Part V

Applying C++

Part Five of this book provides two sample C++ applications. The purpose of this section is twofold. First, the examples help illustrate the benefits of object-oriented programming. Second, they show how C++ can be applied to solve two very different types of programming problems.

Chapter 39

Integrating New Classes: A Custom String Class

This chapter designs and implements a small string class. As you know, Standard C++ provides a full-featured, powerful string class called **basic_string**. The purpose of this chapter is not to develop an alternative to this class, but rather to give you insight into how any new data type can be easily added and integrated into the C++ environment. The creation of a string class is the quintessential example of this process. In the past, many programmers honed their object-oriented skills developing their own personal string classes. In this chapter, we will do the same.

While the example string class developed in this chapter is much simpler than the one supplied by Standard C++, it does have one advantage: it gives you full control over how strings are implemented and manipulated. You may find this useful in certain situations. It is also just plain fun to play with!

The StrType Class

Our string class is loosely modeled on the one provided by the standard library. Of course, it is not as large or as sophisticated. The string class defined here will meet the following requirements:

- Strings may be assigned by using the assignment operator.
- Both string objects and quoted strings may be assigned to string objects.
- Concatenation of two string objects is accomplished with the + operator.
- Substring deletion is performed using the – operator.
- String comparisons are performed with the relational operators.
- String objects may be initialized by using either a quoted string or another string object.
- Strings must be able to be of arbitrary and variable lengths. This implies that storage for each string is dynamically allocated.
- A method of converting string objects to null-terminated strings will be provided.

Although our string class will, in general, be less powerful than the standard string class, it does include one feature not defined by **basic_string**: substring deletion via the – operator.

The class that will manage strings is called **StrType**. Its declaration is shown here:

```
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 *
};
```

The private part of **StrType** contains only two items: **p** and **size**. When a string object is created, memory to hold the string is dynamically allocated by using **new**, and a pointer to that memory is put in **p**. The string pointed to by **p** will be a normal, null-terminated character array. Although it is not technically necessary, the size of the string is held in **size**. Because the string pointed to by **p** is a null-terminated string, it would be possible to compute the size of the string each time it is needed. However, as you will see, this value is used so often by the **StrType** member functions that the repeated calls to **strlen( )** cannot be justified.

The next several sections detail how the **StrType** class works.

The Constructors and Destructors

A **StrType** object may be declared in three different ways: without any initialization, with a quoted string as an initializer, or with a **StrType** object as an initializer. The constructors that support these three operations are shown here:

```
// 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);
}
```

When a **StrType** object is created with no initializer, it is assigned a null-string. Although the string could have been left undefined, knowing that all **StrType** objects contain a valid, null-terminated string simplifies several other member functions.

When a **StrType** object is initialized by a quoted string, first the size of the string is determined. This value is stored in **size**. Then, sufficient memory is allocated by **new** and the initializing string is copied into the memory pointed to by **p**.

When a **StrType** object is used to initialize another, the process is similar to using a quoted string. The only difference is that the size of the string is known and does not have to be computed. This version of the **StrType** constructor is also the class' copy constructor. This constructor will be invoked whenever one **StrType** object is used to initialize another. This means that it is called when temporary objects are created and when objects of type **StrType** are passed to functions. (See Chapter 14 for a discussion of copy constructors.)

Given the three preceding constructors, the following declarations are allowed:

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

The **StrType** destructor simply frees the memory pointed to by **p**.

I/O on Strings

Because it is common to input or output strings, the **StrType** class overloads the << and >> operators, as shown here:

```
// 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;
}
```

As you can see, output is very simple. However, notice that the parameter **o** is passed by reference. Since **StrType** objects may be quite large, passing one by reference is more efficient than passing one by value. For this reason, all **StrType** parameters are passed by reference. (Any function you create that takes **StrType** parameters should probably do the same.)

Inputting a **string** proves to be a little more difficult than outputting one. First, the string is read using the **getline( )** function. The length of the largest string that can be input is limited to 254 plus the null terminator. As the comments indicate, you can change this if you like. Characters are read until a newline is encountered. Once the string has been read, if the size of the new string exceeds that of the one currently held by **o**, that memory is released and a larger amount is allocated. The new string is then copied into it.

The Assignment Functions

You can assign a **StrType** object a string in two ways. First, you can assign another **StrType** object to it. Second, you can assign it a quoted string. The two overloaded **operator=( )** functions that accomplish these operations are shown here:

```
// 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;
  }
```

These two functions work by first checking to see if the memory currently pointed to by **p** of the target **StrType** object is sufficiently large to hold what will be copied to it. If not, the old memory is released and new memory is allocated. Then the string is copied into the object and the result is returned. These functions allow the following types of assignments:

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

y = x; // StrType object to StrType object

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

Each assignment function returns the value assigned (that is, the right-hand value) so that multiple assignments like this can be supported:

```
StrType x, y, z;

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

Concatenation

Concatenation of two strings is accomplished by using the + operator. The **StrType** class allows for the following three distinct concatenation situations:

- Concatenation of a **StrType** object with another **StrType** object
- Concatenation of a **StrType** object with a quoted string
- Concatenation of a quoted string with a **StrType** object

When used in these situations, the + operator produces as its outcome a **StrType** object that is the concatenation of its two operands. It does not actually modify either operand.

The overloaded **operator+( )** functions are shown here:

```
// 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;
}
```

All three functions work basically in the same way. First, a temporary **StrType** object called **temp** is created. This object will contain the outcome of the concatenation, and it is the object returned by the functions. Next, the memory pointed to by **temp.p** is freed. The reason for this is that when **temp** is created, only 1 byte of memory is allocated (as a placeholder) because there is no explicit initialization. Next, enough memory is allocated to hold the concatenation of the two strings. Finally, the two strings are copied into the memory pointed to by **temp.p**, and **temp** is returned.

Substring Subtraction

A useful string function not found in **basic_string** is *substring subtraction.* As implemented by the **StrType** class, substring subtraction removes all occurrences of a specified substring from another string. Substring subtraction is accomplished by using the – operator.

The **StrType** class supports two cases of substring subtraction. One allows a **StrType** object to be subtracted from another **StrType** object. The other allows a quoted string to be removed from a **StrType** object. The two **operator–( )** functions are shown here:

```
// 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;
}
```

These functions work by copying the contents of the left-hand operand into **temp**, removing any occurrences of the substring specified by the right-hand operand during the process. The resulting **StrType** object is returned. Understand that neither operand is modified by the process.

The **StrType** class allows substring subtractions like these:

```
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"
```

The Relational Operators

The **StrType** class supports the full range of relational operations to be applied to strings. The overloaded relational operators are defined within the **StrType** class declaration. They are repeated here for your convenience:

```
// 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; }
```

The relational operations are very straightforward; you should have no trouble understanding their implementation. However, keep in mind that the **StrType** class implements comparisons between two **StrType** objects or comparisons that have a **StrType** object as the left operand and a quoted string as the right operand. If you want to be able to put the quoted string on the left and a **StrType** object on the right, you will need to add additional relational functions.

Given the overloaded relational operator functions defined by **StrType**, the following types of string comparisons are allowed:

```
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";
```

Miscellaneous String Functions

The **StrType** class defines three functions that make **StrType** objects integrate more completely with the C++ programming environment. They are **strsize( )**, **makestr( )**, and the conversion function **operator char *( )**. These functions are defined within the **StrType** declaration and are shown here:

```
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 *
```

The first two functions are easy to understand. As you can see, the **strsize( )** function returns the length of the string pointed to by **p**. Since the length of the string might be different than the value stored in the **size** variable (because of an assignment of a shorter string, for example), the length is computed by calling **strlen( )**. The **makestr( )** function copies into a character array the string pointed to by **p**. This function is useful when you want to obtain a null-terminated string given a **StrType** object.

The conversion function **operator char *( )** returns **p**, which is, of course, a pointer to the string contained within the object. This function allows a **StrType** object to be used anywhere that a null-terminated string can be used. For example, this is valid code:

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

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

Recall that a conversion function is automatically executed when an object is involved in an expression for which the conversion is defined. In this case, because the prototype for the **strcpy( )** function tells the compiler that its second argument is of type **char ***, the conversion from **StrType** to **char *** is automatically performed, causing a pointer to the string contained within **x** to be returned. This pointer is then used by **strcpy( )** to copy the string into **s**. Because of the conversion function, you can use an **StrType** object in place of a null-terminated string as an argument to any function that takes an argument of type **char ***.

Note *The conversion to **char** * does circumvent encapsulation, because once a function has a pointer to the object's string, it is possible for that function to modify the string directly, bypassing the **StrType** member functions and without that object's knowledge. For this reason, you must use the conversion to **char** * with care. You can prevent the underlying string from being modified by having the conversion to **char** * return a **const** pointer. With this approach, encapsulation is preserved. You might want to try this change on your own.*

The Entire StrType Class

Here is a listing of the entire **StrType** class along with a short **main( )** function that demonstrates its features:

```
#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;
}
```

The preceding program produces this output:

```
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
```

This output assumes that the string "I like C++" was entered by the user when prompted for input.

To have easy access to the **StrType** class, remove the **main( )** function and put the rest of the preceding listing into a file called STR.H. Then, just include this header file with any program in which you want to use **StrType**.

Using the StrType Class

To conclude this chapter, two short examples are given that illustrate the **StrType** class. As you will see, because of the operators defined for it and because of its conversion function to **char ***, **StrType** is fully integrated into the C++ programming environment. That is, it can be used like any other type defined by Standard C++.

The first example creates a simple thesaurus by using **StrType** objects. It first creates a two-dimensional array of **StrType** objects. Within each pair of strings, the first contains the key word, which may be looked up. The second string contains a list of alternative or related words. The program prompts for a word, and if the word is in the thesaurus, alternatives are displayed. This program is very simple, but notice how clean and clear the string handling is because of the use of the **StrType** class and its operators. (Remember, the header file STR.H contains the **StrType** class.)

```
#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;
}
```

The next example uses a **StrType** object to check if there is an executable version of a program, given its filename. To use the program, specify the filename without an extension on the command line. The program then repeatedly tries to find an executable file by that name by adding an extension, trying to open that file, and reporting the results. (If the file does not exist, it cannot be opened.) After each extension is tried, the extension is subtracted from the filename and a new extension is added. Again, the **StrType** class and its operators make the string manipulations clean and easy to follow.

```
#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;
}
```

For example, if this program is called ISEXEC, and assuming that TEST.EXE exists, the command line **ISEXEC TEST** produces this output:

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

One thing to notice about the program is that an **StrType** object is used by the **ifstream** constructor. This works because the conversion function **char *( )** is automatically invoked. As this situation illustrates, by the careful application of C++ features, you can achieve significant integration between C++'s standard types and types that you create.

Creating and Integrating New Types in General

As the **StrType** class has demonstrated, it is actually quite easy to create and integrate a new data type into the C++ environment. To do so, just follow these steps.

1. Overload all appropriate operators, including the I/O operators.
2. Define all appropriate conversion functions.
3. Provide constructors that allow objects to be easily created in a variety of situations.

Part of the power of C++ is its extensibility. Don't be afraid to take advantage of it.

A Challenge

Here is an interesting challenge that you might enjoy. Try implementing **StrType** using the STL. That is, use a container to store the characters that comprise a string. Use iterators to operate on the strings, and use the algorithms to perform the various string manipulations.