Assignment 6
C/C++ Programming I

Exercise 0 *(6 points – 1 point per question – No program required)*

Language standards compliance and appropriate header file inclusion is assumed. Testing code by running it is sometimes misleading due to implementation dependence. These are not trick questions and there is only one correct answer to each. Applicable notes from the course book are listed.

1. If **char**s are 8 bits, **int**s are 16 bits, and ***int** *ip = (**int** *)20,* predict the value of *++ip* (Note 6.14)
   A. 20
   B. 21
   C. 22
   D. 23
   E. none of above or implementation dependent

2. Which one of the following does not access *abc[5]* or is syntactically incorrect? (Note 6.16)
   A. *abc + 5
   B. *(8 + abc - 3)
   C. *(abc + 5)
   D. 5[abc]
   E. abc[2 * 4 - 3]

3. What is the biggest problem with the following?
```
static int array[5] = { 1, 2, 3, 4, 5 };
int *ip, *ptr = array;
for (*ip = 0; *ip < 5; (*ip)++)
    printf("%d ", *ptr++);
```
   (Note 6.6)
   A. (*ip)++ is not a portable expression
   B. **ptr++* is ambiguous and should be written **(ptr++)*.
   C. **(array + *ip)* would be valid in place of **ptr++* in the *printf* statement.
   D. *array[*ip]* would be valid in place of **ptr++* in the *printf* statement.
   E. there is a pointer problem involving *ip*.

4. For ***int** p[9];* the data types passed to function *xyz* by *xyz(p, &p)* left-to-right are: (Note 6.16)
   A. "array of 9 **int**" and "address of **int**"
   B. "array of 9 **int**" and "pointer to **int**"
   C. "pointer to **int**" and "pointer to array of 9 **int**"
   D. "**int**" and "pointer to **int**"
   E. implementation dependent

5. Predict what gets printed if *one two three* is entered in response to the *scanf*:
```
char a[32], b[32], c[32];
int x =
  scanf("\n%31[a-t ]\n%[^t] %s", a, b, c);
printf("%d %s%s%s", x, a, b, c);
```
   (Note 7.3)
   A. garbage
   B. *3 one two three*
   C. garbage *one two three*
   D. *3 one three*
   E. none of above or implementation dependent

6. Predict the output:
```
string buf("Length");
buf += " of" + " this is: ";
string buf2("Strange characters" + 8,
            "Strange characters" + 18);
cout << buf << buf2.size() << ' ' << buf2;
```
   (Notes 6.14, 6.16, 7.1, & 7.7)
   A. garbage
   B. The code will not compile
   C. *Length of this is: 10 characters*
   D. *Length of this is: Strange characters*
   E. none of above or implementation dependent

Submitting your solution

Using the format below place your answers in a plain text file named **C1A6E0_Quiz.txt** and send it to the Assignment Checker with the subject line **C1A6E0_*ID***, where ***ID*** is your 9-character UCSD student ID.

-- Place an appropriate "Title Block" here --
1. A
2. C
etc.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

What is *size_t* ?

size_t (note 2.12) is an implementation-defined data type typically used to count things related to memory, such as the amount of storage needed for an object or the number of objects available or to be processed. The actual data type that **size_t** represents is at the discretion of the compiler designer and may be any of **unsigned char**, **unsigned short**, **unsigned int**, **unsigned long**, or **unsigned long long**. Any assumptions on the part of the applications programmer about which one of these it actually is are not portable.

"Lexicographical" String Comparison

A string is defined as a sequence of characters ending with a null terminator character. Two strings are considered equal if they are the same length and all of their corresponding characters are equal. The term "corresponding characters" refers to the 1st character in one string compared to the 1st character in the other string, the 2nd character in one string compared to the 2nd character in the other string, etc. Two strings are not equal if they are different lengths or if any corresponding characters are not equal. For non-equal strings the string considered to be greater is not necessarily the longer string. Instead, which is greater is determined entirely by the relative numeric values of the first two corresponding characters that are not equal. For example, the greatest of strings "Heat" and "Hi" is "Hi" because the second two corresponding characters ('e' and 'i') are not equal and the value of 'i' is greater than the value of 'e' (ASCII character set assumed). This method of comparing strings is known as "lexicographical" (dictionary) comparison.

Inputting an Entire (Possibly Empty) User Line in C

All strings stored in a C program end with the null terminator character, `'\0'`, and an "empty" string contains **only** that character. The standard library `strlen` function will report the length of an empty string to be 0. Because of the sometimes inconsistent behavior of the `scanf` function between compilers I recommend against using it to read empty user input lines. Instead, both empty and non-empty user lines can be reliably read using the `fgets` function. `fgets` reads and stores an entire input line as a string, including the newline character that terminates that line. Often, however, that newline character is not wanted and must be removed, and a common way to do it is to simply overwrite it. Because there are occasionally cases in which there is not a newline character, this should be tested first or there will be a risk of overwriting a character you actually want:

```
char buffer[BUF_SIZE];
size_t length;
fgets(buffer, BUF_SIZE, stdin);
length = strlen(buffer);
if (length != 0 && buffer[length - 1] == '\n')
    buffer[--length] = '\0';
```

In C++ the `getline` function should be used instead of `fgets` since it has the advantage of automatically discarding the newline character, thereby eliminating the need to remove it manually.

Exercise 1 *(4 points – C Program)*

Exclude any existing source code files that may already be in your IDE project and add two new ones, naming them **C1A6E1_MyStrlen.c** and **C1A6E1_main.c**. Do not use `#include` to include either of these files in each other or in any other file. However, you may use it to include any appropriate header file(s) you need.

File **C1A6E1_MyStrlen.c** must contain a function named `MyStrlen` that has the same syntax and functionally as the standard library `strlen` function. If you aren't familiar with `strlen` look it up in your IDE's help, the course book, any good C textbook, or online. `MyStrlen` must:

1. have the syntax (prototype): `size_t MyStrlen(const char *s1);`
2. return a count of the number of characters in the string in `s1`, not including the null terminator;
3. use only one variable other than its formal parameter `s1`. That variable:
    a. must be of type "pointer to **const char**";
    b. must be initialized when declared and not changed after that;
4. not call any functions, use any macros, or display anything;
5. not use the **sizeof** operator (it wouldn't help anyway).

File **C1A6E1_main.c** must contain function `main`, which must:

1. prompt the user to enter a string (which may be empty or contain spaces);
2. call `strlen` to determine that string's length;
3. call `MyStrlen` to determine that string's length;
4. display the string and its length as determined by both `strlen` and `MyStrlen` in the following 2-line format, where `ABC` is the string used in this example and where the question marks represent the integral decimal numeric values returned by the functions. By sure to enclose the string in double-quotes:

```
strlen("ABC") returned  ?
MyStrlen("ABC") returned ?
```

Manually re-run your program several times, testing with at least the following 4 strings (the last string is empty):

1. `a`
2. `HELLO`
3. `C/C++ Programming I`
4. *(an empty string)*

Submitting your solution

Send your two source code files to the Assignment Checker with the subject line **C1A6E1_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

Hints:

Do you know what **size_t** is? If not consider reviewing note 2.12. Do you know what pointer subtraction is? If not consider reviewing note 6.14. For an example of using a pointer to walk through a string see notes 6.17 and 7.2. No special case is needed for an empty string. Set the extra pointer variable you are allowed to declare equal to the parameter pointer variable then increment one of these pointers as you step through the input string looking for the null terminator character, `'\0'`. When you find it, subtract the two pointers to find the string length and return that difference. Type cast the return expression to **size_t** to avoid a compiler warning. Most library functions that compute values, including `strlen`, do no printing.

Exercise 2 *(4 points – C Program)*

Exclude any existing source code files that may already be in your IDE project and add two new ones, naming them **C1A6E2_MyStrcmp.c** and **C1A6E2_main.c**. Do not use `#include` to include either of these files in each other or in any other file. However, you may use it to include any appropriate header file(s) you need.

File **C1A6E2_MyStrcmp.c** must contain a function named `MyStrcmp` that has the same syntax and functionally as the standard library `strcmp` function. If you aren't familiar with `strcmp` look it up in your IDE's help, the course book, any good C textbook, or online. `MyStrcmp` must:

1. have the syntax (prototype): `int MyStrcmp(const char *s1, const char *s2);`
2. return:
    a. any value < 0 if the string in `s1` is lexicographically less than the string in `s2`;
    b. 0 if the string in `s1` is equal to the string in `s2`;
    c. any value > 0 if the string in `s1` is lexicographically greater than the string in `s2`.
3. not use any variables other than its two formal parameters `s1` and `s2`;
4. not call any functions, use any macros, or display anything;
5. not use the **sizeof** operator (it wouldn't help anyway).

File **C1A6E2_main.c** must contain function `main`, which must:

1. use two separate user prompts to obtain two strings (both of which may be empty or contain spaces);
2. call `strcmp` to compare the two strings;
3. call `MyStrcmp` to compare the two strings;
4. display the relationship between the two strings as determined by both `strcmp` and `MyStrcmp` in the following 2-line format, where `ABCXYZ` and `DEF` are the strings in this example and where the question marks represent the integral decimal numeric values returned by the functions. Be sure to enclose the strings in double-quotes:

```
strcmp("ABCXYZ", "DEF") returned ?
MyStrcmp("ABCXYZ", "DEF") returned ?
```

Note that the values returned by `strcmp` and `MyStrcmp` don't have to match as long as they are both < 0, both == 0, or both > 0.

Manually re-run your program several times, testing with at least the following 4 string pairs (the last pair consists of two empty strings):

1. `a` and `B`
2. `HE` and `HELLO`
3. `HE` and `EHLLO`
4. *(an empty string)* and *(an empty string)*

Submitting your solution

Send your two source code files to the Assignment Checker with the subject line **C1A6E2_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

Hints:

See note 7.2 for an example of using a pointer to walk through a string and note 7.6 for some examples of string comparisons. No special case is needed for empty strings. The value obtained by subtracting the values of the two characters currently being compared is the most straightforward value to return when a return is required. Merely simultaneously step through both strings character-at-a-time, comparing the corresponding characters in each. Return when the first pair of non-equal characters is encountered or when a null terminator character, `'\0'`, is reached in either string.

Exercise 3 *(6 points – C Program)*

Exclude any existing source code files that may already be in your IDE project and add two new ones, naming them **C1A6E3_GetSubstring.c** and **C1A6E3_main.c**. Do not use `#include` to include either of these files in each other or in any other file. However, you may use it to include any appropriate header file(s) you need.

File **C1A6E3_GetSubstring.c** must contain a function named `GetSubstring` whose purpose is to create a new string from characters copied from an existing string. Its syntax (prototype) is:

```
char *GetSubstring(const char source[], int start, int count, char result[]);
```

where `source` represents the string from which to copy the characters, `start` is the index in `source` of the first character to copy, `count` is the number of characters to copy, and `result` represents an array into which the characters are to be copied. For example, the call

```
GetSubstring("investments", 2, 4, resultArray)
```

will copy the characters `vest` from string `"investments"` into `resultArray` and append a `'\0'`.

Function `GetSubstring` must:

1. handle the following three situations:
   a. If `start` is within the `source` string and `count` does not extend beyond the end of it –
      Copy `count` characters into the `result` array and append a `'\0'`.
   b. If `start` is within the `source` string but `count` does extend beyond the end of it –
      Copy all characters remaining in `source` into the `result` array and append a `'\0'`.
   c. If `start` is beyond the end of the `source` string –
      Store only a `'\0'` in the `result` array.
2. return a pointer to the first element of the `result` array;
3. use only one variable other than formal parameters `source`, `start`, `count`, and `result`; it must be an automatic variable of type "pointer to **char**".
4. not call any functions, use any macros, or display anything;
5. not use the **sizeof** operator (it wouldn't help anyway);
6. not use index or pointer offset expressions like `pointer[i]` and `*(pointer + i)`. Compact or moving pointer expressions like `*pointer++` and `pointer++` are okay. If you have trouble with this it may help to write the program using index notation first, then convert to compact or moving pointers.
7. Remember that for parameter declarations only the forms **type** `name[]` and **type** `*name` are equivalent in all ways and both mean "`name` is a pointer to **type**".

File **C1A6E3_main.c** must contain function `main`, which must:

1. prompt the user twice:
   a. first to enter a source string, which may be empty or contain spaces (don't put quotes around it);
   b. second to enter a start index and a character count, space separated on the same line;
2. call `GetSubstring` using the user entries as its first three arguments and a 256-element character array as its 4th;
3. display the results as shown in the following example, with the user-entered string first, the user-entered index next, the user-entered length next, and the extracted substring last. For readability the quotes, commas, and the literal word **extracts** are all required. For example, for the call:

   ```
   GetSubstring("investments", 2, 4, resultArray)
   ```

   the `main` function must display:

   ```
   "investments", 2, 4, extracts "vest"
   ```

   The pointer returned by `GetSubstring`, not its 4th argument, must be used to display the extracted substring.

Manually re-run your program several times, testing with at least the 7 user entry sets shown on the next page:

source	start	count	You should get
This is really fun	2	800	is is really fun
This is really fun	261	9	*(an empty string)*
This is really fun	0	12	This is real
one two three	5	87	wo three
one two three	18	7	*(an empty string)*
one two three	6	5	o thr
one two three	0	3	one
(an empty string)	3	23	*(an empty string)*

Submitting your solution

Send your two source code files to the Assignment Checker with the subject line **C1A6E3_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

Hints:

Save the `result` pointer first so you can return it later. Don't mistakenly return a pointer to the end of your substring. Don't repeatedly perform unneeded math operations, such as `source + start` or `start + count`. Although not required, an optimal solution for `GetSubstring` will contain the statement:

```
*result++ = *source++;
```

The following 2-loop algorithm is recommended:

1. Save a copy of `result`.
2. Starting at the character specified by `source`, loop on each character in `source` until either the end of the string is found or the offset specified by `start` is reached. Increment `source` and decrement `start` as you proceed as appropriate.
3. After the previous loop completes use another loop to copy each character from the updated value of `source` from step 2 into `result` until the null terminator character is reached (don't copy this character) or until `count` characters have been copied, whichever comes first. Increment `source` and `result` as you proceed as appropriate.
4. Copy a null terminator character, `'\0'`, into `*result`.
5. Return the copy of the original value of `result` made in step 1.

Get a Consolidated Assignment Report (optional)

If you would like to receive a consolidated report containing the results of the most recent version of each exercise submitted for this assignment, send an empty email to the assignment checker with the subject line **C1A6_*ID***, where ***ID*** is your 9-character UCSD student ID. Inspect the report carefully since it is what I will be grading. You may resubmit exercises and report requests as many times as you wish before the assignment deadline.