1. No, they are not contradictory. When there are consecutive identical operators within an expression, associativity determines which subexpressions are arguments of which operators. It does not determine the order in which those subexpressions are evaluated. For example, left associativity for subtraction determines that \( f(a) - g(b) - h(c) \) groups as \( (f(a) - g(b)) - h(c) \) (rather than \( f(a) - (g(b) - h(c)) \)), but it does not determine whether \( f \) or \( g \) is called first.

2. Let \( \sim \) represent unary negation (yes, it’s needed).
Postfix: \( \sim b \ b \ * \ 4 \ a \ * \ c \ * \ - \ \sqrt{} \ + \ 2 \ a \ * \ / \).
Prefix: \( / \ + \ \sim b \ \sqrt{} \ - \ * \ b \ b \ * \ * \ 4 \ a \ c \ * \ 2 \ a \).

3. Without parentheses, would \(- 2 \ 3 \ * \ 4 \ 5 \ 6\) evaluate to \((- 2 \ 3 \ (* \ 4 \ 5 \ 6)) = -27\) or to \((- 2 \ 3 \ (* \ 4 \ 5 \ 6)) = -121\)?
More accurately, issues of precedence and associativity do not arise with prefix or postfix notation in which each operator takes a fixed number of operands.

4.
```c
first_zero_row = -1;
for (i = 0; i < n && first_zero_row == -1; i++) {
    first_zero_row = i;
    for (j = 0; j < n; j++) {
        if (A[i][j]) {  // * not C */
            first_zero_row = -1;
            break;
        }
    }
}
```
If C allowed us to use a continue statement to jump to the next iteration of a specific, named loop, we could write the following:
```c
first_zero_row = -1; /* none */
outer: for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        if (A[i][j]) continue outer; /* not C */
    }
    first_zero_row = i;
    break;
}
```
5.

\[ l := 1 \]
\[ h := n + 1 \]
while \( l < h \)
\[ m = \text{floor} \left( \frac{(l+h)}{2} \right) \]
if \( t = X[m] \)
exit
if \( t < X[m] \)
\[ h := m \]
else
\[ l := m + 1 \]

6. No. The macro must either call a subroutine or execute a loop. It must also take
the form of an expression in order to return a value. C provides no way for an expression
to contain a loop. Many C implementations, however (including gcc) extend the language
to include “statement expressions” that eliminate this restriction. The following macro works
with gcc:

```c
#define GCD(a, b)        
{ while ((a) != (b)) {      
  if ((a) > (b)) (a) = (a) - (b);  
  else (b) = (b) - (a);   
}        
(a);      
}
```