11

Strings
字符串

本章讲述 C++ 标准程序库中的 *string*（字符串）型别，包括针对基本 template class `basic_string<>` 及其标准特化型别 `string` 和 `wstring` 的详细内容。

String 可能是困惑的根源，因为这个字眼的确切含义比较模糊，可能是指一个型别为 `char*`（亦可加上 `const` 饰词）的字符数组（charater array），也可能是 class `string` 的一个实体，或泛指代表字符串的某个对象。本章之中我对术语 "*string*" 的定义是：*C++ 标准程序库中某个字符串型别（`string` 或 `wstring`）的对象*。至于一般字符串，也就是 `char*` 或 `const char*`，我用的术语是 "*C-string*"。

注意，字符串字面常数（例如 `"hello"`）会被转为 `const char*`。为了向下兼容，它们可以被隐式转换为 `char*`，不过这种转换并不值得赞赏。

11.1 动机

C++ 标准程序库中的 *string* class 使你可以将 *string* 当做一个一般型别而不会令用户感觉有任何问题。你可以像对待基本型别那样地复制、赋值和比较 string，再也不必担心内存是否足够、占用的内存实际长度等问题，只需运用操作符操作函数即可，例如以 `=` 进行赋值动作，以 `==` 进行比较动作，以 `+` 进行串联动作。简而言之，C++ 标准程序库对于 *string* 的设计思维就是，让它的行为尽可能像基本型别，不会在操作上引起什么麻烦（至少原则如此）。现今世界的数据处理大部分就是字符串处理，所以对于从 C, Fortran 或类似语言一路走来的程序员而言，这是非常重要的进步，因为 *string* 在那些语言中往往是烦恼之源。

下面数节给了两个例子，展示 *string* class 的能力和用法。不以实用为目的，而是以示范为目的。

11.1.1 例一：引出一个临时文件名

第一个例子通过命令行（command line）参数产生一个临时文件名。如果你这么启动该程序：

```
string1 prog.dat mydir hello. oops.tmp end.dat
```

输出如下：

```
prog.dat => prog.tmp
mydir => mydir.tmp
hello. => hello.tmp
oops.tmp => oops.xxx
end.dat => end.tmp
```

通常产生的扩展名是.tmp，但如果原本的扩展名就是.tmp，则换成.xxx。

程序如下：

```
// string/string1.cpp

#include <iostream>
#include <string>
using namespace std;

int main (int argc, char* argv[])
{
    string filename, basename, extname, tmpname;
    const string suffix("tmp");

    /* for each command-line argument
    * (which is an ordinary C-string)
    */
    for (int i=1; i<argc; ++i) {
        // process argument as file name
        filename = argv[i];

        // search period in file name
        string::size_type idx = filename.find('.');
        if (idx == string::npos) {
            // file name does not contain any period
            tmpname = filename + '.' + suffix;
        }
        else {
```

```
            /* split file name into base name and extension
            * - base name contains all characters before the period
            * - extension contains all characters after the period
            */
            basename = filename.substr(0, idx);
            extname = filename.substr(idx+1);
            if (extname.empty()) {
                // contains period but no extension: append tmp
                tmpname = filename;
                tmpname += suffix;
            }
            else if (extname == suffix) {
                // replace extension tmp with xxx
                tmpname = filename;
                tmpname.replace (idx+1, extname.size(), "xxx");
            }
            else {
                // replace any extension with tmp
                tmpname = filename;
                tmpname.replace (idx+1, string::npos, suffix);
            }
        }

        // print file name and temporary name
        cout << filename << " => " << tmpname << endl;
    }
}
```

首先：

```
#include <string>
```

用来将定义有标准 C++ *string* 类别(s) 的头文件含入。一如惯例，这些类别被声明在 `std` 命名空间中。

以下声明式会产生四个 *string* 变量：

```
string filename, basename, extname, tmpname;
```

既然没有传入参数，它们均使用 *string* 的 default 构造函数，这会使它们被初始化为空字符串。

以下声明式产生一个 string 常数 `suffix`：

```
const string suffix("tmp");
```

在本程序中，它被用来作为临时文件名的正常后缀。此字符串以一个 *C-string* 初始化，其值为 tmp。注意，在几乎所有“两个 strings 对象的组合操作”场合中，你可以改而使用“一个 *C-string* 和一个 string 对象”来进行组合操作。例如在这个程序里，你可以直接以 "tmp" 取代所有的 suffix。

每当 for 循环被执行一次，其中的语句：

```
filename = argv[i];
```

对字符串变量 filename 赋予一个新值。本例中这个新值是个 *C-string*，不过你也可以把另一个 string 对象或型别为 char 的单个字符赋值给它。

以下语句：

```
string::size_type idx = filename.find('.');
```

在字符串 filename 中搜寻第一个 '.' 字符。有好几个函数可以在字符串内实施搜寻功能，函数 find() 是其中之一。你也可以从后向前搜寻，或是搜寻子字符串，或是在字符串的某个范围内搜寻，或是同时搜寻数个字符。所有这些搜寻函数都返回第一个匹配位置（一个索引）。没错，返回值是个整数，而不是迭代器。字符串的一般接口并不依赖 STL 概念。然而字符串的确也提供了数种迭代器（参见 11.2.12 节, p497）。所有搜寻函数的返回型别都是 string::size_type，这是 string class 定义的一个无正负号整数型别[1]。当然啦，第一个字符的索引值为 0，最后一个字符的索引值是 numberOfCharacters-1。注意，numberOfCharacters 并不是一个有效索引。和 *C-string* 不同，string 对象的字符串尾部并没有一个特殊字符 '\0'。

如果搜寻失败，必须返回一个特殊值来表示，该值就是 npos，定义于 *string* class 中。所以下面这一行可用来检验搜寻动作是否失败：

```
if (idx == string::npos)
```

请特别注意，当你打算检验搜寻函数的返回值时，应该使用 string::size_type 型别而不是 int 或 unsigned。否则上述与 string::npos 的比较动作将无法有效运行。细节见 11.2.12 节, p495。

本例如果搜寻失败，表示该文件没有扩展名。此时，文件名应该是原文件名之后接上一个句点，再接上先前定义的后缀：

```
tmpname = filename + '.' + suffix;
```

你可以用 operator+ 直接串连两个字符串。也可以将字符串与 *C-string* 以及单个字符串联起来。

---

[1] 更明确地说，字符串的 size_type 型别系根据字符串类别的内存模型而定。见 11.3.12 节, p526。

如果找到 '.'，else 部分就要发挥作用了。这里，句点所在的位置（索引）用来将文件名分割成主文件名和扩展名，由成员函数 substr() 完成：

```
basename = filename.substr(0, idx);
extname = filename.substr(idx+1);
```

substr() 的第一参数是起点索引，可有可无的第二参数是字符个数（而非终点索引）。如果没有指定第二参数，那么所有剩余字符都将被视为子字符串返回。

凡是“以一个索引和一个长度作为参数”的地方，字符串行为遵循下面两项规则：

1. 索引值必须合法。该值必须小于字符串的字符个数（第一字符的索引是 0）。最后一个字符的下一个位置（的索引值）可用来标明结束位置。

   大部分情况下，如果你指定的索引超过实际字符数，会引发 out_of_range 异常。不过，用以搜寻单一字符或某个位置的所有搜寻函数，均可接受任意索引。如果索引超过实际字符数，这些函数会返回 string::npos（表示没找到）。

2. 字符数量（长度）可为任意值。如果其值大于实际剩余的字符数，则这些剩余字符都会被用到。如果使用 string::npos，相当于指明“剩余所有字符”。

所以，如果找不到 '.'，以下表达式会抛出一个异常：

```
filename.substr(filename.find('.'))
```

但如果找不到 '.'，以下表达式不会抛出异常：

```
filename.substr(0, filename.find('.'))
```

而是会返回整个文件名。

即使找到了 '.'，如果其后没有任何字符，substr() 返回的扩展名将为空。下面这个式子对这种情况实施检查：

```
if (extname.empty())
```

如果这个条件式得到 true，产生的临时文件名将由其主文件名加上一般扩展名组成：

```
tmpname = filename;
tmpname += suffix;
```

这里，operator+= 的作用是在尾端附加扩展名。

原扩展名可能就是临时文件的标准扩展名。我们可以用 operator== 来比较这两个字符串：

```
if (extname == suffix)
```

如果比较结果是 true，则以 "xxx" 作为临时扩展名：

```
tmpname = filename;
tmpname.replace (idx+1, extname.size(), "xxx");
```

其中的：

```
extname.size()
```

返回字符串 extname 的字符个数。你也可以使用 length()获得相同结果，其行为和 size()完全一样。是的，size()和 length()都返回字符串的字符个数。注意，size()的结果和字符串实际占用的内存无关[2]。

当所有特殊情况都照顾到了之后，接下来就可以进行常规处理了。程序以标准扩展名替换原文件的扩展名：

```
tmpname = filename;
tmpname.replace (idx+1, string::npos, suffix);
```

此处的 string::npos 表示“剩余的所有字符”。所以句点之后的所有字符被替换为 suffix。这一替换对于“句号之后为空字符串”的文件名同样有效，相当于以 suffix 来替换“无物”。

以下语句输出原文件名和新产生的临时文件名。看，你可以使用一般的 stream output 操作符来打印字符串（惊讶吧）：

```
cout << filename << " => " << tmpname << endl;
```

11.1.2 例二：引出一段文字并逆向打印

第二个例子从标准输入装置取得一个个英文单词，然后将其中各个字符（字母）逆序印出。单词和单词之间以一般空格符（换行符号 newline、空格符 space 或定位符号 tab）或逗号、句号、分号分隔开来。

```
// string/string2.cpp

#include <iostream>
#include <string>
using namespace std;

int main (int argc, char** argv)
{
    const string delims(" \t,.;");
    string line;

    // for every line read successfully
    while (getline(cin,line)) {
```

[2] 这里出现的两个成员函数依不同的设计原则执行了相同的动作。length()传回字符串长度，就好像 *C-strings* 以 strlen()所得结果一样。size()则是根据 STL 习惯而设的成员函数，用来表明元素数量。

```
        string::size_type begIdx, endIdx;

        // search beginning of the first word
        begIdx = line.find_first_not_of(delims);

        // while beginning of a word found
        while (begIdx != string::npos) {
            // search end of the actual word
            endIdx = line.find_first_of (delims, begIdx);
            if (endIdx == string::npos) {
                // end of word is end of line
                endIdx = line.length();
            }

            // print characters in reverse order
            for (int i=endIdx-1; i>=static_cast<int>(begIdx); --i) {
                cout << line[i];
            }
            cout << ' ';

            // search beginning of the next word
            begIdx = line.find_first_not_of (delims, endIdx);
        }
        cout << endl;
    }
}
```

本程序中，所有间隔字符被定义于一个特殊的字符串常数中：

```
const string delims(" \t,.;");
```

换行符号也是一个间隔字符，但这里不必特别在意，因为程序本身就是一行一行地读取标准输入装置。

外层循环不断地将新行读入字符串 line 之中：

```
string line;
while (getline(cin,line)) {
   ...
}
```

getline()是一个可以从 stream 读取字符串的特殊函数，它逐字读取，直到一行结束（通常以换行符号作为标示）。换行符号可以由你自定，你可以将自己喜欢的换行符号作为第三参数传入，令 getline()在该符号所区隔出来的各个语汇单元（token）中一一读取。

在外层循环内，各个单词分别被搜寻和打印。第一个语句：

```
begIdx = line.find_first_not_of(delims);
```

搜寻第一个单词的起始位置。函数 find_first_not_of()返回“不隶属参数所指字符串”的第一个字符的索引，所以本例返回第一个“分隔符（delims）以外”的字符。和一般搜寻函数一样，如果没有找到匹配字符，就返回 string::npos。

如果找到了一个单字，就进入内层循环：

```
while (begIdx != string::npos) {
   ...
}
```

内层循环的第一个语句用来搜寻当前单词的结尾：

```
endIdx = line.find_first_of (delims, begIdx);
```

函数 find_first_of()用来搜寻“第一参数所指字符串内的任何字符”的第一次出现位置。可有可无的（可选的）第二参数用来标示搜寻起点。上述动作会找到单词后的第一个分隔符。

如果没找到，就将 endIdx 设定为“行结束标记（*end-of-line*）”：

```
if (endIdx == string::npos) {
    endIdx = line.length();
}
```

这里使用 length()，效果和 size()相同：返回字符个数。

以下语句将所有字符逆序打印出来：

```
for (int i=endIdx-1; i>=static_cast<int>(begIdx); --i) {
     cout << line[i];
}
```

是的，你可以使用 oprator[] 访问字符串的个别字符。请注意，这个操作符并不检查索引是否合法。所以你必须确保索引的合法性（一如上例）。比较安全的做法是利用成员函数 at()来存取字符。不过这种方式会带来效率上的负担，所以一般还是采取不检查态度。

另一个讨厌的问题由字符串索引造成。如果你忘记将 begIdx 转型为 int，程序可能会陷入无穷循环中，甚至崩溃掉。和第一个例子一样，这里的 string::size_type 是一个不带正负号的整数型别。如果不转型，当带正负号的 i 与不带正负号的型别进行比较时，i 会被自动转型为无正负号值，于是如果目前处

理的这个单字位于一行的开头，则表达式：

```
i >= begIdx
```

的结果永远为 true——因为 begIdx 为 0，而任何无正负号值都大于等于 0。这将引发无穷循环，直到存取某个非法地址而崩溃。

因为这个缘故，我并不欣赏 string::size_type 和 string::npos 的设计思维。11.2.12 节, p496 有一个比较安全的方案，但也并非完美。

内层循环中的最后一个语句重新将 begIdx 初始化，使它标示下一个单词的起点（如果有下一个单词的话）：

```
begIdx = line.find_first_not_of (delims, endIdx);
```

和先前对 find_first_not_of()的调用不同，这里把上一个单词的终点当做搜寻起点。如果上一个单词是该行的最后一个单词，则 endIdx 就是该行的终点。这意味着搜寻将从字符串终点开始，结果当然是返回 string::npos。

让我们来试试这个“有用而且重要”☺ 的程序。下面是输入：

```
pots & pans
I saw a reed
```

输出如下：

```
stop & snap
I was a deer
```

如果能找到更多有趣的例子，我很愿意在本书下一版收录进来。

11.2 String Classes 细部描述

11.2.1 String 的各种相关型别

表头文件

表头文件 <string> 定义了所有的字符串型别和函数。

```
#include <string>
```

一如既往，所有标识符都定义于命名空间 std 之中。

模板类别（template class）basic_string<>

在 <string> 之中，basic_string<> 被定义为所有字符串型别的基本模板类别（basic template class）：

```
namespace std {
   template<class charT,
            class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
   class basic_string;
}
```

此一类别将字符型别、字符型别特性（traits）、内存模型（memory model）加以参数化：

- 第一参数是单个字符所属型别（译注：也许是 ASCII 字符或 Unicode 字符…）。
- 带默认值的第二参数是个特性类别（traits class），提供字符串类别中所有的字符核心操作。此种特性类别规定了“复制字符”或“比较字符”的做法（详见 p687, 14.1.2 节），如果没有指定它，就会根据现有的字符型别采用缺省的特性类别。p503, 11.2.14 节实作出一个使用者自定的特性类别，让字符串以“不分大小写”的方式进行各种操作。
- 带默认值的第三参数定义了字符串类别所采用的内存模式，通常设定为“缺省的内存模型 allocator”（详见 p31, 3.4 节和第 15 章）[3]。

string 型别和 wstring 型别

C++ 标准程序库提供了两个 basic_string<> 特化版本：

1. string 是针对 char 而预先定义的特化版本：
```
namespace std {
   typedef basic_string<char> string;
}
```
2. wstring 是针对 wchar_t 而预先定义的特化版本：
```
namespace std {
   typedef basic_string<wchar_t> wstring;
}
```
如此一来你就可以使用宽字符集，例如 Unicode 或某些亚洲字符集（国际化议题详见第 14 章）。

以下数节的讨论并不区分如上所述不同的字符串类型。由于所有字符串类型都采用相同接口，所以用法和问题都一样。我将以 "*string*" 表示任何字符串型别，包括 string 和 wstring。由于一般软件开发大多顺应欧美环境，所以本书的例子大多采用 string 型别。

---

[3] 如果你的系统不支持 default template parameters（缺省模板参数），则第三参数通常会被省略。

11.2.2 操作函数（Operations）综览

表 11.1 列出针对字符串而设计的所有操作函数。

表 11.1 字符串的各种操作函数

操作函数（Operation）	效果（Effect）
构造函数（*constructors*）	产生或复制字符串
析构函数（*destructors*）	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（使之为空）
resize()	改变字符数量（在尾端删除或添加字符）
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()	保留一定量内存以容纳一定数量的字符
[ ], at()	存取单一字符
>>, getline()	从 stream 中读取某值
<<	将某值写入 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（character array）形式返回
substr()	返回某个子字符串（substring）
搜寻函数（*find functions*）	搜寻某个子字符串或字符
begin(), end()	提供正常的（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（*allocator*）

字符串操作函数的参数

STL 提供了很多字符串操作函数。其中许多往往具有数个重载版本，分别以一个、两个或三个参数来指定新值。表 11.2 整理出所有字符串操作的参数规格。

表 11.2 字符串操作函数的参数规格

参数（Arguments）	含义
`const string & str`	整个 `str` 字符串
`const string & str, size_type idx, size_type num`	大部分情况下是指字符串 `str` 中以 `idx` 开始的 `num` 个字符
`const char* cstr`	整个 C-string `cstr`
`const char* chars, size_type len`	字符数组 `chars` 中的 `len` 个字符
`char c`	字符 `c`
`size_type num, char c`	`num` 个字符 `c`
`iterator beg, iterator end`	区间 `[beg;end)` 内所有字符

注意，只有在单参数版本中，才将 `char*` 字符`'\0'` 当做字符串结尾特殊符号来处理，其它所有情况下 `'\0'` 都不被视为特殊字符：

```
std::string s1("nico"); // initializes s1 with: 'n' 'i' 'c' 'o'
std::string s2("nico",5); // initializes s2 with: 'n' 'i' 'c' 'o' '\0'
std::string s3(5,'\0'); // initializes s3 with: '\0' '\0' '\0' '\0' '\0'
s1.length()  // yields 4
s2.length()  // yields 5
s3.length()  // yields 5
```

因此，一般而言一个字符串内可以包含任何字符，甚至可以包含二进制文件的内容。

至于各个操作函数对应何种参数，请见表 11.3。所有操作符都只能把对象当做单一值来处理，因此如果要赋值、比较或添加 *string*（或 *C-string*）的一部分，就必须采用相应的函数。

未提供的操作函数

C++ 标准程序库的 *string* class 并没有解决所有可能遇到的字符串问题。事实上它并没有提供下列问题的解决之道：

- 正则表达式（Regular expressions）
- 文本处理（例如大写化、大小写不计的字符串比较动作等等）

不过，文本处理方面的问题不大，参见 11.2.13 节, p497 的例子。

表 11.3 拥有字符串参数的各种操作函数

	Full String	Part of String	C-string (char*)	char Array	Single char	num chars	Iterator Range
constructors	Yes	Yes	Yes	Yes	—	Yes	Yes
=	Yes	—	Yes	—	Yes	—	—
assign()	Yes	Yes	Yes	Yes	—	Yes	Yes
+=	Yes	—	Yes	—	Yes	—	—
append()	Yes	Yes	Yes	Yes	—	Yes	Yes
push_back()	—	—	—	—	Yes	—	—
insert(), index version	Yes	Yes	Yes	Yes	—	Yes	—
insert(), iterator version	—	—	—	—	Yes	Yes	Yes
replace(), index version	Yes	Yes	Yes	Yes	Yes	Yes	—
replace(), iterator version	Yes	—	Yes	Yes	—	Yes	Yes
find functions	Yes	—	Yes	Yes	Yes	—	—
+	Yes	—	Yes	—	Yes	—	—
==, !=, <, <=, >, >=	Yes	—	Yes	—	—	—	—
compare()	Yes	Yes	Yes	Yes	—	—	—

11.2.3 构造函数和析构函数（Constructors and Destructors）

表 11.4 列出 *strings* 的所有构造函数和析构函数。本节将一一介绍它们。“利用迭代器指出的区间进行初始化”的构造函数将在第 11.2.13 节, p497 介绍。

表 11.4 strings 的构造函数和析构函数

表达式	效果
strings	生成一个空字符串 s
string s(str)	copy 构造函数，生成字符串 str 的一个复制品
string s(str,stridx)	将字符串 str 内“始于位置 stridx”的部分，当做字符串 s 的初值。
string s(str,stridx,strlen)	将字符串 str 内“始于位置 stridx 且长度顶多 strlen”的部分，当做字符串 s 的初值。
string s(cstr)	以 C-string cstr 作为字符串 s 的初值。
string s(chars,chars_len)	以 C-string cstr 的前 chars_len 个字符作为字符串 s 的初值。
string s(num,c)	生成一个字符串，包含 num 个 c 字符
string s(beg,end)	以区间[beg;end)内的字符作为字符串 s 的初值。
s.~string()	销毁所有字符，释放内存

注意，你不能以单一字符来初始化某个字符串，但是你可以这么做：

```
std::string s('x');          // ERROR
std::string s(1,'x'); // OK, creates a string that has one character 'x'
```

这表示编译器提供了一个从 const char* 到 string 的自动型别转换功能，但不存在一个从 char 到 string 的自动型别转换功能。

11.2.4 Strings 和 C-Strings

C++ *Standard* 将字符串字面常数（string literals）的型别由 char* 改为 const char*。为了提供向下兼容性，C++ *Standard* 规定了一个颇有争议的隐式转换，可从 const char* 隐式转为 char*。由于字符串字面常数的型别并非 string，因此在新的 string object 和传统的 *C-strings* 之间必须存在一种强烈关系：在“*strings* 和 *string*-like object 共通的操作场合”（例如比较、追加、插入等等动作）都应该可以使用 *C-strings*。或者具体地说，存在一个从 const char* 到 strings 的隐式型别转换。然而却不存在一个从 *string* object 到 *C-string* 的自动型别转换。这是出于安全考虑，防止意外转型导致奇异行为（char* 经常有奇异的行为）和模棱两可（例如在一个结合了 string 和 *C-string* 的表达式中，既可以把 string 转化为 char*，也可以反其道而行，这就导致模棱两可）。有好几种办法可以产生或改写/复制 *C-string*。更明确地说，c_str()可以得到“*string* 对应的 *C-string*”，所得结果和“以 '\0' 为结尾的字符数组一样。运用 copy()，你也可以将字符串内容复制或写入既有的 *C-string* 或字符数组内。

请注意，'\0' 在 *string* 之中并不具有特殊意义，但在一般 *C-string* 中却用来标识字符串结束。在 *string* 中，字符 '\0' 和其它字符的地位完全相同。

请注意，千万不要以 null 指标（NULL）取代 char* 作为参数，这样会导致奇异行为，因为 NULL 具有整数型别，在单整数型别的重载函数版本上会被解释为数字 0 或“其值为 0”的字符。

有三个函数可以将字符串内容转换为字符数组或 *C-String*：

1. data() 以字符数组的形式返回字符串内容。由于并未追加 '\0' 字符，所以返回型别并非有效的 *C-string*。
2. c_str() 以 *C-string* 形式返回字符串内容，也就是在尾端添加 '\0' 字符。
3. copy() 将字符串内容复制到“调用者提供的字符数组”中。不添加 '\0' 字符。

注意，data()和 c_str()返回的字符数组由该字符串拥有。也就是说调用者千万不可修改它或释放其内存。例如：

```
std::string s("12345");

atoi(s.c_str())             // convert string into integer
f(s.data(),s.length())      // call function for a character array
                            // and the number of characters

char buffer[100];
s.copy(buffer,100);    // copy at most 100 characters of s into buffer
s.copy(buffer,100,2); // copy at most 100 characters of s into buffer
                      // starting with the third character of s
```

一般而言，整个程序中你应该坚持使用 *strings*，直到你必须将其内容转化为 char*时才把它们转换为 *C-string*。请注意 c_str()和 data()的返回值有效期限在下一次调用 non-const 成员函数时即告终止。

```
std::string s;
...
foo(s.c_str());  // s.c_str() is valid during the whole statement

const char* p;
p = s.c_str();   // p refers to the contents of s as a C-string p
foo(p);          // OK (p is still valid)
s += "ext";      // invalidates p
foo(p);          // ERROR: argument p is not valid
```

11.2.5 大小（Size）和容量（Capacity）

为了高效无误地运用 *strings*，你应该理解 *strings* 的大小和容量是如何配合的。一个 *strings* 存在三种“大小”：

1. size()和 length()

   返回 *string* 中现有的字符个数。上述两个函数等效[4]。

   成员函数 empty()用来检验字符数是否为 0，亦即字符串是否为空。你应该优先使用该函数，因为它比 length()或 size()来得快。

[4] 这里两个成员函数所做的事情相同。根据 STL 概念，size()是获取容器元素个数的通用成员函数，length()则对应于一般 C-string strlen()函数，传回字符串长度。

2. `max_size()`

   此函数返回一个 *string* 最多能够包含的字符数。一个 *string* 通常包含一块单独内存区块内的所有字符，所以可能跟 PC 机器本身的限制有关系。返回值一般而言是索引型别的最大值减 1。之所以“减 1”有两个原因：(a) 最大值本身是 `npos`；(b) 具体实作中，可因此轻易在内部缓冲区之后添加一个`'\0'`，以便将这个 *string* 当做 *C-string* 使用（例如透过 `c_str()`）。一旦某个操作函数使用一个长度大于 `max_size()` 的 *string*，length_error 异常就会被抛出来。

3. `capacity()`

   重新分配内存之前，*string* 所能包含的最大字符数。

让 *string* 拥有足够的容量是很重要的，原因有二：

1. 重新分配会造成所有指向 *string* 的 references、pointers 和 iterators 失效。
2. 重新分配（reallocation）很耗时间。

因此，如果程序要用到指向 *string*（或其内部字符）的 references、pointers 和 iterators，抑或需要很快的执行速度，就必须考虑容量（capacity）问题。成员函数 `reserve()` 就是用来避免重分配行为。`reserve()` 使你得以预留一定容量，并确保该容量尚有余裕之时，reference 能够一直保持有效：

```
std::string s; // create empty string
s.reserve(80); // reserve memory for 80 characters
```

容量概念应用于 *string* 和应用于 vector 是相同的（参见第 6.2.1 节, p149）：但有一个显著差异：面对 *string* 你可以调用 `reserve()` 来缩减实际容量，而 vector 的 `reserve()` 却没有这项功能。拿一个“小于现有容量”的参数来调用 `reserve()`，实际上就是一种非强制性缩减请求（nonbinding shrink request）— 如果参数小于现有字符数，则这项请求被视为非强制性适度缩减请求（nonbinding shrink-to-fit request）。也就是说你可能想要缩减容量至某个目标，但不保证你一定可以如愿。*String* 的 `reserve()` 参数默认值为 0，所以调用 `reserve()` 并且不给参数，就是一种“非强制性适度缩减请求”：

```
s.reserve(); // "would like to shrink capacity to fit the current size"
```

为什么缩减动作是非强制性的呢？因为“如何获取最佳性能”系由实作者定义。具体实作 *string* 时，如何处理速度和内存耗用量之间关系可能有不同的设计思路。因此任何实作作品都可以以较大的魄力增加容量，并且永不缩减。

C++ *Standard* 规定，唯有在响应 `reserve()` 调用时，容量才有可能缩减。因此即使发生“字符被删除或被改变”的事情，任何其它字符只要位于“被操作字符”之前，指向它们身上的那些 references、pointers 和 iterators 就仍然保持有效。

11.2.6 元素存取（Element Access）

String 允许我们对其所包含的字符进行读写。有两种方法可以访问单一字符：subscript（下标）操作符[]和成员函数 at()。两者都返回某指定索引的对应位置上的字符。一如既往，第一个字符索引为 0，最后的字符索引为 length()-1。但是请注意以下区别：

- operator[]并不检查索引是否有效，at()则会检查。如果调用 at()时指定的索引无效，系统会抛出 *out_of_range* 异常。如果调用 operator[]时指定的索引无效，其行为未有定义——可能存取非法内存，因而引起某些讨厌的边缘效应或甚至崩溃（崩溃了还算运气好，因为你好歹知道出错了）。
- 对于 operator[]的 const 版本，最后一个字符的后面位置也是有效的。此时的实际字符数是有效索引。在此情况下 operator[]的返回值是“由 char 型别之 default 构造函数所产生”的字符。因此，对于型别为 string 的对象，返回值为 '\0' 字符。

  其它任何情况（包括成员函数 at()和 operator[]的 non-const 版本），实际字符数都是个无效索引。如果使用该索引，会引发异常，或导致未定义行为。

  举个例子：

```
const std::string cs("nico");   // cs contains: 'n' 'i' 'c' 'o'
std::string s("abcde");         // s contains: 'a' 'b' 'c' 'd' 'e'

s[2]          // yields 'c'
s.at(2)       // yields 'c'

s[100]        // ERROR: undefined behavior
s.at(100)     // throws out_of_range

s[s.length()]          // ERROR: undefined behavior
cs[cs.length()]        // yields '\0'
s.at(s.length())       // throws out_of_range
cs.at(cs.length())     // throws out_of_range
```

为了允许更改 *string* 内容，operator[]的 non-const 版本和 at()都返回字符的 reference。一旦发生重分配行为，那些 reference 立即失效：

```
std::string s("abcde"); // s contains: 'a' 'b' 'c' 'd' 'e'

char& r = s[2];  // reference to third character
char* p = &s[3]; // pointer to fourth character
```

```
r = 'X';                // OK, s contains: 'a' 'b' 'X' 'd' 'e'
*p = 'Y';               // OK, s contains: 'a' 'b' 'X' 'Y' 'e'

s = "new long value"; // reallocation invalidates r and p

r = 'X';                // ERROR: undefined behavior
*p = 'Y';               // ERROR: undefined behavior
```

在这里，为了避免运行时出错，我们应该在 r 和 p 被初始化之前，先运用 reserve() 保留足够的容量。

以下操作可能导致指向字符的 references 和 pointers 失效：

- 以 swap() 交换两值
- 以 operator>>() 或 getline() 读入新值
- 以 data() 或 c_str() 输出内容
- 调用 operator[],at(),begin(),rbegin(),end() 或 rend() 之外的任何 non-const 成员函数
- 调用任何函数并于其后跟着 operator[],at(),begin(),rbegin(),end() 或 rend()。

以上讨论同样适用于迭代器（参见 11.2.13 节，p497）。

11.2.7 比较（Comparisons）

Strings 支持常见的比较（comparison）操作符，操作数可以是 *strings* 或 *C-strings*：

```
std::string s1, s2;
...

s1 == s2     // returns true if s1 and s2 contain the same characters
s1 < "hello" // return whether s1 is less than the C-string "hello"
```

如果以 <, <=, >, >= 来比较 *strings*，得到的结果是根据“当前字符特性（current character traits）”将字符依字典顺序逐一比较。例如以下所有比较结果均为 true：

```
std::string("aaaa") < std::string("bbbb")
std::string("aaaa") < std::string("abba")
std::string("aaaa") < std::string("aaaaaa")
```

你可以使用成员函数 compare() 来比较子字符串，此函数针对一个 *string* 可使用多个参数进行处理，如此一来就可以采用索引和长度，双管齐下定位出子字符串。请注意 compare() 返回的是整数值而非布尔值。返回值意义如下：0 表示相等，小于 0 表示小于，大于 0 表示大于。例如：

```
std::string s("abcd");

s.compare("abcd")      // returns 0
s.compare("dcba")      // returns a value < 0 (s is less)
s.compare("ab")        // returns a value > 0 (s is greater)

s.compare(s)           // returns 0 (s is equal to s)
s.compare(0,2,s,2,2)   // returns a value < 0 ("ab" is less than "cd")
s.compare(1,2,"bcx",2) // returns 0 ("bc" is equal to "bc")
```

如果想采用不同的比较准则，你也可以自己定义，并采用 STL 的比较算法（参见 11.2.13 节, p499 示例），或使用特殊的字符特性（character traits）完成“不计大小写”的比较动作。不过由于“具备特殊 traits class”的 *string* 是另一个不同的数据型别，所以不能将这种 *string* object 拿来和 string object 共同处理。参见第 11.2.14 节, p503 示例。

针对国际市场而制作的程序，可能需要按特殊的当地（国别, locale）规则来比较字符串。为简化这个问题，locale class 提供了圆括号操作符（参见 p703），采用 string collation facet，根据对应的当地惯例来比较字符串，从而进行排序。详见 14.4.5 节, p724。

11.2.8 更改内容（Modifiers）

你可以运用不同的成员函数或操作符来更改字符串内容。

赋值（Assignments）

可运用 operator= 来对字符串赋新值。新值可以是 *string*、*C-string* 或单一字符。如果需要多个参数来描述新值，可采用成员函数 assign()。举个例子：

```
const std::string aString("othello");
std::string s;

s = aString;           // assign "othello"
s = "two\nlines";      // assign a C-string
s = ' ';               // assign a single character

s.assign(aString);     // assign "othello" (equivalent to operator =)
s.assign(aString,1,3);     // assign "the"
s.assign(aString,2,std::string::npos);   // assign "hello"

s.assign("two\nlines");
        // assign a C-string (equivalent to operator =)
s.assign("nico",5);
        // assign the character array: 'n' 'i' 'c' 'o' '\0'
s.assign(5,'x');
```

```
// assign five characters: 'x' 'x' 'x' 'x' 'x'
```

也可以运用两个迭代器定义出来的字符区间进行赋值动作，详见 11.2.13 节, p497。

交换（Swapping Values）

和许多“非寻常的（nontrivial）”型别一样，string 型别提供了一个特殊的 swap() 函数，用来交换两字符串内容(4.4.2 节, p67 介绍全局函数 swap())。这个用于 *strings* 的特殊 swap()保证常数复杂度，所以如果赋值之后不再需要旧值，你应该利用它进行交换，从而达成赋新值的目的。

令 Strings 成空

许多动作都可以令字符串成为空字符串，例如：

```
std::string s;

s = "";      // assign the empty string
s.clear();   // clear contents
s.erase();   // erase all characters
```

安插（Inserting）和移除（Removing）字符

Strings 提供许多成员函数用于安插（insert）、移除（remove）、替换（replace）、擦除（erase）字符。另有 operator+=, append()和 push_back()可添加字符。下面是个实例：

```
const std::string aString("othello");
std::string s;

s += aString;          // append "othello"
s += "two\nlines";     // append C-string
s += '\n';             // append single character

s.append(aString);     // append "othello" (equivalent to operator +=)
s.append(aString,1,3); // append "the"
s.append(aString,2,std::string::npos); // append "hello"

s.append("two\nlines"); // append C-string (equivalent to operator +=)
s.append("nico",5);    // append character array: 'n' 'i' 'c' 'o' '\0'
s.append(5,'x');       // append five characters: 'x' 'x' 'x' 'x' 'x'

s.push_back('\n'); // append single character (equivalent to operator +=
```

operator+= 将单一参数添加在 *string* 尾部，append()可使用多个参数指定添加值。还有一个 append()版本可以将两个迭代器指定的字符区间添加在 *string* 尾部（参见 11.2.13 节，p497）。push_back()是为了支持 back inserters 而设，STL 算法可由此往 *string* 尾部添加字符（关于 back inserters，详见 7.4.2 节，p272。11.2.13 节，p502 有其运用实例）。

和 append()类似，成员函数 insert()也允许你安插字符。使用 insert()时需知道安插位置的索引，新字符将安插于此位置之后。

```
const std::string aString("age");
std::string s("p");

s.insert(1,aString);        // s: page
s.insert(1,"ersifl"); // s: persiflage
```

注意，成员函数 insert()不接受“索引 + 单独字符”的参数组合，所以你必须传入一个 *string* 或一个额外数字：

```
s.insert(0,' ');        // ERROR
s.insert(0," ");        // OK
```

也许你还想试试这样：

```
s.insert(0,1,' ');      // ERROR: ambiguous
```

然而由于 insert()具有以下重载形式，导致令人厌烦的模棱两可现象：

```
insert (size_type idx, size_type num, charT c); // position is index
insert (iterator pos, size_type num, charT c); // position is iterator
```

string 的 size_type 通常被定义为 unsigned，string 的 iterator 通常被定义为 char*。这种情况下，第一个参数 0 有两种转换可能，不分优劣。为了获得正确操作，你必须如此：

```
s.insert((std::string::size_type)0,1,' ');     // OK
```

刚才提到的模棱两可的第二形式，恰恰给出了一个运用迭代器来安插字符的范例。如果你想运用迭代器来指定安插位置，有三种情况：安插一个字符、安插多个相同字符、安插“两个迭代器所指区间”内的字符（参见 11.2.13 节，p497）。

和 append()及 insert()类似，移除字符用的 erase()函数也有好几个，替换字符用的 replace()函数也有好几个。例如：

```
std::string s = "i18n";                    // s: i18n
s.replace(1,2,"nternationalizatio"); // s: internationalization
s.erase(13);                               // s: international
s.erase(7,5);                              // s: internal
s.replace(0,2,"ex");                       // s: external
```

你可以使用 resize()改变 *string* 的字符数量。如果参数所指定的大小比现有字符数少，则尾部字符会被移除。如果参数所指定的大小比现有字符数多，则以某个字符填充尾部。你可以传入一个字符参数，作为填充用的字符，否则就采用字符型别的 default 构造函数产生填充字符（对于型别 char*，其值为 '\0'）。

11.2.9 子串和字符串接合（concatenation）

你可以使用成员函数 substr()从 *string* 身上提取出子字符串。例如：

```
std::string s("interchangeability");

s.substr()              // returns a copy of s
s.substr(11)            // returns string("ability")
s.substr(5,6)           // returns string("change")
s.substr(s.find('c')) // returns string("changeability")
```

你可以使用 operator+ 把两个 *strings*（或 *C-strings*）接合起来。例如：

```
std::string s1("enter");
std::string s2("nation");
std::string i18n;
i18n = 'i' + s1.substr(1) + s2 + "aliz" + s2.substr(1);
std::cout << "i18n means: " + i18n << std::endl;
```

输出如下：

```
i18n means: internationalization
```

11.2.10 I/O 操作符

Strings 定义了常用的 I/O 操作符：

- operator >> 从 input stream 读取一个 *string*。
- operator << 把一个 *string* 写到 output stream 中。

这些操作符的使用方法和面对一般 *C-strings* 时相同。更明确地说，operator>> 的执行方式如下：

1. 如果设置了 skipws 标志（参见 13.7.7 节, p625），则跳过开头空格。
2. 持续读取所有字符，直到发生以下情形之一：
   - 下个字符为空格符（whitespace）
   - stream 不再处于 good 状态（例如遇到 *end-of-file*）

- stream 的 width()结果大于 0（13.7.3 节, p618），而目前已读出 width()个字符。
- 已读取 max_size()个字符。

3. stream width()被设为 0。

一般而言，input 操作符读入下一个字时，会跳过前导的空格符。所谓空格符是指任何令 isspace(c,strm.getloc())结果值为 true 的字符（关于 isspace()，请见 14.4.4 节, p718）。

output 操作符通常也会考虑 stream width()。如果 width()大于 0，则 operator<<至少要写入 width()个字符。

String classes 在命名空间 std 内还提供了一种用于逐行读取的特殊函数：std::getline()。该函数读取所有字符，包括开头的空格符，直到遭遇分行符号或 *end-of-file*。分行符号可指定，不可添加。缺省情况下分行符号为 newline 字符，但你也可以把自己喜欢的分行符号作为参数传给 getline()[5]：

```
std::string s;
while (getline(std::cin,s)) {   // for each line read from cin
   ...
}
while (getline(std::cin,s,':')) {  // for each token separated by ':'
   ...
}
```

注意，如果你是逐一读取语汇单元，那么 newline 字符不被视为特殊字符。因此语汇单元中也可能包含 newline 字符。

11.2.11 搜索和查找（Searching and Finding）

Strings 提供了许多用于搜索和查找字符及子字符串的函数[6]。你可以：

- 搜寻单一字符、字符区间（子字符串）、或若干字符中的一个
- 前向搜寻和后向搜寻
- 从字符串头部或内部任何地方开始搜寻

---

[5] 由于 "Koenig lookup" 搜寻法则会在函数被调用时考虑参数类别定义所处的命名空间（参见 p17），所以我们不必在 getline()之前加上 std::。

[6] 在这里我写搜索和查找（searching and finding），但是请不要被弄胡涂了，它们（几乎）同义。搜索(search)函数的名称同样有 find 这个字眼，但它们并不保证能找到任何东西。所以事实上它们只是试图找到某些东西。我以术语 *search* 来说明这些函数的行为，而在涉及其函数名字时，采用 *find*。译注：中译本两者不分，一律译为搜寻。

另外，如果配上迭代器，STL 的所有搜寻（search）算法都可派上用场。

所有搜寻函数的名字中都有 *find* 这个字。它们试图找到“与参数传入的 *value* 值相等”的字符所处位置，如表 11.5 所示。

表 11.5 *Strings* 搜寻函数

string 函数	效果
`find()`	搜寻第一个与 *value* 相等的字符
`rfind()`	搜寻最后一个与 *value* 相等的字符（逆向搜寻）
`find_first_of()`	搜寻第一个“与 *value* 中的某值相等”的字符
`find_last_of()`	搜寻最后一个“与 *value* 中的某值相等”的字符
`find_first_not_of()`	搜寻第一个“与 *value* 中任何值都不相等”的字符
`find_last_not_of()`	搜寻最后一个“与 *value* 中任何值都不相等”的字符

所有搜寻函数都返回符合搜寻条件之字符区间内的第一个字符的索引。如果搜寻不成功（没找到目标），则返回 `npos`。这些搜寻函数都采用下面的参数方案：

- 第一参数总是被搜寻的对象。
- 第二参数（可有可无）指出 *string* 内的搜寻起点（索引）。
- 第三参数（可有可无）指出搜寻的字符个数。

不幸的是上面这个参数方案与其它 *string* 相关函数不同。其它 *string* 函数的第一个参数是起点索引，随后是数值和长度。特别要指出的是，每个搜寻函数都以下面的参数集进行重载：

- `const` ***string&*** *value*
  搜寻对象为 *value*（一个 *string*）。
- `const` ***string&*** *value*, ***size_type*** *idx*
  从`*this` 的 *idx* 索引位置开始，搜寻 *value*（一个 *string*）。
- `const` ***char**** *value*
  搜寻 *value*（一个 *C-string*）。
- `const` ***char**** *value*, ***size_type*** *idx*
  从`*this` 的 *idx* 索引位置开始，搜寻 *value*（一个 *C-string*）。
- `const` ***char**** *value*, ***size_type*** *idx*, ***size_type*** *value_len*
  从`*this` 的 *idx* 索引位置开始，搜索 *value*（一个 *C-string*）内的前 *value_len* 个字符所组成的字符区间。*value* 内的 null 字符（`'\0'`）将不复特殊意义。
- `const` ***char*** *value*
  搜寻 *value*（一个字符）。
- `const` ***char*** *value*, ***size_type*** *idx*
  从`*this` 的 *idx* 索引位置开始，搜寻 *value*（一个字符）。

例如：

```
std::string s("Hi Bill, I'm ill, so please pay the bill");

s.find("il")      // returns 4 (first substring "il")
s.find("il",10)   // returns 13 (first substring "il" starting from s[10])
s.rfind("il")     // returns 37 (last substring "il")
s.find_first_of("il") // returns 1 (first char 'i' or 'l')
s.find_last_of("il")       // returns 39 (last char 'i' or 'l')
s.find_first_not_of("il")  // returns 0 (first char neither 'i' nor 'l')
s.find_last_not_of("il")   // returns 36 (last char neither 'i' nor 'l')
s.find("hi")               // returns npos
```

你也可以通过 STL 算法来搜寻 *string* 内的字符或子字符串。这些算法通常都允许你使用自己的比较准则（实例参见 11.2.13 节, p499）。但是请注意，STL 搜索算法的命名方式和 *string* 搜寻函数的命名方式非常不同（详见 9.2.2 节, p324）。

11.2.12 数值 npos 的意义

如果搜寻函数失败，就会返回 *string*::npos。试看下面例子：

```
std::string s;
std::string::size_type idx; // be careful: don't use any other type!
...

idx = s.find("substring");
if (idx == std::string::npos) {
   ...
}
```

只有当"substring" 不是字符串 s 的子字符串时，if 语句才会得到 true 值。使用 *string* 的 npos 值及其型别时要格外小心：若要检查返回值，一定要使用型别 string::size_type，不能以 int 或 unsigned 作为返回值型别；否则返回值与 string::npos 之间的比较可能无法正确执行。这是因为 npos 被设计为 -1：

```
namespace std {
    template<class charT,
             class traits = char_traits<charT>,
             class Allocator = allocator<charT> >
    class basic_string {
    public:
```

```
        typedef typename Allocator::size_type size_type;
        ...
        static const size_type npos = -1;
        ...
    };
}
```

不幸的是 size_type（由字符串配置器 allocator 定义）需为无正负号整数型别。因为缺省配置器以型别 size_t 作为 size_type（参见 15.3 节，p732）。于是 -1 被转换为无正负号整数型别，npos 也就成了该型别的最大无符号值。不过实际数值还是取决于型别 size_type 的实际定义。不幸的是这些最大值都不相同。事实上(unsigned long)-1 和(unsigned short)-1 不同（前提是两者型别大小不同）。因此，以下比较式：

```
idx == std::string::npos
```

如果 idx 的值为-1，由于 idx 和字符串 string::npos 型别不同，比较结果可能得到 false：

```
std::string s;
...
int idx = s.find("not found"); // assume it returns npos
if (idx == std::string::npos) { // ERROR: comparison might not work
   ...
}
```

避免这种错误的办法之一就是直接检验搜寻是否失败：

```
if (s.find("hi") == std::string::npos) {
   ...
}
```

但由于我们常常需要用到匹配字符的位置索引，所以另一个简单的解决方法是自行定义对应于 npos 的带正负号数值：

```
const int NPOS = -1;
```

前述的比较式必须略加修改（但方便多了）如下：

```
if (idx == NPOS) { // works almost always
   ...
}
```

遗憾的是如果 idx 的型别为 unsigned short，抑或索引大于 int 最大值，上述比较式就会失败，所以这种解法并不完善（也因此，*C++ Standard* 并没有按这种方式定义）。但是这两种情况很少发生，所以此解法在大多数情况下都有效。如果你希望你的程序代码有高度移植性，你应该对 *string* 型别的任何索引都采用 *string*::size_type。若要获得完善解法，还需考虑精确型别 *string*::size_type 的重载函数。我希望将来 *C++ Standard* 能提出更好的解决方案。

11.2.13 Strings 对迭代器的支持

string 是字符的有序群集（ordered collection）。所以 C++ 标准程序库为 *strings* 提供了相应接口，以便将字符串当做 STL 容器使用[7]。

更明确地说，你可以调用常用的成员函数，取得“能够遍历 *string* 内所有字符”的迭代器。如果对迭代器不熟悉，可以把它们看做是指向字符串内部单个字符的东西，就像普通指针之于 *C-string*。采用迭代器，你便可以通过调用 C++ 标准程序库提供（或用户自行定义）的算法，遍历 *string* 内的全部字符。你可以因此对字符串内的字符进行排序、逆向重排、找出最大值（字符）等等动作。

String 迭代器是 random access（随机存取）迭代器。也就是说它支持随机存取，所以任何一个 STL 算法都可与它搭配（关于迭代器的分类，请见 5.3.2 节, p93 和 7.2 节, p251）。通常 *string* 的“迭代器型别”（`iterator`, `const_iterator` 等）由 *string* class 本身定义，确切型别则由实作作品定义，但通常被简单定义为一般指针。“以指针实作而成”和“以 class 实作而成”的迭代器之间有着难对付的差别，参见 7.2.6 节, p258。

对迭代器而言，如果发生重分配（reallocation），或其所指值发生某些变化，迭代器就会失效，详见 p487, 11.2.6 节。

String 的迭代器相关函数

表 11.6 列出 *strings* 在迭代器方面提供的所有成员函数。通常，`beg` 和 `end` 所规范的区间包括 `beg` 但不包括 `end`，是个半开区间，常写作`[beg;end)`，参见 5.3 节, p83]。

为了支持运用 back inserter，*string* 定义了 `push_back()`。关于 back inserter，详见 7.4.2 节, p272，它们在 *string* 中的使用实例请见 p502。

String 迭代器的运用实例

String 迭代器可以做一件非常有用的事情：透过一个简单的语句，把 *string* 内的所有字符都转为大写或小写。例如：

```
// string/iter1.cpp

#include <string>
#include <iostream>
#include <algorithm>
#include <cctype>
using namespace std;
```

[7] 第 5 章已介绍过 STL。

表 11.6 *string* 的迭代器操作函数

表达式	效果
`s.begin()`	返回一个随机存取迭代器，指向第一字符
`s.end()`	返回一个随机存取迭代器，指向最后一个字符的下一个位置
`s.rbegin()`	返回一个逆向迭代器，指向倒数第一个字符（亦即最后一个字符）
`s.rend()`	返回一个逆向迭代器，指向倒数最后一个字符的下一位置（亦即第一字符的前一位置）
`string s(beg,end)`	以区间 `[beg;end]` 内的所有字符作为 *string* `s` 的初值
`s.append(beg,end)`	将区间 `[beg;end]` 内的所有字符添加于 `s` 尾部
`s.assign(beg,end)`	将区间 `[beg;end]` 内的所有字符赋值给 `s`
`s.insert(pos,c)`	在迭代器 `pos` 所指之处插入字符 `c`，并返回新字符的迭代器位置
`s.insert(pos,num,c)`	在迭代器 `pos` 所指之处插入 `num` 个字符 `c`，并返回第一个新字符的迭代器位置
`s.insert(pos,beg,end)`	在迭代器 `pos` 所指之处插入区间 `[beg;end]` 内的所有字符
`s.erase(pos)`	删除迭代器 `pos` 所指字符，并返回下一个字符位置
`s.erase(beg,end)`	删除区间 `[beg;end]` 内的所有字符，并返回下一个字符的下一位置
`s.replace(beg,end,str)`	以 string `str` 内的字符替代 `[beg;end]` 区间内的所有字符
`s.replace(beg,end,cstr)`	以 *C-string* `cstr` 内的字符替代 `[beg;end]` 区间内的所有字符
`s.replace(beg,end,cstr,len)`	以字符数组 `str` 的前 `len` 个字符替代 `[beg;end]` 区间内的所有字符
`s.replace(beg,end,num,c)`	以 `num` 个字符 `c` 替代 `[beg;end]` 区间内的所有字符
`s.replace(beg,end,` `newBeg,newEnd)`	以 `[newBeg;newEnd)` 区间内的所有字符替代 `[beg;end)` 区间内的所有字符

```
int main()
{
    // create a string
    string s("The zip code of Hondelage in Germany is 38108");
    cout << "original: " << s << endl;

    // lowercase all characters
    transform (s.begin(), s.end(),   // source
               s.begin(),            // destination
               tolower);             // operation
    cout << "lowered: " << s << endl;

    // uppercase all characters
    transform (s.begin(), s.end(),   // source
               s.begin(),            // destination
               toupper);             // operation
    cout << "uppered: " << s << endl;
}
```

程序输出如下：

```
original: The zip code of Hondelage in Germany is 38108
lowered:  the zip code of hondelage in germany is 38108
uppered:  THE ZIP CODE OF HONDELAGE IN GERMANY IS 38108
```

请注意，tolower()和 toupper()用的是旧式的 C 全局函数。如果国别(locale)不同或程序涵盖多种国别，应采用 tolower()和 toupper()的新形式。详见 14.4.4 节，p718。

以下例子说明 STL 如何使用你自己定义的搜寻和排序准则，以“不计大小写”的方式对 *string* 进行比较和搜寻：

```
// string/iter2.cpp

#include <string>
#include <iostream>
#include <algorithm>
using namespace std;

bool nocase_compare (char c1, char c2)
{
  return toupper(c1) == toupper(c2);
}
```

```
int main()
{
    string s1("This is a string");
    string s2("STRING");

    // compare case insensitive
    if (s1.size() == s2.size() &&    // ensure same sizes
        equal (s1.begin(),s1.end(),  // first source string
               s2.begin(),           // second source string
               nocase_compare)) {    // comparison criterion
        cout << "the strings are equal" << endl;
    }
    else {
        cout << "the strings are not equal" << endl;
    }

    // search case insensitive
    string::iterator pos;
    pos = search (s1.begin(),s1.end(),    // source string in which to search
                  s2.begin(),s2.end(),    // substring to search
                  nocase_compare);        // comparison criterion
    if (pos == s1.end()) {
        cout << "s2 is not a substring of s1" << endl;
    }
    else {
        cout << '"' << s2 << "\" is a substring of \""
             << s1 << "\" (at index " << pos - s1.begin() << ")"
             << endl;
    }
}
```

注意，equal()的调用者必须保证第二区间至少要和第一区间具有一样多的元素（字符）。因此先比较字符串的大小是必要的，否则可能导致未定义的行为。

你可以在最后一条语句中处理两个 *string* 迭代器间的差距（difference），用以获取字符位置的索引：

```
pos - s1.begin()
```

之所以能够这么做，因为 *string* 迭代器是一种 Random Access（随机存取）迭代器。将索引转换为迭代器位置也是一样：只要简单地把索引值加到迭代器身上即可。

本例中，使用者自行定义的辅助函数 nocase_compare()用于“大小写不分”的字符串比较方式。其实你也可以组合运用某些函数适配器（function adapter）并采用以下表达式替换先前的 nocase_compare：

```
compose_f_gx_hy(equal_to<int>(),
                ptr_fun(toupper),
                ptr_fun(toupper))
```

细节请见 p309 和 p318。

如果在 sets 或 maps 中使用 *strings*，也许你会想要一个特定的排序准则，让这些容器能够以“大小写不分”的方式对 *string* 排序。相应例子请见 p213。

以下程序示范 *string* 迭代器的另一种运用：

```
// string/iter3.cpp

#include <string>
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
    // create constant string
    const string hello("Hello, how are you?");

    // initialize string s with all characters of string hello
    string s(hello.begin(),hello.end());

    // iterate through all of the characters
    string::iterator pos;
    for (pos = s.begin(); pos != s.end(); ++pos) {
         cout << *pos;
    }
    cout << endl;

    // reverse the order of all characters inside the string
    reverse (s.begin(), s.end());
    cout << "reverse: " << s << endl;

    // sort all characters inside the string
    sort (s.begin(), s.end());
    cout << "ordered: " << s << endl;
```

```
    /* remove adjacent duplicates
    * - unique() reorders and returns new end
    * - erase() shrinks accordingly
    */
    s.erase (unique(s.begin(),
                    s.end()),
             s.end());
    cout << "no duplicates: " << s << endl;
}
```

程序输出如下：

```
Hello, how are you?
reverse:       ?uoy era woh ,olleH
ordered:          ,?Haeehlloooruwy
no duplicates:  ,?Haehloruwy
```

以下例子采用 back inserters，从标准输入装置读取数据到 *strings* 内：

```
// string/unique.cpp

#include <iostream>
#include <string>
#include <algorithm>
#include <locale>
using namespace std;

class bothWhiteSpaces {
   private:
      const locale& loc; // locale
   public:
      /* constructor
      * - save the locale object
      */
      bothWhiteSpaces (const locale& l) : loc(l) {
      }
      /* function call
      * - returns whether both characters are whitespaces
      */
      bool operator() (char elem1, char elem2) {
         return isspace(elem1,loc) && isspace(elem2,loc);
      }
};
```

```
int main()
{
    string contents;

    // don't skip leading whitespaces
    cin.unsetf (ios::skipws);

    // read all characters while compressing whitespaces
    unique_copy(istream_iterator<char>(cin),   // beginning of source
                istream_iterator<char>(), // end of source
                back_inserter(contents),      // destination
                bothWhiteSpaces(cin.getloc()));
                   // criterion for removing

    // process contents
    // - here: write it to the standard output
    cout << contents;
}
```

透过 unique_copy()算法（参见 p384, 9.7.2 节），input stream cin 的所有字符都被安插到字符串 contents 中。仿函数 bothWhiteSpaces 用于检查两个相邻字符是否都是空格——为了实现这个功能，我们以“cin 的 locale（译注：上述粗体部分）”作为其初值，并调用 isspace()检查字符是否为空格（详见 14.4.4 节, p718）。unique_copy()以 bothWhiteSpaces 为行为准则，删除相邻重复空格。p385 可以找到类似的例子。

11.2.14 国际化（Internationalization）

正如先前（11.2.1 节, p479）介绍 *string* classes 时所说，template *string* class basic_string<> 乃是以“字符型别”完成参数化；string 是一个针对 char 的特化体（specialization），wstring 则是针对 wchar_t 的一个特化体。

为了应付那些“答案取决于字符型别”的技术问题，*strings* 使用所谓的字符特性（character traits）提供相关细节。我们需要一个新的 class，因为我们无法改变内建型别（例如 char 和 wchar_t）的接口。关于特性类别（traits classes）的详细讨论请见 14.1.2 节, p687。

下面展示一个为 *strings* 打造的特性类别（traits classes），使字符可以在“不计大小写”的方式下被操作：

```
// string/icstring.hpp

#ifndef ICSTRING_HPP
#define ICSTRING_HPP
#include <string>
#include <iostream>
#include <cctype>      // 译注：for toupper()

/* replace functions of the standard char_traits<char>
* so that strings behave in a case-insensitive way
*/
struct ignorecase_traits : public std::char_traits<char> {
   // return whether c1 and c2 are equal
   static bool eq(const char& c1, const char& c2) {
      return std::toupper(c1)==std::toupper(c2);
   }
   // return whether c1 is less than c2
   static bool lt(const char& c1, const char& c2) {
      return std::toupper(c1)<std::toupper(c2);
   }
   // compare up to n characters of s1 and s2
   static int compare(const char* s1, const char* s2,
                      std::size_t n) {
      for (std::size_t i=0; i<n; ++i) {
         if (!eq(s1[i],s2[i])) {
            return lt(s1[i],s2[i])?-1:1;
         }
      }
      return 0;
   }
   // search c in s
   static const char* find(const char* s, std::size_t n,
                           const char& c) {
      for (std::size_t i=0; i<n; ++i) {
         if (eq(s[i],c)) {
              return &(s[i]);
         }
      }
      return 0;
   }
};
```

```
// define a special type for such strings
typedef std::basic_string<char,ignorecase_traits> icstring;

/* define an output operator
* because the traits type is different than that for std::ostream
*/
inline
std::ostream& operator << (std::ostream& strm, const icstring& s)
{
    // simply convert the icstring into a normal string
    return strm << std::string(s.data(),s.length());
}
#endif // ICSTRING_HPP
```

由于 C++ *Standard* 只为“采用相同字符型别和特性类别”的 stream 定义 I/O 操作，而此处的特性类别并不相同，所以我们不得不定义自己的 output 操作符。同样道理也适用于 Input 操作符。

下面这个程序说明如何使用这些特殊字符串：

```
// string/icstring1.cpp

#include "icstring.hpp"

int main()
{
    using std::cout;
    using std::endl;

    icstring s1("hallo");
    icstring s2("otto");
    icstring s3("hALLo");

    cout << std::boolalpha;
    cout << s1 << " == " << s2 << " : " << (s1==s2) << endl;
    cout << s1 << " == " << s3 << " : " << (s1==s3) << endl;

    icstring::size_type idx = s1.find("All");
    if (idx != icstring::npos) {
        cout << "index of \"All\" in \"" << s1 << "\": "
             << idx << endl;
    }
    else {
        cout << "\"All\" not found in \"" << s1 << endl;
    }
}
```

程序输出如下：

```
hallo == otto : false
hallo == hALLo : true
index of "All" in "hallo": 1
```

关于国际化（internationalization）议题，详见第 14 章。

11.2.15 效率（Performance）

C++ Standard 并未规定如何实作 *string* classes，只是定义了其界面。

由于理念和侧重点各有不同，不同的实作作品可能在速度和内存占用方面存在显著的差异。

如果你希望速度更快，请确认你所使用的 *string* classes 采用了类似 *reference counting*（引用计数）概念（译注：根据我对 SGI STL 源码的理解，可确认 SGI *string* 支持 *reference counting*）。这种手法可以加速 *string* 的复制和赋值（赋值），因为在实作之中不再是对字符串内容进行操作，而仅仅是复制和赋值字符串的 reference（本书有一个可适用于任何型别的智能型指标便是运用这种手法，详见 6.8 节，p222）。通过 *reference counting*，你甚至不必透过 const reference 来传递字符串；不过基于灵活性和可移植性的考虑，一般还是应该采用 const reference 来传递参数。

11.2.16 Strings 和 Vectors

Strings 和 vectors 很相似，这也不奇怪，因为它们都是一种动态数组。因此，可以把 *strings* 视为一种“以字符作为元素”的特定 vectors。实用上可把 *string* 当做 STL 容器使用，这在 p497, 11.2.13 节已有介绍。但由于 *string* 和 vectors 之间有许多本质上的不同，所以把 *string* 当做特殊的 vectors 还是存在一定的危险。最重要的差异在于两者的主要目标：

- vectors 首要目标是处理和操作容器内的元素，而非容器整体。因此实作时通常会为“容器元素的操作行为”进行优化。
- *strings* 主要是把整个容器视为整体，进行处理和操作，因此实作时通常会为“整个容器的赋值和传递”进行优化。

不同的目标导致完全不同的实作手法。例如 *strings* 通常采用 *reference counting*（引用计数）手法，vectors 则决对不会如此。然而我们还是可以把 vectors 当做一般的 *C-strings* 来使用，详见 6.2.3 节，p155。

11.3 细说 String Class

本节如果出现 *string*，指的是实际的 string class，可以是 string，wstring 或 basic_string<> 的任何一种特定形式；*char* 则是指实际字符型别，亦即 string 所使用的 char 和 wstring 所使用的的 wchar_t。其它以斜体字标出的型别和实值的定义都取决于字符型别或特性类别（traits class）的个别定义。14.1.2 节，p687 详细介绍了所谓特性类别。

11.3.1 内部的型别定义和静态值

string::**traits_type**

- 字符特征（character traits）的型别。
- basic_string<> 的第二个 template 参数。
- 对型别 string 而言，此值等价于 char_traits<char>。

string::**value_type**

- 字符型别。
- 等价于 traits_type::char_type。
- 对型别 string 而言，此值等价于 char。

string::**size_type**

- 未带正负号的整数型别，用来指定大小值和索引。
- 等价于 allocator_type::size_type。
- 对型别 string 而言，此值等价于 size_t。

string::**difference_type**

- 带正负号的整数型别，用来指定差值（距离）。
- 等价于 allocator_type::difference_type。
- 对型别 string 而言，此值等价于 ptrdiff_t。

string::**reference**

- 字符的 references 型别。
- 等价于 allocator_type::reference。
- 对型别 string 而言，此值等价于 char&。

string::**const_reference**

- 常数型的字符 references 型别。
- 等价于 allocator_type::const_reference。
- 对型别 string 而言，此值等价于 const char&。

string::**pointer**

- 字符的 pointers 型别。
- 等价于 allocator_type::pointer。
- 对型别 string 而言，此值等价于 char*。

string::**const_pointer**

- 常数型的字符 pointers 型别。
- 等价于 allocator_type::const_pointer。
- 对型别 string 而言，此值等价于 const char&。

string::**iterator**

- 迭代器型别。
- 确切型别由实作作品负责定义。
- 对型别 string 而言通常为 char*。

string::**const_iterator**

- 常数型迭代器型别。
- 确切型别由实作作品负责定义。
- 对型别 string 而言通常为 const char*。

string::**reverse_iterator**

- 逆向迭代器（reverse iterators）型别。
- 等价于 reverse_iterator<iterator>。

string::**const_reverse_iterator**

- 常数型逆向迭代器（constant reverse iterators）型别。
- 等价于 reverse_iterator<const_iterator>。

static const *size_type string*::**npos**

- 这是一个特殊值，表示下列情形：
  - “未找到”
  - “所有剩余字符”
- 初始值为 -1（一个无正负号整数值）。
- 使用 npos 时要十分小心，详见 11.2.3 节, p495。

11.3.2 生成（Create）、拷贝（Copy）、销毁（Destroy）

string::**string** ()

- default（缺省）构造函数。
- 产生一个空字符串。

```
string::string (const string& str)
```

- copy（拷贝）构造函数。
- 产生一个新字符串，是 *str* 的副本。

```
string::string (const string& str, size_type str_idx)
string::string (const string& str, size_type str_idx,
                size_type str_num)
```

- 产生一个新字符串，其初值为“*str* 内，从索引 *sr_idx* 开始的最多 *str_num* 个字符”。
- 如果没有指定 *str_num*，则取用“从 *str_idx* 开始到 *str* 尾部”的所有字符。
- 如果 *str_idx* > *str*.size()，抛出 out_of_range 异常。

```
string::string (const char* cstr)
```

- 产生一个字符串，并以 *C-string cstr* 作为初值。
- 初值为 *cstr* 内以 '\0' 为结束符号（但不包括 '\0'）的所有字符。
- *cstr* 不可为 NULL 指标。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

```
string::string (const char* chars, size_type chars_len)
```

- 产生一个字符串，以字符数组 *chars* 内的 *chars_len* 个字符为初值。
- *chars* 必须至少包含 *chars_len* 个字符。这些字符可以为任意值，'\0' 无特殊意义。
- 如果 *chars_len* 等于 *string*::npos，抛出 length_error 异常。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

```
string::string (size_type num, char c)
```

- 产生一个字符串，其内容初值为 *num* 个字符 *c*。
- 如果 *num* 等于 *string*::npos，抛出 length_error 异常。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

```
string::string (InputIterator beg, InputIterator end)
```

- 产生一个字符串，以区间 [*beg*;*end*) 内的字符为初值。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

```
string::~string ()
```

- 析构函数。
- 销毁所有字符并释放内存。

大部分构造函数都可以接受配置器作为附加参数传入（见 11.3.12 节, p526）。

11.3.3 大小（Size）和容量（Capacity）

关于大小

```
size_type string::size () const
size_type string::length () const
```

- 两个函数都返回现有字符的个数。
- 二者等价（*equivalent*）。
- 如果想要检查字符串是否为空，应采用速度更快的 `empty()`。

```
bool string::empty () const
```

- 判断字符串是否为空，亦即是否未包含任何字符。
- 等价于 `string::size()==0`，但可能更快。

```
size_type string::max_size () const
```

- 返回字符串可含的最大字符数目。
- 任何操作一旦产生长度大于 `max_size()` 的字符串，就抛出 length_error 异常。

关于容量

```
size_type string::capacity () const
```

- 返回重分配之前字符串所能包含的最多字符个数。

```
void string::reserve ()
void string::reserve (size_type num)
```

- 第二种形式用以保留“至少能容纳 *num* 个字符”的内存。
- 如果 *num* 小于目前实际容量，调用这个函数相当于“非强制性容量缩减请求”。
- 如果 *num* 小于目前字符量，调用这个函数相当于“非强制性适度缩减（*shink-to-fit*）请求”，其意义是请求将容量缩小至实际字符数的大小。
- 如果没有传递参数（上述第一形式），调用该函数相当于一个“非强制性适度缩减（*shink-to-fit*）请求”。
- 容量永远不能小于实际字符数。
- 每次重分配都会造成所有 references、pointers 和 iterators 失效，并耗费一定时间。因此可事先调用 `reserve()` 来加快速度，并因此保持 references、pointers 和 iterators 的有效性（详见第 11.2.5 节, p486）。

11.3.4 比较（Comparisons）

```
bool comparison (const string& str1, const string& str2)
bool comparison (const string& str, const char* cstr)
bool comparison (const char* cstr, const string& str)
```

- 第一种形式返回两个 *strings* 的比较结果。
- 后两种形式返回 *string* 和 *C-string* 的比较结果。
- ***comparison*** 指的是以下任何一种动作：

```
operator ==
operator !=
operator <
operator >
operator <=
operator >=
```

- 按字典次序（lexicographically）进行比较（参见 p488）。

```
int string::compare (const string& str) const
```

- 把*this 拿来和 *str* 进行比较。
- 返回值：
  - 0，表示两端字符串相等
  - < 0，表示*this 小于 *str*（按字典次序）
  - > 0，表示*this 大于 *str*（按字典次序）
- 以 traits::compare() 为比较准则（参见 14.1.2 节, p689）。
- 详见 11.2.7 节, p488。

```
int string::compare (size_type idx, size_type len, const string& str) const
```

- 将*this 之内“从 *idx* 开始的最多 *len* 个字符”拿来和 *str* 比较
- 如果 *idx* > size()，抛出 out_of_range 异常。
- 比较动作和前述的 compare(*str*) 相同。

```
int string::compare (size_type idx, size_type len,
                     const string& str, size_type str_idx,
                     size_type str_len) const
```

- 将*this 之内“从 *idx* 开始的最多 *len* 个字符”拿来和 *str* 之内“从 *str_idx* 开始的最多 *str_len* 个字符”相较。
- 如果 idx > size()，抛出 out_of_range 异常。
- 如果 str_idx > str.size()，抛出 out_of_range 异常。
- 比较动作和前述的 compare(*str*) 相同。

```
int string::compare (const char* cstr) const
```

- 将*this 的字符和 *C-string* cstr 的字符进行比较。
- 比较动作和前述的 compare(str) 相同。

```
int string::compare (size_type idx, size_type len, const char* cstr) const
```

- 将*this 之内“从 idx 开始的最多 len 个字符”拿来和 *C-string* cstr 的所有字符比较[8]。
- 比较动作和前述的 compare(str) 相同。
- cstr 绝不可为 null 指针。

```
int string::compare (size_type idx, size_type len,
                     const char* chars, size_type chars_len) const
```

- 将*this 之内“从 idx 开始的最多 len 个字符”拿来和字符数组 chars 内的 chars_len 个字符相较。
- 比较动作和前述的 compare(str) 相同。
- chars 必须至少包含 chars_len 个字符（可为任意值）。'\0' 没有特殊意义。
- 如果 chars_len 等于 string::npos，抛出 length_error 异常。

11.3.5 字符存取（Character Access）

```
char& string::operator[] (size_type idx)
char string::operator[] (size_type idx) const
```

- 两种形式都返回索引 idx 所指示的字符（首字符的索引为 0）。
- 常量字符串的 length() 是一个有效索引，上述函数会因此返回一个“由字符型别的缺省构造函数所产生”的值（对 string 而言为 '\0'）。
- 非常量（non-const）字符串的 length() 是一个无效索引。
- 无效索引会导致未定义的行为。
- 非常量（non-const）字符串返回的 reference 会因为字符串的修改或重分配而失效（详见 11.2.6 节, p487）。
- 如果调用者无法确定索引有效，就应该采用 at()。

[8] C++ *Standard* 对于此形式的 compare() 的描述与本书不同：它将 cstr 视为字符数组而非 *C-string*，并以 npos 作为其长度（实际上是呼叫 compare() 的后继形式并以 npos 为附加参数）。这是 C++ *Standard* 的一个错误（它理应抛出 length_error 异常）。

```
char& string::at (size_type idx)
const char& string::at (size_type idx) const
```

- 两种形式都返回索引 *idx* 所指示的字符（首字符的索引为 0）。
- 对于所有 *strings*，length()均为非法（无效）索引。
- 传递无效索引（小于 0 或大于等于 size()）会导致 out_of_range 异常。
- 非常量（non-const）字符串返回的 reference 会因为字符串的修改或重分配而失效（详见 11.2.6 节, p487）。
- 如果调用者确定索引是有效的，可采用操作符[]，速度更快。

11.3.6 产生 C-*string* 和字符数组（Character Arrays）

```
const char* string::c_str () const
```

- 将 *string* 的内容以 *C-string*（一个字符数组，尾部添加 '\0'）形式返回。
- 返回值隶属于该 *string*，所以调用者不能修改、释放或删除该返回值。
- 唯有当 *string* 存在，并且用来处理该返回值的函数是个“常数型函数”时，这个返回值才保持有效。

```
const char* string::data () const
```

- 将 *string* 的内容以字符数组的形式返回。
- 返回值内含 *string* 的所有字符，完全未加修改或扩充。更明确地说，并没有附加 null 字符。因此这个返回值往往不是有效的 *C-string*。
- 返回值隶属于该 *string*，所以调用者不能修改、释放或删除该返回值。
- 唯有当 *string* 存在，并且用来处理该返回值的函数是个“常数型函数”时，这个返回值才保持有效。

```
size_type string::copy (char* buf, size_type buf_size) const
size_type string::copy (char* buf, size_type buf_size, size_type idx) const
```

- 以上两种形式都将字符串*this（从索引 *idx* 开始）内最多 *buf_size* 个字符复制到字符数组 *buf* 中。
- 返回被复制的字符数。
- 不添加 null 字符。因此函数执行后的 *buf* 内容可能不是有效的 *C-string*。
- 调用者必须确保 *buf* 有足够的内存；否则会导致未定义行为。
- 如果 *idx* > size()，抛出 out_of_range 异常。

11.3.7 更改内容

赋值（Assignments）

```
string& string::operator= (const string& str)
string& string::assign (const string& str)
```

- 以上两种形式都将 str 的值赋给 *this。
- 都返回*this。

```
string& string::assign (const string& str, size_type str_idx,
                        size_type str_num)
```

- 将字符串 str 之内“从索引 str_idx 开始最多 str_num 个字符”赋值给*this。
- 返回*this。
- 如果 str_idx > str.size()，抛出 out_of_range 异常。

```
string& string::operator= (const char* cstr)
string& string::assign (const char* cstr)
```

- 以上两种形式都将 *C-string* 的值赋给 *this。
- 赋值“以 '\0' 结尾（但不包括 '\0'）”的 cstr 所有字符。
- 都返回*this。
- cstr 不得为 null 指标。
- 如果所得结果超出最大字符数，两个函数都抛出 length_error 异常。

```
string& string::assign (const char* chars, size_type chars_len)
```

- 将数组 chars 内的 chars_len 个字符赋值给*this。
- 返回*this。
- chars 至少必须包含 chars_len 个字符，字符可为任意值，'\0' 并无特殊意义。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

```
string& string::operator= (char c)
```

- 将字符 c 赋值给*this。
- 返回 *this。
- 调用后，*this 只含这一个字符。

```
string& string::assign (size_type num, char c)
```

- 将 num 个字符 c 赋值给 *this。
- 返回 *this。
- 如果 num 等于 string::npos，抛出 length_error 异常。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

```
void string::swap (string& str)
void swap (string& str1, string& str2)
```

- 两种形式都用来交换两个 *string*s：
  - 成员函数版本用来交换*this 和 str 的内容。
  - 全局函数版本用来交换 str1 和 str2 的内容。
- 你应该尽可能采用这些函数取代赋值操作（assignment），因为它们更快。它们具有常数复杂度，详见 11.2.8 节, p490。

添加（Appending）字符

```
string& string::operator+= (const string& str)
string& string::append (const string& str)
```

- 两种形式都将 str 的字符添加到*this 尾部。
- 都返回*this。
- 如果所得结果超出最大字符数，两者都抛出 length_error 异常。

```
string& string::append (const string& str, size_type str_idx,
                        size_type str_num)
```

- 将 str 之内“从 str_idx 开始最长 str_num 个字符”添加到*this 尾部。
- 返回*this。
- 如果 str_idx > str.size()，抛出 out_of_range 异常。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

```
string& string::operator+= (const char* cstr)
string& string::append (const char* cstr)
```

- 都将 *C-string* 内的字符添加到*this 尾部。
- 都返回*this。
- cstr 不能为 null 指标。
- 如果所得结果超出最大字符数，两者都抛出 length_error 异常。

```
string& string::append (const char* chars, size_type chars_len)
```

- 将字符数组 chars 之内的 chars_len 个字符添加到*this 尾部。
- 返回*this。
- chars 必须包含至少 chars_len 个字符，字符可为任意值，'\0' 无特殊含义。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

string& *string*::**append** (size_type *num*, *char c*)

- 将 *num* 个字符 *c* 添加到*this 尾部。
- 返回*this。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

string& *string*::**operator+=** (*char c*)
void *string*::**push_back** (*char c*)

- 将字符 *c* 添加到*this 尾部。
- operator+= 返回*this。
- 如果所得结果超出最大字符数，两者都抛出 length_error 异常。

string& *string*::**append** (InputIterator *beg*, InputIterator *end*)

- 将区间[*beg*;*end*)内所有字符添加到*this 尾部。
- 返回*this。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

安插（inserting）字符

string& *string*::**insert** (size_type *idx*, const *string*& *str*)

- 将 *str* 插入*this 之内，新增字符从索引 *idx* 处开始安插。
- 返回*this。
- 如果 *idx* > size()，抛出 out_of_range 异常。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

string& *string*::**insert** (size_type *idx*, const *string*& *str*,
size_type *str_idx*, size_type *str_num*)

- 将 *str* 之内“从 *str_idx* 开始最多 *str_num* 个字符”插入*this，新增字符从索引 *idx* 处开始安插。
- 返回*this。
- 如果 *idx* > size()，抛出 out_of_range 异常。
- 如果 *str_idx* > *str*.size()，抛出 out_of_range 异常。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

string& *string*::**insert** (size_type *idx*, const *char** *cstr*)

- 将 *C-string cstr* 插入*this，新增字符从索引 *idx* 处开始安插。
- 返回*this。
- *cstr* 不得为 null 指标。
- 如果 *idx* > size()，抛出 out_of_range 异常。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

```
string& string::insert (size_type idx, const char* chars,
                        size_type chars_len)
```

- 将字符数组 *chars* 之内的 *chars_len* 个字符插入*this，新增字符从索引 *idx* 处开始安插。
- 返回*this。
- *chars* 必须包含至少 *chars_len* 个字符，字符可为任意值，'\0' 无特殊含义。
- 如果 *idx* > size()，抛出 out_of_range 异常。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

```
string& string::insert (size_type idx, size_type num, char c)
void string::insert (iterator pos, size_type num, char c)
```

- 两种形式分别在 *idx* 或 *pos* 指定的位置上安插 *num* 个字符 *c*。
- 第一形式将新字符插入 *str*，新增字符从索引 *idx* 处开始。
- 第二形式在迭代器 *pos* 所指字符之前方插入新字符。
- 这两个函数构成重载（overloaded）形式，可能导致模棱两可。如果你以 0 为第一参数，由于 0 可被视为索引（通常被转换为 unsigned），也可被视为迭代器（通常被转换为 char*），因而导致模棱两可。这种情况下你应该明确告知参数是个“索引”。例如：

```
std::string s;
...
s.insert(0,1,' ');   // ERROR: ambiguous
s.insert((std::string::size_type)0,1,' ');  // OK
```

- 两种形式都返回*this。
- 如果 *idx* > size()，两种形式都抛出 out_of_range 异常。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

```
iterator string::insert (iterator pos, char c)
```

- 在迭代器 *pos* 所指字符之前插入字符 *c* 的副本。
- 返回新被插入的字符的位置。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

```
void string::insert (iterator pos, InputIterator beg,
                     InputIterator end)
```

- 在迭代器 *pos* 所指字符之前插入区间 [*beg*;*end*) 的所有字符。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

擦除（Eraseing）字符

```
void string::clear ()
string& string::erase ()
```

- 两个函数都会删除（delete）字符串的所有字符，因此调用后字符串成空。
- erase()返回*this。

```
string& string::erase (size_type idx)
string& string::erase (size_type idx, size_type len)
```

- 两种形式都删除*this之内从索引 *idx* 开始的最多 *len* 个字符。
- 都返回*this。
- 如果未指定 *len*，则删除 *idx* 之后的所有字符。
- 如果 *idx* > size()，两种形式都抛出 out_of_range 异常。

```
string& string::erase (iterator pos)
string& string::erase (iterator beg, iterator end)
```

- 两种形式分别删除 *pos* 所指的单一字符或 [*beg*;*end*) 区间内的所有字符。
- 两者都返回最后一个被删除字符的下一个字符（因此第二形式返回 *end*）[9]。

改变大小

```
void string::resize (size_type num)
void string::resize (size_type num, char c)
```

- 两种形式都将*this 的字符数改为 *num*。也就是说如果 *num* 不等于目前的 size()，则函数将在尾部添加或删除足够字符，使字符数量等于新的大小 *num*。
- 如果字符数增加，则以 *c* 作为初值。如果未指定 *c*，则使用“字符型别”的 default 构造函数来为新字符设初值（对 string 而言将是 '\0'）。
- 如果 *num* 等于 *string*::npos，两者都抛出 length_error 异常。
- 如果所得结果超出最大字符数，两者都抛出 length_error 异常。

[9] C++ *Standard* 规定此一函数的第二形式传回 *end* 之后的位置，那是一种错误描述。

替换（Replacing）字符

```
string& string::replace (size_type idx, size_type len, const string& str)
string& string::replace (iterator beg, iterator end, const string& str)
```

- 第一种形式将*this 之内“从 idx 开始，最长为 len”的字符替换为 str 内的所有字符。
- 第二种形式将[beg;end)区间内的字符替换为 str 的所有字符。
- 返回*this。
- 如果 idx > size()，抛出 out_of_range 异常。
- 如果所得结果超出最大字符数，两者都抛出 length_error 异常。

```
string& string::replace (size_type idx, size_type len,
                         const string& str, size_type str_idx, size_type str_num)
```

- 将*this 之内“从 idx 开始，最长为 len”的字符替换为 str 之内“从 str_idx 开始，最长为 str_num”的所有字符。
- 返回*this。
- 如果 idx > size()，抛出 out_of_range 异常。
- 如果 str_idx > str.size()，抛出 out_of_range 异常。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

```
string& string::replace (size_type idx, size_type len, const char* cstr)
string& string::replace (iterator beg, iterator end, const char* cstr)
```

- 两种形式分别将*this 之中“以 idx 开始，最长为 len”的字符，或[begin; end)区间内的字符替换为 *C-string* cstr 中的所有字符。
- 都返回*this。
- cstr 不得为 null 指标。
- 如果 idx > size()，两者都抛出 out_of_range 异常。
- 如果所得结果超出最大字符数，两者都抛出 length_error 异常。

```
string& string::replace (size_type idx, size_type len,
                         const char* chars, size_type chars_len)
string& string::replace (iterator beg, iterator end,
                         const char* chars, size_type chars_len)
```

- 两种形式分别将*this 之中“以 idx 开始，最长为 len”的字符或[begin; end)区间内的字符，替换为字符数组 chars 的 chars_len 个字符。
- 都返回*this。
- chars 必须包含至少 chars_len 个字符，字符可为任意值，'\0' 无特殊含义。
- 如果 idx > size()，两种形式都抛出 out_of_range 异常。
- 如果所得结果超出最大字符数，两者都抛出 length_error 异常。

```
string& string::replace (size_type idx, size_type len,
                         size_type num, char c)
string& string::replace (iterator beg, iterator end,
                         size_type num, char c)
```

- 两种形式分别将*this之内"从 *idx* 开始，最长为 *len*"的字符，或区间[*begin*;*end*)内的字符，替换为 *num* 个字符 *c*。
- 都返回*this。
- 如果 *idx* > size()，两种形式都抛出 out_of_range 异常。
- 如果结果大小超出最大字符数，则两种形式都抛出 length_error。

```
string& string::replace (iterator beg, iterator end,
                         InputIterator newBeg, InputIterator newEnd)
```

- 以区间[*newBeg*;*newEnd*)内的所有字符替换区间[*beg*;*end*)内的所有字符。
- 返回*this。
- 如果所得结果超出最大字符数，抛出 length_error 异常。

11.3.8 搜寻（Searching and Finding）

搜寻单一字符

```
size_type string::find (char c) const
size_type string::find (char c, size_type idx) const
size_type string::rfind (char c) const
size_type string::rfind (char c, size_type idx) const
```

- 这些函数都从索引 *idx* 开始搜索第一个或最后一个字符 *c*。
- 函数 find()正向（*forward*）搜寻，并返回第一个搜寻结果。
- 函数 rfind()逆向（*backward*）搜索，并返回最后一个搜寻结果。
- 如果这些函数成功，就返回字符索引；否则返回 *string*::npos。

搜寻子字符串

```
size_type string::find (const string& str) const
size_type string::find (const string& str, size_type idx) const
size_type string::rfind (const string& str) const
size_type string::rfind (const string& str, size_type idx) const
```

- 这些函数都从索引 *idx* 开始搜寻第一个或最后一个子字符串 *str*。
- 函数 find() 正向（*forward*）搜寻，并返回第一个子字符串。
- 函数 rfind() 逆向（*backward*）搜索，并返回最后一个子字符串。
- 如果这些函数成功，就返回子字符串的第一字符索引；否则返回 string::npos。

```
size_type string::find (const char* cstr) const
size_type string::find (const char* cstr, size_type idx) const
size_type string::rfind (const char* cstr) const
size_type string::rfind (const char* cstr, size_type idx) const
```

- 这些函数都从索引 *idx* 开始搜寻“与 *C-string cstr* 内容相同”的第一个或最后一个子字符串。
- 函数 find() 正向（*forward*）搜寻，并返回第一个子字符串。
- 函数 rfind() 逆向（*backward*）搜索，并返回最后一个子字符串。
- 如果这些函数成功，就返回子字符串的第一字符索引；否则返回 string::npos。
- *cstr* 不得为 **null** 指标。

```
size_type string::find (const char* chars, size_type idx,
                        size_type chars_len) const
size_type string::rfind (const char* chars, size_type idx,
                         size_type chars_len) const
```

- 这些函数都从索引 *idx* 开始搜寻“与字符数组 *chars* 内的 *chars_len* 个字符内容相同”的第一个或最后一个子字符串。
- 函数 find() 正向（*forward*）搜寻，并返回第一个子字符串。
- 函数 rfind() 逆向（*backward*）搜索，并返回最后一个子字符串。
- 如果这些函数成功，就返回子字符串的第一字符索引；否则返回 string::npos。
- *chars* 必须包含至少 *chars_len* 个字符，字符可为任意值，'\0' 无特殊含义。

搜寻第一个匹配字符

```
size_type string::find_first_of (const string& str) const
size_type string::find_first_of (const string& str, size_type idx) const
size_type string::find_first_not_of (const string& str) const
size_type string::find_first_not_of (const string& str, size_type idx) const
```

- 这些函数从索引 *idx* 处开始搜寻 *this 之中属于（或不属于）*str* 的第一个字符。
- 如果函数成功，就返回子字符串或字符索引；否则返回 *string*::npos。

```
size_type string::find_first_of (const char* cstr) const
size_type string::find_first_of (const char* cstr, size_type idx) const
size_type string::find_first_not_of (const char* cstr) const
size_type string::find_first_not_of (const char* cstr, size_type idx) const
```

- 这些函数从索引 *idx* 处开始搜寻*this 之中属于（或不属于）*C-string cstr* 的第一个字符。
- 如果函数成功，就返回字符索引；否则返回 *string*::npos。
- *cstr* 不得为 null 指针。

```
size_type string::find_first_of (const char* chars, size_type idx,
                                 size_type chars_len) const
size_type string::find_first_not_of (const char* chars, size_type idx,
                                     size_type chars_len) const
```

- 这些函数从索引 *idx* 处开始搜寻*this 之中“属于（或不属于）字符数组 *chars* 内的前 *chars_len* 个字符”的第一个字符。
- 如果函数成功，就返回字符索引；否则返回 *string*::npos。
- *chars* 必须包含至少 *chars_len* 个字符，字符可为任意值，'\0' 无特殊含义。

```
size_type string::find_first_of (char c) const
size_type string::find_first_of (char c, size_type idx) const
size_type string::find_first_not_of (char c) const
size_type string::find_first_not_of (char c, size_type idx) const
```

- 这些函数从索引 *idx* 处开始搜寻*this 之中等于（或不等于）字符 *c* 的第一个字符。
- 如果函数成功，就返回字符索引；否则返回 *string*::npos。

搜寻最后一个匹配字符

```
size_type string::find_last_of (const string& str) const
size_type string::find_last_of (const string& str, size_type idx) const
size_type string::find_last_not_of (const string& str) const
size_type string::find_last_not_of (const string& str, size_type idx) const
```

- 这些函数从索引 *idx* 处开始搜寻 *this 之中属于（或不属于）*str* 的最后一个字符。
- 如果函数成功，就返回子字符串或字符索引；否则返回 *string*::npos。

```
size_type string::find_last_of (const char* cstr) const
size_type string::find_last_of (const char* cstr, size_type idx) const
size_type string::find_last_not_of (const char* cstr) const
size_type string::find_last_not_of (const char* cstr, size_type idx) const
```

- 这些函数从索引 *idx* 处开始搜寻*this 之中属于（或不属于）*C-string cstr* 的最后一个字符。
- 如果函数成功，就返回字符索引；否则返回 *string*::npos。
- *cstr* 不得为 null 指标。

```
size_type string::find_last_of (const char* chars, size_type idx,
                                size_type chars_len) const

size_type string::find_last_not_of (const char* chars, size_type idx,
                                    size_type chars_len) const
```

- 这些函数从索引 *idx* 处开始搜寻*this 之中“属于（或不属于）字符数组 *chars* 内的前 *chars_len* 个字符”的最后一个字符。
- 如果函数成功，就返回字符索引；否则返回 *string*::npos。
- *chars* 必须包含至少 *chars_len* 个字符，字符可为任意值，'\0' 无特殊含义。

```
size_type string::find_last_of (char c) const
size_type string::find_last_of (char c, size_type idx) const
size_type string::find_last_not_of (char c) const
size_type string::find_last_not_of (char c, size_type idx) const
```

- 这些函数从索引 *idx* 处开始搜寻*this 之中等于（或不等于）字符 *c* 的最后一个字符。
- 如果函数成功，就返回字符索引；否则返回 *string*::npos。

11.3.9 子字符串及字符串接合（String Concatenation）

```
string string::substr () const
string string::substr (size_type idx) const
string string::substr (size_type idx, size_type len) const
```

- 这几种形式都返回*this 之内“从索引 idx 开始的最多 len 个字符”所组成的子字符串。
- 如果没有 len，则将“余下的所有字符”当做子字符串返回。
- 如果没有 idx 和 len，则返回字符串副本。
- 如果 idx > size()，则抛出 out_of_range 异常。

```
string operator+ (const string& str1, const string& str2)
string operator+ (const string& str, const char* cstr)
string operator+ (const char* cstr, const string& str)
string operator+ (const string& str, char c)
string operator+ (char c, const string& str)
```

- 所有形式都可以接合两个操作数内的所有字符，并返回接合后的字符串。
- 操作数可以是下列任意一种：
  - 一个 *string*
  - 一个 *C-string*
  - 单一字符
- 如果接合结果超出最大字符数，则所有形式都抛出 length_error 异常。

11.3.10 I/O 函数

```
ostream& operator << (ostream& strm, const string& str)
```

- 将 str 内的字符写入 stream strm。
- 如果 strm.width()大于 0，则至少写入 width()个字符，然后 width()被设置为 0。
- ostream 的型别是 basic_ostream<char>，具体型别取决于字符型别（参见 13.2.1 节，p588）。

```
istream& operator >> (istream& strm, string& str)
```

- 从 strm 读取下一个单字（字符串）的所有字符，放到 str 中。
- 如果 strm 的 skipws 标志被设立，则前导空格将忽略不计。
- 字符读取动作遇到下面情形之一即结束：
  - strm.width()大于 0，且已存入 width()个字符
  - strm.good()为 false（可能导致相应的异常）

- 对下一个字符 *c*，`isspace(c,strm.getloc())`为 true。
- 已存入 *str*.`max_size()`个字符。

- 视情况重新分配内存。
- *istream* 的型别是 `basic_istream<char>`，具体型别取决于字符型别（参见 13.2.1 节, p588）。

```
istream& getline (istream& strm, string& str)
istream& getline (istream& strm, string& str, char delim)
```

- 从 *strm* 读取下一整行的所有字符到字符串 *str* 内。
- 读取所有字符（包括前导空格）直到下列情形之一发生：
  - *strm*.`good()`为 false（可能导致相应的异常）
  - 读到 *delim* 或 *strm*.`widen('\n')`。
  - 已读入 *str*.`max_size()`个字符。
- 行分隔符（line delimiter）乃是从参数中获取。
- 视情况重新分配内存。
- *istream* 的型别是 `basic_istream<char>`，具体型别取决于字符型别（参见 13.2.1 节, p588）。

11.3.11 产生迭代器

```
iterator string::begin ()
const_iterator string::begin () const
```

- 两种形式都返回一个 random access（随机存取）迭代器，指向字符串头部（首字符位置）。
- 如果字符串为空，以上调用等价于 `end()`。

```
iterator string::end ()
const_iterator string::end () const
```

- 两种形式都返回一个 random access（随机存取）迭代器，指向字符串尾部（最后字符的下一个位置）。
- end 处并未定义字符，所以`*s.end()`会导致未定义行为。
- 如果字符串为空，以上调用等价于 `begin()`。

```
reverse_iterator string::rbegin ()
const_reverse_iterator string::rbegin () const
```

- 两种形式都返回一个 reverse random access（逆向随机存取）迭代器，指向倒数第一个字符（亦即最后一个字符位置）。
- 如果字符串为空，以上调用等价于 rend()。
- 关于逆向迭代器，详见 7.4.1 节, p264。

```
reverse_iterator string::rend ()
const_reverse_iterator string::rend () const
```

- 两种形式都返回一个 reverse random access（逆向随机存取）迭代器，指向倒数最后一个元素的下一个位置（亦即第一个字符的前一个位置）。
- *rend* 处并未定义字符，所以*s.rend()会导致未定义行为。
- 如果字符串为空，以上调用等价于 rbegin()。
- 关于逆向迭代器，详见 7.4.1 节, p264。

11.3.12 对配置器（allocator）的支持

就像其它运用配置器的 classes 一样，*strings* 也提供常见的配置器相关支持。

```
string::allocator_type
```

- 这是配置器型别。
- 同时也是 basic_string<> 的第三个 template 参数。
- 对 string 型别而言，等价于 allocator<char>。

```
allocator_type string::get_allocator () const
```

- 返回字符串的内存模型（memory model）。

Strings 的所有构造函数都具有可有可无的配置器参数。根据 *C++ Standard*，下面列出所有 *string* 构造函数：

```
namespace std {
   template<class charT,
            class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
   class basic_string {
     public:
        // default constructor
        explicit basic_string(const Allocator& a = Allocator());

        // copy constructor and substrings
```

```
        basic_string(const basic_string& str,
                     size_type str_idx = 0,
                     size_type str_num = npos);
        basic_string(const basic_string& str,
                     size_type str_idx, size_type str_num,
                     const Allocator&);

        // constructor for C-strings
        basic_string(const charT* cstr,
                     const Allocator& a = Allocator());

        // constructor for character arrays
        basic_string(const charT* chars, size_type chars_len,
                     const Allocator& a = Allocator());

        // constructor for num occurrences of a character
        basic_string(size_type num, charT c,
                     const Allocator& a = Allocator());

        // constructor for a range of characters
        template<class InputIterator>
        basic_string(InputIterator beg, InputIterator end,
                     const Allocator& a = Allocator());
        ...
    };
}
```

这些构造函数的行为一如 11.3.2 节, p508 所介绍，并带有额外机能：允许你传递你自己的内存模型对象（memory model object）。如果 *string* 系以另一个 *string* 为初值，配置器也会被复制[10]。关于配置器，详见第 15 章。

[10] *C++ Standard* 最初版本规定：当字符串被复制时，应采用预设配置器。但此说法并无多大意义，所以又有人提议对此行为进行修改，最后形成目前的结果。