

- 1(b) The number  $255 = 11111111_2$  is the largest that can be represented in an 8-bit unsigned binary system. Hence,  $255 + 1$  overflows to give  $00000000_2 = 0$ .
- 2(a) Since its second argument is passed by value, `foo(4,z)` does nothing to the value of `z`. `bar(2.3,z)`, on the other hand, multiplies `z` by 2. Note that the fractional part of 2.3 is truncated when passed as an integer.
- 3(d) Identifiers (here, the functions names) that begin with a digit are invalid.
- 4(b) Since identifiers are case-sensitive, all three function definitions are distinct and there is no ambiguity about which is called. Remember that unary minus has high precedence so that `2*n+-1` is interpreted as `2*n+(-1)`.
- 5(d) The variable `l` is out of scope in the statement `prod += 1`; The code would be valid if the last two statements were enclosed in a block: `{ prod*= k; prod *= 1;}`.
- 6(c) You might be tempted to guess `false, false, false` for the output, since none of the triplets are ordered. But that would be the correct interpretation only if the comparison read `x < y and y < z` rather than `x < y < z`. Instead the *binary* comparisons are grouped left to right so that `2 < 1 < 3` becomes `(2 < 1) < 3 == int(false) < 3 == 0 < 3 == true` and `1 < 2 < 0` becomes `(1 < 2) < 0 == int(true) < 0 == 1 < 0 == false`.
- 7(c) The two nested `?:` operators are logically equivalent to the following.
- ```
if (x > 1.0) // evaluates to true
    if (x < 2.0) // evaluates to false
        n = -1;
    else
        n = 0; // this branch executed
else
    n = -2;
```
- 8(c) There are missing `break` statements following each `case` in the `switch`. Regardless of the value of `c`, the last assignment is always `c = 'D'`.
- 9(a) The indentation notwithstanding, the `else` statement is attached to the second `if`.
- 10(b) Since both `1` and `n` are integers, the division `1/n` evaluates to 1 when `n == 1` and 0 otherwise. To sum the series as intended, you'd have to write `1.0/n`.
- 11(d) The function `poly2` used before it is declared.
- 12(c) C strings are terminated by an implied sentinel character, `'\0'`.
- 13(c) In the comparison slot of the `for` statement, `*p` is nonzero (and thus cast to `true`) so long as `p` does not point to the sentinel at the end of `"This is a test"`. Every letter that is not a space is advanced by one.
- 14(d) C arrays use zero indexing: the elements of `odds` are `odds[0], ..., odds[4]`. This code will compile but trigger an error when executed.
- 15(d) There is a syntax error in the function definitions: `= x*x;` should read `{ return x*x; }`.

- 16(a) Integer division by two rounds down. Since the constants `a` and `b` are `unsigned`, their difference wraps around to a large positive value.
- 17(c) The loop prints every second letter of `text`. The precedence rules dictate that `*p++` is interpreted as `*(p++)` rather than `(*p)++`. (The latter case would trigger an error since `char* p` is flagged as `const`.)
- 18(b) Division of the integers 2 and 3 gives zero.
- 19(a) The dot product is correctly calculated. Note that `--N` in `dot_prod` causes `N` to be decreased to 3 *before* it's used.
- 20(a) The program terminates at `assert(false);`.
- 21(b) The elements `a[1]`, `a[2]`, and `a[3]` are doubled. Changes propagate because arrays are always passed as a pointer.
- 22(d) The `unsigned` type will never be negative.