintptr_t /*pReserved*/)
{
```

```
    // 注意，以下的处理必须是在Debug环境下，因为在Release环境下传递进来的参数都是不正确的，
    // 指针都是x00000000
#if defined(DEBUG)||defined(_DEBUG)
    wcout.imbue( locale("chs") );
    wcout<< L"\n异常："<< expression << L"\n文件："<< file << L"\n行数：" << line << endl;
    wcout.imbue( locale("") );
#endif // #if defined(DEBUG)||defined(_DEBUG)
}

template < typename Pointer >
inline void SafeDeleteArray( Pointer* & p )
{
    if( NULL != p )
    {
        delete []p;
        p = NULL;
    }
}

void Test()
{
    TCHAR* pBuf = NULL;
    __try
    {
        pBuf = new TCHAR[5];
        errno_t tResult = _tcscpy_s( pBuf, 5, _T("哇卡卡卡卡") );
        if( 0 == tResult )
        {
            cout<<"拷贝成功"<<endl;
        }
        else if( ERANGE == tResult )
        {
            SafeDeleteArray( pBuf );

            cout<<"检查到错误的原因是缓冲区太小，尝试重新分配内存！"<<endl;
            pBuf = new TCHAR[20];  // 这里貌似Release需要更大的缓冲区
            tResult = _tcscpy_s( pBuf, 20, _T("哇卡卡卡卡") );

            if( 0 != tResult )
            {
                cerr<<"Error!"<<endl;
            }
            else
            {
                cout<<"Success!"<<endl;
            }
        }
        else
        {
            cout<<"错误"<<endl;
        }
    }
    __finally
    {
        SafeDeleteArray( pBuf );
    }
}
int main()
{
    // 1.关闭这个东东!
    _CrtSetReportMode(_CRT_ASSERT, 0);

    // 3.将它设置为处理的函数!
    _set_invalid_parameter_handler( InvalidHandle );
```

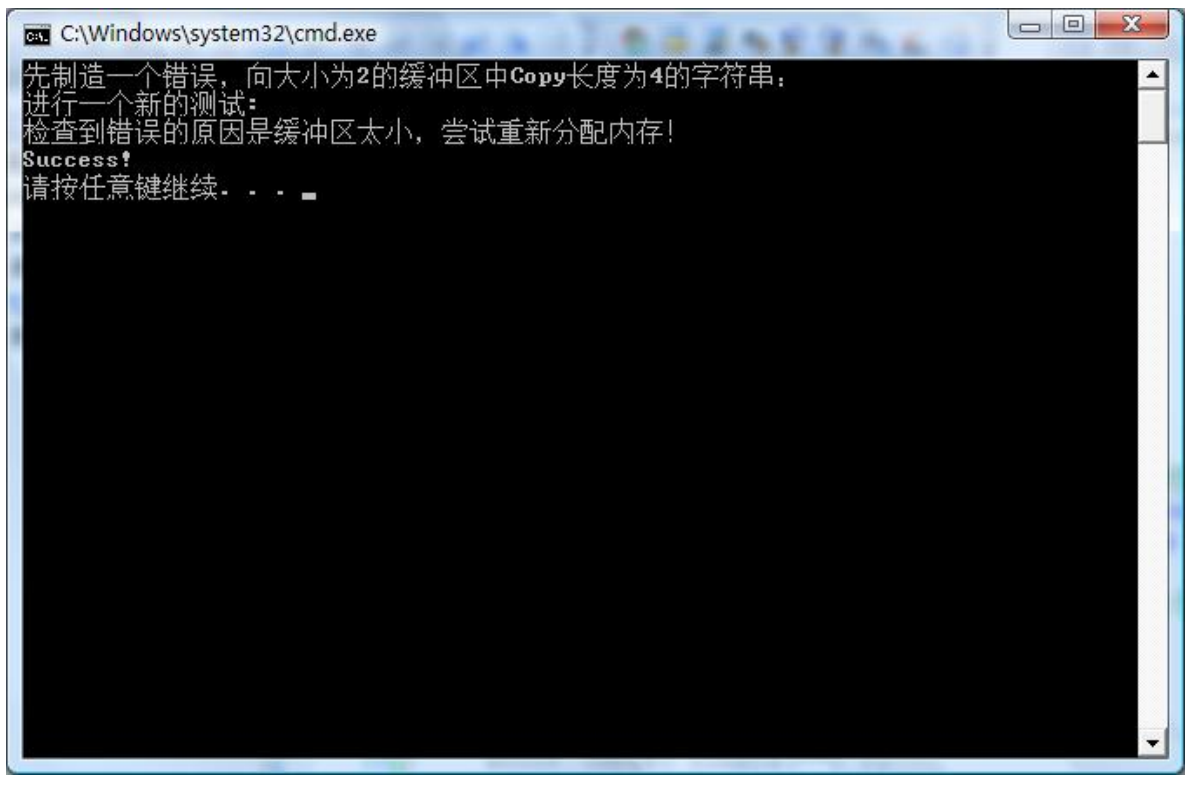
批注 [董波60]: 关于通用字符串函数会在后面做出介绍。

```
        cout<<"先制造一个错误，向大小为的缓冲区中Copy长度为的字符串："<< endl;
        TCHAR sz[2];
        _tcscpy_s( sz, _T("haha") );


        cout<<"进行一个新的测试:"<<endl;
        Test();

        return 0;
    }
```

这个例子是在 VS2008 上面编码和测试的，VS2005 上面没有做测试。Debug 输出：




Release 输出：




关于 C 字符串函数的更多内容请查阅 MSDN，搜索“String Manipulation”即可。

### 3.2.4 兼容

对于 VC2005 以前版本的 VC，由于没有相应的安全版本的字符串处理函数，因此我们应该对此进行包装以提升安全性，下面是我写的一个例子：

```
////////////////////////////////////////////////////////////////////////////
// 版权所有董波
////////////////////////////////////////////////////////////////////////////
// File :  Utility.h
```

```
// Author: Dongbo
// Date:   2009.1.14
// Description: 一些工具包
//////////////////////////////////////////////////////////////////////////

#ifndef _DB_UTILITY_H_
#define _DB_UTILITY_H_

// 包含运行配置头
#include "BuildConfig.h"

// 包含Noncopyable
#include "Noncopyable.h"

// 标准库的utility
#include <xutility>

// 一个宏
#if _MSC_VER > 1310
#   define dbsize_t      std::size_t
#   define dbptrdiff_t    std::ptrdiff_t
#else
#   define dbsize_t      size_t
#   define dbptrdiff_t    ptrdiff_t
#endif

#include <cassert>    // for assert
// 一个断言工具
#ifndef AssertEx
#define AssertEx( exp, msg ) assert( (exp) && (msg) )
#endif

#endif // #ifndef _DB_UTILITY_H_

//////////////////////////////////////////////////////////////////////////
// 版权所有董波
//////////////////////////////////////////////////////////////////////////
// File :  StringOP.h
// Author: Dongbo
// Date:   2009.1.15
// Description: 包装部分常用字符串函数
//////////////////////////////////////////////////////////////////////////

#ifndef _DB_STRINGOP_H_
#define _DB_STRINGOP_H_

#if !defined(_WIN32) || !defined(WIN32)
#error "此策略只适用于Windows平台"
#endif // #if !defined(_WIN32) || !defined(WIN32)

#include "Utility.h"
#include <stdio.h>    // for vnsprintf
#include <string.h>   // for string function
#include <stdarg.h>   // for valist va_start ...

namespace db
{
   // 字符串COPY
   inline char* StrCopy( char* pcDestination, unsigned uDestSize, const char* strSrouce )
   {
#if _MSC_VER > 1310
      char* ptmp = pcDestination;
      strcpy_s( pcDestination, uDestSize, strSrouce );
      return pcDestination;
#else
```

```
        AssertEx( 0!=pcDestination, "目标缓冲区不得为空" );
        AssertEx( 0!=strSrouce, "源字符串不得为空" );   // strlen不能接受NULL

        unsigned uSrcLen = strlen( strSrouce );
        AssertEx( uDestSize > uSrcLen , "缓冲区不够大" );

        unsigned uNeedCopy = ( (uDestSize <= uSrcLen) ? uDestSize : uSrcLen+1 );

        char* pcResult = strncpy( pcDestination, strSrouce, uNeedCopy );
        pcResult[ uDestSize-1 ] = '\0';

        return pcResult;
#endif
    }


    // 字符串添加
    inline char* StrCat( char* pcDest, unsigned uDestSize, const char* strSrouce )
    {
#if _MSC_VER > 1310
        char* ptmp = pcDest;
        strcat_s( pcDest, uDestSize, strSrouce );
        return ptmp;
#else
        // AssertEx( ( 0!=pcDest )&&( 0!=strSrouce ), "输入字串和缓冲区均不得为空"  );
        AssertEx( 0!=pcDest, "缓冲区不得为空" );
        AssertEx( 0!=strSrouce, "输入字符串不得为空" );

        unsigned uSrcLen = strlen( strSrouce );
        unsigned uDestLen = strlen( pcDest );

        AssertEx( uSrcLen + uDestLen < uDestSize, "缓冲区长度不够" );

        unsigned uNeedCopy = uDestSize - 1 - uDestLen;

        char* pcResult = strncat( pcDest, strSrouce, uNeedCopy );
        pcResult[ uDestSize-1 ] = '\0';

        return pcResult;
#endif
    }

    // 字符串格式化系列
    // V打头的用于不定参数...
   inline int StrVsnprintf( char* pcDest, unsigned uDestSize, unsigned uCount, const char* strFormat,
va_list kArgs )
    {
        if( 0 == uDestSize )
        {
            return 0;
        }

        AssertEx( 0!=pcDest, "缓冲区不得为空" );
        AssertEx( 0!=strFormat, "格式化字符串不得为空" );
        AssertEx( uCount < uDestSize || uCount == ( (unsigned)-1 ) , "缓冲区太小" );

        pcDest[0] = '\0';

        int iResult = 0;
        bool bTruncate = ( uCount == ((unsigned)-1) );
#if _MSC_VER > 1310
        iResult = vsnprintf_s( pcDest, uDestSize, uCount, strFormat, kArgs );
#else
        if( bTruncate )
        {
```

```
            uCount = uDestSize - 1;
        }
        iResult = _vsnprintf( pcDest, uCount, strFormat, kArgs );
#endif

        if( -1 == iResult && !bTruncate )
        {
            iResult = uCount;
        }

#if _MSC_VER <=1310
        if( -1 == iResult )
        {
            pcDest[ uDestSize - 1 ] = '\0';
        }
        else
        {
            pcDest[iResult] = '\0';
        }
#endif

        return iResult;
    }

    inline int StrVsprintf( char* pcDest, unsigned uDestSize, const char* strFormat, va_list kArgs )
    {
        return StrVsnprintf( pcDest, uDestSize, ((unsigned)-1), strFormat, kArgs );
    }

    inline int StrSnprintf( char* pcDest, unsigned uDestSize, unsigned uCount, const char*
strFormat, ... )
    {
        AssertEx( 0!= strFormat, "格式化字符串不得为空" );

        va_list kArgs;
        va_start( kArgs, strFormat );
        int iResult = StrVsnprintf( pcDest, uDestSize, uCount, strFormat, kArgs );
        va_end(kArgs);

        return iResult;
    }

    inline int StrSprintf( char* pcDest, unsigned uDestSize, const char* strFormat, ... )
    {
        AssertEx( 0!= strFormat, "格式化字符串不得为空" );

        va_list kArgs;
        va_start( kArgs, strFormat );
        int iResult = StrVsprintf( pcDest, uDestSize, strFormat, kArgs );
        va_end(kArgs);

        return iResult;
    }


    // 内存方面
    inline int Memcpy( void* pvDest, unsigned uDestSize, const void* pvSrc, unsigned uCount )
    {
        int iResult = -1;
#if _MSC_VER > 1310
        iResult = memcpy_s( pvDest, uDestSize, pvSrc, uCount );
#else
        if( uDestSize < uCount )
        {
            iResult = -1;
```

```
        }
        else
        {
            memcpy( pvDest, pvSrc, uCount );
        }
#endif // #if _MSC_VER > 1310

        AssertEx( -1 == iResult, "内存Copy失败! " );

        return iResult;
    }
//-------------------------------------- wchar_t
------------------------------------------------------
//
//
    //
    // 字符串Copy的宽字符串版本
    inline wchar_t* StrCopy( wchar_t* pcDestination, unsigned uDestSize, const wchar_t* strSrouce )
    {
#if _MSC_VER > 1310
        wchar_t* ptmp = pcDestination;
        wcscpy_s( pcDestination, uDestSize, strSrouce );
        return pcDestination;
#else
        AssertEx( 0!=pcDestination, "目标缓冲区不得为空" );
        AssertEx( 0!=strSrouce, "源字符串不得为空" );   // strlen不能接受NULL

        unsigned uSrcLen = wcslen( strSrouce );
        AssertEx( uDestSize > uSrcLen , "缓冲区不够大" );

        unsigned uNeedCopy = ( (uDestSize <= uSrcLen) ? uDestSize : uSrcLen+1 );

        wchar_t* pcResult = wcsncpy( pcDestination, strSrouce, uNeedCopy );
        pcResult[ uDestSize-1 ] = '\0';

        return pcResult;
#endif
    }

    // 追加的宽字符串版本
    inline wchar_t* StrCat( wchar_t* pcDest, unsigned uDestSize, const wchar_t* strSrouce )
    {
#if _MSC_VER > 1310
        wchar_t* ptmp = pcDest;
        wcscat_s( pcDest, uDestSize, strSrouce );
        return ptmp;
#else
        // AssertEx( ( 0!=pcDest )&&( 0!=strSrouce ), "输入字串和缓冲区均不得为空"  );
        AssertEx( 0!=pcDest, "缓冲区不得为空" );
        AssertEx( 0!=strSrouce, "输入字符串不得为空" );

        unsigned uSrcLen = wcslen( strSrouce );
        unsigned uDestLen = wcslen( pcDest );

        AssertEx( uSrcLen + uDestLen < uDestSize, "缓冲区长度不够" );

        unsigned uNeedCopy = uDestSize - 1 - uDestLen;

        wchar_t* pcResult = wcsncat( pcDest, strSrouce, uNeedCopy );
        pcResult[ uDestSize-1 ] = '\0';

        return pcResult;
#endif
    }
```

```
        // 字符串格式化系列
        // V打头的用于不定参数...
        inline int StrVsnprintf( wchar_t* pcDest, unsigned uDestSize, unsigned uCount, const wchar_t*
    strFormat, va_list kArgs )
        {
            if( 0 == uDestSize )
            {
                return 0;
            }

            AssertEx( 0!=pcDest, "缓冲区不得为空" );
            AssertEx( 0!=strFormat, "格式化字符串不得为空" );
            AssertEx( uCount < uDestSize || uCount == ( (unsigned)-1 ) , "缓冲区太小" );

            pcDest[0] = '\0';

            int iResult = 0;
            bool bTruncate = ( uCount == ((unsigned)-1) );
    #if _MSC_VER > 1310
            iResult = _vsnwprintf_s( pcDest, uDestSize, uCount, strFormat, kArgs );
    #else
            if( bTruncate )
            {
                uCount = uDestSize - 1;
            }
            iResult = _vsnwprintf( pcDest, uCount, strFormat, kArgs );
    #endif

            if( -1 == iResult && !bTruncate )
            {
                iResult = uCount;
            }

    #if _MSC_VER <=1310
            if( -1 == iResult )
            {
                pcDest[ uDestSize - 1 ] = L'\0';
            }
            else
            {
                pcDest[iResult] = L'\0';
            }
    #endif

            return iResult;
        }

        inline int StrVsprintf( wchar_t* pcDest, unsigned uDestSize, const wchar_t* strFormat, va_list
    kArgs )
        {
            return StrVsnprintf( pcDest, uDestSize, ((unsigned)-1), strFormat, kArgs );
        }

        inline int StrSnprintf( wchar_t* pcDest, unsigned uDestSize, unsigned uCount, const wchar_t*
    strFormat, ... )
        {
            AssertEx( 0!= strFormat, "格式化字符串不得为空" );

            va_list kArgs;
            va_start( kArgs, strFormat );
            int iResult = StrVsnprintf( pcDest, uDestSize, uCount, strFormat, kArgs );
            va_end(kArgs);

            return iResult;
        }
```

```
    inline int StrSprintf( wchar_t* pcDest, unsigned uDestSize, const wchar_t* strFormat, ... )
    {
        AssertEx( 0!= strFormat, "格式化字符串不得为空" );

        va_list kArgs;
        va_start( kArgs, strFormat );
        int iResult = StrVsprintf( pcDest, uDestSize, strFormat, kArgs );
        va_end(kArgs);

        return iResult;
    }

}

#endif // #ifndef _DB_STRINGOP_H_
```

本人在 VC2008 和 VC6 下面都做了测试，都没有问题。这里只是一些最常用的函数的包装，其它的还没有做。

## 3.3 通用字串函数

### 3.3.1 简述

在 c++开发中我们常用的字符集是以 char 为代表的传统单字节字符集和以 wchar_t 为代表的宽字符集。对于每一个字符集，都有相应的字符串处理函数。其中单字符集通常以 str 打头，宽字符集以 wcs 打头。而对于 sprintf_s 这样的函数，对应的宽字符集版本通常多了一个 w，比如 sprintf_s 对应的 swprintf_s。为了在 wchar_t 和 char 之间灵活的转换，VS 为我们提供了一个 TCHAR，这是一个 typedef：

批注 [董波61]: 特别注意 swprintf/swprintf_s 和 wsprintf 的区别。前者属于 CRT，后者属于系统 API。关于 API 级的字符串处理函数在后面的章节会提到。

```
#if defined(UNICODE)||defined(_UNICODE)
typedef wchar_t TCHAR;
#else
typedef char TCHAR;
#endif
```

这样，TCHAR 就可以根据我们所配置的环境自动的被替换成 wchar_t 或者是 char。而对于我们前面所提到的字串函数，VS 也提供了相应的 typedef。这些 typedef 的特点就是几乎都是以_tcs 开头，比如说 strcpy_s/wcscpy_s 对应的_tcscpy_s，strcat_s/wcscat_s 对应的_tcscat_s。为了得到这些支持，必须包含头文件 TCHAR.h。而且值得一提的是这些支持不仅仅局限于字符串处理函数，对于其它的 C 运行库函数都有相应的 typedef。比如_fopen_s/_wfopen_s 对应的_tfopen_s，它们的特点就是几乎都以_t 开头，又比如：_tprintf_s。

### 3.3.2 简单实例

这是一个简单的例子，其它的用法也 以此类推：

```
#include <iostream>

using namespace std;

#include <tchar.h>

int _tmain()
{
    TCHAR tszBuf[100];
    _tcscpy_s( tszBuf, _T("哇咔咔") ); // 使用_T可以对常量字符串进行自动切换

#if defined(UNICODE)||defined(_UNICODE)
    setlocale( LC_CTYPE, "chs" );  // 在简体中文的操作系统上需要设置Locale才能支持中文
    // 当然是在Unicode环境下才有必要做这个动作// 也可以使用_tprintf_s_l来替换
#endif

    _tprintf_s( tszBuf );

    _stprintf_s( tszBuf, _T("%s %d %d %s"), _T("哈哈"), 100, -1, _T("加油！") );

    _tprintf_s( _T("\n%s\n"), tszBuf );

#if defined(UNICODE)||defined(_UNICODE)
    setlocale( LC_CTYPE, "" );  // 还原
#endif
```

```
        return 0;
    }
```

### 3.3.3 映射表

见附录 4。

## 3.4 API 级的字符串处理

### 3.4.1 简述

Windows 为我们提供了一些 API 级别的字符串处理函数，由于 Windows 系统软件在需要字符串处理的时候使用的是这些 API，因此，我们使用 WindowsAPI 来进行字符串操作通常会具有更高的效率，因为这些 API 存在于内存当中的可能性大大增加，减少了运行时才从 C 运行库的 dll 中导出的开销。但是尽管如此我们也不推荐再继续使用这些旧的字串 API，原因就是安全性问题。这些旧的 API 通常以 lstr 开头，比如 lstrcpy、lstrcat 等等。要使用这些 API 通常要包含 Windows.h 头文件。VC2005 以后，微软又为我们提供了新的更安全的字符串处理函数，比如 StringCchCopy 等等，但是我们还是不推荐使用这一系列的函数，原因就是效率问题，除非你想对你的字符串操作实施更加严格和广泛的控制。要使用这类函数需要包含头文件 strsafe.h。

### 3.4.2 旧的 API

```
LPTSTR lstrcat(
    LPTSTR lpString1,
    LPTSTR lpString2
    );
```

追加字符串。

```
int lstrcmp(
    LPCTSTR lpString1,
    LPCTSTR lpString2
    );
```

比较字符串。

```
int lstrcmpi(
    LPCTSTR lpString1,
    LPCTSTR lpString2
    );
```

不区分大小写的比较。

```
LPTSTR lstrcpy(
    LPTSTR lpString1,
    LPTSTR lpString2
    );
```

字符串拷贝。

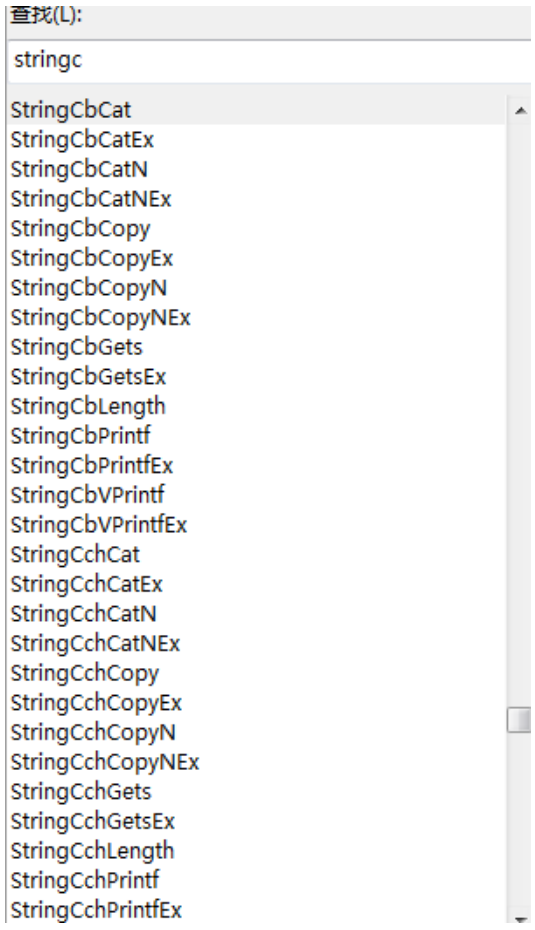
上面罗列的是一些常见的旧的字符串 API，更多内容请自行查阅 MSDN。

### 3.4.3 Shell 字符串函数

微软为我们提供了一些简单好用的 Shell 字符串处理函数，使用这些字符串函数，我们可以更简单方便的对系统相关的一些字符串进行处理。还可以对操作系统相关的数值进行格式化操作，比如 StrFormatKBSize 和 StrFormatKBSize。要使用这些 API 函数需要包含头文件 ShlWApi.h。更多相关内容请查阅：http://msdn.microsoft.com/en-us/library/bb759983.aspx

### 3.4.4 新的安全版字符串处理 API

要使用这些新版的 API 需要包含头文件 strsafe.h。这类字符串处理 API 通常以 String 开头，下面是在 MSDN 中索引得到的：




我们可以看到它们主要分两类:Cb 类和 Cch 类。主要区别在于个数提供的方式,前者是比特,后者是个数,我们通常可以用 sizeof 获取前者，用_countof 获取后者，原生指针除外。

与 Windows 的其它 API 一样，您现在看到的不是真正的 API，而是一个宏。其实它们都有两个版本，一个 A 版本适用于 char，一个 W 版本适用于 wchar_t，比如：

```
HRESULT StringCbCat(
     LPTSTR pszDest,
     size_t cbDest,
     LPCTSTR pszSrc
      );
```

其关系如下：

| String data type | String literal | Function |
|---|---|---|
| char | "string" | StringCbCatA |
| TCHAR | TEXT("string") | StringCbCat |
| WCHAR | L"string" | StringCbCatW |

通常这些 API 还有一个扩展版本：

```
HRESULT StringCbCatEx(
     LPTSTR pszDest,
     size_t cbDest,
     LPCTSTR pszSrc,
     LPTSTR *ppszDestEnd,
     size_t *pcbRemaining,
     DWORD dwFlags
     );
```

这些版本有更多的参数，可以提供更多的控制。下面是一个使用新字符串 API 的例子，其作用是获取并打印所有的环境变量：

```
#include <iostream>
#include <windows.h>
#include <tchar.h>
#include <strsafe.h>
using namespace std;
int main()
{
  // 设置现场以支持中文
  setlocale( LC_CTYPE, "chs" );
  PTSTR pEnvBlock = GetEnvironmentStrings();
  // 检查返回值
```

```
        if( NULL == pEnvBlock )
        {
            _tprintf_s( _T("获取失败\n") );
            return -1;
        }
        // 使用SEH来提升安全性
        __try
        {
            TCHAR szName[MAX_PATH]={0};
            TCHAR szValue[MAX_PATH<<3]={0};
            PTSTR pszCurrent = pEnvBlock;
            HRESULT hr = S_OK;
            PCTSTR pszPos = NULL;
            int current = 0;
            while( pszCurrent != NULL )
            {
                if( *pszCurrent != _T('=') )
                {
                    pszPos = _tcschr( pszCurrent, _T('=') );
                    ++pszPos;
                    size_t cbNameLength = (size_t)pszPos - (size_t)pszCurrent - sizeof(TCHAR);
                    hr = StringCbCopyN( szName, MAX_PATH, pszCurrent, cbNameLength );
                    if( FAILED(hr) )
                    {
                        break;
                    }
                    hr = StringCchCopyN( szValue, MAX_PATH<<3, pszPos, _tcslen(pszPos)+1 );
                    if( SUCCEEDED(hr) )
                    {
                        _tprintf( _T("[%u] %s=%s\r\n"), current, szName, szValue );
                    }
                    else
                    {
                        if( hr == STRSAFE_E_INSUFFICIENT_BUFFER )
                        {   // 这当缓冲区不够大的时候发生!
                            // 但是这样输出是安全的，因为安全版的字符串函数会自动在缓冲区末尾添加
                            _tprintf( _T("[%u] %s=%s\r\n"), current, szName, szValue );
                        }
                        else
                        {
                            _tprintf( _T("[%u] %s=???\r\n"), current, szName );
                        }
                        break;
                    }
                }
                else
                {
                    _tprintf( _T("[%u] %s\r\n"), current, szName );
                }
                ++current;
                while( *pszCurrent != _T('\0') )
                {
                    ++pszCurrent;
                }
                ++pszCurrent;
                if( *pszCurrent == _T('\0') )
                {
                    break;
                }
            }
        }
        __finally
        {
            FreeEnvironmentStrings( pEnvBlock );
            setlocale( LC_CTYPE,"" );
```

```
        }

        return 0;
    }
```

与新版 API 更多的内容请查看：http://msdn.microsoft.com/en-us/library/ms647466.aspx

# 四、 C++字符串使用的建议

以下内容由我个人总结，如有不当之处还请原谅。

1. 优先选择 C++标准库的 string 和 wstring[VC6 的字符串实现使用了引用计数，在多线程下会出现很多问题，故除外]。优先选择 boost 算法库。
2. 在 MFC 环境下优先选择 CString。
3. 对于类 C 字符串处理来说，优先选择通用版字符串处理函数。
4. 对于类 C 字符串处理来说，优先选择安全版的字符串，即_s 版本而非 strsafe 版本。
5. 当需要跨平台的时候请不要使用与 Windows 平台特定相关的函数，包括安全版字符串函数、lstr 系列、strsafe 等等。
6. 当需要跨平台的时候优先选择 C 库和 C++标准库的函数而不是使用条件编译来处理，跟不应该是系统 API。
7. 当你的代码只想在 Windows 下面跑的话对一些情况可以考虑使用 Shell 版本的 API，因为它们非常快。
8. 如果在 Windows 平台下，你想要对越界的字符串进行截断，那么你需要 选择 strsafe 字符串处理 API。
9. 对于团队版的 VS，可以考虑使用/GS 和/RTC 编译器标识来自动检测缓冲区溢出。
10. lstr 系列的 Kernel32 方法应该被彻底抛弃。
11. 对于特殊的应用应该优先选择已经存在的 API，而不是自己写分析程序。比如对于文件路径、XML 元素/属性以及注册表项/值的比较应该选择 CompareStringOrdinal。
12. 如果您还在 VC2005 以前的版本编程，建议您对您的字符串操作函数进行包装。

# 附录 1：参考资料：


1. MSSTL
2. boost 1.37
3. 《泛型编程与 STL》
4. 《C++ STL 中文版》
5. boost 1.36 文档
6. 《Effective STL》
7. 《Windows Via C&C++ Fifth Edition》


# 附录 2： MSSTL 中 basic_string 的部分源码解读

## 2.1 纵览

在 STL 中，我们使用的字符串通常有两个：std::string 和 std::wstring，它们分别是 char 和 wchar_t 的对 std::basic_string 的特化:

```
template class _CRTIMP2_PURE basic_string<char, char_traits<char>,
    allocator<char> >;
template class _CRTIMP2_PURE basic_string<wchar_t, char_traits<wchar_t>,
    allocator<wchar_t> >;
```

也就是说，我们可以自己实例化basic_string，假设我们有另外一种字符叫cchar，那么我们可以这样：

```
template class basic_string< cchar,char_traits<cchar>,allocator<cchar> >;
typedef basic_string<cchar, char_traits<cchar>, allocator<cchar> >    cstring;
```

如果你能保证你的cchar能够有与char有类似的语意的话就可以直接使用cstring了。但是即便是如此，我们的cstring用起来也不会很方便，因为MSSTL不仅仅为我们实现了string，还对string与iostream的结合方面做了很多的工作。

## 2.1 string 的 allocator

### 2.1.1 Allocate 和 Deallocate

allocator 即空间配置器。MSSTL 的 allocator 实现位于头文件 xmemory 中，虽然我们在应用中通常不需要用到这个东东，但是还是有必要提到它的。对于 string，我们的重点将集中在与 char、wchar_t 相关的地方。

我们都知道，因为 allocator 的存在，我们可以将分配内存和初始化这两个步骤分开，而实现这种效果的方法就是分别实现 Allocate 和 Construct，MSSTL 的实现如下：

```
template<class _Ty> inline
    _Ty _FARQ *_Allocate(_SIZT _Count, _Ty _FARQ *)
    {
    if (_Count <= 0)
```

批注 [董波62]: 指 std::badic_string，其实现位于头文件 xstring 中。【仅针对 MSSTL】

批注 [董波63]: 稍后我们会发现 char 和 wchar_t 的 traits 都分别做了特化，这样可以提高效率。至少 MSSTL 是这样的。这里的 cchar 使用的是默认的，效率比较低下。

批注 [董波64]: 《STL 源码剖析》 侯捷

批注 [董波65]: 我的观察重点将集中在我们所需要的部分，而微软提供的安全检查或者是类似的东西将被我忽略。

```
            _Count = 0;
         else if (((_SIZT)(-1) / _Count) < sizeof (_Ty))
            _THROW_NCEE(std::bad_alloc, NULL);

            // allocate storage for _Count elements of type _Ty
         return ((_Ty _FARQ *)::operator new(_Count * sizeof (_Ty)));
         }

            // TEMPLATE FUNCTION _Construct
      template<class _T1,
         class _T2> inline
         void _Construct(_T1 _FARQ *_Ptr, const _T2& _Val)
         {  // construct object at _Ptr with value _Val
         void _FARQ *_Vptr = _Ptr;
         ::new (_Vptr) _T1(_Val);
         }

            // TEMPLATE FUNCTION _Destroy
         template<class _Ty> inline
            void _Destroy(_Ty _FARQ *_Ptr)
            {  // destroy object at _Ptr
            _DESTRUCTOR(_Ty, _Ptr);
            }
```

在这段代码中，_Allocate的参数_Ty _FARQ *并没有实际上的意义，这需要注意。整个模板函数的用途就是分配_Count个_Ty的内存空间，不考虑初始化的问题。_Construct使用了placement new，就是在指定内存的位置上使用_Val来构建_T1对象。而_Destroy就是析构对象，但是并不释放内存。_DESTRUCTOR是这样的一个宏：

```
#define _DESTRUCTOR(ty, ptr)(ptr)->~ty()
```

也就是说，它只是调用了对象的析构函数罢了。

按照习惯我们会实现一个_Deallocator来释放内存，不知道为什么微软没有这么做。

然后微软提供了针对_Destroy 的 char 和 wchar_t 的特化：

```
template<> inline
void _Destroy(char _FARQ *)
{  // destroy a char (do nothing)
}

template<> inline
   void _Destroy(wchar_t _FARQ *)
   {  // destroy a wchar_t (do nothing)
   }
```

为什么要这样？设想一下如果对char指针调用~char()会怎样？

### 2.1.2 allocator 的泛型实现

```
template<class _Ty>
struct _Allocator_base
{  // base class for generic allocators
typedef _Ty value_type;
};

   // TEMPLATE CLASS _Allocator_base<const _Ty>
template<class _Ty>
struct _Allocator_base<const _Ty>
{  // base class for generic allocators for const _Ty
typedef _Ty value_type;
};

template<class _Ty>
class allocator
   : public _Allocator_base<_Ty>
{  // generic allocator for objects of class _Ty
public:
   typedef _Allocator_base<_Ty> _Mybase;
   typedef typename _Mybase::value_type value_type;
   typedef value_type _FARQ *pointer;
   typedef value_type _FARQ& reference;
```

**批注 [董波66]:** 位于头文件 xstddef 中。

**批注 [董波67]:** 微软不这么做不代表这有什么问题，接下来就会看到微软的处理方式，那样也是不会有什么问题的。

如果要提供一个的话，我们可以这么写：

```
template< class _Ty >
inline void _Deallocator( _Ty * _Ptr )
{
    ::operator delete(_Ptr);
}
```

**批注 [董波68]:** 为甚要提供一个const _Ty 的偏特化呢？

因为我们的value_type表达的一定是“值”的含义，而如果没有这个偏特化，那么我们的 value_type 将对 const _Ty 失去作用，无法得到真正的“值”。

**批注 [董波69]:** 这里要说一下偏特化的概念：即对 template 参数施以更加严格的限制而实现的一个特化版本，而且其本身依然是一个 templatized。

**批注 [董波70]:** 为什么要从这里派生？

根据前面一个批注，因为 value_type 的问题，如果我们不从 _Allocator_base 派生，那么我们必须对 allocator 提供 const _Ty 的偏特化，你说这样是不是很麻烦呢？

这样派生之后一切都会变得很简单！

```
typedef const value_type _FARQ *const_pointer;
typedef const value_type _FARQ& const_reference;

typedef _SIZT size_type;
typedef _PDFT difference_type;

template<class _Other>
   struct rebind
   {  // convert an allocator<_Ty> to an allocator <_Other>
   typedef allocator<_Other> other;
   };

pointer address(reference _Val) const
   {  // return address of mutable _Val
   return (&_Val);
   }

const_pointer address(const_reference _Val) const
   {  // return address of nonmutable _Val
   return (&_Val);
   }

allocator() _THROW0()
   {  // construct default allocator (do nothing)
   }

allocator(const allocator<_Ty>&) _THROW0()
   {  // construct by copying (do nothing)
   }

template<class _Other>
   allocator(const allocator<_Other>&) _THROW0()
   {  // construct from a related allocator (do nothing)
   }

template<class _Other>
   allocator<_Ty>& operator=(const allocator<_Other>&)
   {  // assign from a related allocator (do nothing)
   return (*this);
   }

void deallocate(pointer _Ptr, size_type)
   {  // deallocate object at _Ptr, ignore size
   ::operator delete(_Ptr);
   }

pointer allocate(size_type _Count)
   {  // allocate array of _Count elements
   return (_Allocate(_Count, (pointer)0));
   }

pointer allocate(size_type _Count, const void _FARQ *)
   {  // allocate array of _Count elements, ignore hint
   return (allocate(_Count));
   }

void construct(pointer _Ptr, const _Ty& _Val)
   {  // construct object at _Ptr with value _Val
   _Construct(_Ptr, _Val);
   }

void destroy(pointer _Ptr)
   {  // destroy object at _Ptr
   _Destroy(_Ptr);
   }
```

**批注 [董波71]:** 这是一个宏，与它类似的定义有这些:

```
#define _THROW0() throw ()
#define _THROW1(x) throw (...)
#define _THROW(x, y) throw x(y)
#define _THROW_NCEE(x, y) _THROW(x, y)
```

throw()表示一定不会抛出异常
throw(…)表示一定会抛出异常
throw( exp )这样的表示可能会抛出一个 exp 异常，如果抛出的不是 exp 异常，那么程序会直接被 kill 掉。
它们的定义位于 xstddef 中。

```
    _SIZT max_size() const _THROW0()
        {  // estimate maximum array size
        _SIZT _Count = (_SIZT)(-1) / sizeof (_Ty);
        return (0 < _Count ? _Count : 1);
        }
    };
```

类开始是一些类型的typedef，这是C++标准所要求的，你必须提供这样的typedef，因为你需要和其它容器以及其它泛型算法进行交互。rebind结构体是一个必须要定义的结构体，你必须这么做。

另外一个值得注意的是 void deallocate(pointer _Ptr, size_type)。如果我们有实现_Deallocator 函数的话，那么这个函数可能的写法是这样的：

```
void deallocate(pointer _Ptr, size_type)
{
    _Deallocator( _Ptr );
}
```

前面说道了微软没有实现并没有实现它并没有什么影响，事实证明微软只不过是换汤不换药罢了。

接下来是对 allocator 重载了==和!=：

```
template<class _Ty,
    class _Other> inline
    bool operator==(const allocator<_Ty>&, const allocator<_Other>&) _THROW0()
    {  // test for allocator equality (always true)
    return (true);
    }

template<class _Ty,
    class _Other> inline
    bool operator!=(const allocator<_Ty>&, const allocator<_Other>&) _THROW0()
    {  // test for allocator inequality (always false)
    return (false);
    }
```

不要对这种处理方式感到奇怪，因为这是 C++标准要求的。

在头文件 xmemory 中，接下来 MSSTL 对 allocator 进行了对 void 的特化，因为如果不特化的话，模板类的一些操作对于 void 来说是没有意义的。

## 2.1.3 string 与 char_traits

### 2.3.1 traits

traits 编程技法是 STL 的一个显著的特征，通过 traits 我们可以轻松的得到必须要得到的内嵌声明 value_type、reference、pointer 等等。

### 2.3.2 char_traits 的泛型实现

MSSTL 的 char_traits 定义于头文件 iosfwd 中。

```
struct _Char_traits_base
{
    typedef _Unsecure_char_traits_tag _Secure_char_traits;
};
```

首先为我们定义了一个这样的 tag 结构体，它仅仅是在定义 char_traits 的时候起到了作用，如果我们需要自己写一个自己的 char_traits 的话也必须从这个结构体继承。

```
template<class _Elem>
struct char_traits:
    public _Char_traits_base
{  // properties of a string or stream element
typedef _Elem char_type;
typedef long int_type;
typedef streampos pos_type;
typedef streamoff off_type;
typedef _Mbstatet state_type;

static void __CLRCALL_OR_CDECL assign(_Elem& _Left, const _Elem& _Right)
    {  // 其实就是对一个_Elem赋值
    _Left = _Right;
    }

static bool __CLRCALL_OR_CDECL eq(const _Elem& _Left, const _Elem& _Right)
    {  // 判断是否相等
```

批注 [董波72]: 如果你想问为什么请参看《Effective STL》条款 10：注意分配器的协定和约束

批注 [董波73]: 非常抱歉的是这一点，因为我也忘了我是在哪儿看到的了，呵呵。

批注 [董波74]: 关于 traits 的具体内容可以参看《STL 源码剖析》 侯捷

批注 [董波75]: 微软原话：
TEMPLATE STRUCT _Char_traits_base
Used to define the Secure_char_traits tag. This typedef is used only for user defined char_traits. A user defined char_traits should inherit from _Char_traits_base, define _Secure_char_traits as _Secure_char_traits_tag and implement _Copy_s and _Move_s.

批注 [董波76]: 因为这是 char_traits，因为没有 value_type，有的是 char_type，但是这并不是问题的关键。

批注 [董波77]: 微软总是提供各种各样的这些宏来“扰乱”我们，而且在不同的环境下，这些宏总是会有不同的语意，我选择的是基本 Debug 环境，操作系统是 WINDOWS Vista sp1 Ultimate。
比如在这种环境下
__CLRCALL_OR_CDECL 其实就是 __cdecl
#ifndef __CLRCALL_OR_CDECL
#if defined(MRTDLL) || defined(_M_CEE_PURE)
#define __CLRCALL_OR_CDECL __clrcall
#else
#define __CLRCALL_OR_CDECL __cdecl
#endif
#endif
在分析源码的时候，除非必要，否则我会直接忽略这些宏。

```
        return (_Left == _Right);
        }

    static bool __CLRCALL_OR_CDECL lt(const _Elem& _Left, const _Elem& _Right)
        {  // 判断小于关系
        return (_Left < _Right);
        }

    static int __CLRCALL_OR_CDECL compare(_In_count_(_Count) const _Elem *_First1,
        _In_count_(_Count) const _Elem *_First2, size_t _Count)
        {  // 比较_Count个字符串，按照字典顺序，小于返回-1，等于返回0，大于返回1
        for (; 0 < _Count; --_Count, ++_First1, ++_First2)
            if (!eq(*_First1, *_First2))
                return (lt(*_First1, *_First2) ? -1 : +1);
        return (0);
        }

    static size_t __CLRCALL_OR_CDECL length(_In_z_ const _Elem *_First)
        {  // 得到长度 _First必须以_Elem默认构造的值结束，否则无法得到正确的结果。
        size_t _Count;
        for (_Count = 0; !eq(*_First, _Elem()); ++_First)
            ++_Count;
        return (_Count);
        }

    _SCL_INSECURE_DEPRECATE
    static _Elem *__CLRCALL_OR_CDECL copy(_Out_cap_(_Count) _Elem *_First1,
        _In_count_(_Count) const _Elem *_First2, size_t _Count)
        {  // 拷贝_Count个_First2元素到_First1指向的缓冲区中，这里对缓冲区大小做了假定。
        return _Copy_s(_First1, _Count, _First2, _Count);
        }

    static _Elem *__CLRCALL_OR_CDECL _Copy_s(_Out_cap_(_Dest_size) _Elem *_First1, size_t
_Dest_size,
        _In_count_(_Count) const _Elem *_First2, size_t _Count)
        {  // 这是对拷贝行为提供的安全保证
        _SCL_SECURE_CRT_VALIDATE(_Dest_size >= _Count, NULL);
        _Elem *_Next = _First1;
        for (; 0 < _Count; --_Count, ++_Next, ++_First2)
            assign(*_Next, *_First2);
        return (_First1);
        }

    static const _Elem *__CLRCALL_OR_CDECL find(_In_count_(_Count) const _Elem *_First,
        size_t _Count, const _Elem& _Ch)
        {  // 查找字符，找不到的时候返回的0，注意不是最后一个元素的后面!
        for (; 0 < _Count; --_Count, ++_First)
            if (eq(*_First, _Ch))
                return (_First);
        return (0);
        }

    _SCL_INSECURE_DEPRECATE
    static _Elem *__CLRCALL_OR_CDECL move(_Out_cap_(_Count) _Elem *_First1,
        _In_count_(_Count) const _Elem *_First2, size_t _Count)
        {  //
        return _Move_s(_First1, _Count, _First2, _Count);
        }

    static _Elem *__CLRCALL_OR_CDECL _Move_s(_Out_cap_(_Dest_size) _Elem *_First1, size_t
_Dest_size,
        _In_count_(_Count) const _Elem *_First2, size_t _Count)
        {  // move [_First1, _First1 + _Count) to [_First2, ...)
        _SCL_SECURE_CRT_VALIDATE(_Dest_size >= _Count, NULL);
        _Elem *_Next = _First1;
```

批注 [董波78]: 在使用 STL 的时候我们经常会遇到这样的错误：CRT 检测失败，因为访问了已经删除了的元素。

```
        if (_First2 < _Next && _Next < _First2 + _Count)
            for (_Next += _Count, _First2 += _Count; 0 < _Count; --_Count)
                assign(*--_Next, *--_First2);
        else
            for (; 0 < _Count; --_Count, ++_Next, ++_First2)
                assign(*_Next, *_First2);
        return (_First1);
        }

    static _Elem *__CLRCALL_OR_CDECL assign(_Out_cap_(_Count) _Elem *_First,
        size_t _Count, _Elem _Ch)
        {  // assign _Count * _Ch to [_First, ...) 把_First从开始起的_Count个字符设置成_Ch
        _Elem *_Next = _First;
        for (; 0 < _Count; --_Count, ++_Next)
            assign(*_Next, _Ch);
        return (_First);
        }

    static _Elem __CLRCALL_OR_CDECL to_char_type(const int_type& _Meta)
        {  // 下面的就是涉及到类型转换什么的了，就不多说了。
        return ((_Elem)_Meta);
        }

    static int_type __CLRCALL_OR_CDECL to_int_type(const _Elem& _Ch)
        {  // convert character to metacharacter
        return ((int_type)_Ch);
        }

    static bool __CLRCALL_OR_CDECL eq_int_type(const int_type& _Left,
        const int_type& _Right)
        {  // test for metacharacter equality
        return (_Left == _Right);
        }

    static int_type __CLRCALL_OR_CDECL eof()
        {  // return end-of-file metacharacter
        return ((int_type)EOF);
        }

    static int_type __CLRCALL_OR_CDECL not_eof(const int_type& _Meta)
        {  // return anything but EOF
        return (_Meta != eof() ? (int_type)_Meta : (int_type)!eof());
        }
    };
```

批注 [董波79]: 它这里考虑了两个内存部分可能发生的重叠的情况。

### 2.1.4 以 char 和 wchar_t 特化 char_traits

特化它们的原因主要是为了效率上的考虑。上面的 char_traits 是通用版本，考虑的是通用性，而 char 和 wchar_t 都是内置的类型，有很多对它们加速的方法，因此需要特化它们。

```
    template<> struct _CRTIMP2_PURE char_traits<char>:
    public _Char_traits_base
    {  // properties of a string or stream char element
    typedef char _Elem;
    typedef _Elem char_type;
    typedef int int_type;
    typedef streampos pos_type;
    typedef streamoff off_type;
    typedef _Mbstatet state_type;
  // 删除相同或者类似的部分
    static int __CLRCALL_OR_CDECL compare(_In_count_(_Count) const _Elem *_First1,
  _In_count_(_Count) const _Elem *_First2,
        size_t _Count)
        {
        return (::memcmp(_First1, _First2, _Count));
        }
```

批注 [董波80]: 为了节省篇幅，我删除相同或者相似的代码。

```
    static size_t __CLRCALL_OR_CDECL length(_In_z_ const _Elem *_First)
        {
        return (::strlen(_First));
        }
    static _Elem *__CLRCALL_OR_CDECL _Copy_s(_Out_cap_(_Size_in_bytes) _Elem *_First1, size_t
_Size_in_bytes, _In_count_(_Count) const _Elem *_First2,
        size_t _Count)
        {
        _CRT_SECURE_MEMCPY(_First1, _Size_in_bytes, _First2, _Count);
        return _First1;
        }

    static const _Elem *__CLRCALL_OR_CDECL find(_In_count_(_Count) const _Elem *_First, size_t
_Count,
        const _Elem& _Ch)
        {
        return ((const _Elem *)::memchr(_First, _Ch, _Count));
        }

    static _Elem *__CLRCALL_OR_CDECL _Move_s(_Out_cap_(_Size_in_bytes) _Elem *_First1, size_t
_Size_in_bytes, _In_count_(_Count) const _Elem *_First2,
        size_t _Count)
        {
        _CRT_SECURE_MEMMOVE(_First1, _Size_in_bytes, _First2, _Count);
        return _First1;
        }

    static _Elem *__CLRCALL_OR_CDECL assign(_Out_cap_(_Count) _Elem *_First, size_t _Count, _Elem
_Ch)
        {  // assign _Count * _Ch to [_First, ...)
//      _DEBUG_POINTER(_First);
        return ((_Elem *)::memset(_First, _Ch, _Count));
        }
    };
```

基本上就是使用 C 库的一些函数来代替了通用版本。wchar_t 的特化版本和 char 的也是非常的类似，差别在于调用的是处理 wchar_t 的库函数罢了，因此这里就不罗列了。

# 附录 3：Boost.Format 中文文档

## 2.1 大纲

一个 format 对象从一个格式化字符串构造，它以重复的%操作符给出参数。接着每个参数转换成字符串，它们被按照格式合成一个字符串。

```
cout << boost::format("writing %1%,  x=%2% : %3%-th try") % "toto" % 40.23 % 50;
// prints "writing toto,  x=40.230 : 50-th try"
```

## 2.2 它是如何工作的

当你调用 format(s)，这里 s 是一个用于格式化的字符串。它从格式化字符串中分析并且查找里面的所有指示并且为下一步准备内部结构。

接着，要么马上，像

```
cout << format("%2% %1%") % 36 % 77 )
```

或者迟些，像

```
format fmter("%2% %1%");fmter % 36; fmter % 77;
```

你将变量传送给格式化器。

这些变量被存进一个内部 stream，它的状态按照格式化字符串中的格式化选项来设定（如果有的话）。Format 对象保留最后的字符串结果。

一旦是所有的参数都已经传送给格式化对象了，你可以把格式化对象存进一个 stream 中，可以用 str()成员函数得到它的字符串值，或者使用 boost 命名空间中的自由函数 str(const format& )。format 对象中的结果字符串保持可取直到另一个参数被传递进来，这时它将重新初始化。

```
// fmter was previously created and fed arguments, it can print the result :
cout << fmter ;
// You can take the string result :
string s  = fmter.str();
// possibly several times :
```

```
s = fmter.str( );
// You can also do all steps at once :
cout << boost::format("%2% %1%") % 36 % 77;
// using the str free function :
string s2 = str( format("%2% %1%") % 36 % 77 );
```

可选的，第三步之后，你可以重用 format 对象并且在第二步重新开始: fmter % 18 % 39; 用格式化字符串格式化新的变量，保存有关第一步的复杂的处理。

总而言之，format 类把一个格式化字符串转换成（最后使用类 prinf 指示）对一个内部 stream 的操作，最后以字符串格式返回，或者直接输出到一个输出流。

---

例子

```
using namespace std;
using boost::format;
using boost::io::group;
```

重新排列的简单输出：

```
cout << format("%1% %2% %3% %2% %1% \n") % "11" % "22" % "333"; // 'simple' style.
```

输出 : "11 22 333 22 11 \n"

更精确的格式化，带 Posix-printf 位置指示符：

```
    cout << format("(x,y) = (%1$+5d,%2$+5d) \n") % -23 % 35;   // Posix-Printf style
```

输出 : "(x,y) = ( -23, +35) \n"

经典的 printf 指示符，没有重排：

```
    cout << format("writing %s,  x=%s : %d-th step \n") % "toto" % 40.23 % 50;
```

输出 : "writing toto, x=40.23 : 50-th step \n"

处理同一事情的几种方法：

```
    cout << format("(x,y) = (%+5d,%+5d) \n") % -23 % 35;
    cout << format("(x,y) = (%|+5|,%|+5|) \n") % -23 % 35;
    cout << format("(x,y) = (%1$+5d,%2$+5d) \n") % -23 % 35;
    cout << format("(x,y) = (%|1$+5|,%|2$+5|) \n") % -23 % 35;
```

输出均为 : "(x,y) = ( -23, +35) \n"

使用操纵子修改格式化串：

```
    format fmter("_%1$+5d_ %1$d \n");
    format fmter2("_%1%_ %1% \n");
    fmter2.modify_item(1, group(showpos, setw(5)) );
    cout << fmter % 101 ;
    cout << fmter2 % 101 ;
```

输出均为 : "_ +101_ 101 \n"

使用带参数的操纵子：

```
    cout << format("_%1%_ %1% \n") % group(showpos, setw(5), 101);
```

在每个 %1% 出现时都要应用操纵子, 输出 : "_ +101_ +101 \n"

新增的格式化特性：'绝对表格', 用在循环中以确保输出的字段在每一行中的相同位置，即使其前面参数的宽度是可变的。

```
    for(unsigned int i=0; i < names.size(); ++i)
        cout << format("%1%, %2%, %|40t|%3%\n") % names[i] % surname[i] % tel[i];
```

对于一些 std::vector names, surnames, 和 tel (见 sample_new_features.cpp) 输出 :

```
Marc-François Michel, Durand,          +33 (0) 123 456 789
Jean, de Lattre de Tassigny,           +33 (0) 987 654 321
```

## 2.3 语法

### 2.3.1 boost::format( format-string ) % arg1 % arg2 % ... % argN

格式化字符串包含的字符文本,在它当中特殊的指示器将被字符串所替换,这些字符串由所给的参数产生。

从 c，c++世界继承来的语法其中之一是使用 printf，这样 format 能直接使用 printf 的格式化字符串，并且产生一样的结果(几乎包含所有方面，细节请查看 Incompatibilities with printf)。这些核心语法已被扩展，来允许新特性，但是同时适合 c++流的上下文。所以，format 接受在格式化字符串中的一系列的指示：

1. 继承自 printf 的格式化字符串:%spec

   这里 spec 是个 printf 格式规则， spec 传递格式选项，如宽度，对齐，用来格式化数字的基数。它仅仅在内部 stream 上设置合适的标志，和/或格式化参数，但是不要求合适的参数为一类特殊的类型。 比如：$x 这样的规格，对于 printf 意味着"打印参数数字，它是个进制的整数"，对于 format 来说仅仅意味着"打印基于进制的第

二个参数”。

2. %|spec|

这里 spec 是一个打印格式规则。框架已经介绍了，为了增加格式化字符串的可读性，但是重要的是，使在 spec 中的类型转换字符任意可选。这些信息对于 c++变量来说是不够的，但是为了直接支持 printf 语法，有必要总是给出一个类型转换字符，仅仅是因为这个字符用来判断格式化字符串的末尾是至关重要的。 比如："%|-5|"将下一个变量设置为宽度为，左对齐，就像"%-5g", "%-5f", "%-5s" ..这样的 printf 格式方式。

3. %N%

这个简单的位置标记需要第 N 个参数的格式化——而不需要任何格式化选项。（它不过是 printf 的位置标志的一个简单表示（就像“%N$s”），但是一个主要的好处是它有更强的可读性，而且不用一个“类型转换”的字符）。标准 printf 格式化规格之上，新的特性被增加进来了，比如居中对齐。细节参看 new format specification .

### 2.3.2 printf 格式化规则

Boost.format 所支持的格式化规则严格按照 Unix98 的文档 Open-group printf，而非标准 c 的 printf，它不支持有位置信息的参数。（普通的标志在前两者中有同样的含义，这样都不会任何人带来麻烦）。注意：在同一个格式化字符串中同时使用有位置信息（e.g. %3$+d）的格式化规则和没有位置信息（e.g. %+d）的格式化规则是会出错的。 在 Open-group 规则中，使用同一参数多次(e.g. "%1$d %1$d")会导致未定义的行为。Boost.format 在这些情况中允许多次使用同一参数。唯一的约束是它期望准确的参数个数，P 是格式化用到的最大参数个数（e.g., for "%1$d %10$d", P = 10）。 如果提供多余或者少于 P 数量的参数会引起异常（除非它被另外设定，查看 exceptions 异常 一节）

一个规范的 spec 有这样的格式: [ N$ ] [ flags ] [ width ] [ . precision ] type-char

方括号中的参数是可选的。下面将一一介绍每个参数域：

N $ （可选域）指定了应用于第 N 个参数的格式规则。（它称之为一个位置格式规则）如果这不满足，参数将被一一提取（并且将会在后面提供参数数时产生一个错误）。

标志为下面序列中的任一：

| 标志 | 含义 | 在内部 stream 的作用 |
|---|---|---|
| '-' | 左对齐 | N/A (作用于后面的 string) |
| '=' | 居中对齐 | N/A (作用于后面的 string)<br>- *注意 ： 新增的特性，printf 中是没有的* - |
| '_' | 内部对齐 | 设置内部对齐<br>- *注意:新增特性，printf 中没有* - |
| '+' | 显示符号无论是否正数 | 设置 *showpos* |
| '#' | 显示数字的基，和小数点 | 设置 *showbase* 和 *showpoint* |
| '0' | 后加 0 (插在符号或者基指示器后) | 如非左对齐，调用 *setfill('0')* 并且设置 *internal*<br>当 stream 转换成 user-defined output 后执行额外的操作. |
| ' ' | 如果字符串不是以+,-开头，在已转换的字符串前插入空格 | N/A (作用于后面的 string)<br>不同于 printf 的行为:它不受内部对齐的影响 |

width

指定转换而来的字符串的最小宽度。如果有必要，字符串将被对齐填补并且通过用操纵子填充设置在 stream 上字符或者被格式化字符串指定的字符(比如,标志‘’,‘-’,..)。注意长度不仅设置在转换 stream 上。为了支持 user-defined types 的输出（可能对多个成员多次调用<<操作），宽度在 stream 将整个参数对象转换之后被传递进去，它是在 format 类中的代码实现的。

precision（通过加小数点），设置 stream 的精度。

当输出一个浮点类型数字时，它设置数字的最大数字个数

当它是确定的或者科学计数模式时在小数点之后

当它是默认模式时为总数（‘普通模式’,像%g）

当使用字符类型 s 或者 S 时，它包含了另外的含义：转换字符串被压缩成第一类字符的精度。（注意在压缩之后才将后面的填充到 width 宽度）

类型字符。它没有强迫相关的参数来为一个限制的类型集合，只是为相关的类型规则设置标志。

| 字符类型 | 含义 | 对 stream 的作用 |
|---|---|---|
| p or x | 十六进制输出 | 设置 *hex* |
| o | 十进制输出 | 设置 *oct* |
| e | 科学计数浮点格式 | 设置浮点域位为 *scientific* |

| | | |
|---|---|---|
| **f** | 固定浮点格式 | 设置浮点域位为 *fixed* |
| **g** | 默认浮点格式 | 所有浮点域位归位 |
| **X, E** or **G** | 跟小写一样，但是使用大写字符作为输出（exponents, hex digits, ..） | 除了大写，跟小写‘x’，‘e’，‘g’一样 |
| **d, i** or **u** | **decimal** 类型输出 | 设置进制位为 *dec* |
| **s** or **S** | 字符串输出 | 精度格式归位，它的值传给内部域并作用后面的 truncation（查看前面的 precision 注释） |
| **c** or **C** | 单字符输出 | 只使用转换字符串的第一个字符 |
| **%** | 输出字符 % | N/A |

注意‘n’类型规则被忽略（相关的参数也一样），因为它不符合这里的内容。同样的，printf 的‘l’, ‘L’或‘h’（来表示宽度、long 或 short 类型）支持修改符（对于内部流没有影响）。

### 2.3.3 新的格式规则

就像标志表所列的那样，新加了中间对齐的标志(‘ = '和‘_')。

%{nt} ，就像标志表所列的那样，新加了中间对齐的标志(‘ = '和‘_')。(见 examples 例子 )

%{nTX} 以同样的方式插入一个制表符，但是使用 X 作为填充字符以取代当前流的‘fill’域所指示的字符（默认为空格）。

# 附录 4 TCHAR.h 映射表

| Generic-text routine name | SBCS (_UNICODE & MBCS not defined) | _MBCS defined | _UNICODE defined |
|---|---|---|---|
| _cgetts | _cgets | _cgets | _cgetws |
| _cgetts_s | _cgets_s | _cgets_s | _cgetws_s |
| _cputts | _cputs | _cputs | _cputws |
| _fgettc | fgetc | fgetc | fgetwc |
| _fgettchar | _fgetchar | _fgetchar | _fgetwchar |
| _fgetts | fgets | fgets | fgetws |
| _fputtc | fputc | fputc | fputwc |
| _fputtchar | _fputchar | _fputchar | _fputwchar |
| _fputts | fputs | fputs | fputws |
| _ftprintf | fprintf | fprintf | fwprintf |
| _ftprintf_s | fprintf_s | fprintf_s | fwprintf_s |
| _ftscanf | fscanf | fscanf | fwscanf |
| _ftscanf_s | fscanf_s | fscanf_s | fwscanf_s |
| _gettc | getc | getc | getwc |
| _gettch | _getch | _getch | _getwch |
| _gettchar | getchar | getchar | getwchar |
| _gettche | _getche | _getche | _getwche |
| _getts | gets | gets | getws |
| _getts_s | gets_s | gets_s | getws_s |
| _istalnum | isalnum | _ismbcalnum | iswalnum |

| | | | |
|---|---|---|---|
| _istalpha | isalpha | _ismbcalpha | iswalpha |
| _istascii | isascii | isascii | iswascii |
| _istcntrl | iscntrl | iscntrl | iswcntrl |
| _istdigit | isdigit | _ismbcdigit | iswdigit |
| _istgraph | isgraph | _ismbcgraph | iswgraph |
| _istlead | Always returns false | _ismbblead | Always returns false |
| _istleadbyte | Always returns false | isleadbyte | Always returns false |
| _istlegal | Always returns true | _ismbclegal | Always returns true |
| _istlower | islower | _ismbclower | iswlower |
| _istprint | isprint | _ismbcprint | iswprint |
| _istpunct | ispunct | _ismbcpunct | iswpunct |
| _istspace | isspace | _ismbcspace | iswspace |
| _istupper | isupper | _ismbcupper | iswupper |
| _istxdigit | isxdigit | isxdigit | iswxdigit |
| _itot | _itoa | _itoa | _itow |
| _itot_s | _itoa_s | _itoa_s | _itow_s |
| _ltot | _ltoa | _ltoa | _ltow |
| _ltot_s | _ltoa_s | _ltoa_s | _ltow_s |
| _puttc | putc | putc | putwc |
| _puttch | _putch | _putch | _putwch |
| _puttchar | putchar | putchar | putwchar |
| _putts | puts | puts | _putws |
| _sctprintf | _scprintf | _scprintf | _scwprintf |
| _sntprintf | _snprintf | _snprintf | _snwprintf |
| _sntprintf_s | _snprintf_s | _snprintf_s | _snwprintf_s |
| _sntscanf | _snscanf | _snscanf | _snwscanf |
| _sntscanf_s | _snscanf_s | _snscanf_s | _snwscanf_s |
| _stprintf | sprintf | sprintf | swprintf |
| _stprintf_s | sprintf_s | sprintf_s | swprintf_s |
| _stscanf | sscanf | sscanf | swscanf |
| _stscanf_s | sscanf_s | sscanf_s | swscanf_s |
| _taccess | _access | _access | _waccess |
| _taccess_s | _access_s | _access_s | _waccess_s |
| _tasctime | asctime | asctime | _wasctime |
| _tasctime_s | asctime_s | asctime_s | _wasctime_s |

| | | | |
|---|---|---|---|
| _tccmp | Maps to macro or inline function | _mbsncmp | Maps to macro or inline function |
| _tccpy | Maps to macro or inline function | _mbccpy | Maps to macro or inline function |
| _tccpy_s | strcpy_s | _mbccpy_s | wcscpy_s |
| _tchdir | _chdir | _chdir | _wchdir |
| _tclen | Maps to macro or inline function | _mbclen | Maps to macro or inline function |
| _tchmod | _chmod | _chmod | _wchmod |
| _tcprintf | _cprintf | _cprintf | _cwprintf |
| _tcprintf_s | _cprintf_s | _cprintf_s | _cwprintf_s |
| _tcreat | _creat | _creat | _wcreat |
| _tcscanf | _cscanf | _cscanf | _cwscanf |
| _tcscanf_s | _cscanf_s | _cscanf_s | _cwscanf_s |
| _tcscat | strcat | _mbscat | wcscat |
| _tcscat_s | strcat_s | _mbscat_s | wcscat_s |
| _tcschr | strchr | _mbschr | wcschr |
| _tcsclen | strlen | _mbslen | wcslen |
| _tcsclen_s | strlen_s | _mbslen_s | wcslen_s |
| _tcscmp | strcmp | _mbscmp | wcscmp |
| _tcscoll | strcoll | _mbscoll | wcscoll |
| _tcscpy | strcpy | _mbscpy | wcscpy |
| _tcscpy_s | strcpy_s | _mbscpy_s | wcscpy_s |
| _tcscspn | strcspn | _mbscspn | wcscspn |
| _tcsdec | _strdec | _mbsdec | _wcsdec |
| _tcsdup | _strdup | _mbsdup | _wcsdup |
| _tcserror | strerror | strerror | _wcserror |
| _tcserror_s | strerror_s | strerror_s | _wcserror_s |
| _tcsftime | strftime | strftime | wcsftime |
| _tcsicmp | _stricmp | _mbsicmp | _wcsicmp |
| _tcsicoll | _stricoll | _mbsicoll | _wcsicoll |
| _tcsinc | _strinc | _mbsinc | _wcsinc |
| _tcslen | strlen | strlen | wcslen |
| _tcslen_s | strlen_s | strlen_s | wcslen_s |
| _tcslwr | _strlwr | _mbslwr | _wcslwr |
| _tcslwr_s | _strlwr_s | _mbslwr_s | _wcslwr_s |
| _tcsnbcnt | _strncnt | _mbsnbcnt | _wcsncnt |
| _tcsncat | strncat | _mbsnbcat | wcsncat |

| | | | |
|---|---|---|---|
| _tcsncat_s | strncat_s | _mbsnbcat_s | wcsncat_s |
| _tcsnccat | strncat | _mbsncat | wcsncat |
| _tcsnccmp | strncmp | _mbsncmp | wcsncmp |
| _tcsnccmp_s | strncmp_s | _mbsncmp_s | wcsncmp_s |
| _tcsnccoll | _strncoll | _mbsncoll | _wcsncoll |
| _tcsncmp | strncmp | _mbsnbcmp | wcsncmp |
| _tcsnccnt | _strncnt | _mbsnccnt | _wcsncnt |
| _tcsnccpy | strncpy | _mbsncpy | wcsncpy |
| _tcsnccpy_s | strncpy_s | _mbsncpy_s | wcsncpy_s |
| _tcsncicmp | _strnicmp | _mbsnicmp | _wcsnicmp |
| _tcsncicoll | _strnicoll | _mbsnicoll | _wcsnicoll |
| _tcsncpy | strncpy | _mbsnbcpy | wcsncpy |
| _tcsncpy_s | strncpy_s | _mbsnbcpy_s | wcsncpy_s |
| _tcsncset | _strnset | _mbsnset | _wcsnset |
| _tcsnextc | _strnextc | _mbsnextc | _wcsnextc |
| _tcsnicmp | _strnicmp | _mbsnbicmp | _wcsnicmp |
| _tcsnicoll | _strnicoll | _mbsnbicoll | _wcsnicoll |
| _tcsninc | _strninc | _mbsninc | _wcsninc |
| _tcsnccnt | _strncnt | _mbsnccnt | _wcsncnt |
| _tcsnset | _strnset | _mbsnbset | _wcsnset |
| _tcspbrk | strpbrk | _mbspbrk | wcspbrk |
| _tcsspnp | _strspnp | _mbsspnp | _wcsspnp |
| _tcsrchr | strrchr | _mbsrchr | wcsrchr |
| _tcsrev | _strrev | _mbsrev | _wcsrev |
| _tcsset | _strset | _mbsset | _wcsset |
| _tcsspn | strspn | _mbsspn | wcsspn |
| _tcsstr | strstr | _mbsstr | wcsstr |
| _tcstod | strtod | strtod | wcstod |
| _tcstoi64 | _strtoi64 | _strtoi64 | _wcstoi64 |
| _tcstok | strtok | _mbstok | wcstok |
| _tcstok_s | strtok_s | _mbstok_s | wcstok_s |
| _tcstol | strtol | strtol | wcstol |
| _tcstoui64 | _strtoui64 | _strtoui64 | _wcstoui64 |
| _tcstoul | strtoul | strtoul | wcstoul |
| _tcsupr | _strupr | _mbsupr | _wcsupr |

| | | | |
|---|---|---|---|
| _tcsupr_s | _strupr_s | _mbsupr_s | _wcsupr_s |
| _tcsxfrm | strxfrm | strxfrm | wcsxfrm |
| _tctime | ctime | ctime | _wctime |
| _tctime_s | ctime_s | ctime_s | _wctime_s |
| _tctime32 | _ctime32 | _ctime32 | _wctime32 |
| _tctime32_s | _ctime32_s | _ctime32_s | _wctime32_s |
| _tctime64 | _ctime64 | _ctime64 | _wctime64 |
| _tctime64_s | _ctime64_s | _ctime64_s | _wctime64_s |
| _texecl | _execl | _execl | _wexecl |
| _texecle | _execle | _execle | _wexecle |
| _texeclp | _execlp | _execlp | _wexeclp |
| _texeclpe | _execlpe | _execlpe | _wexeclpe |
| _texecv | _execv | _execv | _wexecv |
| _texecve | _execve | _execve | _wexecve |
| _texecvp | _execvp | _execvp | _wexecvp |
| _texecvpe | _execvpe | _execvpe | _wexecvpe |
| _tfdopen | _fdopen | _fdopen | _wfdopen |
| _tfindfirst | _findfirst | _findfirst | _wfindfirst |
| _tfindnext | _findnext | _findnext | _wfindnext |
| _tfindnext32 | _findnext32 | _findnext32 | _wfindnext32 |
| _tfindnext64 | _findnext64 | _findnext64 | _wfindnext64 |
| _tfindnexti64 | _findnexti64 | _findnexti64 | _wfindnexti64 |
| _tfindnexti6432 | _findnexti6432 | _findnexti6432 | _wfindnexti6432 |
| _tfindnext32i64 | _findnext32i64 | _findnext32i64 | _wfindnext32i64 |
| _tfopen | fopen | fopen | _wfopen |
| _tfopen_s | fopen_s | fopen_s | _wfopen_s |
| _tfreopen | freopen | freopen | _wfreopen |
| _tfreopen_s | freopen_s | freopen_s | _wfreopen_s |
| _tfsopen | _fsopen | _fsopen | _wfsopen |
| _tfullpath | _fullpath | _fullpath | _wfullpath |
| _tgetcwd | _getcwd | _getcwd | _wgetcwd |
| _tgetdcwd | _getdcwd | _getdcwd | _wgetdcwd |
| _tgetenv | getenv | getenv | _wgetenv |
| _tgetenv_s | getenv_s | getenv_s | _wgetenv_s |
| _tmain | main | main | wmain |

| | | | |
|---|---|---|---|
| _tmakepath | _makepath | _makepath | _wmakepath |
| _tmakepath_s | _makepath_s | _makepath_s | _wmakepath_s |
| _tmkdir | _mkdir | _mkdir | _wmkdir |
| _tmktemp | _mktemp | _mktemp | _wmktemp |
| _tmktemp_s | _mktemp_s | _mktemp_s | _wmktemp_s |
| _topen | _open | _open | _wopen |
| _topen_s | _open_s | _open_s | _wopen_s |
| _totlower | tolower | _mbctolower | towlower |
| _totupper | toupper | _mbctoupper | towupper |
| _tperror | perror | perror | _wperror |
| _tpopen | _popen | _popen | _wpopen |
| _tprintf | printf | printf | wprintf |
| _tprintf_s | printf_s | printf_s | wprintf_s |
| _tputenv | _putenv | _putenv | _wputenv |
| _tputenv_s | _putenv_s | _putenv_s | _wputenv_s |
| _tremove | remove | remove | _wremove |
| _trename | rename | rename | _wrename |
| _trmdir | _rmdir | _rmdir | _wrmdir |
| _tsearchenv | _searchenv | _searchenv | _wsearchenv |
| _tsearchenv_s | _searchenv_s | _searchenv_s | _wsearchenv_s |
| _tscanf | scanf | scanf | wscanf |
| _tscanf_s | scanf_s | scanf_s | wscanf_s |
| _tsetlocale | setlocale | setlocale | _wsetlocale |
| _tsopen | _sopen | _sopen | _wsopen |
| _tsopen_s | _sopen_s | _sopen_s | _wsopen_s |
| _tspawnl | _spawnl | _spawnl | _wspawnl |
| _tspawnle | _spawnle | _spawnle | _wspawnle |
| _tspawnlp | _spawnlp | _spawnlp | _wspawnlp |
| _tspawnlpe | _spawnlpe | _spawnlpe | _wspawnlpe |
| _tspawnv | _spawnv | _spawnv | _wspawnv |
| _tspawnve | _spawnve | _spawnve | _wspawnve |
| _tspawnvp | _spawnvp | _spawnvp | _wspawnvp |
| _tspawnvpe | _spawnvpe | _spawnvpe | _wspawnvpe |
| _tsplitpath | _splitpath | _splitpath | _wsplitpath |
| _tstat | _stat | _stat | _wstat |

| | | | |
|---|---|---|---|
| _tstat32 | _stat32 | _stat32 | _wstat32 |
| _tstati32 | _stati32 | _stati32 | _wstati32 |
| _tstat64 | _stat64 | _stat64 | _wstat64 |
| _tstati64 | _stati64 | _stati64 | _wstati64 |
| _tstof | atof | atof | _wtof |
| _tstoi | atoi | atoi | _wtoi |
| _tstoi64 | _atoi64 | _atoi64 | _wtoi64 |
| _tstol | atol | atol | _wtol |
| _tstrdate | _strdate | _strdate | _wstrdate |
| _tstrdate_s | _strdate_s | _strdate_s | _wstrdate_s |
| _tstrtime | _strtime | _strtime | _wstrtime |
| _tstrtime_s | _strtime_s | _strtime_s | _wstrtime_s |
| _tsystem | system | system | _wsystem |
| _ttempnam | _tempnam | _tempnam | _wtempnam |
| _ttmpnam | tmpnam | tmpnam | _wtmpnam |
| _ttmpnam_s | tmpnam_s | tmpnam_s | _wtmpnam_s |
| _ttoi | atoi | atoi | _wtoi |
| _ttoi64 | _atoi64 | _atoi64 | _wtoi64 |
| _ttol | atol | atol | _wtol |
| _tunlink | _unlink | _unlink | _wunlink |
| _tutime | _utime | _utime | _wutime |
| _tutime32 | _utime32 | _utime32 | _wutime32 |
| _tutime64 | _utime64 | _utime64 | _wutime64 |
| _tWinMain | WinMain | WinMain | wWinMain |
| _ui64tot | _ui64toa | _ui64toa | _ui64tow |
| _ui64tot_s | _ui64toa_s | _ui64toa_s | _ui64tow_s |
| _ultot | _ultoa | _ultoa | _ultow |
| _ultot_s | _ultoa_s | _ultoa_s | _ultow_s |
| _ungettc | ungetc | ungetc | ungetwc |
| _ungettch | _ungetch | _ungetch | _ungetwch |
| _vftprintf | vfprintf | vfprintf | vfwprintf |
| _vftprintf_s | vfprintf_s | vfprintf_s | vfwprintf_S |
| _vsctprintf | _vscprintf | _vscprintf | _vscwprintf |
| _vsctprintf_s | _vscprintf_s | _vscprintf_s | _vscwprintf_S |
| _vsntprintf | _vsnprintf | _vsnprintf | _vsnwprintf |

| | | | |
|---|---|---|---|
| _vsntprintf_s | _vsnprintf_s | _vsnprintf_s | _vsnwprintf_s |
| _vstprintf | vsprintf | vsprintf | vswprintf |
| _vstprintf_s | vsprintf_s | vsprintf_s | vswprintf_s |
| _vtprintf | vprintf | vprintf | vwprintf |
| _vtprintf_s | vprintf_s | vprintf_s | vwprintf_s |

## 日志：

1.0 版

基本内容的完成

1.1 版

1. 修改了“at 和[]一样”的 Bug。
2. 添加了 string 和流。
3. 添加了格式化字符串的内容。
4. 添加了 MFC 中使用 string 的一些小内容。

2009/1/6

1.2 版

1. 修改了 wstring 和 string 相互转换的程序。
2. 在附录中添加部分 Boost.Format 的文档。

2009/1/18

2.0 版

1. 修复了 wstring 和 string 相互转换的程序的一个小 bug。
2. 添加了类 C 字符串处理的一大章节。
3. 添加了 TCHAR 映射表作为附录资料

2009/1/30