**STl string字符串类详解**

本章讲述 C++ STL 中的字符串类模板 string。  
在最初的 C 语言中，头文件 string.h 提供了一系列字符串函数。早期的 C++ 也为处理字符串提供了类。string 类由头文件 <string> 支持，该类包含了大量方法及若干构造函数，用于将字符串赋给变量、合并字符串、比较字符串和访问各个元素的重载操作符、查找字符和子字符串的方法等。到目前为止，string 已被用户广泛接受及使用。  
 C++ 从 C 继承的字符串概念仍然是以 '\0' 为结束符的 char 数组。C++ 标准库中的 string class 可以将 string 作为一个型别，可以实现复制、赋值和比较，不必担心内存大小及占用内存实际长度等具体问题。  
 现今，数据处理大部分是字符串处理，相较于早期的 C 语言和 Fortran 语言，这是非常重要的进步。在这些语言中，字符串的处理是非常复杂的。  
 本章将详细讲述字符串类库简述、字符的特点、字符串类模板（basic\_String），字符串通用函数、字符串联接、字符串 IO 操作、搜索和查找、字符串对迭代器的支持以及字符串对配置器的支持等内容。

**1 C++ string类库简介**

字符串的表现形式多种多样，如 TCHAR，std::string、BSTR 等。字符串类均起源于 C 语言的字符串，而最初 C 语言的字符串是字符的数组。单字节字符串顺序存放各个字符串，并用 \0 表示字符串的结束。在 C 语言中，已存在部分字符串处理函数，例如 strcpy()，sprintf() , stoi() 等，只能用于单字节字符串。在标准库中，还有仅用于 Unicode 字符串的函数，如 wcscpy()，swprintf() , \_wtol() 等。  
 多数人都惯于使用[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank) ++ 和 -- 操作符来遍历字符串。使用数组处理字符串中的字符也非常方便。无论 ASCII 码字符串还是 Unicode 字符串，使用指针均能够正确无误地返回要寻求的字符位置。  
 [STL](http://c.biancheng.net/stl/) 中只有一个字符串类，即 basic\_string。类 basic\_string 实现管理以 \0 结尾的字符数组，字符类型由模板参数决定。  
 通常，basic\_string 被处理为不透明的对象，靠获得只读指针来访问缓冲区，写操作是由 basic\_string 的成员函数实现的。STL 的 [C++](http://c.biancheng.net/cplus/) 标准程序库中的 string 类，使用时不必担心内存是否充足、字符串长度等问题。  
 string 作为类出现，其集成的操作函数足以完成多数情况下的需要。可以使用 "=" 进行赋值，使用 "==" 进行等值比较，使用 "+" 做串联。  
 要使用 string 类，必须包含头文件 <string>。在 STL 库中，basic\_string 有两个预定义类型：包含 char 的 string 类型和包含 wchar 的 wstring 类型。  
 string 类的 s**tring::npos 可同时定义字符串的最大长度，通常设置为无符号 int 的最大值。string 类包含了 6 个构造函数。string 类支持 cin 方式和 getline() 方式两种输入方式。**简单示例如下：

string stuff;

cin >> stuff;

getline(cin, stuff);

上述三行代码，第一行是声明 string 类的对象 stuff，第二行是从屏幕读入输入的字符串，第三行同样实现第二行代码的功能。  
 string 库提供了许多其他功能，如删除字符串的部分或全部，用一个字符的部分或全部替换另一个字符串的部分或全部，插入、删除字符串中数据，比较、提取、复制、交换等。

**2 浅谈C++ auto\_ptr智能指针**

[STL](http://c.biancheng.net/stl/) 还提供了另一个模板类：auto\_ptr 类。该类主要用于管理动态内存分配。  
 如果使用 new() 函数分配堆中的内存，而又不记得回收这部分内存，会导致内存泄漏。因此必须使用 delete 语句释放该内存块。即使在函数末端添加了 delete 语句释放内存，还需要在任何跳出该函数的语句（如抛出异常）之前添加释放内存的处理，例如 goto 语句和 throw 语句。  
 auto\_ptr 模板定义了类似[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank)的对象，将 new 获得的地址赋给该对象。当 auto\_ptr 对象过期时，析构函数将使用 delete 来释放内存。如果将 new 返回的地址赋值给 auto\_ptr 对象，无须记住还需要释放这些内存。在 auto\_ptr 对象过期时，内存将自动被释放。  
 在 [C++](http://c.biancheng.net/cplus/) 语言中，要使用 STL 中的 auto\_ptr 对象，必须包含头文件 <memory>，该文件包括 auto\_ptr 模板。使用通常的模板句法来实例化所需类型的指针。auto\_ptr 构造函数是显式的，不存在从指针到 auto\_ptr 对象的隐式类型转换。

auto\_ptr <double> pd;  
double \*p\_reg = new double;  
pd = p\_reg; // 不允许  
pd = auto\_ptr <double> (p\_reg); //允许  
auto\_ptr <double> panto =p\_reg; //不允许  
auto\_ptr <double> pauto (p\_reg); //允许

模板可以通过构造函数将 auto\_ptr 对象初始化为一个常规指针。auto\_ptr 是一个智能指针，但其特性远比指针要多。值得注意的是，在使用 auto\_ptr 时，只能配对使用 new 和 delete。

提示，只能对 new 分配的内存使用 auto\_ptr 对象，不要对由 new() 分配的或通过声明变量分配的内存使用它。

**总结**

C++ 库中的 auto\_ptr 对象是一种智能指针。智能指针是一种类，即其对象的特征结类似于指针。智能指针存储 new 分配的内存地址，也可以被解除引用。  
 智能指针是一个类对象，可以修改和扩充简单指针的行为。智能指针可以建立引用计数，这使得多个对象可共享由智能指针跟踪的同一个值。当使用该值的对象数为 0 时，智能指针将删除该值。  
智能指针可以提高内存的使用效率，帮助防止内存泄露。

1. **浅谈C++ auto\_ptr智能指针**

"字符" 本身是个有趣的抽象概念。例如，在纸上或者屏幕上，字符 "C" 仅仅是一段曲线而已。在计算机中，用一个 8 Byte 存储该字符，并赋值 67；字符 "C" 还是拉丁字母的第三个字母；在化学专业，字符 "C" 是原子碳的缩写形式；在计算机学科中，字符 "C" 又被用来表示一种程序设计语言的名字。  
目前，在计算机领域，字符集合是在字符与整数值之间的一种映射关系。  
 [C++](http://c.biancheng.net/cplus/) 程序员通常假定能够使用美国字符集（ASCII），但 C++ 允许程序员缺少某些字符的可能性。如果在程序开发过程中，源代码含有 ASCII 里所没有的字符，这是比较麻烦的，许多语言（例如中文、丹麦文、法文、冰岛文、日文）无法用 ASCII 中的字符正常写出来，即使扩充到 16 位字符集，也无法将人类所知的所有字符放在同一字符集中。据说已经出现的 32 位字符集能保存每一个字符，但因字符数量庞大，不便于使用，尚未得到推广。  
 C++ 语言允许程序员使用任何字符集作为字符串的字符，也允许程序员使用扩充字符集或可移植的数值编码。从原则上讲，字符串能以任何（带有正确的复制操作）类型作为其字符类型。标准字符串类 string 要求其中的字符不能包含用户自定义复制操作，有助于字符串 I/O 的简化与高效率。   
 字符类型的性质是由字符特征类（char\_traits）定义的。字符特征类是下述模板的特例：

template <class Ch> struct char\_traits { }

所有字符特征类均定义在名称空间 std 中，标准的字符特征类由头文件 <String> 给出。通用字符串特征类 char\_traits 本身不具有任何属性，只有针对特定字符类型的专门 char\_traits 才具有属性。  
 标准字符串模板的实例类 basic\_string 依赖于诸多类型和函数。若一个类型作为 basic\_string 字符类型，必须提供支持上述功能的字符特征类（char\_traits）。  
 C++ 的字符串模板实例化类 basic\_string 中，还集合了大量的字符串处理函数，与标准容器类似。该类模板的声明如下：

template <class Ch, class Tr = char\_traits <Ch>, class A=allocator <Ch>> class std::basic\_string

{

public:

...

}

在上述模板声明中，第一个参数（class Ch）是说明单个字符（Ch）所属型别（class）。  
第二个参数（class Tr = char\_traits <Ch>）是特性类别，用以提供字符串类别中的所有字符核心操作。该特性类别规定了“复制字符”或“比较字符”的做法；如果不指定该特性类别，系统会根据现有的字符型别采用默认的特性类别。  
 第三个参数带有默认值（class A = allocator<Ch>），用以定义字符串类别所采用的内存模式，通常设定为“默认内存模型 allocator”。该模板及其相关功能都定义在名称空间 std 中，由头文件 <string> 给出，其中包含了两个定义类型，可以为最常用的串类型提供便于使用的名称，即 C++ [STL](http://c.biancheng.net/stl/) 提供了两个 basic\_string<> 实例化版本：

typedef basic\_string <char> string;  
typedef basic\_string <wchar> wstring;

其中，wstring 类是为了便于使用宽字符集，例如 Unicode 或某些欧洲字符集。但所有字符串类型均使用相同接口，其用法和问题是相同的。在本教程中，仍以 string 表示任何字符串型别。  
 basic\_string 和 vector 类似，而 basic\_string 还提供典型的字符串操作，例如子串检索。basic\_string 没有提供一组完整的操作函数。通常 string 不能直接使用数组或者 vector，为了更好地支持 string 的常见应用，程序员在实现过程中需要尽量减少复制。尤其对于较短的字符串，不应使用自由存储空间，但允许对较长的字符串进行简单修改。  
 basic\_string<T> 没有虚函数，这点和其他标准库类型一致。当需要设计更复杂的文字处理类时，可考虑用它加以实现。  
 与其他标准容器相似，basic\_string 提供了一组成员类型名，程序员能使用这些与串相关的类型。例如：

typedef Tr traits\_type;  
typedef typename Tr::char\_type value\_type  
typedef A allocator\_type  
……

basic\_string 除支持最简单的 basic\_string <char> 之外，还支持许多不同种类的字符串，例如：

typedef basic\_string <unsigned char> Ustring;  
tyepdef basic\_string <Jchar> Jstring; //日文字符串

无论如何定义字符串，模板 basic\_string 的大量函数均可便捷地使用。模板 basic\_string<Ch> 能够存放集合 Ch 中的任何字符，特别是 string 中的 '0'。"字符类型" Ch 的行为必须像字符，但它不能包含用户确定的复制构造函数、析构函数和复制赋值。  
 虽然字符串类 string 包含了诸多的成员和函数，但个别功能没能够实现，例如正则表达式和较复杂的文本处理功能。  
 总体而言，string 类似的字符串操作逐渐变得简单了。程序员可以定义 string 类型的对象、string 类的重载操作符和成员函数，这使字符串操作变得非常容易。

**4 C++ string类成员函数汇总（超全）**

在定义 string 类对象时，string 类自身可以管理内存，程序员不必关注内存的分配细节。  
 string 类提供的各种操作函数大致分为八类：构造器和析构器、大小和容量、元素存取、字 符串比较、字符串修改、字符串接合、I/O 操作以及搜索和查找。  
 下表列出了 string 类的所有成员函数及它们的功能。

|  |  |
| --- | --- |
| **表 1 string 类的所有成员函数** | |
| **函数名称** | **功能** |
| **构造函数** | **产生或复制字符串** |
| **析构函数** | **销毁字符串** |
| **=，assign** | **赋以新值** |
| **Swap** | **交换两个字符串的内容** |
| **+ =，append( )，push\_back()** | **添加字符** |
| **insert ()** | **插入字符** |
| **erase()** | **删除字符** |
| **clear ()** | **移除全部字符** |
| **resize ()** | **改变字符数量** |
| **replace()** | **替换字符** |
| **+** | **串联字符串** |
| **==，！ =，<，<=，>，>=，compare()** | **比较字符串内容** |
| **size()，length()** | **返回字符数量** |
| **max\_size ()** | **返回字符的最大可能个数** |
| **empty ()** | **判断字符串是否为空** |
| **capacity ()** | **返回重新分配之前的字符容量** |
| **reserve()** | **保留内存以存储一定数量的字符** |
| **[],at()** | **存取单一字符** |
| **>>，getline()** | **从 stream 中读取某值** |
| **<<** | **将值写入 stream** |
| **copy()** | **将内容复制为一个 C - string** |
| **c\_str()** | **将内容以 C - string 形式返回** |
| **data()** | **将内容以字符数组形式返回** |
| **substr()** | **返回子字符串** |
| **find()** | **搜寻某子字符串或字符** |
| **begin( )，end()** | **提供正向迭代器支持** |
| **rbegin()，rend()** | **提供逆向迭代器支持** |
| **get\_allocator()** | **返回配置器** |

**5 C++ string构造函数和析构函数详解**

构造函数有四个参数，其中三个具有默认值。要初始化一个 string 类，可以使用 C 风格字符串或 string 类型对象，也可以使用 C 风格字符串的部分或 string 类型对象的部分或序列。

注意，不能使用字符或者整数去初始化字符串。

常见的 string 类构造函数有以下几种形式：

string strs //生成空字符串  
string s(str) //生成字符串str的复制品  
string s(str, stridx) //将字符串str中始于stridx的部分作为构造函数的初值  
string s(str, strbegin, strlen) //将字符串str中始于strbegin、长度为strlen的部分作为字符串初值  
string s(cstr) //以C\_string类型cstr作为字符串s的初值  
string s(cstr,char\_len)    //以C\_string类型cstr的前char\_len个字符串作为字符串s的初值  
strings(num, c) //生成一个字符串，包含num个c字符  
strings(strs, beg, end)    //以区间[beg, end]内的字符作为字符串s的初值

析构函数的形式如下：

~string() //销毁所有内存，释放内存

如果字符串只包含一个字符，使用构造函数对其初始化时，使用以下两种形式比较合理：

std::string s('x');    //错误  
std::string s(1, 'x');    //正确

或

std::string s("x");    //正确

C\_string 一般被认为是常规的 [C++](http://c.biancheng.net/cplus/) 字符串。目前，在 C++ 中确实存在一个从 const char \* 到 string 的隐式型别转换，却不存在从 string 对象到 C\_string 的自动型别转换。对于 string 类型的字符串，可以通过 c\_str() 函数返回该 string 类对象对应的 C\_string。  
 通常，程序员在整个程序中应坚持使用 string 类对象，直到必须将内容转化为 char\* 时才将其转换为 C\_string。  
  
【例 1】

#include <iostream>

#include <string>

u[sin](http://c.biancheng.net/ref/sin.html)g namespace std;

int main ()

{

    string str ("12345678");

    char ch[ ] = "abcdefgh";

    string a; //定义一个空字符串

    string str\_1 (str); //构造函数，全部复制

    string str\_2 (str, 2, 5); //构造函数，从字符串str的第2个元素开始，复制5个元素，赋值给str\_2

    string str\_3 (ch, 5); //将字符串ch的前5个元素赋值给str\_3

    string str\_4 (5,'X'); //将 5 个 'X' 组成的字符串 "XXXXX" 赋值给 str\_4

    string str\_5 (str.begin(), str.end()); //复制字符串 str 的所有元素，并赋值给 str\_5

    cout << str << endl;

    cout << a << endl ;

    cout << str\_1 << endl;

    cout << str\_2 << endl;

    cout << str\_3 << endl;

    cout << str\_4 << endl;

    cout << str\_5 << endl;

    return 0;

}

程序运行结果为：

12345678  
  
12345678  
34567  
abcde  
XXXXX  
12345678

使用 cout 输出 string 类型对象 a 时，输出为空。这是因为没有给 string 类型对象 a 赋值。  
 通过上述内容的学习，读者应对 string 类的构造函数和析构函数有了初步了解，并能够使用构造函数创建 string 类型对象。

**6 C++获取字符串长度详解**

String 类型对象包括三种求解字符串长度的函数：size() 和 length()、 maxsize() 和 capacity()：

size() 和 length()：这两个函数会返回 string 类型对象中的字符个数，且它们的执行效果相同。

max\_size()：max\_size() 函数返回 string 类型对象最多包含的字符数。一旦程序使用长度超过 max\_size() 的 string 操作，编译器会拋出 length\_error 异常。

capacity()：该函数返回在重新分配内存之前，string 类型对象所能包含的最大字符数。

string 类型对象还包括一个 reserve() 函数。调用该函数可以为 string 类型对象重新分配内存。重新分配的大小由其参数决定。reserve() 的默认参数为 0。  
  
上述几个函数的使用方法如下程序所示：

#include <iostream>

#include <string>

u[sin](http://c.biancheng.net/ref/sin.html)g namespace std;

int main ()

{

    int size = 0;

    int length = 0;

    unsigned long maxsize = 0;

    int capacity=0;

    string str ("12345678");

    string str\_custom;

    str\_custom = str;

    str\_custom.resize (5);

    size = str\_custom.size();

    length = str\_custom.length();

    maxsize = str\_custom.max\_size();

    capacity = str\_custom.capacity();

    cout << "size = " << size << endl;

    cout << "length = " << length << endl;

    cout << "maxsize = " << maxsize << endl;

    cout << "capacity = " << capacity << endl;

    return 0;

}

程序执行结果为：

size = 8  
length = 8  
maxsize = 2147483647  
capacity = 15

由此程序可知，string 类型对象 str\_custom 调用 reserve() 函数时，似乎并没有起到重新分配内存的目的（笔者所用编译器为 Visual [C++](http://c.biancheng.net/cplus/)6.0）。  
修改上述代码，删除语句 str\_custom.reserve (5)，在代码 str\_custom = str 之后如下添加代码：

str\_custom.resize (5);

修改后程序的执行结构如下：

size = 5  
length = 5  
maxsize = 2147483647  
capacity = 15

重新设置 string 类型对象 str\_custom 的大小之后，重新求解 str\_custom 的大小，其执行效果与设置的数值一致（均为 5）。

**7 C++ string获取字符串元素：[]和at()**

在通常情况下，string 是 [C++](http://c.biancheng.net/cplus/) 中的字符串。字符串是一种特殊类型的容器，专门用来操作字符序列。  
 字符串中元素的访问是允许的，一般可使用两种方法访问字符串中的单一字符：下标操作符[] 和 成员函数at()。两者均返回指定的下标位置的字符。第 1 个字符索引（下标）为 0，最后的字符索引为 length()-1。  
 需要注意的是，这两种访问方法是有区别的：

下标操作符 [] 在使用时不检查索引的有效性，如果下标超出字符的长度范围，会示导致未定义行为。对于常量字符串，使用下标操作符时，字符串的最后字符（即 '\0'）是有效的。对应 string 类型对象（常量型）最后一个字符的下标是有效的，调用返回字符 '\0'。

函数 at() 在使用时会检查下标是否有效。如果给定的下标超出字符的长度范围，系统会抛出 out\_of\_range 异常。

【例 1】

#include <iostream>

#include <string>

int main()

{

const std::string cS ("c.biancheng.net");

std::string s ("abode");

char temp =0;

char temp\_1 = 0;

char temp\_2 = 0;

char temp\_3 = 0;

char temp\_4 = 0;

char temp\_5 = 0;

temp = s [2]; //"获取字符 'c'

temp\_1 = s.at(2); //获取字符 'c'

temp\_2 = s [s.length()]; //未定义行为，返回字符'\0'，但Visual C++ 2012执行时未报错

temp\_3 = cS[cS.length()]; //指向字符 '\0'

temp\_4 = s.at (s.length ()); //程序异常

temp\_5 = cS.at(cS.length ()); //程序异常

std::cout << temp <<temp\_1 << temp\_2 << temp\_3 << temp\_4 << temp\_5 << std::endl;

return 0;

}

通过对上述代码的分析可知，要理解字符串的存取需要多实践、多尝试，并且要牢记基础知识和基本规则。  
 为修改 string 字符串的内容，下标操作符 [] 和函数 at() 均返回字符的“引用”。但当字符串的内存被重新分配以后，可能会发生执行错误。  
  
【例 2】

#include <iostream>

#include <string>

int main()

{

std::string s ("abode");

std::cout << s << std::endl ;

char& r = s[2] ; //建立引用关系

char\*p=&s[3]; //建立引用关系

r='X' ;//修改内容

\*p='Y' ;//修改内容

std::cout << s << std::endl; //输出

s = "12345678"; //重新赋值

r ='X'; //修改内容

\*p='Y'; //修改内容

std::cout << s << std::endl; //输出

return 0;

}

程序输出结果为：

abode  
abXYe  
12XY5678

在例 2 中，使用 Visual C++ 2012 编译器编译，字符串被重新赋值后，修改其中某位置字符的值，仍然成功。这与前面所述的“可能会发生执行错误”其实并不矛盾。因为，从意义上讲，字符串被重新赋值后，只是其原来的引用关系已经没有意义了。

**8 C++ string字符串比较方法详解**

字符串可以和类型相同的字符串相比较，也可以和具有同样字符类型的数组比较。  
Basic\_string 类模板既提供了  >、<、==、>=、<=、!= 等比较运算符，还提供了 compare() 函数，其中 compare() 函数支持多参数处理，支持用索引值和长度定位子串进行比较。该函数返回一个整数来表示比较结果。如果相比较的两个子串相同，compare() 函数返回 0，否则返回非零值。

compare()函数

类 basic\_string 的成员函数 compare() 的原型如下：

int compare (const basic\_string& s) const;  
int compare (const Ch\* p) const;  
int compare (size\_type pos, size\_type n, const basic\_string& s) const;  
int compare (size\_type pos, size\_type n, const basic\_string& s,size\_type pos2, size\_type n2) const;  
int compare (size\_type pos, size\_type n, const Ch\* p, size\_type = npos) const;

如果在使用 compare() 函数时，参数中出现了位置和大小，比较时只能用指定的子串。例如：

s.compare {pos,n, s2);

若参与比较的两个串值相同，则函数返回 0；若字符串 S 按字典顺序要先于 S2，则返回负值；反之，则返回正值。下面举例说明如何使用 string 类的 compare() 函数。  
  
【例 1】

#include <iostream>

#include <string>

u[sin](http://c.biancheng.net/ref/sin.html)g namespace std;

int main ()

{

string A ("aBcdef");

string B ("AbcdEf");

string C ("123456");

string D ("123dfg");

//下面是各种比较方法

int m=A.compare (B); //完整的A和B的比较

int n=A.compare(1,5,B,4,2); //"Bcdef"和"AbcdEf"比较

int p=A.compare(1,5,B,4,2); //"Bcdef"和"Ef"比较

int q=C.compare(0,3,D,0,3); //"123"和"123"比较

cout << "m = " << m << ", n = " << n <<", p = " << p << ", q = " << q << endl;

cin.get();

return 0;

}

程序的执行结果为：

m = 1, n = -1, p = -1, q = 0

由此可知，string 类的比较 compare() 函数使用非常方便，而且能区分字母的大小写。建议读者多使用此函数。

比较运算符

String 类的常见运算符包括 >、<、==、>=、<=、!=。其意义分别为"大于"、"小于"、"等于"、"大于等于"、"小于等于"、"不等于"。  
 比较运算符使用起来非常方便，此处不再介绍其函数原型，读者直接使用即可。下面以例 2 进行说明。  
  
【例 2】

#include <iostream>

#include <string>

using namespace std;

void TrueOrFalse (int x)

{

cout << (x?"True":"False")<<endl;

}

int main ()

{

string S1 = "DEF";

string CP1 = "ABC";

string CP2 = "DEF";

string CP3 = "DEFG";

string CP4 ="def";

cout << "S1= " << S1 << endl;

cout << "CP1 = " << CP1 <<endl;

cout << "CP2 = " << CP2 <<endl;

cout << "CP3 = " << CP3 <<endl;

cout << "CP4 = " << CP4 <<endl;

cout << "S1 <= CP1 returned ";

TrueOrFalse (S1 <=CP1);

cout << "S1 <= CP2 returned ";

TrueOrFalse (S1 <= CP2);

cout << "S1 <= CP3 returned ";

TrueOrFalse (S1 <= CP3);

cout << "CP1 <= S1 returned ";

TrueOrFalse (CP1 <= S1);

cout << "CP2 <= S1 returned ";

TrueOrFalse (CP2 <= S1);

cout << "CP4 <= S1 returned ";

TrueOrFalse (CP4 <= S1);

cin.get();

return 0;

}

程序运行结果为：

S1= DEF  
CP1 = ABC  
CP2 = DEF  
CP3 = DEFG  
CP4 = def  
S1 <= CP1 returned False  
S1 <= CP2 returned True  
S1 <= CP3 returned True  
CP1 <= S1 returned True  
CP2 <= S1 returned True  
CP4 <= S1 returned False

由上述内容可知，使用比较运算符可以非常容易地实现字符串的大小比较。在使用时比较运算符时，读者应注意，对于参加比较的两个字符串，任一个字符串均不能为 NULL，否则程序会异常退出。

**9 C++ string字符串修改和替换方法详解**

字符串内容的变化包括修改和替换两种。本节将分别讲解字符串内容的修改和字符串内容的替换。

字符串内容的修改

可以通过使用多个函数修改字符串的值。例如 assign()，operator=，erase()，交换（swap），插入（insert）等。另外，还可通过 append() 函数添加字符。  
  
下面逐一介绍各成员函数的使用方法。

assign()函数

使用 assign() 函数可以直接给字符串赋值。该函数既可以将整个字符串赋值给新串，也可以将字符串的子串赋值给新串。其在 basic\_string 中的原型为：

basic\_string& assign (const E\*s); //直接使用字符串赋值  
basic\_string& assign (const E\*s, size\_type n);  
basic\_string& assign (const basic\_string & str, size\_type pos, size\_type n);  
//将str的子串赋值给调用串  
basic\_string& assign (const basic\_string& str);    //使用字符串的“引用”賦值  
basic\_string& assign (size\_type n, E c) ; //使用 n个重复字符賦值  
basic\_string& assign (const\_iterator first, const\_iterator last);    //使用迭代器赋值

以上几种方法在例 1 中均有所体现。请读者参考下述代码。  
  
【例 1】

#include <iostream>

#include <string>

u[sin](http://c.biancheng.net/ref/sin.html)g namespace std;

int main()

{

string str1 ("123456");

string str;

str.assign (str1); //直接赋值

cout << str << endl;

str.assign (str1, 3, 3); //赋值给子串

cout << str << endl;

str.assign (str1,2,str1.npos);//赋值给从位置 2 至末尾的子串

cout << str << endl;

str.assign (5,'X'); //重复 5 个'X'字符

cout << str << endl;

string::iterator itB;

string::iterator itE;

itB = str1.begin ();

itE = str1.end();

str.assign (itB, (--itE)); //从第 1 个至倒数第 2 个元素，赋值给字符串 str

cout << str << endl;

return 0;

}

operator= 函数

operator= 的功能就是赋值。

erase()函数

erase() 函数的原型为：

iterator erase (iterator first, iterator last);  
iterator erase (iterator it);  
basic\_string& erase (size\_type p0 = 0, size\_type n = npos);

erase() 函数的使用方法为：

str.erase (str\* begin(), str.end());  
或 str.erase (3);

**swap()函数**

swap()函数的原型为：

void swap (basic\_string& str);

swap()函数的使用方法为：

string str2 ("abcdefghijklmn");  
str.swap (str2);

**insert()函数**

insert () 函数的原型为：

basic\_string& insert (size\_type p0 , const E \* s); //插人 1 个字符至字符串 s 前面  
basic\_string& insert (size\_type p0 , const E \* s, size\_type n); // 将 s 的前 3 个字符插入p0 位置  
basic\_string& insert (size\_type p0, const basic\_string& str);  
basic\_string& insert (size\_type p0, const basic\_string& str,size\_type pos, size\_type n); //选取 str 的子串  
basic\_string& insert (size\_type p0, size\_type n, E c); //在下标 p0 位置插入  n 个字符 c  
iterator insert (iterator it, E c); //在 it 位置插入字符 c  
void insert (iterator it, const\_iterator first, const\_iterator last); //在字符串前插入字符  
void insert (iterator it, size\_type n, E c) ; //在 it 位置重复插入 n 个字符 c

insert() 函数的使用方法为：

string A("ello");

string B ;

B.insert(1,A);

cout << B << endl;

A = "ello";

B = "H";

B.insert (1,"yanchy ",3);

cout<< B <<endl;

A = "ello";

B = "H";

B.insert (1,A,2,2);

cout << B << endl;

A="ello";

B.insert (1 , 5 , 'C');

cout << B << endl;

A = "ello";

string::iterator it = B.begin () +1;

const string:: iterator itF = A.begin();

const string:: iterator itG = A.end();

B.insert(it,itF,itG);

cout << B << endl;

append 函数

append() 函数的原型为：

basic\_string& append (const E \* s); //在原始字符串后面追加字符串s  
basic\_string& append (const E \* s, size\_type n);//在原始字符串后面追加字符串 s 的前 n 个字符  
basic\_string& append (const basic\_string& str, size\_type pos,size\_type n);//在原始字符串后面追加字符串 s 的子串 s [ pos,…,pos +n -1]  
basic\_string& append (const basic\_string& str);  
basic\_string& append (size\_type n, E c); //追加 n 个重复字符  
basic\_string& append (const\_iterator first, const\_iterator last); //使用迭代器追加

append() 函数的使用方法为：

A = "ello";

cout << A << endl;

cout << B << endl;

B.append(A);

cout << B << endl;

A = "ello";

cout << A << endl;

cout << B << endl;

B.append("12345",2);

cout << B << endl;

A = "ello";

cout << A << endl;

cout << B << endl;

B.append("12345",2,3);

cout << B << endl;

A = "ello";

B = "H";

cout << A << endl;

cout << B << endl;

B.append (10, 'a');

cout << B << endl;

A = "ello";

B = 'H';

cout << A << endl ;

cout << B << endl;

B.append(A.begin(), A, end());

cout << B << endl;

下面通过一个完整的例子介绍这些函数的使用：

#include <iostream>

#include <string>

using namespace std;

int main ()

{

string str1 ("123456");

string str2 ("abcdefghijklmn");

string str;

str.assign(str1);

cout << str << endl;

str.assign (str1 , 3, 3);

cout << str << endl;

str.assign (str1, 2, str1.npos);

cout << str << endl;

str.assign (5, 'X');

cout << str << endl;

string::iterator itB;

string::iterator itE;

itB = str1.begin ();

itE = str1.end();

str.assign (itB, (--itE));

cout << str << endl;

str = str1;

cout << str << endl;

str.erase(3);

cout << str << endl;

str.erase (str.begin (), str.end());

cout << ":" << str << ":" << endl;

str.swap(str2);

cout << str << endl;

string A ("ello");

string B ("H");

B.insert (1, A);

cout << B << endl;

A = "ello";

B ='H';

B.insert (1, "yanchy ", 3);

cout << "插入: " << B << endl;

A = "ello";

B = "H";

B.insert(1,A,2,2);

cout << "插入:" << B << endl;

A = "ello";

B = "H";

B.insert (1,5,'C');

cout << "插入:" << B << endl;

A = "ello";

B = "H";

string::iterator it = B.begin () +1;

const string::iterator itF = A.begin ();

const string::iterator itG = A.end ();

B.insert(it,itF,itG);

cout<<"插入："<< B << endl;

A = "ello";

B = "H";

cout << "A = " << A <<", B = " << B << endl ;

B.append (A);

cout << "追加：" << B << endl;

B = "H";

cout << "A = "<< A << ", B= " << B << endl;

B.append("12345", 2);

cout << "追加：" << B << endl;

A = "ello";

B = "H";

cout << "A = " << A << ", B= " << B << endl;

B.append ("12345", 2, 3);

cout << "追加：" << B << endl;

A = "ello";

B = "H";

cout << "A = " << A <<", B = " << B << endl;

B.append (10 , 'a');

cout << "追加:"<< B << endl;

A = "ello";

B = "H";

cout << "A = " << A << ", B = " << B << endl;

B.append(A.begin() , A.end());

cout << "追加:" << B << endl;

cin.get();

return 0;

}

程序运行结果：

123456  
456  
3456  
XXXXX  
12345  
123456  
123  
::  
abcdefghijklmn  
Hello  
插入: Hyan  
插入:Hlo  
插入:HCCCCC  
插入：Hello  
A = ello, B = H  
追加：Hello  
A = ello, B= H  
追加：H12  
A = ello, B= H  
追加：H345  
A = ello, B = H  
追加:Haaaaaaaaaa  
A = ello, B = H  
追加:Hello

字符串内容的替换

如果在一个字符串中标识出具体位置，便可以通过下标操作修改指定位置字符的值，或者替换某个子串。完成此项操作需要使用 string 类的成员函数 replace()。  
  
**replace() 函数的原型如下：**

basic\_string& replace (size\_type p0, size\_type n0, const E \* s); //使用字符串 s 中的 n 个字符，从源串的位置 P0 处开始替换  
basic\_string& replace (size\_type p0, size\_type n0, const E \*s, size\_type n); //使用字符串 s 中的 n 个字符，从源串的位置 P0 处开始替换 1 个字符  
basic\_string& replace (size\_type p0, size\_type n0, const basic\_string& str); //使用字符串 s 中的 n 个字符，从源串的位置 P0 处开始替换  
basic\_string& replace (size\_type p0, size\_type n0, const basic\_string& str, size\_type pos, size\_type n); //使用串 str 的子串 str [pos, pos + n-1] 替换源串中的内容，从位置 p0 处开始替换，替换字符 n0 个  
basic\_string& replace (size\_type p0, size\_type n0, size\_type n, E c); //使用 n 个字符 'c' 替换源串中位置 p0 处开始的 n0 个字符  
basic\_string& replace (iterator first0, iterator last0, const E \* s);//使用迭代器替换，和 1) 用法类似  
basic\_string& replace (iterator first0, iterator last0, const E \* s, size\_type n);//和 2) 类似  
basic\_string& replace (iterator first0, iterator last0, const basic\_string& str); //和 3) 类似  
basic\_string& replace (iterator first0, iterator last0, size\_type n, E c); //和 5) 类似  
basic\_string& replace (iterator first0, iterator last0, const\_iterator first, const\_iterator last); //使用迭代器替换

该函数的使用方法参照下面的程序：

#include <iostream>

#include <string>

using namespace std;

int main ()

{

string var ("abcdefghijklmn");

const string dest ("1234");

string dest2 ("567891234");

var.replace (3,3, dest);

cout << "1: " << var << endl;

var = "abcdefghijklmn";

var.replace (3,1, dest.c\_str(), 1, 3);

cout << "2: " << var << endl;

var ="abcdefghijklmn";

var.replace (3, 1, 5, 'x');

cout << "3: " << var << endl;

string::iterator itA, itB;

string::iterator itC, itD;

itA = var.begin();

itB = var.end();

var = "abcdefghijklmn";

var.replace (itA, itB, dest);

cout << "4: " << var << endl;

itA = var.begin ();

itB = var.end();

itC = dest2.begin () +1;

itD = dest2.end ();

var = "abodefghijklmn";

var.replace (itA, itB, itC, itD);

cout << "5: " << var << endl;

var = "abcdefghijklmn";

var.replace (3, 1, dest.c\_str(), 4); //这种方式会限定字符串替换的最大长度

cout <<"6: " << var << endl;

return 0;

}

程序执行结果为：

1: abc1234ghijklmn  
2: abc234efghijklmn  
3: abcxxxxxefghijklmn  
4: 1234  
5: 67891234efghijklmn  
6: abc1234efghijklmn

本节讲述了诸多可进行字符串内容的修改和替换的函数及其使用方法，并给出了例题。由于每个函数可能有多个原型，希望读者根据自己的情况，掌握其中的一种或两种，以满足自己使用的需要。同时，希望读者能够对照例题的执行效果，认真阅读本章节中的源代码，彻底掌握本节内容。

**10 C++字符串输入输出操作详解**

"<<" 和 ">>" 提供了 [C++](http://c.biancheng.net/cplus/) 语言的字符串输入和字符串输出功能。"<<" 可以将字符读入一个流中（例如 ostream）；">>" 可以实现将以空格或回车为 "结束符" 的字符序列读入到对应的字符串中，并且开头和结尾的空白字符不包括进字符串中。  
  
还有一个常用的 getline() 函数，该函数的原型包括两种形式：

template <class CharType, class Traits, class Allocator > basic\_istream<CharType, Traits>& getline (basic\_istream<CharType, Traits>& \_Istr,basic\_string <CharType，Traits, Allocator> &\_Str);  
//上述原型包含 2 个参数：第 1 个参数是输入流；第 2 个参数是保存输入内容的字符串  
template <class CharType, class Traits, class Allocator> basic\_istream<CharType, Traits>& getline (basic\_istream <CharType, Traits>&\_ Istr, basic\_string <CharType, Traits, Allocator>& \_Str,CharType\_Delim);  
//上述原型包含 3 个参数：第 1 个参数是输入流，第 2 个参数保存输入的字符串，第 3 个参数指定分界符。

该函数可将整行的所有字符读到字符串中。在读取字符时，遇到文件结束符、分界符、回车符时，将终止读入操作，且文件结束符、分界符、回车符在字符串中不会保存；当已读入的字符数目超过字符串所能容纳的最大字符数时，将会终止读入操作。  
  
下面分别按上述两种函数原型举例说明，参见下述程序：

#include <iostream>

#include <string>

u[sin](http://c.biancheng.net/ref/sin.html)g namespace std;

void main ()

{

string s1, s2;

getline(cin, s1);

getline(cin, s2, ' ');

cout << "You inputed chars are: " << s1 << endl;

cout << "You inputed chars are: " << s2 << endl;

}

程序的执行结果为：

123456  
asdfgh klj  
You inputed chars are: 123456  
You inputed chars are: asdfgh

注意，程序中输入的第二行字符中间包含空格字符，而空格之后的字符没有被存储到字符串 s2 中。

**11 C++字符串查找函数详解**

在 C 语言和 [C++](http://c.biancheng.net/cplus/) 语言中，可用于实现字符串查找功能的函数非常多。在 [STL](http://c.biancheng.net/stl/) 中，字符串的查找功能可以实现多种功能，比如说：

搜索单个字符、搜索子串；

实现前向搜索、后向搜索；

分别实现搜索第一个和最后一个满足条件的字符（或子串）；  
若查找 find() 函数和其他函数没有搜索到期望的字符（或子串），则返回 npos；若搜索成功，则返回搜索到的第 1 个字符或子串的位置。其中，npos 是一个无符号整数值，初始值为 -1。当搜索失败时， npos 表示“没有找到（not found）”或“所有剩佘字符”。  
 值得注意的是，所有查找 find() 函数的返回值均是 size\_type 类型，即无符号整数类型。该返回值用于表明字符串中元素的个数或者字符在字符串中的位置。  
下面分别介绍和字符查找相关的函数。

find()函数和 rfind()

find() 函数的原型主要有以下 4 种：

size\_type find (value\_type \_Chr, size\_type \_Off = 0) const;  
//find()函数的第1个参数是被搜索的字符、第2个参数是在源串中开始搜索的下标位置  
size\_type find (const value\_type\* \_Ptr , size\_type \_Off = 0) const;  
//find()函数的第1个参数是被搜索的字符串，第2个参数是在源串中开始搜索的下标位置  
size\_type find (const value\_type\* \_Ptr, size\_type \_Off = 0, size\_type \_Count) const;  
//第1个参数是被搜索的字符串，第2个参数是源串中开始搜索的下标，第3个参数是关于第1个参数的字符个数，可能是 \_Ptr 的所有字符数，也可能是 \_Ptr 的子串宇符个数  
size\_type find (const basic\_string& \_Str, size\_type \_Off = 0) const;  
//第1个参数是被搜索的字符串，第2参数是在源串中开始搜索的下标位置

rfind() 函数的原型和find()函数的原型类似，参数情况也类似。只不过 rfind() 函数适用于实现逆向查找。  
  
find() 函数和 rfind() 函数的使用方法参见如下程序：

#include <iostream>

#include <string>

u[sin](http://c.biancheng.net/ref/sin.html)g namespace std;

int main ()

{

string str\_ch (" for");

string str (" Hi, Peter, I'm sick. Please bought some drugs for me.");

string::size\_type m= str.find ('P', 5);

string::size\_type rm= str.rfind('P', 5);

cout << "Example - find() : The position (forward) of 'P' is: " << (int) m << endl;

cout << "Example - rfind(): The position (reverse) of 'P' is: " << (int) rm << endl;

string::size\_type n = str.find (" some", 0);

string::size\_type rn = str.rfind (" some", 0);

cout << "Example - find () : The position (forward) of 'some' is: " << (int) n << endl;

cout << "Example - rfind () : The position (reverse) of 'some' is: " << (int) rn << endl;

string::size\_type mo = str.find (" drugs", 0, 5);

string::size\_type rmo = str.rfind (" drugs", 0, 5);

cout << "Example - find(): The position (forward) of 'drugs' is: " << (int) mo << endl;

cout << "Example - rfind(): The position (reverse) of 'drugs' is: " << (int) rmo << endl;

string::size\_type no = str.find (str\_ch, 0);

string::size\_type rno = str.rfind(str\_ch, 0);

cout << "Example - find (): The position of 'for' is: " << (int) no << endl;

cout << "Example - rfind(): The position of 'for' is: " << (int) rno << endl;

cin.get ();

}

程序的运行结果为：

Example - find() : The position (forward) of 'P' is: 5  
Example - rfind(): The position (reverse) of 'P' is: 5  
Example - find () : The position (forward) of 'some' is: 35  
Example - rfind () : The position (reverse) of 'some' is: -1  
Example - find(): The position (forward) of 'drugs' is: 40  
Example - rfind(): The position (reverse) of 'drugs' is: -1  
Example - find (): The position of 'for' is: 46  
Example - rfind(): The position of 'for' is: -1

find\_first\_of()函数和 find\_last\_of()函数

find\_first\_of() 函数可实现在源串中搜索某字符串的功能，该函数的返回值是被搜索字符串的第 1 个字符第 1 次出现的下标（位置）。若查找失败，则返回 npos。  
  
find\_last\_of() 函数同样可实现在源串中搜索某字符串的功能。与 find\_first\_of() 函数所不同的是，该函数的返回值是被搜索字符串的最后 1 个字符的下标（位置）。若查找失败，则返回 npos。  
  
上述两个函数的原型分别为：

size\_type find\_first\_not\_of (value\_type\_Ch, size\_type\_Off = 0) const; size\_type find\_first\_of (const value\_type\* \_Ptr, size\_type \_Off = 0) const;  
size\_type find\_first\_of (const value\_type\* \_Ptr, size\_type\_Off, size\_type\_Count) const;  
size\_type find\_first\_of (const basic\_string & \_Str, size\_type\_Off = 0) const;  
size\_type find\_last\_of (value\_type \_Ch, size\_type\_Off = npos) const;  
size\_type find\_last\_of (const value\_type\* \_Ptr, size\_type\_Off = npos) const;  
size\_type find\_last\_of (const value\_type\* \_Ptr, size\_type \_Off, size\_type \_Count) const;  
size\_type find\_last\_of (const basic\_string& \_Str, size\_type\_Off = npos) const;

下面的程序示例详细阐述了 find\_first\_of() 函数和 find\_last\_of() 函数的使用方法。这两个函数和 find() 函数及 rfind() 函数的使用方法相同，具体参数的意义亦相同。

#include <iostream>

#include <string>

using namespace std;

int main ()

{

string str\_ch ("for");

string str("Hi, Peter, I'm sick. Please bought some drugs for me. ");

int length = str.length();

string::size\_type m = str.find\_first\_of ('P', 0);

string::size\_type rm = str.find\_last\_of ('P', (length - 1));

cout << "Example - find\_first\_of (): The position (forward) of 'P' is: " << (int) m << endl;

cout << "Example - find\_last\_of (): The position (reverse) of 'P' is： " << (int) rm << endl;

string:: size\_type n = str.find\_first\_of ("some", 0);

string:: size\_type rn = str.find\_last\_of ("some", (length -1));

cout << "Example - find\_first\_of(): The position (forward) of 'some' is: " << (int) n << endl;

cout << "Example - find\_last\_of(): The position (reverse) of 'some' is: " << (int) rn << endl;

string:: size\_type mo = str.find\_first\_of ("drugs", 0, 5);

string:: size\_type rmo = str.find\_last\_of ("drugs", (length-1), 5);

cout << "Example - find\_first\_of () : The position (forward) of 'drugs' is: " << (int) mo << endl;

cout << "Example - find\_last\_of () : The position (reverse) of 'drugs' is: " << (int) rmo << endl;

string::size\_type no = str.find\_first\_of (str\_ch, 0);

string::size\_type rno = str.find\_last\_of (str\_ch, (length -1));

cout << "Example - find\_first\_of() : The position of 'for' is: " << (int) no << endl;

cout << "Example - find\_last\_of () : The position of 'for' is: " << (int) rno << endl;

cin.get();

return 0;

}

程序执行结果：

Example - find\_first\_of (): The position (forward) of 'P' is: 4  
Example - find\_last\_of (): The position (reverse) of 'P' is： 21  
Example - find\_first\_of(): The position (forward) of 'some' is: 5  
Example - find\_last\_of(): The position (reverse) of 'some' is: 51  
Example - find\_first\_of () : The position (forward) of 'drugs' is: 8  
Example - find\_last\_of () : The position (reverse) of 'drugs' is: 48  
Example - find\_first\_of() : The position of 'for' is: 8  
Example - find\_last\_of () : The position of 'for' is: 48

find\_first\_not\_of()函数和 find\_last\_not\_of()函数

find\_first\_not\_of() 函数的函数原型为：

size\_type find\_first\_not\_of (value\_type \_Ch, size\_type\_Off = 0) const;  
size\_type find\_first\_not\_of (const value\_type \* \_Ptr, size\_type\_Off = 0) const;  
size\_type find\_first\_not\_of (const value\_type\* \_Ptr, size\_type\_Off, size\_type\_Count) const;  
size\_type find\_first\_not\_of (const basic\_string & \_Str, size\_type\_Off = 0) const;

find\_first\_not\_of() 函数可实现在源字符串中搜索与指定字符（串）不相等的第 1 个字符；find\_last\_not\_of() 函数可实现在源字符串中搜索与指定字符（串）不相等的最后 1 个字符。这两个函数的参数意义和前面几个函数相同，它们的使用方法和前面几个函数也基本相同。详见下面的程序：

[纯文本复制](http://c.biancheng.net/view/1453.html)

#include < iostream >

#include <string>

using namespace std;

int main ()

{

string str\_ch (" for");

string str ("Hi, Peter, I'm sick. Please bought some drugs for me.");

int length = str.length ();

string::size\_type m= str.find\_first\_not\_of ('P',0);

string::size\_type rm= str.find\_last\_not\_of ('P', (length -1);

cout << "Example - find\_first\_of (): The position (forward) of 'P' is: " << (int) m << endl;

cout << "Example - find\_last\_of (): The position (reverse) of 'P' is: " << (int) rm << endl;

string:: size\_type n = str.find\_first\_not\_of ("some", 0);

string:: size\_type rn = str.find\_last\_not\_of ("some", (length -1));

cout << "Example - find\_first\_of (): The position (forward) of 'some' is: " << (int) n << endl;

cout << "Example - find\_last\_of (): The position (reverse) of 'some' is: " << (int) rn << endl;

string:: size\_type mo = str.find\_first\_not\_of ("drugs", 0, 5);

string:: size\_type rmo = str.find\_last\_not\_of ("drugs", (length-1), 5);

cout << "Example - find\_first\_of (): The position (forward) of 'drugs' is: " << (int) mo << endl;

cout << "Example - find\_last\_of (): The position (reverse) of 'drugs' is: " << (int) rno << endl;

string::size\_type no = str.find\_first\_not\_of (str\_ch, 0);

string::size\_type rno = str.find\_last\_not\_of (str\_ch, (length-1));

cout << "Example - find\_first\_of (): The position of 'for' is: " << (int) no << endl;

cout << "Example - find\_last\_of () : The position of 'for' is: " << (int) rno << endl;

cin.get ();

return 0;

}

程序运行结果为：

Example - find\_first\_of (): The position (forward) of 'P' is: 0  
Example - find\_last\_of (): The position (reverse) of 'P' is: 52  
Example - find\_first\_of (): The position (forward) of 'some' is: 0  
Example - find\_last\_of (): The position (reverse) of 'some' is: 52  
Example - find\_first\_of (): The position (forward) of 'drugs' is: 0  
Example - find\_last\_of (): The position (reverse) of 'drugs' is: 52  
Example - find\_first\_of (): The position of 'for' is: 0  
Example - find\_last\_of () : The position of 'for' is: 52

本小节主要讲述 C++ STL 中的字符串查找函数。对于所述的 6 个查找函数，它们的使用形式大致相同，对于每个函数均配备了实例作为参考。请读者能认真对照例题，深刻理解这 6 个函数的使用方法，仔细体会函数每个参数的意义。

**12 C++ string支持迭代器方法详解**

理解迭代器是理解 [STL](http://c.biancheng.net/stl/) 的关键所在。模板使得算法独立于存储的数据类型，而迭代器使得算法独立于使用的容器类型。  
  
STL 定义了 5 种迭代器，根据所需的迭代器类型对算法进行描述。这 5 种迭代器分别是：输入迭代器、输出迭代器、正向迭代器、双向迭代器和随机访问迭代器。对于这 5 种迭代器不仅可以执行解除引用操作（\* 操作符），还可进行比较。本节主要讲述 basic\_string（或 string 类）中迭代器的使用。  
  
basic.string 和 string 类均提供了常规的迭代器和反向迭代器。string 是字符的有序群集。  
  
[C++](http://c.biancheng.net/cplus/) 标准库为 string 提供了相应接口，便于将字符串作为 STL 容器使用。可以使用迭代器遍历 string 内的所有字符；也可以使用其他算法遍历 string 内的所有字符。而且能够访问字符串中的每个字符，并对这些字符进行排序、逆向重排、找出最大值等操作。  
  
string 类的迭代器是随机存取迭代器，即支持随机存取。任何一个 STL 算法都可与其搭配使用。通常 string 的“迭代器型别”由 string class 本身定义，通常可以被简单地定义为一般[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank)。对迭代器而言，如果发生重新分配，或其所指的值发生某些变化时，迭代器会失效。  
  
string 类中和使用迭代器相关的成员函数是很多的，主要包括 begin()、end()、rbegin ()、rend()、append()、assign()、insert()、erase()、replace() 等。  
  
下面通过一个列子说明迭代器在 string 类中的使用方法。

#include <iostream>

#include <string>

#include <algorithm>

u[sin](http://c.biancheng.net/ref/sin.html)g namespace std;

int main ()

{

string s ("The zip code of Hondelage in Germany is 38108.");

cout << "Original: " << s << endl;

string sd(s.begin(),s.end ()); //构造函数中使用迭代器

cout << "Destination: " << sd << endl;

transform (sd.begin(), sd.end(), sd.begin(), [toupper](http://c.biancheng.net/ref/toupper.html)); //算法中使用迭代器(仿函数)

cout << "Destination (All Toupper)): " << sd << endl;

string sd1;

sd1.append (sd.begin(),(sd.end() -7)); //append()函数中使用迭代器

cout << "Destination sd1: " << sd1 << endl;

string sd2;

string::reverse\_iterator iterA;

string temp = "0";

for (iterA = sd.rbegin (); iterA != sd.rend (); iterA++) //reverse\_iterator

{

temp=\* iterA;

sd2.append (temp);

}

cout << "Destination sd2: " << sd2 << endl;

sd2.erase (0, 15); //erase()函数中使用迭代器

cout << "Destination sd2 (Erased 15 chars) : " << sd2 << endl;

string::iterator iterB = sd2.begin ();

string sd3 = string ("12345678");

sd2.insert (sd2.begin(), sd3.begin(), sd3.end()); //insert()函数中使用迭代器

cout << "Destination sd2 (Insert 8 chars) : " << sd2 << endl;

sd2.replace (sd2.begin (), sd2.end(), "This is an Exarrple of Replace"); //Replace

cout <<"Destination sd2 (Replace All): " << sd2 << endl; // replace ()函数中使用迭代器

}

程序运行结果为：

Original: The zip code of Hondelage in Germany is 38108.  
Destination: The zip code of Hondelage in Germany is 38108.  
Destination (All Toupper)): THE ZIP CODE OF HONDELAGE IN GERMANY IS 38108.  
Destination sd1: THE ZIP CODE OF HONDELAGE IN GERMANY IS  
Destination sd2: .80183 SI YNAMREG NI EGALEDNOH FO EDOC PIZ EHT  
Destination sd2 (Erased 15 chars) : EG NI EGALEDNOH FO EDOC PIZ EHT  
Destination sd2 (Insert 8 chars) : 12345678EG NI EGALEDNOH FO EDOC PIZ EHT  
Destination sd2 (Replace All): This is an Exarrple of Replace

13 C++ string支持配置器方法详解

配置器是 [STL](http://c.biancheng.net/stl/) 的重要内容。使用 STL 必然会涉及容器，而容器中存储了大量的数值，必然需要分配内存空间。配置器的作用就是为容器分配内存。  
  
配置器最早是为将内存模型抽象化而提出的。所以使用内存配置器分配内存时，是按对象的个数进行的，而不是按字节数。这有别于原来的 new [] 和 new 操作符。配置器最大的优点在于，配置器实现了将算法、容器与物理存储细节分隔。配置器可以提供一套分配与释放内存的标准方式，并提供用作[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank)类型和引用类型的标准名称。目前而言，配置器仅是一种纯粹的抽象。行为上类似分配器的类型都可看作配置器。  
  
[C++](http://c.biancheng.net/cplus/) STL 提供了标准分配器，目的是为用户提供更多的服务。basic\_string 模板以及 string 类均提供了对常见配置器的相关支持。basic\_string 类模板中包含 1 个配置器类型的成员 allocator\_type。对于 string 对象，allocator\_type 可以作为配置器类的对象使用；对 string 类而言，allocator\_type 等价于 allocator<char>，即分配数据类型为 char 的内存，便于 string 类的对象存储 char 型字符。  
  
同时 basic\_string 类模板的第 3 个参数也是配置器模板参数。basic\_string 类模板的形式如下：

template <class CharType, class Traits = char\_traits <CharType>, class Allocator=allocator<CharType>> class basic\_string

而 string 类的声明形式如下：

typedef basic\_string <char> string;

对于 basic\_string 类模板，其第 1 个参数是 CharType，第 2 个参数和第 3 个参数的默认值和 CharType 均相关。在声明 string 类时，参数 char 取代模板中的 CharType，string 即成为模板的实例，同时模板中的第 3 个参数成为 "class Allocator = allocator <char>"，其意义为 string 中对象的内存类型为 char 型。  
  
string 类还提供了 1 个和配置器相关的函数 get\_allocator()，其函数原型为：

allocator\_type string:: get\_allocator () const

函数返回 string 类的内存模型对象，可以用于构造新的字符串。以如下程序为例介绍该函数的使用方法。

#include <iostream>

#include <string>

#include <memory>

u[sin](http://c.biancheng.net/ref/sin.html)g namespace std;

int main ()

{

string s("abed");

basic\_string <char> s1 (s.get\_allocator());

basic\_string <char> :: allocator\_type aT = s1.get\_allocator ();

string::size\_type se = s1.size ();

cout << se << endl;

}

由于在 string 类中，allocator 是保护成员，难以直接调用对内存的直接配置。对于配置器，一般情况下都是使用默认配置器。对于本节内容，读者了解即可。