Manipulating Strings 15

Key Concepts

C-strings | The string class | Creating string objects | Manipulating strings | Relational operations on strings | Comparing strings | String characteristics | Swapping strings

15.1 INTRODUCTION

A string is a sequence of characters. We know that C++ does not support a built-in string type. We have used earlier null-terminated character arrays to store and manipulate strings. These strings are called *C-strings* or *C-style strings*. Operations on C-strings often become complex and inefficient. We can also define our own string classes with appropriate member functions to manipulate strings. This was illustrated in Program 7.4 (Mathematical Operation of Strings).

ANSI standard C++ now provides a new class called **string**. This class improves on the conventional C-strings in several ways. In many situations, the string objects may be used like any other built-in type data. Further, although it is not considered as a part of the STL, **string** is treated as another container class by C++ and therefore all the algorithms that are applicable for containers can be used with the **string** objects. For using the **string** class, we must include **<string>** in our program.

The **string** class is very large and includes many constructors, member functions and operators. We may use the constructors, member functions and operators to achieve the following:

- Creating string objects
- Reading string objects from keyboard
- Displaying string objects to the screen
- Finding a substring from a string
- Modifying string objects
- Comparing string objects
- Adding string objects
- Accessing characters in a string
- Obtaining the size of strings
- Many other operations

Table 15.1 gives prototypes of three most commonly used constructors and Table 15.2 gives a list of important member functions. Table 15.3 lists a number of operators that can be used on **string** objects.

Table 15.1 *Commonly used string constructors*

Constructor	Usage
String();	For creating an empty string
String(const chat *str);	For creating a string object from a null-terminated string
String(const string & str);	For creating a string object from other string object

Table 15.2 *Important functions supported by the string class*

Function	Task
append()	Appends a part of string to another string
Assign()	Assigns a partial string
at()	Obtains the character stored at a specified location
Begin()	Returns a reference to the start of a string
capacity()	Gives the total elements that can be stored.
compare()	Compares string against the invoking string
empty()	Returns true if the string is empty; Otherwise returns false
end()	Returns a reference to the end of a string
erase()	Removes characters as specified
find()	Searches for the occurrence of a specified substring
insert()	Inserts characters at a specified location
length()	Gives the number of elements in a string
max size()	Gives the maximum possible size of a string object in a give system
replace()	Replace specified characters with a given string
resize()	Changes the size of the string as specified
size()	Gives the number of characters in the string
swap()	Swaps the given string with the invoking string

Table 15.3 *Operators for string objects*

Operator	Meaning
=	Assignment
+	Concatenation
+=	Concatenation assignment
==	Equality
!=	Inequality
<	Less than
<=	Less than or equal

(Contd.)

Table 15.3 (Contd.)

>	Greater than
>=	Greater than or equal
[]	Subscription
<<	Output
>>	Input

15.2 CREATING (string) OBJECTS

We can create **string** objects in a number of ways as illustrated below:

```
string s1;              // Using constructor with no argument
string s2("xyz");       // Using one-argument constructor
s1 = s2;                // Assigning string objects
s3 = "abc" + s2         // Concatenating strings
cin >> s1;              // Reading through keyboard (one word)
getline(cin, s1);       // Reading through keyboard a line of text
```

The overloaded + operator concatenates two string objects. We can also use the operator += to append a string to the end of a string. Examples:

```
s3 += s1;               // s3 = s3 + s1
s3 += "abc";            // s3 = s3 + "abc"
```

The operators << and >> are overloaded to handle input and output of string objects. Examples:

```
cin >> s2;              // Input to string object (one word)
cout << s2;             // Displays the contents of s2
getline(cin, s2);       // Reads embedded blanks
```

Note *Using cin and >> operator we can read only one word of a string while the getline() function permits us to read a line of text containing embedded blanks.*

Program 15.1 demonstrates the several ways of creating string objects in a program.

Program 15.1 Creating String Objects

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

using namespace std;

int main()
{
        // Creating string objects
        string s1;              // Empty string object
        string s2(" New");      // Using string constant
```

(Contd.)

```
    string s3(" Delhi");

    // Assigning value to string objects
    s1 = s2;                        // Using string object
    cout << "S1 = " << s1 << "\n";

    // Using a string constant
    s1 = "Standard C++";
    cout << "Now S1 = " << s1 << "\n";

    // Using another object
    string s4(s1);
    cout << "S4 = " << s4 << "\n\n";

    // Reading through keyboard
    cout << "ENTER A STRING \n";
    cin >> s4;                      // Delimited by blank space
    cout << "Now S4 = " << s4 << "\n\n";

    // Concatenating strings
    s1 = s2 + s3;
    cout << "S1 finally contains: " << s1 << "\n";

    return 0;
}
```

The output of Program 15.1 would be:

```
S1 = New
Now S1 = Standard C++
S4 = Standard C++
ENTER A STRING
COMPUTER CENTRE
Now S4 = COMPUTER

S1 finally contains: New Delhi
```

15.3 MANIPULATING STRING OBJECTS

We can modify contents of **string** objects in several ways, using the member functions such as **insert()**, **replace()**, **erase()**, and **append()**. Program 15.2 demonstrates the use of some of these functions.

Program 15.2 Modifying String Objects

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

using namespace std;

int main()
{
```

(Contd.)

```
    string s1("12345");
    string s2("abcde");

    cout << "Original Strings are: \n";
    cout << "S1: " << s1 << "\n";
    cout << "S2: " << s2 << "\n\n";

    // Inserting a string into another
    cout << "Place S2 inside S1 \n";
    s1.insert(4,s2);
    cout << "Modified S1: " << s1 << "\n\n";

    // Removing characters in a string
    cout << "Remove 5 Characters from S1 \n";
    s1.erase(4,5);
    cout << "Now S1: " << s1 << "\n\n";

    // Replacing characters in a string
    cout << "Replace Middle 3 Characters in S2 with S1 \n";
    s2.replace(1,3,s1);
    cout << "Now S2: " << s2 << "\n";

    return 0;
}
```

The output of Program 15.2 given below illustrates how strings are manipulated using string functions.

```
Original Strings are:
S1: 12345
S2: abcde

Place S2 inside S1
Modified S1: 1234abcde5

Remove 5 Characters from S1
Now S1: 12345

Replace Middle 3 Characters in S2 with S1
Now S2: a12345e
```

Note *Analyze how arguments of each function used in this program are implemented.*

15.4 RELATIONAL OPERATIONS

A number of operators that can be used on strings are defined for **string** objects (Table 15.3). We have used in the earlier examples the operators = and + for creating objects. We can also apply the relational operators listed in Table 15.3. These operators are overloaded and can be used to compare **string** objects. The **compare()** function can also be used for this purpose.

Program 15.3 shows how these operators are used.

Program 15.3 Relational Operations on String Objects

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

using namespace std;

int main()
{
string s1("ABC");
string s2("XYZ");
string s3 = s1 + s2;

if(s1!=s2)
     cout<<s1<<" is not equal to "<<s2<<"\n";

if(s1>s2)
     cout<<s1<<" is greater than "<<s2<<"\n";
else
     cout<<s2<<" is greater than "<<s1<<"\n";

if(s3==s1+s2)
     cout<<s3<<" is equal to "<<s1+s2<<"\n\n";

int x = s1.compare(s2);
if(x==0)
     cout<<s1<<" = "<<s2<<"\n";

else if(x>0)
     cout<<s1<<" > "<<s2<<"\n";
else
     cout<<s1<<" < "<<s2<<"\n";

return 0;
}
```

The output of Program 15.3 would be:

```
ABC is not equal to XYZ
XYZ is greater than ABC
ABCXYZ is equal to ABCXYZ

ABC < XYZ
```

15.5 STRING CHARACTERISTICS

Class **string** supports many functions that could be used to obtain the characteristics of strings suc as size, length, capacity, etc. The size or length denotes the number of elements currently stored a given string. The capacity indicates the total elements that can be stored in the given string. Anoth characteristic is the *maximum size* which is the largest possible size of a string object that the give system can support. Program 15.4 illustrates how these characteristics are obtained and used in application.

Program 15.4 Obtaining String Characteristics

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

using namespace std;

void display(string &str)
{
        cout << "Size = " << str.size() << "\n";
        cout << "Length = " << str.length() << "\n";
        cout << "Capacity = " << str.capacity() << "\n";
        cout << "Maximum Size = " << str.max_size() << "\n";
        cout << "Empty: " << (str.empty() ? "yes" : "no");
        cout << "\n\n";
}

int main()
{
        string str1;

        cout << "Initial status: \n";
        display(str1);

        cout << "Enter a string (one word) \n";
        cin >> str1;
        cout << "Status now: \n";
        display(str1);

        str1.resize(15);
        cout << "Status after resizing: \n";
        display(str1);
        cout << "\n";

        return 0;
}
```

The output of Program 15.4 would be:

```
Initial status:
Size = 0
Length = 0
Capacity = 0
Maximum Size = 4294967293
Empty: yes
Enter a string (one word)
INDIA
Status now:
Size = 5
Length = 5
Capacity = 31
```

```
Maximum Size = 4294967293
Empty: no

Status after resizing:
Size = 15
Length = 15
Capacity = 31
Maximum Size = 4294967293
Empty: no
```

The size and length of 0 indicate that the string **str1** contains no characters. The size and length are always the same. The **str1** has a capacity of zero initially but its capacity has increased to 31 when a string is assigned to it. The maximum size of a string in this system is 4294967293. The function **empty()** returns **true** if **str1** is empty; otherwise **false.**

15.6 ACCESSING CHARACTERS IN STRINGS

We can access substrings and individual characters of a string in several ways. The **string** class supports the following functions for this purpose:

```
at()            for accessing individual characters
substr()        for retrieving a substring
find()          for finding a specified substring
find_first_of() for finding the location of first occurrence of the
                specified character(s)
find_last_of()  for finding the location of last occurrence of the
                specified character(s)
```

We can also use the overloaded [ ] operator (which makes a **string** object look like an array) to access individual elements in a string. Program 15.5 demonstrates the use of some of these functions.

Program 15.5 Accessing and Manipulating Characters

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

using namespace std;

int main()
{
    string s("ONE TWO THREE FOUR");
    cout << "The string contains: \n";
    for(int i=0;i<s.length();i++)
        cout << s.at(i);              // Display one character
    cout << "\nString is shown again: \n";
    for(int j=0;j<s.length();j++)
        cout << s[j];

    int x1 = s.find("TWO");
    cout << "\n\nTWO is found at: " << x1 << "\n";
```

(Contd.)

```
    int x2 = s.find_first_of('T');
    cout << "\nT is found first at: " << x2 << "\n";
    int x3 = s.find_last_of('R');
    cout << "\nR is last found at: " << x3 << "\n";

    cout << "\nRetrieve and print substring TWO \n";

    cout << s.substr(x1,3);
    cout << "\n";

    return 0;
}
```

The output of Program 15.5 would be:

```
The string contains:
ONE TWO THREE FOUR
String is shown again:
ONE TWO THREE FOUR

TWO is found at: 4

T is found first at: 4

R is last fount at: 17

Retrieve and print substring TWO
TWO
```

We can access individual characters in a string using either the member function **at()** or the subscript operator [ ]. This is illustrated by the following statements:

```
cout << s.at(i);
cout << s[i];
```

The statement

```
int x1 = s.find("TWO");
```

locates the position of the first character of the substring "TWO". The statement

```
cout << s.substr(x1,3);
```

finds the substring "TWO". The first argument **x1** specifies the location of the first character of the required substring and the second argument gives the length of the substring.

15.7 COMPARING AND SWAPPING

The **string** supports functions for comparing and swapping strings. The **compare()** function can be used to compare either two strings or portions of two strings. The **swap()** function can be used for swapping the contents of two **string** objects. The capabilities of these functions are demonstrated in Program 15.6.

Program 15.6 Comparing and Swapping Strings

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
        string s1("Road");
        string s2("Read");
        string s3("Red");
        cout << "s1 = " << s1 << "\n";
        cout << "s2 = " << s2 << "\n";
        cout << "s3 = " << s3 << "\n";

        int x = s1.compare(s2);
        if(x == 0)
            cout << "s1 == s2" << "\n";
        else if(x > 0)
            cout << "s1 > s2" << "\n";
        else
            cout << "s1 < s2" << "\n";

        int a = s1.compare(0,2,s2,0,2);
        int b = s2.compare(0,2,s1,0,2);
        int c = s2.compare(0,2,s3,0,2);
        int d = s2.compare(s2.size()-1,1,s3,s3.size()-1,1);

        cout << "a = " << a << "\n" << "b = " << b << "\n";
        cout << "c = " << c << "\n" << "d = " << d << "\n";

        cout << "\nBefore swap: \n";
        cout << "s1 = " << s1 << "\n";
        cout << "s2 = " << s2 << "\n";
        s1.swap(s2);
        cout << "\nAfter swap: \n";
        cout << "s1 = " << s1 << "\n";
        cout << "s2 = " << s2 << "\n";

        return 0;
}
```

The output of Program 15.6 would be:

```
s1 = Road
s2 = Read
s3 = Red
s1 > s2
a = 1
b = -1
c = 0
d = 0
```

```
Before swap:
s1 = Road
s2 = Read

After swap:
s1 = Read
s2 = Road
```

The statement

```
int x = s1.compare(s2);
```

compares the string **s1** against **s2** and **x** is assigned 0 if the strings are equal, a positive number if **s1** is *lexicographically* greater than **s2** or a negative number otherwise.

The statement

```
int a = s1.compare(0,2,s2,0,2);
```

compares portions of **s1** and **s2**. The first two arguments give the starting subscript and length of the portion of **s1** to compare to **s2**, that is supplied as the third argument. The fourth and fifth arguments specify the starting subscript and length of the portion of **s2** to be compared. The value assigned to **a** is 0, if they are equal, 1 if substring of **s1** is greater than the substring of **s2**, -1 otherwise.

The statement

```
s2.swap(s2);
```

exchanges the contents of the strings **s1** and **s2**.

SUMMARY

- Manipulation and use of C-style strings become complex and inefficient. ANSI C++ provides a new class called **string** to overcome the deficiencies of C-strings.
- The string class supports many constructors, member functions and operators for creating and manipulating string objects. We can perform the following operations on the strings:
  - Reading strings from keyboard
  - Assigning strings to one another
  - Finding substrings
  - Modifying strings
  - Comparing strings and substrings
  - Accessing characters in strings
  - Obtaining size and capacity of strings
  - Swapping strings
  - Sorting strings