CMPT 115: Principles of Computer Science

Lab 3: Strings and Arrays in Procedural C++

Department of Computer Science
University of Saskatchewan

January 25 – 29, 2016

Laboratory 3 Overview

Part 0	:	Pre-Lab Reading
Part 1	:	Strings in Procedural C++
Part 2	:	String Manipulation Functions
Part 3	:	Strings Searching Functions
Hand in	:	Exercises (to hand in by the end of the week)

1. Exercise 1: Pg 34
2. Exercise 2: Pg 38

Part I

Pre-Lab 3 Reading

What is Procedural C++?

- C++ is a big language; bigger than we've seen in CMPT 111/116 or CMPT 115 so far
- We've studied only the core of C++: variables, conditionals, loops, functions, arrays, etc
- The core of C++ is virtually identical to a number of other related languages: C, Java, C++, C#; many other languages take inspiration from C, but are not "virtually identical"
- This core needs a name. We could have called it "C with nice things" or "Primitive Java." A good name is "Procedural C++."
- The term "procedural" implies that we are not yet using the object-oriented programming features that C++ has. We'll see them later in the term, though.

Strings

- A *string* is a term for a sequence of characters (including digits, and punctuation)
- In CMPT 111/116, we used strings very simply (abstractly, without being concerned about details). E.g.,

  ```
  string astring  = "Hello!";
  ```

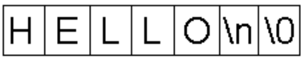
- In CMPT 115 we're going to peel away the abstraction and look more closely.
-

Important Note

From now on, we're going to stop using the `string` type. This will help us understand concepts of memory, references and other related concepts.

C-strings in C/C++

- A "C-string" is a sequence of zero or more characters, terminated by `'\0'`, a character called the *null* character:



H	E	L	L	O	\n	\0

- A C-string is always stored in an array of characters.

```
char bstring[7]  = "Hello\n";
```

- A C-string literal is the same as a string literal: `"Hello"`
- The null character is *implicit* in every string literal, e.g., `"Hello"` has a null character after the `o`.

Important Note

The null character is **not implicit** everywhere else. We have to put it in deliberately everywhere except C-string literals and initializations.

Why do we use a null character?

- The null character is the "end of string" marker, often called a *tombstone* character.
- Sometimes, the array fits the string exactly (as above). But often, a string is stored in an array that is larger than the sequence requires.

Important Note

The null character indicates the end of the C-string.

- Advantage: we don't need to store the length of the C-string! The C-string data is everything before the `'\0'`
- Advantage: when we send a C-string to a function, we do not need to send the size of the array too!
- Disadvantage: we have to be very careful to put the null character in the right place!
  If the `'\0'` is missing, the C-string has no end!
- Good or bad, that is the reality of C-strings. We have to learn it!

Object-Oriented Strings

Object-Oriented C++ contains the type `string` and a `class String` that provides a more robust, complete, and useful version of strings of characters.

Important Note

Students in CMPT 115 cannot use the type `string` or the class `String` for assignments or exams.

The `string` type and the `String` class hide most of the internal understanding of memory and pointers that this course teaches you. Hence, using `String` eliminates your opportunity to learn that fundamental and transferable knowledge.

Declaring a C-string

- A C-string variable can be declared in two ways:

```
char mystring[10];
```

or

```
char *mystring;
```

  where `mystring` is the name of the C-string.

- The first declaration allocates local memory needed to store a maximum of 9 characters and a `'\0'`.
  - Of course, we could put any number of characters less than 9 characters followed by a `'\0'` into our C-string instead.
- The second declaration provides only a *character pointer*, space to hold a memory address. Defining a character pointer does not actually put any of the computer's memory aside for the string.
  - Before we can use the variable `s` defined in this way, we have to make sure that the pointer is pointing to some allocated memory.

C-Strings initialized with string literals

- A C-string is usually declared as an array using one of the following methods:

  ```
  char astring[] = "Welcome to C/C++!";
  char bstring[18] = "Welcome to C/C++!";
  char cstring[30] = "Welcome to C/C++!";
  ```

- Notice that the string `"Welcome to C/C++!"` is 17 visible characters long and that we have declared room for 18. The 18th character is the `'\0'` character, which is never displayed.
- In the first example, we did not put an array size inside the square brackets. The compiler creates an array of exactly the right length automatically (including the null character!).
- *Question:* What happens if `bstring` were declared to have only 10 elements, instead of 18, as above? Try it and see!

C-Strings initialized with string literals cont'd

- Since a C-string is stored in an array, we can look at each character.
- *Indexing* is used to look at a particular character in the C-string, i.e., `s[0]` would give the first character in the string.
- Using the previous example, `char astring[] = "Welcome to C/C++!";` we have
  - `astring[0]` is `'W'`
  - `astring[1]` is `'e'`
  - ...
  - `astring[16]` is `'!'`
  - `astring[17]` is `'\0'`
- If `s[0]=='\0'`, then the C-string has a length of 0 (but the array has a capacity to hold 17 characters)

C-Strings initialized with string literals cont'd

- Since a C-string is stored in an array, we can look at each character.
- *Indexing* is used to look at a particular character in the C-string, i.e., `s[0]` would give the first character in the string.
- Using the previous example,
  `char astring[] = "Welcome to C/C++!";` we have
  - `astring[0]` is `'W'`
  - `astring[1]` is `'e'`
  - ...
  - `astring[16]` is `'!'`
  - `astring[17]` is `'\0'`
- If `s[0]=='\0'`, then the C-string has a length of 0 (but the array has a capacity to hold 17 characters)

Notions for equality

- Consider two identical copies of a book.
  They are equal because their contents are identical.
- Consider two ways to refer to the same thing, e.g., “Superman” and “Man of Steel”.
  They are equal because they refer to the same thing.
- Consider two expressions, $3 + 4$ and $14/2$.
  They are equal because they have the same value.

Equality and Arrays

Important Note

There are two ways to think about equality for C-strings.

1. Two different pointer variables, containing a reference to the same array.
2. Two different arrays, but with the same sequence of characters.

1. Pointer equality, is called *shallow* equality and is fast. It checks whether the two arrays are stored at the same address. We use == for this.
2. Contents equality, is called *deep* equality, and is slower. This checks if the two C-strings are identical by content, even if they are not stored in the same place. We will introduce a function called `strcmp()` for this.

Equality Arrays cont'd

Example:

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

int main() {
  char c1[] = "hi";
  char c2[] = "hi";  // exactly the same contents, different location
  char *c3;

  if (c2==c1) {
    cout << "c2 == c1" << endl
         << "i.e, the two variables point to the same array!" << endl;
  } else {
    cout << "c2 != c1" << endl
         << "i.e, the two variables point to different arrays!" << endl;
  }

  c3 = c1;

  if (c3==c1) {
    cout << "c3 == c1" << endl
         << "i.e, the two variables point to the same array!" << endl;
  } else {
    cout << "c3 != c1" << endl
         << "i.e, the two variables point to different arrays!" << endl;
  }

  return EXIT_SUCCESS;
}
```

Functions for C-Strings

You can find lots of useful functions for C-strings when you include

`<cstring>`

including ones to

- compare C-strings based on their character sequences
- compute the length
- append one C-string onto another
- copy one C-string to the space pointed to by another
- make a different C-string with the same character sequence

String Manipulation Functions: The first set

```
int strcmp(const char s1[],
           const char s2[])
```

compare C-string s1 to C-string s2; returns < 0 if s1 is alphabetically before s2, returns > 0 if s1 is alphabetically after s2 and returns $==$ 0 if s1 and s2 point to (possibly different) C-strings that have the same value.

```
int strlen(const char s[])
```

returns the number of characters in the C-string s (which is different from the capacity of the C-string array!) not including the terminating null.

Note: The word `const` means that the function cannot change the data stored in the array!

strlen

C-String Length vs C-string Capacity

Observe the important difference between

- the **length** of a C-string – (`strlen`) the number of characters in front of the null character, and
- the **capacity** of a C-string array – the number of characters in the whole array.

String Manipulation Functions: The second set

These three functions return a reference to a character array, where the result of the function is stored.

`char* strcat(char dest[],`
`             const char src[])`
	copy string `src` to end of string `dest`; return `dest`. Legnth of array `dest` must be at least strlen(dest) + strlen(src) + 1

`char* strcpy(char dest[],`
`             const char src[])`
	copy string `src` to string `dest`, including `'\0'`; return `src`. Length of array `dest` must be at least strlen(src) + 1.

`char *strdup(const char s[])`	copies the null-terminated string `s` into a newly allocated string. Returns a pointer to the duplicated string, or `NULL` if insufficient memory was available.

String Searching Functions

Also included in the header file `<cstring>` are functions used for searching:

- `char *strchr(const char *s1, int c);`
  - return a pointer to the first occurrence of the character `c` in the C-string `s`, or `NULL` if the character is not found.

- `char *strstr(const char *s1, const char *s2);`
  - return pointer to first occurrence of C-string `s2` in `s1`, or `NULL` if not present.

Note: In both cases, the return value is a *pointer to somewhere in the C-string s1*.

strstr Example

```
#include <iostream>
#include <cstring>

using namespace std;

int main() {
  char string1[] = "needle in a haystack";
  char string2[] = "needle";
  char string3[] = "haystack";
  char *result;

  result = strstr(string1, string2);

  // ’result’ is now a pointer to the start
  // of ’string1’.  Thus, result = a pointer to
  // "needle in a haystack"
  cout << result << endl;

  result = strstr(string1, string3);

  // Now ’result’ points into ’result’, but not at the beginning.
  // Thus, result = a pointer to "haystack"
  cout << result << endl;

  // Note: if we were to modify ’result’ (say by using strcat()),
  // we would also be modifying ’string1’ since ’result’ points
  // into memory that is also part of ’string1’

  return EXIT_SUCCESS;
}
```

Part II

Strings in Procedural C++

C-Strings initialized with string literals

```
char astring[] = "Welcome to C/C++!";
char bstring[18] = "Welcome to C/C++!";
char cstring[30] = "Welcome to C/C++!";
```

Activity: What happens if `bstring` were declared to have only 10 elements, instead of 18, as above? Try it and see!

Statically Allocated Arrays

ACTIVITY: Predict the behaviour of the following program: What does it display?

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

int main() {
    char word[10];
    word[0] = ’H’;
    word[1] = ’e’;
    word[2] = ’l’;
    word[3] = ’l’;
    word[4] = ’o’;
    word[5] = ’\0’;
    cout << "The contents of word[] is " << word << endl;

    return EXIT_SUCCESS;
}
```

Statically Allocated Arrays cont'd

ACTIVITY: In the previous program, delete the line assigning `word[5] = '\0';`. Compile and run on the command line! What happens?

Equality Arrays cont'd

ACTIVITY: Compile and run the following program

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

int main() {
  char c1[] = "hi";
  char c2[] = "hi";  // exactly the same contents, different location
  char *c3;

  if (c2==c1) {
    cout << "c2 == c1" << endl
         << "i.e, the two variables point to the same array!" << endl;
  } else {
    cout << "c2 != c1" << endl
         << "i.e, the two variables point to different arrays!" << endl;
  }

  c3 = c1;

  if (c3==c1) {
    cout << "c3 == c1" << endl
         << "i.e, the two variables point to the same array!" << endl;
  } else {
    cout << "c3 != c1" << endl
         << "i.e, the two variables point to different arrays!" << endl;
  }

  return EXIT_SUCCESS;
}
```

Part III

String Manipulation Functions

strcmp Example

Activity: Compile and run the following program.

```
#include <iostream>
#include <cstring> /* strcmp */

using namespace std;

int main() {
  char s[] = "CMPT111";
  char t[] = "CMPT115";
  char u[] = "CMPT115";
  int compareResult = strcmp(s, t);

  // Which message is printed?
  if (compareResult == 0) {
    cout << "The two strings have same contents.\n";
  } else {
    cout << "The strings are have different contents.\n";
  }

  // Which message is printed?
  if (t == u) {
    cout << "The two strings are equal.\n";
  } else {
    cout << "The strings are NOT equal.\n";
  }

  return EXIT_SUCCESS;
}
```

strlen Example

Activity: Compile and run the following program.

```
#include <iostream>
#include <cstring> /* strcmp */

using namespace std;

int main() {
  char s1[20] =  "Hello ";
  char s2[10] =  "world.\n";

  int length_s1, length_s2;

  strcat(s1, s2);
  length_s1 = strlen(s1);
  length_s2 = strlen(s2);

  // What gets output?
  cout << length_s1 << endl;
  cout << length_s2 << endl;
  cout << s1 << s2 << endl;

  return EXIT_SUCCESS;
}
```

strcat Example

ACTIVITY: Compile and run the following program.

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

int main() {
   char a[20] = "Hello";

   strcat(a, ", world.");
   cout << a << endl;

   return EXIT_SUCCESS;
}
```

What happens if the 20-byte array is not big enough?

strcpy Example

Activity: Compile and run the following program.

```
#include <iostream>
#include <cstring>

using namespace std;

int main() {
   char a[20];
   char b[] = " 42.\n";

   strcpy(a, "The answer is: ");
   cout << a << b;

   // What happens here?
   strcpy(b, a);

   cout << a << b;

   return EXIT_SUCCESS;
}
```

What is the length of a? What is the capacity of a?
How does cout know to print only the first `strlen(a)` chars of a?

strdup Example

Activity: Compile and run the following program.

```
using namespace std;

#include <iostream>
#include <cstring>

using namespace std;

int main() {
  char oldstr[] = "This is a copy.";
  char *newstr;

  // Make newstr point to a duplicate of str */
  newstr = strdup(oldstr);
  if (newstr == NULL) {
    return EXIT_FAILURE;
  }

  cout << "The new string is: " << newstr << endl;
  newstr[8] = '!';
  cout << "The new string is: " << newstr << endl;
  cout << "The old string is: " << oldstr << endl;

  /* Observe how this is different from: */
  newstr = oldstr;

  return EXIT_SUCCESS;
}
```

Exercise 1

ACTIVITY: Start with the following program, then complete it as indicated in the comments:

```
#include <iostream>
#include <cstring>

using namespace std;
int main(void) {
    char dir[] = "usr";
    char subdir[] = "bin";
    char file[] = "firefox";
    char path[100];

    // Use strcpy() and strcat() to combine the strings stored in
    // 'dir', 'subdir', and 'file' to create somethign that looks
    // like a "full path",  e.g., /usr/bin/firefox
    // this string must be stored in the variable 'path'

    // put the code here!

    cout << "The full path is: " << path << endl;

    return EXIT_SUCCESS;
}
```

Part IV

Strings Searching Functions

String Searching Functions

Also included in the header file <cstring> are functions used for searching:

- `char *strchr(const char *s1, int c);`
  - return a pointer to the first occurrence of the character `c` in the string `s`, or `NULL` if the character is not found.
- `char *strstr(const char *s1, const char *s2);`
  - return pointer to first occurence of string `s2` in `s1`, or `NULL` if not present.

Note: In both cases, the return value is a *pointer to somewhere in the string s1*.

Bonus Example: Working with Strings

Activity: Compile and run the following program.

```
#include <iostream>
#include <cstring>

using namespace std;

void reverse(char *s) {
  int i, j;

  for (i=0, j=strlen(s)-1;
       i < j;
       i++, j--) {
    // swap s[i] and s[j] */
    char temp = s[i];
    s[i] = s[j];
    s[j] = temp;
  }
}

int main() {
  char text[20] = "This is CMPT115";

  cout << "String is: " << text << endl;
  reverse(text);
  cout << "String is: " << text << endl;

  return EXIT_SUCCESS;
}
```

Exercise 2

ACTIVITY: Write a program that prompts the user to input a string. Determine the middle of the string (rounded down so use integer division), and generate a new string which swaps the two halves of the string. Output the result.

Example input:

```
The answer to the great question of life, the
universe, and everything is...42.
```

Correct output:

```
e, the universe, and everything is...42.The answer to
the great question of lif
```

What to hand in

1. Your program for Exercise 1 (building a string from smaller strings) in a file called `lab3e1.cc`.
2. Your program for Exercise 2 (swapping first and last halves of a string) in a file called `lab3e2.cc`.