Supplement V.F: C-Strings

For Introduction to C++ Programming
By Y. Daniel Liang

NOTE
§7.8 gave a brief introduction on C-string from the array's perspectives. This supplement provides a detailed coverage on C-string with use of pointers.

1. Introduction

A C-string is an array of characters ending in the *null terminator* (**'\0'**), which indicates where a string terminates in memory. Recall that a character that begins with the backslash symbol (\) is an escape character. The symbols \ and **0** together represent one character. This character is the first character in the ASCII table.

You can declare a C-string variable using an array. Every string literal is a C-string. Recall that you can assign a string literal to an array. For example, the following statement creates a C-string that contains characters **'D'**, **'a'**, **'l'**, **'l'**, **'a'**, **'s'**, and **'\0'**.

```
char city[7] = "Dallas";
```

Note that the size of the array is **7** and the last character in the array is **'\0'**.

There is a subtle difference between a C-string and an array of characters. A C-string is an array of characters that ends with a *null terminator*. For example, the following two statements are different.

```
char city[] = "Dallas"; // C-string
char city[] = {'D', 'a', 'l', 'l', 'a', 's'}; // Not a C-string
```

The former is a C-string and the latter is just an array of characters. The former has **7** characters including the last null terminator and the latter has **6** characters.

2. Reading Strings

You can read a string from the keyboard using the **cin** object. For example, see the following code:

```
char city[7];
cout << "Enter a city: ";
```

```
cin >> city; // Read to array city
cout << "You entered " << city << endl;
```

CAUTION:
When you read a string to array **city**, make sure to leave room for the null terminator character. Since **city** has a size **7**, your input should not exceed **6** characters.

This approach to read a string is simple, but there is a problem. The input ends with a whitespace character. Suppose you want to enter New York, you have to use an alternative approach. C++ provides the **cin.getline** function in the **iostream** header file, which reads a string into an array. The syntax of the function is:

```
cin.getline(char array[], int size, char delimitChar)
```

The function stops reading characters when the delimiter character is encountered or when the **size - 1** number of characters are read. The last character in the array is reserved for the null terminator (**'\0'**). If the delimiter is encountered, it is read, but not stored in the array. The third argument **delimitChar** has a default value (**'\n'**).

The following code uses the **cin.getline** function to read a string.

```
char city[30];
cout << "Enter a city: ";
cin.getline(city, 30, '\n'); // Read to array city
cout << "You entered " << city << endl;
```

Since the default value for the third argument in the **cin.getline** function is **'\n'**, line 3 can be replaced by

```
cin.getline(city, 30); // Read to array city
```

3. C-string Functions

C++ provides several useful and powerful functions for testing and processing strings, as shown in Table 1. To use these functions, your program needs to include the **cstring** header file.

Table 1
String Functions

Function	Description
size_t strlen(char *s)	Returns the length of the string, i.e., the number of the characters before the null terminator.
char *strcpy(char *s1, const char *s2)	Copies the string s2 to string s1. The value in s1 is returned.
char *strncpy(char *s1, const char *s2, size_t n)	Copies at most n characters from string s2 to string s1. The value in s1 is returned.
char *strcat(char *s1, const char *s2)	Appends string s2 to s1. The first character of s2 overwrites the null terminator in s1. The value in s1 is returned.
char *strncat(char *s1, const char *s2, size_t n)	Appends at most n characters from string s2 to s1. The first character of s2 overwrites the null terminator in s1 and appends a null terminator to the result. The value in s1 is returned.
int strcmp(char *s1, const char *s2)	Returns a value greater than 0, 0, or less than 0 if s1 is greater than, equal to, or less than s2 based on the numeric code of the characters.
int strncmp(char *s1, const char *s2, size_t n)	Returns a value greater than 0, 0, or less than 0 if the n characters in s1 is greater than, equal to, or less than the first n characters in s2 based on the numeric code of the characters.
int atoi(char *s)	Converts the string to an int value.
double atof(char *s)	Converts the string to a double value.
long atol(char *s)	Converts the string to a long value.
void itoa(int value, char *s, int radix)	Converts the value to a string based on specified radix.

NOTE
size_t is a C++ type. For most compilers, it is the same as **unsigned int**.

Functions **strcpy** and **strncpy** can be used to copy a source string in the second argument to a target string in the first argument. The target string must have already been allocated with sufficient memory for the function to work. Function **strncpy** is equivalent to **strcpy** except that **strncpy** specifies the number of characters to be copied from the source. Listing 1 gives a program to demonstrate these two functions.

Listing 1 CopyString.cpp

```
#include <iostream>
```

```
  #include <cstring>
  using namespace std;

  int main()
  {
    char s1[20];
    char s2[] = "Dallas, Texas"; // Let C++ figure out the size
  of s2
    char s3[] = "AAAAAAAAAA"; // Let C++ figure out the size of
  s3

    strcpy(s1, s2);
    strncpy(s3, s2, 6);
    s3[6] = '\0'; // Insert null terminator

    cout << "The string in s1 is " << s1 << endl;
    cout << "The string in s2 is " << s2 << endl;
    cout << "The string in s3 is " << s3 << endl;
    cout << "The length of string s3 is " << strlen(s3) <<
  endl;

    return 0;
  }
```

Sample Output

```
     The string in s1 is Dallas, Texas
     The string in s2 is Dallas, Texas
     The string in s3 is Dallas
     The length of string s3 is 6
```

Three strings **s1**, **s2**, and **s3** are declared in lines 7-9. Line 11 copies **s2** to **s1** using the **strcpy** function. Line 12 copies the first 6 characters from **s2** to **s1** using the **strncpy** function. Note that the null terminator character is not copied in this case. To terminate the string properly, a null terminator is manually inserted at **s3[6]** (the end of the new string). What would happen if you run the program without line 13? String s3 would become **DallasAAAA**.

NOTE:
Both **strcpy** and **strncpy** functions return a string value. But they are invoked as statements in lines 11-12. Recall that a value-returning function may be invoked as a statement if you are not interested in the return value of the function. In this case, the return value is simply ignored.

Functions **strcat** and **strncat** can be used to append the string in the second argument to the string in the first argument. The first string must have already been allocated with sufficient memory for the function to work. Function **strncat** is equivalent to **strcat** except that **strncat** specifies the number of characters to be appended from the second string. Listing 2 gives a program to demonstrate these two functions.

Listing 2 CombineString.cpp

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

int main()
{
  char s1[20] = "Dallas";
  char s2[20] = "Texas, USA";
  char s3[20] = "Dallas";

  strcat(strcat(s1, ", "), s2);
  strncat(strcat(s3, ", "), s2, 5);

  cout << "The string in s1 is " << s1 << endl;
  cout << "The string in s2 is " << s2 << endl;
  cout << "The string in s3 is " << s3 << endl;
  cout << "The length of string s1 is " << strlen(s1) <<
endl;
  cout << "The length of string s3 is " << strlen(s3) <<
endl;

  return 0;
}
```

Sample Output

```
The string in s1 is Dallas, Texas, USA
The string in s2 is Texas, USA
The string in s3 is Dallas, Texas
The length of string s1 is 18
The length of string s3 is 13
```

Three strings **s1**, **s2**, and **s3** are declared in lines 7-9. Line 11 invokes **strcat** twice. First, **strcat(s1, ", ")** appends **", "** to **s1** and returns new **s1**. Second, **strcat(strcat(s1, ", "), s2)** appends **s2** to the new **s1**. So, **s1** is **Dallas, Texas, USA**. Similarly, line 12 appends, **", "** and the first five characters in **s2** to **s3**. So, **s3** is **Dallas, Texas**.

Functions **strcmp** and **strncmp** can be used to compare two strings. How do you compare two strings? You compare their corresponding characters according to their numeric code. Most compilers use the ASCII code for characters.

The function returns the value **0** if **s1** is equal to **s2**, a value less than **0** if **s1** is less than **s2**, and a value greater than **0** if **s1** is greater than **s2**. For example, suppose **s1** is **"abc"** and **s2** is **"abg"**, and **strcmp(s1, s2)** returns **-4**. The first two characters (**a** vs. **a**) from **s1** and **s2** are compared. Because they are equal, the second two characters (**b** vs. **b**) are compared. Because they are also equal, the third two characters (**c** vs. **g**) are compared. Since the character **c** is **4** less than **g**, the comparison returns **-4**.

Listing 3 gives a program to demonstrate these two functions.

Listing 3 CompareString.cpp

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

int main()
{
  char *s1 = "abcdefg";
  char *s2 = "abcdg";
  char *s3 = "abcdg";

  cout << "strcmp(s1, s2) is " << strcmp(s1, s2) << endl;
  cout << "strcmp(s2, s1) is " << strcmp(s2, s1) << endl;
  cout << "strcmp(s2, s3) is " << strcmp(s2, s3) << endl;
  cout << "strncmp(s1, s2, 3) is " << strncmp(s1, s2, 3) <<
endl;

  return 0;
}
```

Sample Output

```
strcmp(s1, s2) is -2
strcmp(s2, s1) is 2
strcmp(s2, s3) is 0
strncmp(s1, s2, 3) is 0
```

Three strings **s1**, **s2**, and **s3** are declared in lines 7-9. Line 11 invokes **strcmp(s1, s2)**, which returns **-2**, because **'e' - 'g'** is **-2**. Line 12 invokes **strcmp(s2, s1)**, which returns **2**, because **'g' - 'e'** is **2**. Line 13 invokes **strcmp(s2, s3)**, which returns **0**, because the two strings are identical. Line 14 invokes **strncmp(s1, s2, 3)**, which returns **0**, because the first three character substrings in both **s1** and **s2** are identical.

NOTE:

With some compilers, functions **strcmp** and **strncmp** always return **1**, **0**, or **-1**.

Functions **atoi**, **atof**, and **atol** convert a string to a numeric value. Function **itoa** converts an integer to a string.

Listing 4 gives a program to demonstrate these functions.

Listing 4 StringConversion.cpp

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

int main()
{
  cout << atoi("4") + atoi("5") << endl;
  cout << atof("4.5") + atof("5.5") << endl;

  char s[10];
  itoa(42, s, 8);
  cout << s << endl;

  itoa(42, s, 10);
  cout << s << endl;

  itoa(42, s, 16);
  cout << s << endl;

  return 0;
}
```

Sample Output

```
    9
    10
    52
    42
    2a
```

Invoking **atoi("4")** returns an **int** value **4** in line 8. Invoking **atof("4.5")** returns a **double** value **4.5** in line 8. Invoking **itoa (42, s, 8)** converts an **int** value **42** to a string based radix **8** in line 11. Invoking **itoa (42, s, 10)** converts an **int** value **42** to a string based radix **10** in line 14. Invoking **itoa (42, s, 16)** converts an **int** value **42** to a string based radix **8** in line 18.

4 Example: Obtaining Substrings

Often it is useful to obtain a substring from a string. But there is no such function in the **<cstring>** header. You can develop your own function for extracting substrings. The header of the function can be specified as:

```
char * substring(char *s, int start, int end)
```

This function returns a string which is a substring in **s** from index **start** to index **end - 1**. For example, invoking

```
substring("Welcome to C++", 2, 6)
```

returns **"lcom"**.

Listing 5 defines and implements the function in a header file.

Listing 5 Substring.h

```
char * substring(const char * const s, int start, int end)
{
  char * pNewString = new char[end - start + 1];

  int j = 0;
  for (int i = start; i < end; i++, j++)
  {
    pNewString[j] = s[i];
  }

  pNewString[j] = '\0'; // Set a null terminator

  return pNewString;
}
```

A new string with the right size **end – start + 1** is created in line 3. **end – start** is the number of characters extracted from the original string to hold the substring. The extra **+1** is for the null terminator. The substring from index **start** to **end** is copied to **pNewString** (lines 6-9). The null terminator is set in line 11.

Listing 6 gives a test program for this function.

Listing 6 TestSubstring.h

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

int main()
{
  char *s = "Atlanta, Georgia";
  cout << substring(s, 0, 7);
```

```
    return 0;
  }
```

Sample Output

```
   Atlanta
```

> **CAUTION**
> The substring returned from this function is created using the **new** operator. The user of the **substring** function should delete the string after it finishes using it to release the memory occupied by this substring. Otherwise, the string exists as long as the program is alive. It is better to redesign the function to pass the result substring as a parameter as follows:

```
  void substring(const char* const s, int start,
    int end, char* substr)
```

5 Case Studies: Checking Palindromes

A string is a palindrome if it reads the same forward and backward. The words “mom,” “dad,” and “noon,” for example, are all palindromes.

How do you write a program to check whether a string is a palindrome? One solution is to check whether the first character in the string is the same as the last character. If so, check whether the second character is the same as the second-last character. This process continues until a mismatch is found or all the characters in the string are checked, except for the middle character if the string has an odd number of characters.

To implement this idea, use two variables, say **low** and **high**, to denote the position of two characters at the beginning and the end in a string **s**, as shown in Listing 7 (lines 26, 29). Initially, **low** is **0** and **high** is **strlen(s) - 1**. If the two characters at these positions match, increment **low** by **1** and decrement **high** by **1** (lines 36-37). This process continues until (**low >= high**) or a mismatch is found.

Listing 7 CheckPalindrome.cpp

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

  // Check if a string is a palindrome
```

```
bool isPalindrome(const char *);

int main()
{
  // Prompt the user to enter a string
  cout << "Enter a string: ";
  char s[80];
  cin.getline(s, 80);

  if (isPalindrome(s))
    cout << s << " is a palindrome" << endl;
  else
    cout << s << " is not a palindrome" << endl;

  return 0;
}

bool isPalindrome(const char * const s)
{
  // The index of the first character in the string
  int low = 0;

  // The index of the last character in the string
  int high = strlen(s) - 1;

  while (low < high)
  {
    if (s[low] != s[high])
      return false; // Not a palindrome

    low++;
    high--;
  }

  return true; // The string is a palindrome
}
```

Sample Output

```
     Enter a string: abccba
     abccba is a palindrome
     Enter a string: abca
     abca is not a palindrome
```

Line 12 declares an array with a size **80**. Line 13 reads a string to the array. Line 15 invokes the **isPalindrome** function to check whether the string is a palindrome.

The **isPalindrome** function is declared in lines 23-41 to return a Boolean value. The address of the string is passed to the pointer **s**. The length of the string is determined by invoking **strlen(s)** in line 29.

6 Improving Deck of Cards Solution

§6.2.7 presented a program that picks four cards randomly from a deck of **52** cards. The program uses two functions **displayRank** and **displaySuit** to display a rank and a suit. These two functions can be replaced by defining the ranks and suits in two arrays of strings.

```
const char *suits[] = {"Clubs", "Diamonds", "Hearts", "Spades"};
const char *ranks[] = {"Ace", "2", "3", "4", "5", "6", "7", "8",
  "9", "10", "Jack", "Queen", "King"};
```

Listing 8 improves the program in §6.2.7.

Listing 8 DeckOfCards.cpp

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

const int NUMBER_OF_CARDS = 52;

int main()
{
  int deck[NUMBER_OF_CARDS];
  const char *suits[] = {"Clubs", "Diamonds", "Hearts", "Spades"}
  const char *ranks[] = {"Ace", "2", "3", "4", "5", "6", "7", "8"
    "9", "10", "Jack", "Queen", "King"};

  // Initialize cards
  for (int i = 0; i < NUMBER_OF_CARDS; i++)
    deck[i] = i;

  // Shuffle the cards
  srand(time(0));
  for (int i = 0; i < NUMBER_OF_CARDS; i++)
  {
    // Generate an index randomly
    int index = rand() % NUMBER_OF_CARDS;
    int temp = deck[i];
    deck[i] = deck[index];
    deck[index] = temp;
  }

  // Display the first four cards
  for (int i = 0; i < 4; i++)
  {
    cout << ranks[deck[i] % 13] << " of "
```

```
             << suits[deck[i] / 13] << "  ";
         }

         return 0;
       }
```

Sample Output

```
     4 of Clubs
     Ace of Diamonds
     6 of Hearts
     Jack of Clubs
```

The program defines an array **suits** for four suits (line 10) and an array ranks for **13** cards in a suits (lines 11-12). Each element in these arrays is a string. Consider the declaration for array **suits**:

```
  const char* suits[] = {"Clubs", "Diamonds", "Hearts", "Spades"};
```

The **suits[]** part in the declaration declares an array. The **const char *** part declares that each element in the array is a pointer to char constants. The four values to be placed in the array are **"Clubs"**, **"Diamonds"**, **"Hearts"**, and **"Spades"**. Each is stored in memory as a null-terminated character string.

The **deck** is initialized with values **0** to **51** in lines 15-16. A deck value **0** represents card Ace of Clubs, **1** represents card 2 of Clubs, **13** represents card Ace of Diamonds, **14** represents card 2 of Diamonds.

Lines 20-27 randomly shuffle the deck. After a deck is shuffled, **deck[i]** contains an arbitrary value. **deck[i] / 13** is **0**, **1**, **2**, or **3**, which determines a suit (line 32). **deck[i] % 13** is a value between **0** and **12**, which determines a rank (line 33).