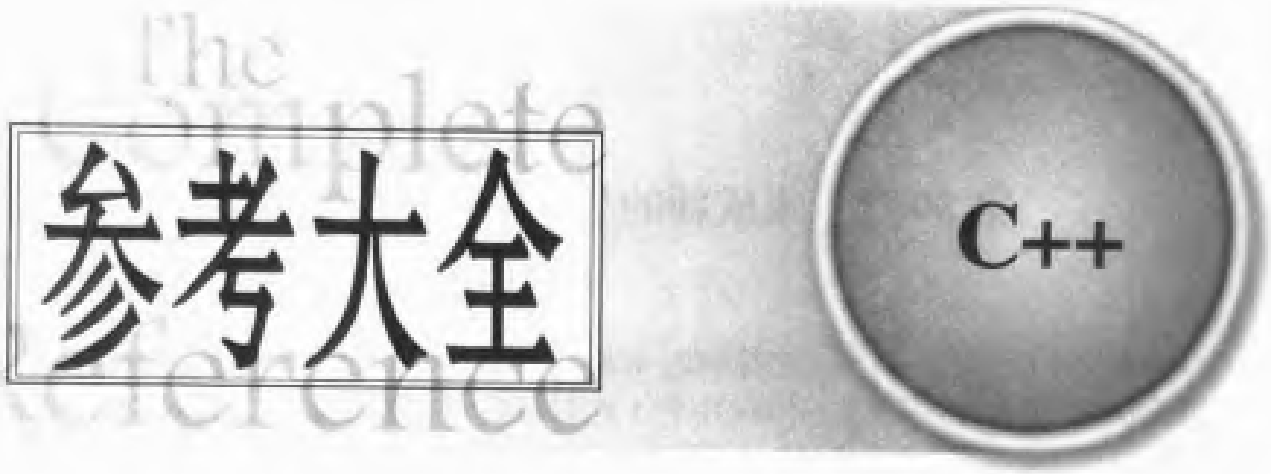

第五部分

C++ 应用程序范例

本书的第五部分提供两个C++应用的范例，目的有两个：首先，这些例子会帮助我们演示面向对象编程的益处；第二，它们也显示了如何使用C++来解决两种不同类型的编程问题。

第39章　集成新的类：自定义字符串类

本章设计并实现了一个小的字符串类。如你所知，标准C++提供了一个称为basic_string的特征全面的、强大的字符串类。本章的目的不是开发这个类的替代物，而是告诉你如何把新的数据类型很容易地添加并集成到C++环境中。字符串类的创建是这个过程中一个典型的例子。在过去，许多程序员在开发自己的字符串类时总是想到他们的面向对象的技术。在本章中，我们将做同样的事情。

虽然本章开发的字符串类的范例比标准C++提供的要简单得多，它确实有一个优点：它让你全面控制字符串的实现与操作。在某些情况下，这很有用。

39.1　StrType类

粗略地讲，我们的字符串类是在标准库所提供的类上建模的。当然，它不是太大，也不复杂。这里定义的字符串类将满足下列要求：

- 可以使用赋值运算符对字符串进行赋值。
- 既可以把字符串对象、也可以把被引用的字符串分配给字符串对象。
- 两个字符串对象的连接是通过+运算符完成的。
- 使用-运算符来执行子串的删除。
- 使用关系运算符进行字符串比较。
- 使用一个引用字符串或另一个字符串对象来初始化字符串对象。
- 字符串必须可以是变长和任意的。这意味着每个字符串的存储是动态分配的。
- 将会提供一种方法来把字符串对象转换为以null结束的字符串。

尽管我们的字符串类一般来讲没有标准字符串类强大，它确实包含了一个没有为basic_string定义的特征：通过-运算符实现的子串删除。

管理字符串的类称为StrType，它的声明如下所示：

```
class StrType {
  char *p;
  int size;
public:
  StrType();
  StrType(char *str);
  StrType(const StrType &o); // copy constructor

  ~StrType() { delete [] p; }

  friend ostream &operator<<(ostream &stream, StrType &o);
  friend istream &operator>>(istream &stream, StrType &o);

  StrType operator=(StrType &o); // assign a StrType object
  StrType operator=(char *s); // assign a quoted string
```

```
  StrType operator+(StrType &o); // concatenate a StrType object
  StrType operator+(char *s); // concatenate a quoted string
  friend StrType operator+(char *s, StrType &o); /* concatenate
                  a quoted string with a StrType object */

  StrType operator-(StrType &o); // subtract a substring
  StrType operator-(char *s); // subtract a quoted substring

  // relational operations between StrType objects
  int operator==(StrType &o) { return !strcmp(p, o.p); }
  int operator!=(StrType &o) { return strcmp(p, o.p); }
  int operator<(StrType &o) { return strcmp(p, o.p) < 0; }
  int operator>(StrType &o) { return strcmp(p, o.p) > 0; }
  int operator<=(StrType &o) { return strcmp(p, o.p) <= 0; }
  int operator>=(StrType &o) { return strcmp(p, o.p) >= 0; }

  // operations between StrType objects and quoted strings
  int operator==(char *s) { return !strcmp(p, s); }
  int operator!=(char *s) { return strcmp(p, s); }
  int operator<(char *s) { return strcmp(p, s) < 0; }
  int operator>(char *s) { return strcmp(p, s) > 0; }
  int operator<=(char *s) { return strcmp(p, s) <= 0; }
  int operator>=(char *s) { return strcmp(p, s) >= 0; }

  int strsize() { return strlen(p); } // return size of string
  void makestr(char *s) { strcpy(s, p); } // make quoted string

  operator char *() { return p; } // conversion to char *
};
```

StrType 的私有部分仅包含两项：p 和 size。在创建字符串对象时，使用 new 来动态分配存储字符串所用的内存，指向那个内存的指针被放到 p 中。p 所指的字符串将是一个正常的以 null 结束的字符数组。尽管技术上并不需要，但字符串的长度存储在 size 中。因为 p 所指的字符串是一个以 null 结束的字符串，所以在每次需要时可以计算字符串的长度。然而，可以看到，StrType 成员函数如此频繁地使用这个值，以致于不能评判对 strlen( )的重复调用是否正确。

下面的几节详细讨论 StrType 类的工作原理。

39.2　构造函数和析构函数

StrType对象有三种声明方法。没有任何初始化说明，可以用引号括起来的字符串做初始化部分或者用一个 StrType 对象作初值来说明。下面的构造函数支持这三种操作。

```
// No explicit initialization.
StrType::StrType() {
  size = 1; // make room for null terminator
  try {
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Allocation error\n";
    exit(1);
  }
  strcpy(p, "");
```

```
}

// Initialize using a quoted string.
StrType::StrType(char *str) {
  size = strlen(str) + 1; // make room for null terminator
  try {
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Allocation error\n";
    exit(1);
  }
  strcpy(p, str);
}

// Initialize using a StrType object.
StrType::StrType(const StrType &o) {
  size = o.size;
  try {
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Allocation error\n";
    exit(1);
  }
  strcpy(p, o.p);
}
```

当创建一个不带初始化部分的StrType时，它被赋予一个null字符串。尽管字符串可能还未定义，但所有的StrType对象包含一个有效的、以null结束的字符串。这样可以简化其他几个成员函数。

当用一个用引号括起来的字符串初始化一个StrType时，首先要决定它的长度，这个值存放在size中。然后用new分配足够的空间，并把初始化字符串复制到p所指的内存。

当用一个StrType对象初始化另一个时，处理过程类似于使用引号括起来的字符串的情况。惟一的差别是，字符串的长度是已知的，不必通过计算来得到。StrType构造函数的这个版本也是这个类的拷贝构造函数。当用一个StrType对象初始化另一个对象时，将调用这个构造函数。这意味着当创建临时对象及当把类型StrType的对象传递给函数时，调用这个构造函数（关于拷贝构造函数的讨论，请参见第14章）。

如果给出前面的三个构造函数，下面的声明是允许的：

```
StrType x("my string"); // use quoted string
StrType y(x); // use another object
StrType z; // no explicit initialization
```

StrType析构函数只释放由p所指的内存空间。

39.3 字符串I/O

因为字符串的输入、输出是很常见的，所以StrType重载了运算符<<和>>，如下所示：

```
// Output a string.
ostream &operator<<(ostream &stream, StrType &o)
```

```
{
  stream << o.p;
  return stream;
}

// Input a string.
istream &operator>>(istream &stream, StrType &o)
{
  char t[255]; // arbitrary size - change if necessary
  int len;

  stream.getline(t, 255);
  len = strlen(t) + 1;

  if(len > o.size) {
    delete[] o.p;
    try {
      o.p = new char[len];
    } catch (bad_alloc xa) {
      cout << "Allocation error\n";
      exit(1);
    }
    o.size = len;
  }
  strcpy(o.p, t);
  return stream;
}
```

可以看出，输出是非常简单的。然而，注意，参数o是通过引用传递的。由于StrType对象可能很大，因此通过引用传递比通过值传递更有效。所以，所有的StrType参数都通过引用传递（你创建的任何带StrType参数的函数应该一样）。

输入一个StrType比输出要稍难一些。首先，使用getline()读字符串。可被输入的最大字符串长度是254加null结束符。正如注释所指示的，如果愿意，可以改变这一点。可读取字符，直到遇到换行。一旦读入了字符串，如果新字符串的长度超过了当前o所含的那个，就释放那个内存并分配一个更大的量，然后把新字符串复制到其中。

39.4 赋值函数

把一个字符串赋给一个StrType对象有两种方法。第一种方法是，可以把一个StrType对象赋给另一个。第二种方法是，可以把用引号括起来的字符串赋给StrType对象。下面两个重载operator=()函数实现了这些操作。

```
// Assign a StrType object to a StrType object.
StrType StrType::operator=(StrType &o)
{
  StrType temp(o.p);

  if(o.size > size) {
    delete[] p; // free old memory
    try {
      p = new char[o.size];
```

```
    } catch (bad_alloc xa) {
      cout << "Allocation error\n";
      exit(1);
    }
    size = o.size;
  }

  strcpy(p, o.p);
  strcpy(temp.p, o.p);

  return temp;
}

// Assign a quoted string to a StrType object.
StrType StrType::operator=(char *s)
{
  int len = strlen(s) + 1;
  if(size < len) {
    delete [] p;
    try {
      p = new char[ len];
    } catch (bad_alloc xa) {
      cout << "Allocation error\n";
      exit(1);
    }
    size = len;
  }
  strcpy(p, s);
  return *this;
}
```

这两个函数首先检查p当前所指的目标StrType对象的内存是否足以存储即将拷入的内容。如果不够，则释放旧的内存，并分配新的内存，然后把字符串复制到对象中并返回结果。这些函数允许下面这些类型的赋值。

```
StrType x("test"), y;

y = x; // StrType object to StrType object

x = "new string for x"; // quoted string to StrType object
```

每一个赋值函数返回所赋的值（即右边的值），以便它支持下列的多重赋值语句：

```
StrType x, y, z;

x = y = z = "test";
```

39.5 连接

两个字符串的连接用运算符 + 实现。StrType 类允许下面三种连接情况：

- 一个 StrType 对象和另一个 StrType 对象连接。
- 一个 StrType 对象和一个用引号括起来的字符串连接。
- 一个用引号括起来的字符串和一个 StrType 对象连接。

在这些情况下，使用运算符 + 产生的结果是两个操作数连接而成的 StrType 对象。它并不实际改变任何一个操作数。

下例是重载的 operator+( )函数。

```
// Concatenate two StrType objects.
StrType StrType::operator+(StrType &o)
{
  int len;
  StrType temp;

  delete [] temp.p;
  len = strlen(o.p) + strlen(p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
  } catch (bad_alloc xa) {
    cout << "Allocation error\n";
    exit(1);
  }
  strcpy(temp.p, p);

  strcat(temp.p, o.p);

  return temp;
}

// Concatenate a StrType object and a quoted string.
StrType StrType::operator+(char *s)
{
  int len;
  StrType temp;

  delete [] temp.p;

  len = strlen(s) + strlen(p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
  } catch (bad_alloc xa) {
    cout << "Allocation error\n";
    exit(1);
  }
  strcpy(temp.p, p);

  strcat(temp.p, s);

  return temp;
}

// Concatenate a quoted string and a StrType object.
StrType operator+(char *s, StrType &o)
{
  int len;
  StrType temp;
```

```
  delete [] temp.p;

  len = strlen(s) + strlen(o.p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
  } catch (bad_alloc xa) {
    cout << "Allocation error\n";
    exit(1);
  }
  strcpy(temp.p, s);

  strcat(temp.p, o.p);

  return temp;
}
```

这三个函数的工作原理基本相同：首先创建一个临时的StrType对象temp，这个对象将存放相加的结果，并且它是由函数返回的对象；接着释放由temp.p所指的内存，其原因是由于没有明确的初始化，所以在创建temp时只分配了一个字节（作为占位符）；然后分配足够的内存以容纳两个字符串的连接结果；最后，把两个字符串复制到temp.p指向的内存并返回temp。

39.6 子字符串减法

子字符串减法是一个不存在于basic_string中的有用的字符串函数。通过StrType类，使子字符串减法可以实现从一个字符串移去指定的子字符串的所有具体值。子字符串减法通过运算符“-”实现。

StrType类支持两种子字符串减法。一种是用一个StrType对象减去另一个StrType对象，另一个是把一个用引号括起来的字符串从一个StrType对象中减去。下面是两个operator-()函数。

```
// Subtract a substring from a string using StrType objects.
StrType StrType::operator-(StrType &substr)
{
  StrType temp(p);
  char *s1;
  int i, j;

  s1 = p;
  for(i=0; *s1; i++) {
    if(*s1!=*substr.p) { // if not first letter of substring
      temp.p[i] = *s1; // then copy into temp
      s1++;
    }
    else {
      for(j=0; substr.p[j]==s1[j] && substr.p[j]; j++) ;
      if(!substr.p[j]) { // is substring, so remove it
        s1 += j;
        i--;
      }
      else { // is not substring, continue copying
        temp.p[i] = *s1;
```

```
        s1++;
      }
    }
  }
  temp.p[i] = '\0';
  return temp;
}

// Subtract quoted string from a StrType object.
StrType StrType::operator-(char *substr)
{
  StrType temp(p);
  char *s1;
  int i, j;

  s1 = p;
  for(i=0; *s1; i++) {
    if(*s1!=*substr) { // if not first letter of substring
      temp.p[i] = *s1; // then copy into temp
      s1++;
    }
    else {
      for(j=0; substr[j]==s1[j] && substr[j]; j++) ;
      if(!substr[j]) { // is substring, so remove it
        s1 += j;
        i--;
      }
      else { // is not substring, continue copying
        temp.p[i] = *s1;
        s1++;
      }
    }
  }
  temp.p[i] = '\0';
  return temp;
}
```

这些函数把左操作数的内容复制到temp中，然后移去在这个过程中所有右操作数指定的子字符串的任何具体值，并返回作为结果的StrType。注意，在这个过程中没有修改任何操作数。

StrType类允许如下的子字符串减法：

```
StrType x("I like C++"), y("like");
StrType z;

z = x - y; // z will contain "I C++"

z = x - "C++"; // z will contain "I like "

// multiple occurrences are removed
z = "ABCDABCD";
x = z -"A"; // x contains "BCDBCD"
```

39.7 关系运算符

StrType类支持所有用于字符串的关系运算符。重载的关系运算符在StrType类的声明中定义。为方便起见，我们在此重复一遍：

```
// relational operations between StrType objects
int operator==(StrType &o) { return !strcmp(p, o.p); }
int operator!=(StrType &o) { return strcmp(p, o.p); }
int operator<(StrType &o) { return strcmp(p, o.p) < 0; }
int operator>(StrType &o) { return strcmp(p, o.p) > 0; }
int operator<=(StrType &o) { return strcmp(p, o.p) <= 0; }
int operator>=(StrType &o) { return strcmp(p, o.p) >= 0; }

// operations between StrType objects and quoted strings
int operator==(char *s) { return !strcmp(p, s); }
int operator!=(char *s) { return strcmp(p, s); }
int operator<(char *s) { return strcmp(p, s) < 0; }
int operator>(char *s) { return strcmp(p, s) > 0; }
int operator<=(char *s) { return strcmp(p, s) <= 0; }
int operator>=(char *s) { return strcmp(p, s) >= 0; }
```

关系运算符直接易懂，理解它的实现原理是不会有什么问题的。但要记住，StrType类是在两个StrType对象之间，或左操作数是一个StrType对象而右操作数是一个用引号括起来的字符串之间进行比较的。如果要把用括号括起来的字符串作为左操作数而把StrType对象作为右操作数，就必须另外增加一个关系函数。

如果给出了由StrType定义的重载关系运算符函数，则下面这些类型的字符串比较是允许的。

```
StrType x("one"), y("two"), z("three");

if(x < y) cout << "x less than y";

if(z=="three") cout << "z equals three";

y = "o";
z = "ne";
if(x==(y+z)) cout << "x equals y+z";
```

39.8 各种字符串函数

StrType类定义了三个函数，以便StrType对象和C++编程环境更彻底地结合起来。它们是strsize( )，makestr( )和转换函数operator char*( )。这些函数在StrType声明中定义，如下所示：

```
int strsize() { return strlen(p); } // return size of string
void makestr(char *s) { strcpy(s, p); } // make quoted string
operator char *(){ return p; } // conversion to char *
```

前两个函数很容易理解。可以看出，函数strsize( )返回p所指的字符串的长度。因为字符串的长度可以不同于存储在size变量中的值（例如，因为一个更短的字符串的赋值），所以通过调用strlen( )计算这个长度。函数makestr( )把p所指的字符串复制到一个字符串数组。如果有了StrType对象，当希望得到以null结束的字符串时，这个函数非常有用。

转换函数 operator char*( )返回对象中指向字符串的指针 p。此函数允许一个 StrType 对象用在可以使用以 null 结束的字符串的任何地方，例如，下面的代码是有效的：

```
StrType x("Hello");
char s[20];

// copy a string object using the strcpy() function
strcpy(s, x); // automatic conversion to char *
```

当定义了转换的表达式中涉及一个对象时，转换函数自动执行。在这种情况下，由于函数 strcpy( )的原型告诉编译器它的第二个变元的类型是 char*，所以自动执行从 StrType 到 char* 的转换，并返回一个指向包含在 x 里的字符串的指针。然后 strcpy( )使用这个指针把字符串复制到 s 中。有了转换函数，就可以把一个 StrType 对象用在一个以 null 结束的字符串作为任何函数的变元的地方，其中的函数带有一个 char* 类型的变元。

注意： 转换为 char* 会破坏封装性。因为函数一旦有了指向对象的字符串的指针，它就有可能直接修改这个字符串，从而绕过 StrType 成员函数，且不为那个对象所知。因此，使用对 char* 的转换时必须十分小心。通过使到 char* 的转换返回一个 const 指针，可以防止字符串被修改。用这种方法，保留了封装性。你可能想亲自试试这种改变。

39.9　完整的 StrType 类

下面是一个完整的 StrType 类列表，函数 main( )演示了它的特点。

```
#include <iostream>
#include <new>
#include <cstring>
#include <cstdlib>
using namespace std;

class StrType {
  char *p;
  int size;
public:
  StrType();
  StrType(char *str);
  StrType(const StrType &o); // copy constructor

  ~StrType() { delete [] p; }

  friend ostream &operator<<(ostream &stream, StrType &o);
  friend istream &operator>>(istream &stream, StrType &o);

  StrType operator=(StrType &o); // assign a StrType object
  StrType operator=(char *s); // assign a quoted string

  StrType operator+(StrType &o); // concatenate a StrType object
  StrType operator+(char *s); // concatenate a quoted string
  friend StrType operator+(char *s, StrType &o); /* concatenate
                a quoted string with a StrType object */
  StrType operator-(StrType &o); // subtract a substring
  StrType operator-(char *s); // subtract a quoted substring
```

```
  // relational operations between StrType objects
  int operator==(StrType &o) { return !strcmp(p, o.p); }
  int operator!=(StrType &o) { return strcmp(p, o.p); }
  int operator<(StrType &o) { return strcmp(p, o.p) < 0; }
  int operator>(StrType &o) { return strcmp(p, o.p) > 0; }
  int operator<=(StrType &o) { return strcmp(p, o.p) <= 0; }
  int operator>=(StrType &o) { return strcmp(p, o.p) >= 0; }

  // operations between StrType objects and quoted strings
  int operator==(char *s) { return !strcmp(p, s); }
  int operator!=(char *s) { return strcmp(p, s); }
  int operator<(char *s) { return strcmp(p, s) < 0; }
  int operator>(char *s) { return strcmp(p, s) > 0; }
  int operator<=(char *s) { return strcmp(p, s) <= 0; }
  int operator>=(char *s) { return strcmp(p, s) >= 0; }

  int strsize() { return strlen(p); } // return size of string
  void makestr(char *s) { strcpy(s, p); } // null-terminated string
  operator char *() { return p; } // conversion to char *
};

// No explicit initialization.
StrType::StrType() {
  size = 1; // make room for null terminator
  try {
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Allocation error\n";
    exit(1);
  }
  strcpy(p, "");
}

// Initialize using a quoted string.
StrType::StrType(char *str) {
  size = strlen(str) + 1; // make room for null terminator
  try {
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Allocation error\n";
    exit(1);
  }
  strcpy(p, str);
}

// Initialize using a StrType object.
StrType::StrType(const StrType &o) {
  size = o.size;
  try {
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Allocation error\n";
    exit(1);
```

```
  }
  strcpy(p, o.p);
}

// Output a string.
ostream &operator<<(ostream &stream, StrType &o)
{
  stream << o.p;
  return stream;
}

// Input a string.
istream &operator>>(istream &stream, StrType &o)
{
  char t[255]; // arbitrary size - change if necessary
  int len;

  stream.getline(t, 255);
  len = strlen(t) + 1;

  if(len > o.size) {
    delete [] o.p;
    try {
      o.p = new char[len];
    } catch (bad_alloc xa) {
      cout << "Allocation error\n";
      exit(1);
    }
    o.size = len;
    }
  strcpy(o.p, t);
  return stream;
}

// Assign a StrType object to a StrType object.
StrType StrType::operator=(StrType &o)
{
  StrType temp(o.p);

  if(o.size > size) {
    delete [] p; // free old memory
    try {
      p = new char[o.size];
    } catch (bad_alloc xa) {
      cout << "Allocation error\n";
      exit(1);
    }
    size = o.size;
  }

  strcpy(p, o.p);
  strcpy(temp.p, o.p);

  return temp;
}
```

```
// Assign a quoted string to a StrType object.
StrType StrType::operator=(char *s)
{
  int len = strlen(s) + 1;
  if(size < len) {
    delete [] p;
    try {
      p = new char[len];
    } catch (bad_alloc xa) {
      cout << "Allocation error\n";
      exit(1);
    }
    size = len;
  }
  strcpy(p, s);
  return *this;
}

// Concatenate two StrType objects.
StrType StrType::operator+(StrType &o)
{
  int len;
  StrType temp;

  delete [] temp.p;
  len = strlen(o.p) + strlen(p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
  } catch (bad_alloc xa) {
    cout << "Allocation error\n";
    exit(1);
  }
  strcpy(temp.p, p);

  strcat(temp.p, o.p);

  return temp;
}

// Concatenate a StrType object and a quoted string.
StrType StrType::operator+(char *s)
{
  int len;
  StrType temp;

  delete [] temp.p;

  len = strlen(s) + strlen(p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
  } catch (bad_alloc xa) {
    cout << "Allocation error\n";
    exit(1);
```

```
  }
  strcpy(temp.p, p);

  strcat(temp.p, s);

  return temp;
}

// Concatenate a quoted string and a StrType object.
StrType operator+(char *s, StrType &o)
{
  int len;
  StrType temp;

  delete [] temp.p;

  len = strlen(s) + strlen(o.p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
  } catch (bad_alloc xa) {
    cout << "Allocation error\n";
    exit(1);
  }
  strcpy(temp.p, s);

  strcat(temp.p, o.p);

  return temp;
}

// Subtract a substring from a string using StrType objects.
StrType StrType::operator-(StrType &substr)
{
  StrType temp(p);
  char *s1;
  int i, j;

  s1 = p;
  for(i=0; *s1; i++) {
    if(*s1!=*substr.p) { // if not first letter of substring
      temp.p[i] = *s1; // then copy into temp
      s1++;
    }
    else {
      for(j=0; substr.p[j]==s1[j] && substr.p[j]; j++) ;
      if(!substr.p[j]) { // is substring, so remove it
        s1 += j;
        i--;
      }
      else { // is not substring, continue copying
        temp.p[i] = *s1;
        s1++;
      }
    }
```

```
  }
  temp.p[i] = '\0';
  return temp;
}

// Subtract quoted string from a StrType object.
StrType StrType::operator-(char *substr)
{
  StrType temp(p);
  char *s1;
  int i, j;

  s1 = p;
  for(i=0; *s1; i++) {
    if(*s1!=*substr) { // if not first letter of substring
      temp.p[i] = *s1; // then copy into temp
      s1++;
    }
    else {
      for(j=0; substr[j]==s1[j] && substr[j]; j++) ;
      if(!substr[j]) { // is substring, so remove it
        s1 += j;
        i--;
      }
      else { // is not substring, continue copying
        temp.p[i] = *s1;
          s1++;
      }
    }
  }
  temp.p[i] = '\0';
  return temp;
}

int main()
{
  StrType s1("A sample session using string objects.\n");
  StrType s2(s1);
  StrType s3;
  char s[80];

  cout << s1 << s2;

  s3 = s1;
  cout << s1;

  s3.makestr(s);
  cout << "Convert to a string: " << s;

  s2 = "This is a new string.";
  cout << s2 << endl;

  StrType s4(" So is this.");
  s1 = s2+s4;
  cout << s1 << endl;
```

```
  if(s2==s3) cout << "Strings are equal.\n";
  if(s2!=s3) cout << "Strings are not equal.\n";
  if(s1<s4) cout << "s1 less than s4\n";
  if(s1>s4) cout << "s1 greater than s4\n";
  if(s1<=s4) cout << "s1 less than or equals s4\n";
  if(s1>=s4) cout << "s1 greater than or equals s4\n";

  if(s2 > "ABC") cout << "s2 greater than ABC\n\n";

  s1 = "one two three one two three\n";
  s2 = "two";
  cout << "Initial string: " << s1;
  cout << "String after subtracting two: ";
  s3 = s1 - s2;
  cout << s3;
  cout << endl;
  s4 = "Hi there!";
  s3 = s4 + " C++ strings are fun\n";
  cout << s3;
  s3 = s3 - "Hi there!";
  s3 = "Aren't" + s3;
  cout << s3;

  s1 = s3 - "are ";
  cout << s1;
  s3 = s1;

  cout << "Enter a string: ";
  cin >> s1;
  cout << s1 << endl;
  cout << "s1 is " << s1.strsize() << " characters long.\n";

  puts(s1); // convert to char *

  s1 = s2 = s3;
  cout << s1 << s2 << s3;

  s1 = s2 = s3 = "Bye ";
  cout << s1 << s2 << s3;

  return 0;
}
```

上述程序产生下列输出：

```
A sample session using string objects.
A sample session using string objects.
A sample session using string objects.
Convert to a string: A sample session using string objects.
This is a new string.
This is a new string. So is this.
Strings are not equal.
s1 greater than s4
s1 greater than or equals s4
s2 greater than ABC
```

```
Initial string: one two three one two three
String after subtracting two: one three one three

Hi there! C++ strings are fun
Aren't C++ strings are fun
Aren't C++ strings fun
Enter a string: I like C++
s1 is 10 characters long.
I like C++
Aren't C++ strings fun
Aren't C++ strings fun
Aren't C++ strings fun
Bye Bye Bye
```

此输出假设字符串“I like C++”是在提示输入时由用户键入的。

为了方便地访问StrType类，可以把函数main( )从前面的程序清单里删去，把剩余的部分放入文件STR.H中。这样，在需要使用StrType类时，只需在程序中包含这个头文件即可。

39.10 使用StrType类

在结束本章时，我们给出两个演示StrType类的小例子。正如你将要看到的，因为为它定义的运算符和它到char *的转换函数，StrType被完全集成进C++编程环境中。即，可以像使用标准C++定义的任何类型那样使用它。

第一个例子用StrType对象创建了一个简单的同义词词典。它首先创建了一个二维的StrType对象数组，在每一组字符串中，第一个包含可查找的关键字；第二个串是一个相关词表。程序提示输入一个词，如果这个词在词典中，就会显示其相关词。程序虽然很简单，但要注意，这里的字符串操作之所以如此清晰明了，是因为使用了StrType类和它的运算符（记住，头文件STR.H包含StrType类）。

```
#include "str.h"
#include <iostream>
using namespace std;

StrType thesaurus[][2] = {
  "book", "volume, tome",
  "store", "merchant, shop, warehouse",
  "pistol", "gun, handgun, firearm",
  "run", "jog, trot, race",
  "think", "muse, contemplate, reflect",
  "compute", "analyze, work out, solve",
  "", ""
};

int main()
{
  StrType x;

  cout << "Enter word: ";
  cin >> x;

  int i;
```

```
  for(i=0; thesaurus[i][0]!=""; i++)
    if(thesaurus[i][0]==x) cout << thesaurus[i][1];

  return 0;
}
```

下面这个例子用一个StrType对象检查是否有一个给定文件名的程序的可执行版本。要使用这个程序，在命令行指定不带扩展名的文件名，然后程序根据那个名字加上扩展名重复查找可执行文件，试着打开它，并报告查找的结果（如果文件不存在，是不能打开的）。在试过某个扩展名之后，就从文件名里减去这个扩展名，然后加上新的扩展名。StrType类及其运算符使字符串操作变得清晰易懂。

```
#include "str.h"
#include <iostream>
#include <fstream>
using namespace std;

// executable file extensions
char ext[3][4] = {
  "EXE",
  "COM",
  "BAT"
};

int main(int argc, char *argv[])
{
  StrType fname;
  int i;

  if(argc!=2) {
    cout << "Usage: fname\n";
    return 1;
  }

  fname = argv[1];

  fname = fname + "."; // add period
  for(i=0; i<3; i++) {
    fname = fname + ext[i]; // add extension
    cout << "Trying " << fname << " ";
    ifstream f(fname);
    if(f) {
      cout << "- Exists\n";
      f.close();
    }
    else cout << "- Not found\n";
    fname = fname - ext[i]; // subtract extension
  }

  return 0;
}
```

例如，如果这个程序称为ISEXEC，并假设TEST.EXE存在，命令行ISEXEC TEST产生下列输出结果：

```
Trying TEST.EXE - Exists
Trying TEST.COM - Not found
Trying TEST.BAT - Not found
```

关于这个程序要注意的一点是，StrType对象被ifstream构造函数所用。这一点能成立是因为转换函数operator char*( )被自动调用了。正如这种情况所说明的，通过谨慎应用C++的这些特性，可以获得C++的标准类型和自定义类型的完美集成。

39.11 创建和集成新类型

正如StrType类所演示的，在C++环境中创建和集成新的数据类型是非常容易的。要这样做，请执行下面的步骤：

1. 重载所有适宜的运算符，包括I/O运算符。
2. 定义所有适宜的转换函数。
3. 提供构造函数，这些构造函数使得在各种情况下很容易创建对象。

C++的强大之处在于它的可扩展性，不要害怕去利用它。

39.12 挑战

这里是一个你会喜欢的有趣的挑战。试着使用STL实现StrType，即，使用一个容器来存储构成字符串的字符。使用迭代器对字符串进行操作，使用算法来执行各种字符串运算。