Started on	Friday, November 4, 2022, 4:38 PM
State	Finished
Completed on	Friday, November 4, 2022, 5:15 PM
Time taken	36 mins 56 secs

Question **1**

Complete

Points out of 3.00

A ***string literal*** or ***string constant*** is a sequence of zero or more characters enclosed in double quotes, as in `"I am a string"` or the empty string `""`.

The quotes are not part of the string, but serve only to delimit it. Other examples of string literals are:

```
"hello world"
"total expenditures: "
"C comments begin with '/*'.\n"
```

String literals can be concatenated at compile time. For example, the following two string literals:

```
"hello, " "world"
```

are equivalent to the string literal

```
"hello, world"
```

Technically, a string literal is an array of characters with the internal representation having a null character `'\0'` at the end. This means that the physical storage required for a string literal is one more than the number of characters written before the quotes. This also means that a string literal containing a single character is not the same as a character constant. The string literal `"a"` is an array of 2 `char` elements with the first `char` element having value `'a'` and the second `char` element having null character `'\0'`. On the other hand, character constant `'a'` has an integral value 97 [in the ASCII character set] and is of type `int`.

Since a string literal is an array of characters, we can use the subscript operator to access individual characters in the array, as in expression `"representation"[2]` which evaluates to value `'p'` of type `char`.

Now, write the ***exact*** value resulting from the evaluation of expression `sizeof("representation")`.

Answer: 15

Question **2**

Complete

Points out of 2.00

Write the ***exact*** value resulting from the evaluation of expression `sizeof('a')`.

Answer: 1

Question **3**

Complete

Points out of 2.00

Given the following definition:

```
char str[] = "Sunny";
```

is the initializer `"Sunny"` a string literal?

Select one:

- True
- False

Question **4**

Complete

Points out of 2.00

Is the following declaration statement legal?

```
char str[] = "Redmond" " " "WA";
```

Select one:

- True
- False

Question **5**

Complete

Points out of 3.00

Write the ***exact*** value resulting from the evaluation of expression `sizeof("representation"[5])`.

Answer: 1

Question **6**

Complete

Points out of 3.00

Write the ***exact*** text printed to standard output stream by the following code fragment. If the code fragment cannot be compiled, write CTE [for *compile-time error*]. If the code fragment generates undefined behavior, write UDB [for *undefined behavior*].

```
#define MAX_STR_LEN (256)

char str[MAX_STR_LEN];
str = "Sunny";
printf("%s", str);
```

Brief side-note on undefined behavior: The C standard says that statements such as `c = (b = a + 2) - (a = 1);` and `c = (b = a + 2) - (a = 1);` cause ***undefined behavior*** [because we don't know whether the left or right operand of operator `-` is evaluated first]. When a program ventures into the realm of undefined behavior, all bets are off. The program may behave differently when compiled with different compilers. But that's not the only thing that can happen. The program may not compile in the first place, if it compiles it may not run, and if it does run, it may crash, behave erratically, or produce meaningless results. In other words, undefined behavior should be avoided like the plague.

Answer: CTE

Question **7**

Complete

Points out of 2.00

Write the ***exact*** value resulting from the evaluation of expression `sizeof("a")`.

Answer: 2

Question **8**

Complete

Points out of 2.00

Write the ***exact*** text printed to standard output stream by the code fragment. If the code fragment cannot be compiled, write CTE [for *compile-time error*]. If the code fragment generates undefined behavior, write UDB [for *undefined behavior*].

```
#define MAX_STR_LEN (256)

char str1[MAX_STR_LEN] = "Sunny", char str2[MAX_STR_LEN];
str2 = str1;
printf("%s", str2);
```

Brief side-note on undefined behavior: The C standard says that statements such as `c = (b = a + 2) - (a = 1);` and `c = (b = a + 2) - (a = 1);` cause ***undefined behavior*** [because we don't know whether the left or right operand of operator `-` is evaluated first]. When a program ventures into the realm of undefined behavior, all bets are off. The program may behave differently when compiled with different compilers. But that's not the only thing that can happen. The program may not compile in the first place, if it compiles it may not run, and if it does run, it may crash, behave erratically, or produce meaningless results. In other words, undefined behavior should be avoided like the plague.

Answer: CTE

Question **9**

Complete

Points out of 2.00

Write the ***exact*** value resulting from the evaluation of expression `sizeof("")`.

Answer: 1

Question **10**

Complete

Points out of 2.00

A short note on function `fgets`

Function `fgets` reads lines from an input stream. The function takes 3 parameters:

- the base address of the buffer to store the read-in line
- the maximum number of characters to read from the file minus one, and
- a pointer to an input file stream

So the call:

```
fgets(buf, 81, infile);
```

says to read up to 80 characters from input file stream `infile` and store these characters into an array whose address is specified by `buf`. The function will read less than 80 characters if it reaches the end of the file or if its reads a newline character first. In any case, `fgets` stores a null character immediately after the last character written to `buf`.

Since `fgets` takes an upper bound on the number of characters to read, it should be the function of choice especially if you're not sure how long the lines you're reading are. For example, `fgets(buf, 81, stdin)` will read up to 80 characters from standard input stream even if there are more characters in the current line.

Finally, `fgets` will return its first parameter on success; otherwise the function will return `NULL` when it reaches the end of the file without reading any characters.

A short note on function `fputs`

Function `fputs` writes a line to a specified stream. That is, the call

```
fputs(buf, outfile);
```

writes the contents of array `buf` to output file stream `outfile`. Note that `fputs` does not append a newline character to the stream. Therefore, the statement

```
fputs(buf, stdout);
```

is equivalent to the statement

```
printf("%s", buf);
```

Question

Consider the execution of the following code fragment:

```
#define MAX_LEN (257)
char str[MAX_LEN];
fputs("Enter a sentence: ", stdout);
fgets(str, MAX_LEN, stdin);
fputs(str, stdout);
```

Suppose in response to the program's prompt, a user enters the text `today is a good day` through the keyboard [followed by the Enter key]. Now, write the ***exact*** text written to the standard output stream by the program's final statement.

Answer: today is a good day

Question **11**

Complete

Points out of 3.00

Write the ***exact*** text printed to standard output stream by the following code fragment. If the code fragment cannot be compiled, write CTE [for *compile-time error*]. If the code fragment generates undefined behavior, write UDB [for *undefined behavior*].

```
char name[4] = {'E','l','m','o'};
fputs(name, stdout);
```

Brief side-note on undefined behavior: The C standard says that statements such as `c = (b = a + 2) - (a = 1);` and `c = (b = a + 2) - (a = 1);` cause ***undefined behavior*** [because we don't know whether the left or right operand of operator `-` is evaluated first]. When a program ventures into the realm of undefined behavior, all bets are off. The program may behave differently when compiled with different compilers. But that's not the only thing that can happen. The program may not compile in the first place, if it compiles it may not run, and if it does run, it may crash, behave erratically, or produce meaningless results. In other words, undefined behavior should be avoided like the plague.

Answer: Elmo

Question **12**

Complete

Points out of 3.00

Write the ***exact*** text printed to standard output stream by the following code fragment. If the code fragment cannot be compiled, write CTE [for *compile-time error*]. If the code fragment generates undefined behavior, write UDB [for *undefined behavior*].

```
#define MAX_STR_LEN (256)

char str1[MAX_STR_LEN] = "Sunny", str2[MAX_STR_LEN] = "days";
if (str2 == str1) { // compare a character array to another character array
  fputs("alike", stdout);
} else {
  fputs("not alike", stdout);
}
```

Brief side-note on undefined behavior: The C standard says that statements such as `c = (b = a + 2) - (a = 1);` and `c = (b = a + 2) - (a = 1);` cause ***undefined behavior*** [because we don't know whether the left or right operand of operator `-` is evaluated first]. When a program ventures into the realm of undefined behavior, all bets are off. The program may behave differently when compiled with different compilers. But that's not the only thing that can happen. The program may not compile in the first place, if it compiles it may not run, and if it does run, it may crash, behave erratically, or produce meaningless results. In other words, undefined behavior should be avoided like the plague.

Answer: not alike

Question **13**

Complete

Points out of 3.00

Write the ***exact*** text printed to standard output stream by the following code fragment:

```
char name[] = {'D','i','g','i','\0','p','e','n'};
fputs(name, stdout);
```

Answer: Digi

Question **14**

Complete

Points out of 3.00

Consider the following code fragment:

```
char name[] = {'E','l','m','o'};
fputs(name, stdout);
```

Now write the additional initializer that must be added to ensure the text printed to standard output stream is `Elmo`. Note that you must ***only*** write the initializer and nothing else!!!

Answer: '\0'

Question **15**

Complete

Points out of 5.00

Consider the execution of the following code fragment:

```
#define MAX_LEN (256)

char str[MAX_LEN];
fputs("Enter a sentence: ", stdout);
scanf("%s", str);
fputs(str, stdout);
```

If the user enters the text **today is a good day** [followed by the Enter keyboard button] in response to the program's prompt, write the ***exact*** text printed to the standard output stream by the code fragment's final statement.

Answer: today is a good day

Question **16**

Complete

Points out of 5.00

Consider the execution of the following code fragment:

```
#define MAX_LEN (13)

char str[MAX_LEN];
fputs("Enter a sentence: ", stdout);
fgets(str, MAX_LEN, stdin);
fputs(str, stdout);
```

If the user enters the text **today is a good day** [followed by the Enter keyboard button] in response to the program's prompt, write the ***exact*** text printed to the standard output stream by the code fragment's final statement.

Answer: today is a g

Question **17**

Complete

Points out of 5.00

Determine the ***exact*** text printed to the standard output stream by the following code fragment:

```
for (int i = 0; "representation"[i]; ++i) {
  fputc("representation"[i] - 'a' + 'A', stdout);
}
```

Select one:

- The code fragment doesn't compile
- **REPRESENTATION**
- No values are printed to the standard output stream
- Nothing is printed because the loop is never executed
- Runtime behavior of the code fragment is undefined
- **representation**

Question **18**

Complete

Points out of 10.00

Consider the execution of the following code fragment:

```
int foo(char const array[]) {
  int i, val;
  for (i = val = 0; array[i]; ++i) {
    val = (array[i] == ' ') ? val+1 : val;
  }
  return val;
}

fputs("Enter a sentence: ", stdout);
char str[256];
fgets(str, 256, stdin);
printf("%i", foo(str));
```

If the user enters the text `today is a good day` [followed by the Enter keyboard button] in response to the program's prompt, write the ***exact*** text printed to the standard output stream by the code fragment's final statement.

Answer: 4

Question **19**

Complete

Points out of 10.00

Write the ***exact*** text printed to the standard output stream by the following code fragment:

```
char foo(int digit) {
  return "0123456789ABCDEF"[digit];
}

char str[16] = {0};
int x = 54321, i = 0;
while (x) {
  str[i++] = foo(x%16);
  x /= 16;
}
str[i] = '\0';

for(--i; i >= 0; --i) {
  fputc(str[i], stdout);
}
```

Answer: D431

Question **20**

Complete

Points out of 10.00

Write the ***exact*** text printed to the standard output stream by the following code fragment:

```
#define MAX_STR_LEN 256

void foo(char array1[], char const array2[]) {
  int i = 0, j = 0;
  while (array1[i++]);

  --i;
  while (array1[i++] = array2[j++]);
}

char str[MAX_STR_LEN] = "FourScore", str2[MAX_STR_LEN] = "AndSeven";
foo(str, str2);
fputs(str, stdout);
```

Answer: FourScoreAndSeven

Question **21**

Complete

Points out of 10.00

Write the ***exact*** text printed to the standard output stream by the following code fragment:

```
#define MAX_STR_LEN 256

void foo(char array1[], char const array2[]) {
  int i = 0;
  while (array1[i] = array2[i]) ++i;
}

char str[MAX_STR_LEN] = "FourScore", str2[MAX_STR_LEN];
foo(str2, str);
fputs(str2, stdout);
```

Answer: FourScore

Question **22**

Complete

Points out of 10.00

Write the ***exact*** text printed to the standard output stream by the following code fragment:

```
int foo(char const array[]) {
  int i = 0;
  while (array[i++]) ;
  return i-1;
}

char str[] = "FourScoreAndSeven";
printf("%i", foo(str));
```

Answer: 17

◄ Lecture 18 [11/02/2022]: Storage duration, Scope, and Linkage of Variables and Functions

Jump to...

Lab 9: Spell Check ►