CS 103 Lab - Party Like A Char Star

1 Introduction

In this lab you will implement a word guessing game where the user is shown blanks representing letter of a word and then tries to guess and fill in the letters with a limited number of guesses.

2 What you will learn

After completing this lab you should be able:

- Manipulate C-string variables (variables of type `char*`)
- Pass C-strings to functions
- Utilize the C-string library (`#include <cstring>`) to perform various string operations
- Use `cin` to read a C-string

3 Background Information and Notes

Remember that C is the older language upon which C++ is based. (Most C programs work in C++ too.) We want to manipulate chunks of text, or **strings**, in our programs. The kinds of strings we'll be dealing with for the next few weeks are the ones from the C language, so-called **C strings**. (Towards the end of the course we'll discuss C++ "string objects".)

As we've seen, a C string can be thought of as an array of characters (a **char[]** variable, see section 7.3.2 of the book, and the lecture notes). But to better understand C strings, recall:

1) C adopts the convention that the last significant character in a string should be followed by the *null character* `'\0'`.
2) When you call a function that takes an array as an input, all it actually sends is the memory address (a *pointer* to the start of the array).

Item 1) makes a lot of sense in light of 2): even though a function only knows the *starting* address of a C string passed to it, it can also figure out where it *ends*, by following each character in sequence one at a time until reaching a null character.

A **char*** variable, or a "char pointer" variable, stores the address of a character. Item 2) means that from the perspective of a function, a **char*** argument is the same as a **char[]** argument. (Due to pointer arithmetic, you can even write **ptr[i]** when **ptr** is a **char*** rather than a **char[]**.) So the standard C/C++ string libraries are written to accept **char*** arguments. (The library generally uses **char*** everywhere rather than **char[]**, one reason being that a function can return **char*** but not **char[]**).

Here are a few of the most useful string functions in the standard <cstring> library:

```
size_t strlen ( const char * str );
char * strcpy ( char * destination, const char * source);
char * strncpy ( char * destination, const char * source, size_t num );
int strcmp ( const char * str1, const char * str2 );
```

strlen():

returns the length of a given string. The return type “size_t” is some kind of unsigned integer (on your VM it is an unsigned 64-bit integer) but you can assign the value returned into an 'int' variable without worrying too much (unless you have strings with over 4 billion characters ☺). The “const” means it is guaranteed not to write memory to the pointer.

strcpy():

This function takes in two pointers: the first to a destination character array, the second to a source character array. It copies the source string into the destination. The problem with strcpy() is that it only stops copying when it hits the NULL character in the source string. If the destination array does not have enough memory allocated, strcpy() will gladly overwrite out-of-bounds memory, which can lead to crashes, security vulnerabilities, or hacks in software that would let some rogue user take over the system. Search YouTube for “Super Mario World Credits Warp” for a closely related vulnerability.

strncpy():

This is a safer alternative to strcpy(). It provides a third argument which is a MAXIMUM number of characters that will be copied. It should usually be passed in as the LENGTH of the destination array to ensure we don't write off the end of the destination array.

What is the return value? Google 'strncpy' and check on cplusplus.com or cppreference.com to find out.

strcmp():

strcmp() takes two pointers to character strings and compares them lexicographically. It returns the integer value 0 if they are equal, a negative number if str1 is "less-than" str2 and a positive number if str1 is "greater than" str2. So comparing "abc" and "abc" would return 0. Comparing "ab" and "abc" would return -1 as would comparing "ab" with "ac". Comparing "ac" with "ab" would return +1 as would comparing "abc" with "ab".

4 Procedure

4.1 [1 pt.] Finding C++ Documentation

Show your TA/CP the documentation for strcmp()

4.2 [2 pts.] Copying strings – Common Errors

Type in the following program to a file (`stringdemo.cpp`). Recall that if we have the following declarations of character arrays and pointers, we cannot simply copy one string to the other via an assignment statement. Similarly pointing one char* at another string means we will just reference the original and not make a copy.

```
 1: #include <iostream>
 2: #include <cstring>
 3:
 4: using namespace std;
 5:
 6: int main()
 7: {
 8:    char mystring[80] = "ComputerScience";
 9:    char yourstring[80];
10:    char* astring;
11:
12:    yourstring = mystring;
13:    strncpy(astring, mystring, 80);
14:
15:    astring = mystring;                  // make a copy?
16:    strncpy(yourstring, mystring, 80);   // make a copy?
17:    cout << astring << endl;
18:    cout << yourstring << endl;
19:
20:    mystring[0] = '*'; // which one actually made a copy?
21:    cout << astring << endl;
22:    cout << yourstring << endl;
23:
24:    return 0;
25: }
```

Try to compile your code:

```
$ compile stringdemo.cpp
```

You should get the following error

```
stringdemo.cpp:12:16: error: array type 'char [80]' is not assignable
       yourstring = mystring;
       ~~~~~~~~~~ ^
```

Recall that array names when used in C yield the starting address. So you are really saying take the starting address of mystring and make it the starting address of yourstring. But yourstring has to stay where it is. Thus you should think of array names as constant pointers that cannot be changed/assigned to.

The moral of the story is you must use the strncpy() function to copy over the characters one at a time. An example of this is line 16 of the program.

So now comment that line (i.e. change it to)

```
//yourstring = mystring;
```

Look at the next line of code and think what this will do.

```
strncpy(astring, mystring, 80);
```

It will attempt to copy the characters one at a time from mystring to the memory that astring points to. But what does astring point to?

Compile it *directly* using **g++** first:

```
$ g++ -g stringdemo.cpp -o stringdemo
$ ./stringdemo
```

This should yield:

```
Segmentation fault (core dumped)
```

Not good, it crashed! The reason is that we haven't initialized `astring`. So whatever garbage bits are in the astring pointer will be used as an address and `strncpy` will attempt to copy the characters there… causing it to crash. Now compile again using the **`compile`** command:

```
$ compile stringdemo.cpp
```

This time it actually warns you (`compile` is just `clang++` with certain warning options and debug information enabled).

The moral of the story is that you can only copy data to pointers that actually POINT at memory you have allocated!

Whenever you get a segmentation fault (which will be a lot in your CS career), you can use GDB to find the line of code causing it. Open the executable in gdb:

```
$ gdb stringdemo
```

At the gdb prompt just type 'run' to run the program in the debugger. When it crashes in gdb, type

```
backtrace
```

This will show you the function calls (and line numbers). You should see that the last function that you recognize (i.e. you wrote) is **main()** at **stringdemo.cpp:13**. This means main() called some other function at line 13 and that other function triggered the error. So this can always be used to tell you where your segmentation fault is.

(If you didn't get a line number, maybe you forgot the **`-g`** when calling `clang++`. That option turns on the debugging information and line number tracking. Gdb will also say "warning: no loadable sections found in added symbol-file". Compile again with `clang++ -g` or `compile`.)

So now comment out line 13 as well:

```
// strncpy(astring, mystring, 80);
```

Look at line 15. We initialize astring to point at mystring. But are we making a copy? No, we're just referencing it (i.e. pointing at the original mystring array). On line 17 we print out astring which is really just printing out the data from mystring (since astring just points to mystring). But in line 20 we change mystring. So when we print out astring again on line 21 we will see the update.

The moral of the story is pointing one pointer to someone else's data DOES NOT make a copy of the data. It only references that data. A change in the original is seen by the referencing pointer.

On line 16 we actually make a copy of all the data from mystring to yourstring. Since we actually allocated an array of characters for yourstring, we have room to put the copied data. Now on line 20 when we change mystring, yourstring is unaffected and when we print it on line 22 we have the original.

Show your TA you know how to find the source of a 'segmentation fault' by uncommenting the line of code that will cause the seg fault to occur and then running gdb in front of your TA to display the line number that caused the fault.

4.3 [7 pts.] Word Game

Go to Lab6: C String in Vocareum
Click on `game.cpp`

Note wordBank, an array of 10 strings (char *s). Your program should do the following:

1. Select a word at random from the wordBank. This is done for you.
2. On each turn display the word, with letters not yet guessed showing as *'s, and letters that have been guessed showing in their correct location
3. The user should have 10 attempts ("lives"). Each unsuccessful guess costs one attempt. Successful guesses do NOT count as a turn.
4. You must use the C-String library (#include <cstring>) functions strlen() and strcmp() in your program [strcmp() can compare when the guessed word matches the target/selected word from the wordBank… meaning the user won/finished]
5. You must complete and use the function

   ```
   int processGuess(char* word, const char* targetWord, char guess)
   ```

   This function should take the pointer to the current value of the word guessed thus far, the pointer to the target/selected word from the wordBank, and the character that the user guessed. It should change any *'s to actual good characters for the letter the user guessed and return a count of how many times the guessed letter appears in that word. In this way, if you return 0, it means the user guessed a letter that did NOT appear and thus should lose a turn back in your main().
6. At every turn display the current version of the guessed word and how many turns remain
7. At the end of the game give a descriptive message whether the user won or lost

Demonstrate your program and show your TA/CP the two functions you wrote explaining how it works.

Sample runs of our solution to this program are shown below.

A winning example:

```
Current word:  *******
10 remain...Enter a letter to guess:
o

Current word:  *******
9 remain...Enter a letter to guess:
l

Current word:  *******
8 remain...Enter a letter to guess:
```

```
c

Current word:  *c***c*
8 remain...Enter a letter to guess:
s

Current word:  sc***c*
8 remain...Enter a letter to guess:
g

Current word:  sc***c*
7 remain...Enter a letter to guess:
i

Current word:  sci**c*
7 remain...Enter a letter to guess:
e

Current word:  scie*ce
7 remain...Enter a letter to guess:
n
The word was:  science.  You win!
```

A losing example:

```
Current word:  ******
10 remain...Enter a letter to guess:
z

Current word:  ******
9 remain...Enter a letter to guess:
z

Current word:  ******
8 remain...Enter a letter to guess:
z

Current word:  ******
7 remain...Enter a letter to guess:
z

Current word:  ******
6 remain...Enter a letter to guess:
z

Current word:  ******
5 remain...Enter a letter to guess:
z

Current word:  ******
4 remain...Enter a letter to guess:
z

Current word:  ******
3 remain...Enter a letter to guess:
z

Current word:  ******
2 remain...Enter a letter to guess:
z

Current word:  ******
```

```
1 remain...Enter a letter to guess:
z
Too many turns...You lose!
```