| | |
|---|---|
| **Started on** | Friday, September 23, 2022, 4:19 PM |
| **State** | Finished |
| **Completed on** | Friday, September 23, 2022, 5:51 PM |
| **Time taken** | 1 hour 31 mins |
| **Grade** | **55.00** out of 55.00 (**100**%) |

Question **1**

Correct

1.00 points out of 1.00

All input and output in a C program is dealt with in the form of streams ✔ . Make sure to avoid extraneous characters in your answer.

The correct answer is: streams

Question **2**

Correct

1.00 points out of 1.00

For lab and personal computers, every C program's input ✔ stream is by default connected to the keyboard. Make sure to avoid extraneous characters in your answer.

The correct answer is: standard input

Question **3**

Correct

1.00 points out of 1.00

On lab and personal computers, every C program's output ✔ stream is by default connected to the computer screen. Make sure to avoid extraneous characters in your answer.

The correct answer is: standard output

Question **4**

Correct

1.00 points out of 1.00

Precise output formatting is accomplished with the C standard library function printf ✔ . Make sure to avoid extraneous characters in your answer.

The correct answer is: printf

Question **5**

Correct

1.00 points out of 1.00

Function **`printf`** has the form

```
printf( format-string, print-list);
```

where ***`format-string`*** describes the output format, and optional ***`print-list`*** consists of comma-separated values that correspond to each *conversion specification* in ***`format-string`***. Each conversion specification begins with a percent sign `%` and ends with a *conversion (or format) specifier*.

Can there be many conversion specifications in one format string?

Select one:

- True ✔
- False

The correct answer is 'True'.

Question **6**

Correct

1.00 points out of 1.00

Any call to function **`printf`** must supply at least one argument (or, input) that is known as the ***format string***.

Select one:

- True ✔
- False

The correct answer is 'True'.

Question **7**

Correct

1.00 points out of 1.00

***Format or conversion specifiers*** in the format string of function **`printf`** begin with the `%` character.

Select one:

- True ✔
- False

The correct answer is 'True'.

Question **8**

Correct

1.00 points out of 1.00

Function **`printf`** may use the conversion specifier _____ or _____ to output a signed decimal integer. Pick *exactly* two choices from the multiple choice list.

Select one or more:

- **`i`** ✔
- **`x`**
- **`f`**
- **`d`** ✔
- **`u`**
- **`e`**

Your answer is correct.

The correct answers are: **`d`**, **`i`**

Question **9**

Correct

1.00 points out of 1.00

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

```
// i is an int variable initialized to value 456
printf("%d", i);

```

Answer: 456 ✔

The correct answer is: 456

Question **10**

Correct

1.00 points out of 1.00

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

```
// i is int variable initialized to value 456
printf("%i", i);


```

Answer: 456 ✔

Conversion specifier **`i`** has same behavior as **`d`** in function **`printf`**.

The correct answer is: 456

Question **11**

Correct

1.00 points out of 1.00

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

```
printf("%d", +456);

```

Answer: 456 ✔

Plus sign doesn't print with **`d`** or **`i`** conversion specifiers.

The correct answer is: 456

Question **12**

Correct

1.00 points out of 1.00

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

```
printf("%d", -456);

```

Answer: -456 ✔

While plus signs don't print, minus sign prints.

The correct answer is: -456

Question **13**

Correct

1.00 points out of 1.00

**h** is a conversion specifier that is called a *length modifier* because it is placed *before* any integer conversion specifier (such as **i** or **d**) to indicate that a value of type **short int** is to be displayed. Write the *exact* sequence of characters printed to standard output by the following code fragment:

```
// assume i is a variable of type int and
// initialized with value 65535
printf("%hd", i);

```

Answer: -1 ✔

The correct answer is: -1

Question **14**

Correct

1.00 points out of 1.00

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

```
// assume i is a variable of type int
// initialized with value 456
printf("%o", i);

```

Answer: 710 ✔

Conversion specifier **o** displays corresponding value as an unsigned octal integer.

The correct answer is: 710

Question **15**

Correct

1.00 points out of 1.00

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

```
// assume i is a variable of type int
initialized with value 456
printf("%x", i);

```

Answer: 1c8 ✔

Conversion specifier **x** displays its corresponding value as an *unsigned hexadecimal integer*.

The correct answer is: 1c8

Question **16**

Correct

1.00 points out of 1.00

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

```
// assume i is a variable of type int
// initialized with value -456
printf("%X", i);

```

Answer: fffffe38 ✔

Conversion specifier **x** displays its corresponding value as an *unsigned hexadecimal integer* with hexadecimal digits represented in capital letters [**A**-**F**].

The correct answer is: FFFFFE38

Question **17**

Correct

1.00 points out of 1.00

We have seen that when function **`printf`** is called, it must be supplied with the format string containing ordinary characters and conversion specifications, followed by an optional *print list* consisting of values that are to be inserted into the string during printing. For example, the following **`printf`** statement:

```
printf("One number: %d", num);
```

contains one conversion specification and one value in the print list. This call to function **`printf`** will display the ordinary characters **`"One number: "`**, then the conversion specification **`%d`** specifies that function **`printf`** is to convert the binary **`int`** value stored in a memory location labeled **`num`** to a sequence of decimal digits representing the decimal value of **`num`**.

Will compilers flag an *error* when compiling source code involving a **`printf`** statement where the number of conversion specifications in a format string doesn't match the number of values in the print list.

Hint: Read page 38 of the text - the textbook description will be the arbiter of the answer irrespective of the behavior of a specific compiler.

Select one:

- True
- False ✔

The correct answer is 'False'.

Question **18**

Correct

1.00 points out of 1.00

Write the *exact* value printed to standard output by the following code fragment?

```
printf("%d", 'd');
```

Answer: 100 ✔

The correct answer is: 100

Question **19**

Correct

1.00 points out of 1.00

What is the output of the following code fragment?

```
// suppose variable ival is defined as type int
// and initialized with value 12
// also, suppose variable fval is defined as type
// float and initialized with value 78.9f
printf("%d %f", fval, ival);
```

Hint: Read page 38 of the text - the textbook description will be the arbiter of the answer irrespective of the behavior of a specific compiler.

Select one:

- Code fragment prints values **`78`** and **`12.000000`** to standard output
- Code fragment doesn't compile
- Code fragment causes program crash at runtime
- Code fragment doesn't link
- The behavior of the code fragment is unspecified - it will print meaningless values ✔

Your answer is correct.

The correct answer is: The behavior of the code fragment is unspecified - it will print meaningless values

Question **20**

Correct

1.00 points out of 1.00

What is the output of the following code fragment?

```
// suppose ival is a variable of type int
// initialized with value 10
// suppose ival2 is a variable of type int
// initialized with value 20
printf("%d", ival, ival2);

```

Hint: Read page 38 of the text - the textbook description will be the arbiter of the answer irrespective of the behavior of a specific compiler.

Select one:

- Code fragment prints **10** followed by a meaningless value to standard output
- Code fragment prints **10** and **20** to standard output
- Code fragment prints **10** to standard output. ✔
- Code fragment causes runtime crash
- Code fragment doesn't link
- Code fragment doesn't compile

Your answer is correct.

The correct answer is: Code fragment prints **10** to standard output.

Question **21**

Correct

1.00 points out of 1.00

If function **printf** is able to successfully write to standard output, it returns an **int** value representing the number of characters written to standard output. What is the value printed to standard output by the *second* **printf** statement in the following code fragment?

```
// assume z is defined as an int variable
z = printf("%d%d", 200, 500);
printf("%d", z);

```

Answer: 6 ✔

The correct answer is: 6

Question **22**

Correct

1.00 points out of 1.00

Function **printf** also provides *flags* to supplement its output formatting capabilities. As discussed in **this** reference, five flags are available for use in format strings: **-**, **+**, *space*, **#**, and **0**. To use a flag in a format string, place the flag immediately to the right of the percent sign, as in **%+d**. Several flags may be combined in one conversion specifier. Research the **#** flag to answer the following question.

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

```
// assume i is an int variable initialized
// with value 20
printf("%#x %#x", i, -i);

```

Answer: 0x14 0xffffffec ✔

The correct answer is: 0x14 0xffffffec

Question **23**

Correct

1.00 points out of 1.00

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

```
// assume x is defined as an int variable
// and initialized with value 31
printf("%+d", -x*-2);

```

Answer: +62 ✔

The correct answer is: +62

Question **24**

Correct

1.00 points out of 1.00

In addition to flags, function **`printf`** allows programmers to specify the *minimum* number of characters to be printed for a value. This is known as the *width* option and is particularly useful for printing tables because small and large numbers can be made to take the same amount of space. An integer representing the width is inserted *between* the percent sign `%` and the conversion specifier, as in `%4d`. If the value being printed is *smaller* than the width, the value will normally be *right justified* with empty spaces used as padding. On the other hand, if the value is *larger* than the width, the width will be increased to accommodate the value. Additional information about the width option can be found at **this** reference.

What is the output produced by the following code fragment? Suppose the answer is a sequence of characters **`abcde`**, write the answer by enclosing it within double quotes like this: **`"abcde"`**

```
printf("%10d", 1234);

```

Answer: "  1234" ✔

The correct answer is: " 1234"

Question **25**

Correct

1.00 points out of 1.00

Combine your knowledge of flags and width option to determine the *exact* output produced by the following code fragment. Suppose the answer is a sequence of characters **`abcde`**, write the answer by enclosing it within double quotes like this: **`"abcde"`**

```
printf("%-10d", 1234);

```

Answer: "1234  " ✔

The correct answer is: "1234 "

Question **26**

Correct

1.00 points out of 1.00

Combine your knowledge of flags and field width to determine the *exact* output produced by the following code fragment.

```
printf("%010d", 1234);

```

Answer: 0000001234 ✔

The correct answer is: 0000001234

Question **27**

Correct

1.00 points out of 1.00

In addition to flags and width option, function **`printf`** also enables you to specify the *precision* with which a value is printed. Precision has different meanings for different data types. When used with integer conversion specifiers, precision indicates the *minimum number of digits to be printed*. If the printed value contains fewer digits than the specified precision, *zeros* are prefixed to the printed value until the total number of digits is equivalent to the precision. To use precision, place a decimal point `.` followed by an integer representing the precision between the percent sign `%` and conversion specifier.

What is the *exact* value printed to standard output by the following call to function **`printf`**?

```
1 printf("%.5d", 1024);
2
```

Answer: 01024 ✔

The correct answer is: 01024

Question **28**

Correct

1.00 points out of 1.00

Combine your knowledge of flags, width and precision options, by writing the *exact* characters written by the following code fragment to standard output.

```
1 // assume i is a variable of type int
2 // initialized with value 456
3 printf("%6.5d", i);
4
```

Use **this** reference page to know more about flags, width, and precision options.

If the characters written to standard output are **`abcde`**, write the answer by enclosing it within a pair of double quotes like this: **`"abcde"`**

Answer: " 00456" ✔

The correct answer is: " 00456"

Question **29**

Correct

1.00 points out of 1.00

Combine your knowledge of flags, width and precision options, by writing the *exact* characters written by the following code fragment to standard output.

```
1 // assume i is a variable of type int
2 // initialized with value 456
3 printf("%-6.5d", i);
4
```

Use **this** reference page to know more about flags width, and precision options.

If the characters written to standard output are **`abcde`**, write the answer by enclosing it in a pair of double quotes like this: **`"abcde"`**

Answer: "00456 " ✔

The correct answer is: "00456 "

Question **30**

Correct

1.00 points out of 1.00

Function **`printf`** will use the conversion specifier _____ to output floating-point values in fixed-point notation.

Pick *exactly* one choice from the multiple choice list.

Select one:

- **`d`**
- **`i`**
- **`e`**
- **`u`**
- **`x`**
- **`f`** ✔

Your answer is correct.

The correct answer is: **`f`**

Question **31**

Correct

1.00 points out of 1.00

Function **`printf`** will use the conversion specifier _____ to output floating-point values in exponential notation.

Pick *exactly* one choice from the multiple choice list.

Select one:

- **`d`**
- **`x`**
- **`u`**
- **`e`** ✔
- **`f`**
- **`i`**

Your answer is correct.

The correct answer is: **`e`**

Question **32**

Correct

1.00 points out of 1.00

Values displayed with conversion specifiers **`f`**, **`e`**, and **`E`** show *six* digits of precision to the right of the decimal point by default. Confirm whether the above statement is true or not by printing values such as **`1.2`**, **`1.23`**, **`1.234`**, **`1.2345`**, **`1.23456`**, **`1.234567`**, and **`1.2345678`**.

Select one:

- True ✔
- False

The correct answer is 'True'.

Question **33**

Correct

1.00 points out of 1.00

The **`e`** and **`E`** conversion specifiers cause the value to be *rounded* in the output and the conversion specifier **`f`** does *not*. Confirm whether the above statement is true or not by printing the floating-point value **`1234567.89`** using these three conversion specifiers.

Select one:

- True ✔
- False

The correct answer is 'True'.

Question **34**

Correct

1.00 points out of 1.00

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

```
printf("%f", 1234.5678);
```

Answer: 1234.567800 ✔

The correct answer is: 1234.567800

Question **35**

Correct

1.00 points out of 1.00

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

```
printf("%E", 1234.5678);
```

Answer: 1.234568E+03 ✔

The correct answer is: 1.234568E+03

Question **36**

Correct

1.00 points out of 1.00

Recall the width option that is provided by function **`printf`** to allow programmers to specify the *minimum* number of characters to be printed for a value. An integer representing the width is inserted *between* the percent sign % and the conversion specifier, as in **`%4d`**. If the value being printed is *smaller* than the width, the value will normally be *right justified* with empty spaces used as padding. On the other hand, if the value is *larger* than the width, the width will be increased to accommodate the value. Additional information about the width option can be found at **this** reference.

What is the output produced by the following code fragment?

```
printf("%10f", .23);
```

Suppose the answer is a sequence of characters **abcde**, write your answer by enclosing it within double quotes like this: **"abcde"**

Answer: " 0.230000" ✔

The correct answer is: " 0.230000"

Question **37**

Correct

1.00 points out of 1.00

Recall that function **`printf`** also enables you to specify the *precision* with which data is printed. Also recall that precision is specified by placing a decimal point **`.`** followed by an integer representing the precision between the percent sign **`%`** and the conversion specifier. We had earlier mentioned that precision has different meanings for different data types. When used with integer conversion specifiers, precision indicates the minimum number of digits to be printed. When used with floating-point conversion specifiers **`f`**, **`e`**, and **`E`**, the precision is the *number of digits to appear after the decimal point*. When a floating-point value is printed with a precision *smaller* than the original number of decimal places in the value, the value is *rounded*. Use **this** reference page for more information about precision.

Now, write the *exact* value printed to standard output by the following code fragment:

```
printf("%.3f", 123.94583);
```

Answer: 123.946 ✔

The correct answer is: 123.946

Question **38**

Correct

1.00 points out of 1.00

The width and precision options can be combined by placing the width, followed by a decimal point, followed by a precision between the percent sign **%** and the conversion specifier, as in the statement

```
// assume f is a variable of type double
// initialized with value 123.456789
printf("%9.3f", f);

```

Use **this** reference page to know more about width and precision options.

Write the *exact* characters printed by the above code fragment to standard output. If the characters written to standard output are **`abcde`**, write your answer by enclosing it within double quotes like this **`"abcde"`**

Answer: " 123.457" ✔

The correct answer is: " 123.457"

Question **39**

Correct

1.00 points out of 1.00

It is possible to specify the field width and the precision using integer expressions in the argument list following the format string. To use this feature, an asterisk **`*`** must be inserted in place of the width or precision or both. The matching argument of type **`int`** in the argument list is evaluated and used in place of the asterisk. The width option's value may be either positive or negative (which causes the output to be left justified in the field, as described in **this** reference). For example, the statement

uses **`9`** for width and **`3`** for precision. The characters printed by this statement would be **`123.457`** right justified with preceding two space characters.

What is the output produced by the following code fragment?

Suppose the answer is the sequence of characters **abcde**, write your answer by enclosing it in a pair of double quotes like this: **"abcde"**

Answer: "98.74 " ✔

The correct answer is: "98.74 "

Question **40**

Correct

1.00 points out of 1.00

Most literal characters to be printed in a printf statement can simply be included in the format string. However, there are several *problem* characters, such as the **%** character or quotation mark **"** that delimits the format string itself. Such problem characters and various control characters, such as newline and tab, must be represented by *escape sequences*. An escape sequence is represented by a backslash ****, followed by a particular escape character. The list of escape characters is provided in Chapter 3 of the textbook.

Write a complete **`printf`** statement that *only* rings the alarm bell of your computer.

Answer: printf("\a"); ✔

The correct answer is: printf("\a");

Question **41**

Correct

1.00 points out of 1.00

Write the complete statement containing a call to function **`printf`** that writes only a single `%` character (and nothing else) to standard output.

Answer: printf("%%");

The correct answer is: printf("%%");

Question **42**

Correct

1.00 points out of 1.00

While function **`printf`** writes formatted characters to standard output stream, function **`scanf`** performs precise formatting on characters in the standard input stream. A call to function **`scanf`** has the following form:

```
scanf( format-string, other-arguments );
```

where ***`format-string`*** describes the formats of the input using conversion specifiers, and ***`other-arguments`*** are pointers to variables (for now think of pointers as memory addresses of variables) in which the input will be stored.

How does **`scanf`** format characters in standard input stream as floating-point or integer values? Consider the following **`scanf`** statement:

Like function **`printf`**, **`scanf`** is controlled by the format string. When it is called, **`scanf`** begins processing the information in the format string, starting at the left which contains conversion specification **`%d`**. For conversion specifier **`d`**, **`scanf`** will try to locate largest group of characters that will comprise a value of type **`int`**. As **`scanf`** searches for the beginning of a number, it will ignore *whitespace* characters (the space, horizontal tab, newline). **`scanf`** will search for a digit, a plus sign, or a minus sign; it then reads digits until it reaches a nondigit. **`scanf`** will then construct a decimal value from the plus or minus sign and the digits which will then be stored in variable **`x`**.

When **`scanf`** encounters a character that can't be part of the current value, the character is put back to be read during the scanning of the next input value specified by the second conversion specification which in the current example is also **`%d`**. The process of scanning characters from standard input stream is continued to format an integer value from the remaining characters and then store the resulting value of type **`int`** into variable **`y`**. Any further characters in the standard input stream will be handled by the next call of **`scanf`**.

If at any time in this process of scanning characters in the standard input stream, **`scanf`** will prematurely return if it is unable to match these characters to the conversion specifier in the format string.

Now, use the following code fragment to answer the question:

Select one:

- The code fragment will compile, link and execute but its runtime behavior is undefined meaning that anything at all can happen - the program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended
- **`x`** has value **`1`** after execution of the code fragment ✔
- **`x`** has value **`0`** after execution of the code fragment
- The code fragment will not compile
- The code fragment will not link

Your answer is correct.

The correct answer is: **`x`** has value **`1`** after execution of the code fragment

Question **43**

Correct

1.00 points out of 1.00

**This** reference page provides the complete list of conversion specifiers. Of particular interest are integer conversion specifiers **`d`** and **`i`**. While these conversion specifiers exhibit similar behavior in function **`printf`**, their behavior is different in function **`scanf`**. Conversion specifier **`d`** reads an optionally signed decimal integer while conversion specifier **`i`** reads an optionally signed decimal, octal, or hexadecimal integer.

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

Answer: -7056112 ✔

The correct answer is: -7056112

Question **44**

Correct

1.00 points out of 1.00

Read page 42 of the textbook to answer this question and assume that warning and error options of your compiler are turned off. Now, use the following code fragment to answer the question.

```
// suppose x is a variable of type int
// assume the user enters the characters: -10*
// where * represents the newline character
scanf("%d", x);
```

Select one:

- The code fragment will not compile
- **`x`** has value **`-10`** after execution of the code fragment
- The code fragment will not link
- The code fragment will compile with a warning message, link and execute but its runtime behavior is undefined. Undefined behavior means anything at all can happen including the program crashing or the program silently generating incorrect results, or the program may fortuitously do exactly what the programmer intended ✔

Your answer is correct.

The correct answer is: The code fragment will compile with a warning message, link and execute but its runtime behavior is undefined. Undefined behavior means anything at all can happen including the program crashing or the program silently generating incorrect results, or the program may fortuitously do exactly what the programmer intended

Question **45**

Correct

1.00 points out of 1.00

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

```
// suppose x is variable of type int initialized with value 5
// assume user enters the characters: -12.3*
// where * represents the newline character
scanf("%d", &x);
printf("%d", x);
```

Answer: -12 ✔

The correct answer is: -12

Question **46**

Correct

1.00 points out of 1.00

Write the *exact* output printed to standard output by the following code fragment. Pay careful attention to the value entered by the user.

Answer: 12 ✔

The correct answer is: 12

Question **47**

Correct

1.00 points out of 1.00

Often it's necessary to skip certain characters in the input stream. For example, a date could be entered as

```
03-31-2012
```

Each number in the date needs to be stored, but the dashes that separate the numbers can be discarded. To eliminate unnecessary characters, include them in the format string of **`scanf`**. You don't have to worry about whitespace characters - such as space, tab, newline - because **`scanf`** ignores whitespace characters. For example, to skip the dashes in the input, use the statement

Now, carefully examine the following code fragment and write the *exact* output printed to standard output:

Answer: 425558 ✔

The correct answer is: 425558

Question **48**

Correct

1.00 points out of 1.00

Study the following code fragment to answer the question:

Select one:

- The code fragment will store value `425` in variable `x` and store value `558` in variable `y`
- The code fragment compiles but will not link
- The code fragment doesn't compile
- The code fragment will store value `425` in variable `x` but will leave variable `y` unchanged ✔
- The code fragment compiles and links but will crash during execution
- The code fragment will leave both variable `x` and variable `y` unchanged

Your answer is correct.

The correct answer is: The code fragment will store value `425` in variable `x` but will leave variable `y` unchanged

Question **49**

Correct

1.00 points out of 1.00

A *width* can be used in a `scanf` conversion specifier to read a specific number of characters from the input stream. The width is specified as an integer value between the percent sign `%` and the conversion specifier. Write the exact output printed to standard output stream by the following code fragment:

Answer: 3 ✔

The correct answer is: 3

Question **50**

Correct

1.00 points out of 1.00

Carefully study the following code fragment and write the *exact* value printed to standard output by the following code fragment.

Answer: 7890456123 ✔

The correct answer is: 7890456123

Question **51**

Correct

1.00 points out of 1.00

It is important to distinguish writing and reading values of type **`double`** using functions **`printf`** and **`scanf`**, respectively. While function **`printf`** uses conversion specifier **`f`** to write values of type **`double`** to standard output stream, function **`scanf`** uses conversion specifier **`lf`** to read values of type **`double`** from standard input stream.

Suppose **`dbl`** is previously defined as a variable of type **`double`**. Write the minimal (meaning the simplest possible C code) **`scanf`** statement that will format characters from the standard input stream and store the resulting value of type **`double`** in variable **`dbl`**.

Answer: scanf("%lf", &dbl); ✔

The correct answer is: scanf("%lf", &dbl);

Question **52**

Correct

1.00 points out of 1.00

Conversion specifier **`lf`** is versatile - it can be used to read double-precision floating-point numbers in both fixed-point (as in **`338.47`**) and in exponential-notation (as in **`3.3847e+2`**). When asked to read a floating-point number, **`scanf`** looks for:

- an optional plus or minus sign, followed by
- a series of digits possibly containing the decimal point, followed by
- an optional exponent. An exponent consists of the letter **`e`** or **`E`**, an optional plus or minus sign, and one or more digits.

In class lectures we learnt that floating-point values are imprecise approximations of real numbers. and the following code fragment confirms this point. Write the *exact* output printed to standard output stream by the following code fragment which illustrates the fact that floating-point values are imprecise.

Answer: 0.000003 ✔

The correct answer is: 0.000003

Question **53**

Correct

1.00 points out of 1.00

Write the *exact* values printed to standard output by the following code fragment.

Answer: 98- ✔

The correct answer is: 98-

Question **54**

Correct

1.00 points out of 1.00

Now that you're aware of how **`scanf`** reads integer and floating-point numbers, write the *exact* output printed to standard output stream by the following code fragment:

Answer: 90.500000 ✔

The correct answer is: 90.500000

Question **55**

Correct

1.00 points out of 1.00

A little known feature of **`scanf`** is that it returns a value of type **`int`** specifying the number of *values* that were successfully read from the standard input stream. For example, in the following code fragment:

function **`scanf`** reads one **`int`** value successfully and therefore returns a value **`1`** of type **`int`**. We will make use of this feature extensively, especially in loops and iterations that read values one by one from a file that is redirected to the standard input stream.

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

Answer: 2 ✔

The correct answer is: 2

◄ Lecture 6 [09/14/2022]: Expressions; Precedence and Associativity of Operators; Assignment and Arithmetic Operators

Jump to...

Lab 3: Problem Solving ►