DUMPSBOSS.

CLA - C Certified Associate Programmer

C++ Institute CLA-11-03

Version Demo

Total Demo Questions: 5

Total Premium Questions: 58

Buy Premium PDF

https://dumpsboss.co

support@dumpsboss.co

support@dumpsboss.co
dumpsboss.co

Topic Break Down

Topic	No. of Questions
Topic 1, Basic Concepts	10
Topic 2, Data Types, Operators, Evaluations, and Basic I/O Operations	13
Topic 3, Flow Control – Loops and Conditional Blocks	5
Topic 4, Pointers, Arrays, and Strings	12
Topic 5, Functions	10
Topic 6, Structures, Unions, Enumerated Types, and Bit Operations	8
Total	58

QUESTION NO: 1

What happens if you try to compile and run this program?

```
#include

int main (int argc, char *argv[]) {

int i = 7 || 0 ;

printf("%d", !! i);

return 0;

}
```

Choose the right answer:

A. The program outputs -1

B. The program outputs 7

C. Compilation fails

D. The program outputs 1

E. The program outputs 0

ANSWER: D

Explanation:

Assuming the displayed header directive represents the intended standard `#include <stdio.h>` line, the program compiles and prints `1`. In C, the logical OR operator `||` evaluates to an integer result of either `0` or `1`; it does not preserve the operand value. The left operand, `7`, is nonzero and therefore true, so `7 || 0` evaluates to `1`. That result is assigned to `i`. The expression `!!i` applies logical negation twice. The first negation converts nonzero `i` to `0`; the second negation converts that zero to `1`. Consequently, `printf` receives the integer value `1` for its `%d` conversion and writes `1`, with no newline. This behavior follows the C language definition of logical operators: each produces either zero or one, and a nonzero operand is considered true. See cppreference's documentation for C logical operators and the C standard library description of printf.

QUESTION NO: 2

What happens if you try to compile and run this program?

```
#include

int f1(int n) {

return n = n * n;

}

int f2(int n) {

return n = f1(n) * f1(n);

}

int main(int argc, char ** argv) {

printf ("%d \n", f2(1));

return 0;

}
```

-

Select the correct answer:

A. The program outputs 8

B. The program outputs 2

C. Execution fails

D. The program outputs 4

E. The program outputs 1

ANSWER: E

Explanation:

The program outputs 1. The function `f1` receives its parameter by value, so its local `n` is initialized with the argument supplied by the caller. Its return statement first evaluates `n * n`, then assigns that result to its local parameter, and the assignment expression has the assigned value. Therefore, `f1(1)` returns `1`.

The expression in `f2` evaluates two calls to `f1` and multiplies their returned integer values. Both calls receive the value `1`, so the multiplication is `1 * 1`, producing `1`. Consequently, `f2(1)` returns `1`, and the `%d` conversion specification in `printf` writes that integer followed by a newline.

This follows C's rules that an assignment expression evaluates to the value stored after assignment and that function arguments initialize the corresponding parameters for the call. See the C standard's assignment-operator specification in ISO C11 draft, section 6.5.16 and the function-call rules in section 6.5.2.2.

QUESTION NO: 3

What happens if you try to compile and run this program?

```
#include <stdio.h>

int i = 1;

int fun(void) {
    return ++i;
}

int main(void) {
    int i = 5;

    printf("%d", i + fun());
    return 0;
}
```

Choose the right answer:

A. The program outputs 6

B. The program outputs 7

C. The program outputs 8

D. The program outputs 10

E. The behavior is undefined.

ANSWER: B

Explanation:

The program outputs `7`. The declaration inside `main` creates a local variable named `i` with the value 5. It hides the global variable named `i` only within the block of `main`.

The function `fun` has no local variable named `i`, so `++i` modifies the global object. That object changes from 1 to 2, and `fun` returns 2. The expression in `main` therefore adds its local value 5 to 2, producing 7.

QUESTION NO: 4

What happens if you try to compile and run this program?

```
#include

#include <string.h>

int main (int argc, char *argv[]) {

int a = 0, b = 1, c;

c = a++ && b++;

printf("%d",b);

return 0;

}
```

Choose the right answer:

A. The program outputs 3

B. The program outputs 1

C. The program outputs 2

D. Compilation fails

E. The program outputs 0

ANSWER: D

Explanation:

Compilation fails is correct because the first preprocessing directive is malformed: `#include <stdio.h></stdio.h>`. An `#include` directive must specify a header name, such as `<stdio.h>`, and then end at the newline. Here, the additional `</stdio.h>` text follows the valid header-name token on the same directive line. It is not part of a valid include-header-name and constitutes extra tokens in the directive. A conforming C implementation must issue at least one diagnostic for a preprocessing directive that does not satisfy its syntax requirements; consequently, this source cannot be treated as a cleanly compilable program. Since successful compilation is required before the executable can run, no output is guaranteed from this source as written.

If the first line were corrected to `#include <stdio.h>`, the logical-AND expression would use short-circuit evaluation, but that runtime behavior is irrelevant until the invalid preprocessing directive has been fixed. The C standard's rules for preprocessing directives and diagnostics are available in the WG14 C standard draft: N1570, C11 draft standard. See also the GCC documentation on preprocessing directives: Include Syntax.

QUESTION NO: 5

What happens if you try to compile and run this program?

```
#include
```

```
int main (int argc, char *argv[]) {

int i = 'A' - 'B';

int j = 'b' - 'a';

printf("%d",i / j);

return 0;

}
```

Choose the right answer:

A. Execution fails

B. Compilation fails

C. The program outputs 1

D. The program outputs -1

E. The program outputs 0

ANSWER: D

Explanation:

The program outputs -1. In the character encoding assumed by this introductory C exercise, consecutive uppercase letters have consecutive numeric values, so `'A' - 'B'` evaluates to `-1`. Likewise, consecutive lowercase letters have consecutive values, making `'b' - 'a'` evaluate to `1`. Both variables are of type `int`, because character constants such as `'A'` have type `int` in C. The expression passed to `printf` is therefore integer division: `-1 / 1`. Dividing an integer value by one preserves that value, so the result is `-1`. The `%d` conversion specification correctly prints that `int` result as a signed decimal integer, with no newline appended. The C standard specifies the form and integer type of character constants in its lexical-elements rules, and its arithmetic rules define integer division for integer operands. See the C standard draft sections on character constants and multiplicative operators at N1570, and the C arithmetic operators reference.