Hard-Level Pseudocode MCQs – Logical Operators

Assume short-circuit evaluation and C-style operator precedence (!>&&>||).

All variables are booleans unless declared otherwise.

Only ONE choice is guaranteed correct.

1.

```
bool p = true, q = false, r = true;
bool x = !p || q && r || !q;
print x;
```

A. false

B. true

C. depends on short-circuit order

D. compile-time error

2.

```
int a = 0, b = 1, c = 2;
bool z = (a++ && b++) || (c-- && a--);
print a, b, c, z;
```

A. 0 1 2 false

B. 1 1 2 true

C. 1 2 1 true

D. 1 1 1 false

3.

```
bool f(bool x) { print x; return !x; }
bool p = false, q = true;
bool y = f(p) && f(q) || f(p);
print y;
```

(Assume f is only called when its argument is evaluated.)

A. false, then true, then false → final true

B. false, then true → final true

C. false → final true

D. false, then true, then false → final false

4.

```
int m = 3, n = 5;
bool k = (m > n) || (m++ < n) && (n++ > m);
print m, n, k;
```

A. 3 5 false

B. 4 6 true

C. 4 5 true

D. 3 6 false

5.

```
bool a = true, b = false, c = true;
bool res = !(a && b) && (b || !c) || (a && !a);
print res;
```

A. false

B. true

C. indeterminate

D. runtime error

6.

```
int x = 7, y = 0;
bool flag = x && (y = x / 2) && (x /= y);
print flag, x, y;
```

A. true 3 3

B. true 2 3

C. false 7 0

D. false 7 3

7.

```
bool p = true;
bool q = p && (p = false) || p;
print p, q;
```

A. false true

B. false false

C. true true

D. true false

8.

```
int i = 4, j = 3;
bool t = (i& j) && (i | j) ^ (i&& j);
print t;
```

A. 0

B. 1

C. depends on endianness

D. undefined

9.

```
bool a = false, b = true, c = false;
bool expr = a ? b || c : !b && c;
print expr;
```

A. false

B. true

C. syntax error

D. indeterminate

10.

```
int cnt = 0;
bool mystery = !(cnt++ || cnt++) && !(cnt++ &&cnt++);
print cnt, mystery;
```

A. 2 false

B. 3 false

C. 4 true

D. 4 false

Answers

1 B

2 B

3 C

4 C

5 A

6 A

7 A

8 B

9 A

10 D

What does short-circuit evaluation mean?

Short-circuit evaluation is a strategy used by many programming languages when evaluating **logical expressions** involving the operators AND (&&) and OR (||):

- **For AND (&&)**:
  The second operand is evaluated **only if** the first operand is true.
  If the first operand is false, the whole expression is immediately false, so the second operand is skipped.
- **For OR (||)**:
  The second operand is evaluated **only if** the first operand is false.
  If the first operand is true, the whole expression is immediately true, so the second operand is skipped.

Why it matters:

- **Efficiency**: Avoids unnecessary computation.
- **Safety**: Prevents errors like null-pointer dereferencing or divide-by-zero.

- **Side effects**: Skipped expressions won't execute any side effects (e.g., function calls or increments).

Example:

c

```c
int x =0;
bool result = (x !=0) && (10/ x >1);
```

- If x == 0, the second part (10 / x > 1) is **never evaluated**, preventing division by zero.

Share

What are the benefits of short-circuit evaluation in programming?

Benefits of short-circuit evaluation (&&, ||, and, or, etc.)

1. Performance
   • Skips the right-hand operand when the final truth value is already known, saving CPU cycles.
2. Safety / Defensive programming
   • Guards against run-time faults:
   if (ptr&&ptr->value > 0) – the dereference occurs only when ptr is non-null.
   if (denom == 0 || num / denom> 2) – avoids division-by-zero.
3. Control-flow shorthand
   • Replaces small conditional blocks with one-liners:
   ok &&doSomething();
   !ok || die("fatal");
4. Correctness with side-effecting expressions
   • Prevents unintended side effects when the skipped operand would have changed state or I/O.
5. Idiomatic readability
   • Expresses pre-conditions succinctly:
   if (isOpen() &&readHeader()) …
6. Enables lazy / conditional evaluation patterns
   • Useful in configuration checks, logging, or expensive computations:
   debug && log("details");

7. Reduced branching in generated machine code
   • Compilers often emit shorter instruction sequences for short-circuit operators.
8. Maintains logical intent without auxiliary variables
   • Complex boolean expressions stay concise and self-documenting.