

Chapter 6 – Strings

1. The Cavat

While literal string constants (lists of characters inside of double quotes) are supported, string is not a native type in C++. Nevertheless, programmers need to be able to put string information into an object. Authors of introductory programming books often avoid or put off the issues involved in using strings.

All of this avoidance could lead to the conclusion that strings are very difficult, but that would be only partly true. This chapter will not attempt to tackle all of the issues involved in strings. That is better left to later chapters when the student has had experience with fundamental algorithms and data structures. Instead, this chapter will serve as an introduction to strings.

2. C-Strings

As has been demonstrated in previous chapters, `cout` can handle output of *string* constant expressions. `cin` can handle input of strings, but can only copy data into a variable object. Alas, since there is no type `string`, variable objects of type `string` cannot be created. Instead, variable objects called C-strings are made by ‘stringing’ objects of type `char` together. (*Note for students with previous programming language experience: This creates an array of characters.*) This is done in several ways. Here is how a C-string is declared:

```
char object[n];
```

where *n* stands for the number of characters that the object will be able to store

Here is an example declaration of a C-string

```
char firstName[15];
```

In this declaration, the object *firstName* is created. In all declarations, computer memory is allocated from available reserves, called the *free store*. A *char* object receives enough memory to store one *char*, just as a *double* object receives enough memory to store the largest possible defined value for a *double*. In the declaration of *firstName*, enough memory to store 15 *chars* is allocated from the free store to *firstName*. Alas, the programmer will only have use of 14 of the spaces, as the end of the string data will need to be represented by a null character (`'\0'`). (The good news is that the null character is automatically assigned and the programmer does not have to do so. Note that C-strings are often called *null terminated strings*.)

Like other object variables, string object variables can be assigned a value when being declared.

```
char object[n] = “ ... a string constant ...”;
```

For example: `char name[25] = "Robert Hall";`

The object *name* cannot be assigned a value longer than 24 *chars*. Room must be left for the null character.

The number of character spaces can go unspecified if the value of a string object variable is assigned when the variable is declared. For example, the object *name* can be declared as follows:

```
char name = "Robert Hall";
```

Which will create a c-string variable object consisting of 12 *char* spaces, 11 for the data and one for the null character.

3. Restrictions C-Strings

Since C-strings are not objects of a simple type, various operations do not work with them. For example, the assignment statement only works for C-strings as part of the declaration of the C-string object. While

```
char color[50] = "red";
```

or

```
char color = "red"
```

is correct and assigns the variable *color* the value of "red", the following two examples do not work.

```
char color[50];  
color = "red";  
  
char color;  
color = "red";
```

4. Operations on C-Strings

Including the file *string.h* into a program brings in a set of free functions that manipulate *C-strings*. Since most standard operators don't work on *C-strings*, this becomes a standard inclusion in most C++ programs that use *C-strings*. The most immediately useful are the following three functions: (see next page)

strcpy(string1, string2) – the function *strcpy* (pronounced “string copy”) copies the contents of *string2* into *string1*, any value in *string1* is replaced with the value in *string2*, *string1* must be the same size or larger than *string2*

```
example:   char s2[10] = "High";
           char s1[15];
           strcpy(s1, s2);      // "High" is copied into s2, replacing
                               // any value in s1
```

strcat(string1, string2) – the function *strcat* (pronounced “string concatenation”) combines the contents of *string1* and *string2* into new memory allocated from the free store, the new area of memory is then given to *string1*

```
example:   char s1[10] = ", Mom!";
           char s2[20] = "High";
           strcat(s2, s1);     // ", Mom!" is added to the value in
                               // s2, giving s2 the value
                               // "High, Mom!"
```

strlen(string) – the function *strlen* (pronounced “string length”) returns the number of characters in the string

```
example:   char s = "Henry Ford";
           int len = strlen(s); // the number of characters of s (10) is
                               // returned and assigned to the integer
                               // variable len
```

The following three functions are included here for the sake of completeness and for the convenience of having all string functions in one location for lookup later. They will not be of use until many chapters later.

strcmp(string1, string2) – the function *strcmp* (pronounced “string compare”) returns an integer less than 0 if *string1* is less than *string2*, returns 0 if *string1* equals *string2*, returns an integer greater than 1 if *string1* greater than *string2*

```
example:   char s = "dynamical";
           int compare = strcmp(s, "static");
                               // compare receives an integer value
                               // less than 0 because the character 'd'
                               // has a value less than 's' in the ASCII
                               // collating sequence
```

strchr(string, character) – the function *strchr* returns a pointer to the first occurrence of *character* in *string*

strstr(string1, string2) – the function *strstr* returns a pointer to the first occurrence of *string2* in *string1*

Here is an example program:

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

void main() {
    char firstName[15];
    char fullName[10];
    strcpy(firstName, "Robert");
    strcpy(fullName, "Hall");
    strcat(fullName, ", ");
    strcat(fullName, firstName);

    cout << "My name is " << fullName << endl;
}
```

When run, this program produces the following output:

My name is Hall, Robert

5. The C++ Standard Class *string*

Given the origins of C++, *C-strings* are very common. This is so, even though C++ operators cannot work on them directly. Consequentially, operations on them require the use of free functions and, because of that, are somewhat clumsy.

The modern C++ library of code includes a class called *string*, which is found in *string*. (Note: This file is different from *string.h*!) This class exists to allow most standard C++ operators to work with object variables of class *string*. The class *string* is very large and only a small portion of it will be covered here.

There are three commonly used means of declaring an object of class *string*. These are :

```
string object-name; // create a string object with no value

string object-name(null-terminated-string); // create a string object and copy the
// value in a null terminated string
// into it

string object-name(string-object); // create a string object and copy the
// value in another string object into it
```

Here are some examples of their use:

```
string s1; // creates the string object s1 with no value

string s2("Hello, world!"); // creates the string object s2 with the value
// "Hello, world!"

char ns[10] = "Bye, world!"; // creates the null terminated string object ns with the
// value "Bye, world!"
string s3(ns); // creates the string object s3 and copies the value
// "Bye, world!" into it

string s4(s2); // creates the string object s4 and copies the value
// "Hi, world!" into it
```

Variables of the class *string* can have values assigned to them in the normal C++ manner using the assignment operator. For example:

```
s1 = "Hello, Graceland!"; // allocates memory for the variable object s1 and
// copies the value "Hello, Graceland!" into it
```

The following is a program example using *string* class objects.

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

void main( )
{
    char ns[10] = "test 1";
    string s1;
    s1 = "test 2";
    string s2("test 3");
    string s3(ns);
    string s4(s2);

    cout << "ns is " << ns << endl
         << "s1 is " << s1 << endl
         << "s2 is " << s2 << endl
         << "s3 is " << s3 << endl
         << "s4 is " << s4 << endl;
}
```

When run, this program produces the following output"

```
ns is test 1
s1 is test 2
s2 is test 3
s3 is test 1
s4 is test 4
```

Objects of class *string* use the addition operator for concatenation. This is in the form of + and +=. For example, the following code

```
string s("My dog ");
s = s + "Spot";
```

produces the string literal "My dog Spot" and stores it in *string* class variable *s1*. The next code example is identical in outcome.

```
string s("My dog ");
s += "Spot";
```

In addition, objects of class *string* come with several member functions. These are:

length() - returns an integer representing the number of characters stored in the object

```
example:    string s="one potatoe";
            int len = s.length(); // len will receive 11
```

size() - same function and usage as *length*

find(string) - returns an integer representing the position of the first occurrence of the value in *string* in the string; first character is in location 0; for example:

```
example:    string a="12345";
            cout << a.find("34");
```

outputs: 2

substr(int1, int2) - returns the string starting at position *int1* that is *int2*] characters long

```
example:    string a="12345";
            cout << a.substr(a.find("34"), 3);
```

outputs: 345

6. Cin with Strings

cin accepts string input and directs it to string variable objects of either type. *cin* will read string data until a space or the end-of-line, such as a pressed Enter key, is encountered. Here are two examples of *string* input with *cin*:

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

void main( ) {
    char firstName[15];
    char fullName[30];

    cout << "Enter your first and last names: ";
    cin >> firstName >> fullName;
    strcat(fullName, ", ");
    strcat(fullName, firstName);

    cout << "My name is " << fullName << endl;
}
```

If the user enters the string “Robert Hall” during a program run, the program run would look like the following:

```
Enter your first and last names: Robert Hall
My name is Hall, Robert
```

Here is a slightly different example:

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

void main( ) {
    string firstName, fullName;

    cout << "Enter your first name: ";
    cin >> firstName;
    cout << "Enter your last name: ";
    cin >>fullName;

    fullName += " " + firstName;
    cout << "My name is " << fullName << endl;
    cout << "The length of my name string is: " << fullName.length( )
        << endl;
}
```

If the user enters the *strings* “Robert” and “Hall” during a program run, the program run would look like the following:

```
Enter your first name: Robert
Enter your last name: Hall
My name is Hall, Robert
The length of my name string is: 12
```

7. Cin with getline

cin can be used with a member function of class *istream* called *getline*, which reads all characters on a line, including spaces. The member function *getline* can only be used with C-strings. It is written:

```
cin.getline(C-string, length, terminator);
```

where *string* is a C-string variable

length is the maximum length of the string or the maximum number of characters to be read, it is an *int*

terminator is the character that indicates the end of the string (usually ‘\n’ for end-of-line)

Here is a fragment example of the use of *getline*:

```
void main( )
{
    char name[25];
    cout << “Enter your first and last name: “;
    cin.getline(name, 25, ‘\n’);
    cout << “My name is “ << name << endl;
    cout << “The length of my name string is: “ << strlen(name)
        << endl;
}
```

If the user enters the strings “Robert” and “Hall” during a program run, the program run would look like the following:

```
Enter your first and last name: Robert Hall
Enter your last name: Hall
My name is Robert Hall
The length of my name string is: 11
```

8. The Free Function getline

There is a free function also named *getline* that is included with the *string* library and works with objects of class *string*. It is used in place of *cin*. It is written:

```
getline(input-stream-object, string-object);
```

where *input-stream-object* is the object of the input stream that will have a line extracted from it (usually *cin*, at first);
string-object is the string variable that will receive the line from the input stream

fragment example:

```
void main( )
{
    string name;
    cout << "Enter your first and last name: ";
    getline(cin, name);
    cout << "My name is " << name << endl;
    cout << "The length of my name string is: " << name.length( )
        << endl;
}
```

If the user enters the strings “Robert” and “Hall” during a program run, the program run would look like the following:

```
Enter your first and last name: Robert Hall
Enter your last name: Hall
My name is Robert Hall
The length of my name string is: 11
```

9. Character Input Issues

Though not a string input issue, character input has its problems, too. For example, the following code will not accept a space as a character:

```
char c;
cin >> c;
cout << c << endl;
```

If executed, the program will not move past "cin >> c;" until the users inputs a printable character and presses the Enter key. To enable the input of all characters (such as space), the member function of *istream* called *get* is used with *cin*. This is written:

```
cin.get(char-variable); // where char-variable receives one character
```

For example, the following will respond to a space or any other character input, followed by the Enter key - or even just the Enter key!

```
char c;
cin.get(c);
cout << c << endl;
```

Exercises 6.1

1. What is an object of type C-string?
2. What is an object of type string?
3. Is the following declaration valid? If not, why?

```
char astring[10] = "stuff";
```

4. Is the following set of statements valid? If not, why?

```
char astring[10];
astring = "stuff";
```

5. Assuming the a program has `#include <string>`, is the following declaration valid? If not, why?

```
string astring = "stuff";
```

6. Assuming the a program has `#include <string>`, is s the following set of statements valid? If not, why?

```
string astring;
astring = "stuff";
```

7. What is the value in the variable *first* after the following statements are performed?

```
char first[20] = "one";
char second[10] = "potatoe";
streat(first, second);
```

8. What is the value in the variable *first* after the following statements are performed?

```
char first[20] = "one";
char second[10] = "potatoe";
strcpy(first, second);
```

9. What is the value in the variable *first* after the following statements are performed?

```
string first = "one";
string second = "potatoe";
first += second;
```

10. What is the value in the variable *first* after the following statements are performed?

```
string first = "one";  
string second = "potatoe";  
first = second;
```

Programming Assignment 6.1

Write a program asks the user for three words, then outputs them in reverse order.

Programming Assignment 6.2

Write a program that accepts a first name, last name, street address, city, state, and zip code from the program user, then uses these names as part of the following letter written to that individual.

```
<name>  
<street address>  
<city>, <state> <zip>
```

Dear <first name>,

You can be the proud owner of a new Zapperflot 100 for a small monthly payment of \$200 for only 5 years. Just think, <first name>, how your neighbors will turn green with envy over your ability to afford this incredible lawn mower.

Yours truly,
Justa Husker