For the next five questions, consider the following implementation of a function to find the maximum value in an array of integers. The for loop is required to be implemented using pointers to access array elements rather than direct array indexing.

```c
// Returns largest value in array Data.
// Pre: Data[0:Sz-1] have been initialized.
// Returns: maximum( Data[0:Sz-1] )
int maxEntry(const int* const Data, const int Sz) {          // Line 1
    if ( Data == NULL || Sz <= 0 ) return INT_MIN;            //      2
    // Set hiSoFar to point to the first array element:
    const int *hiSoFar = ___________;                         //      3
    // Set Current to point to the second array element:
    const int *Current = ___________;                         //      4
    for ( int Count = 1; Count < Sz; __________________) {    //      5
        if ( ___________________ )                             //      6
            hiSoFar = Current;                                  //      7
    }                                                       //      8
    return ( ____________ );                                  //      9
}
```

1. How should the blank in Line 3 be filled?
   1) Data
   2) *Data
   3) &Data
   4) &Data[0]
   5) Data[0]
   6) 1 or 4 only
   7) 2 or 4 only
   8) 3 or 4 only
   9) 3 or 5 only
   10) None of these

2. How should the blank in Line 4 be filled?
   1) hiSoFar
   2) hiSoFar++
   3) Data++
   4) &Data[1]
   5) Data[1]
   6) 2 or 4 only
   7) 2 or 5 only
   8) None of these

3. How should the blank in Line 5 be filled?
   1) Count++
   2) Current++
   3) Count++, Current++
   4) It should be left blank.
   5) None of these

4. How should the blank in Line 6 be filled?
   1) Current > hiSoFar
   2) &Current > &hiSoFar
   3) *Current > *hiSoFar
   4) *Current < *hiSoFar
   5) None of these
5. How should the blank in Line 8 be filled?

1) *hiSoFar
2) &hiSoFar
3) hiSoFar
4) It should be left blank.
5) None of these

6. Consider implementing a function to dynamically allocate an array of integers and set all its elements to zero:

```c
// Allocates array of zeroes of dimension Size and sets target of A
// to point to it.
// Pre: Size > 0
// Post: *A points to an array of zeroes of dimension Size

void ZeroIt( __________ A, const int Size ) {
    int* C = malloc( Size * sizeof(int));
    for (int Idx = 0; Idx < Size; Idx++) {
        C[Idx] = 0;
    }
    *A = C;
}
```

Which of the following choices for the blank preceding the formal parameter \texttt{A} is best?

1) int*
2) int**
3) const int*
4) const int**
5) const int* const
6) const int** const
7) None of these

7. Assuming all the allocations are granted, what logical error(s) occur in the following code fragment?

```c
int *p1 = malloc(sizeof(int));  // Line 1
*p1 = 1;  // 2
int *p2 = malloc(sizeof(int));  // 3
*p2 = 2;  // 4
p2 = p1;  // 5
free(p1);  // 6
```

1) A dangling pointer is created in line 5.
2) A dangling pointer is created in line 6.
3) A memory leak occurs in line 5.
4) A memory leak occurs in line 6.
5) All of them
6) 1 and 3 only
7) 1 and 4 only
8) 2 and 3 only
9) 2 and 4 only
10) None of these
Recall that in C, a string is represented by a `char` array of ASCII codes, with a 0-byte marking the logical end of the string.

In many applications we need an array of C-strings, and that is achieved by using an array of `char*` (char pointers). For example:

```c
char* Words[5] = {0}; // array of char pointers; initially all are NULL
char W0[] = "pointers"; // some C-strings
char W1[] = "are";
char W2[] = "good";
char W3[] = "for";
char W4[] = "you";

Words[0] = W0; // make the array cells point to the strings
Words[1] = W1;
Words[2] = W2;
```

The code above would create the following logical structure:

```
   Words
      |   'p'   'o'   'i'   'n'   't'   'e'   'r'   's'   '\0'
      |   'a'   'r'   'e'   '\0'
      |   'g'   'o'   'd'   '\0'
      |   'f'   'o'   'r'   '\0'
      |   'y'   'o'   'u'   '\0'
```

Recall that an array name is a pointer to the 0-th cell of the array, so the pointer type is actually `pointer-to-whatever-kind-of-thing-is-in-the-array`.

In the situation above, `Words[0]` is a `char*`, and so `Words` is a `char**`... a pointer to a pointer to a `char`.

Understanding all of this will help with the following questions...
For the next three questions, consider the function to search an array of C-strings for matches to a given C-string.

```c
// Counts the number of times the string toMatch occurs in the array A.
// Pre:
//    A points to an array of Sz char pointers, each of which points
//    to a properly initialized C-string (0-terminated char array).
//    toMatch points to a properly initialized C-string
// Post:
// Returns:  number of pointers in A that point to C-strings that equal
//           the target of toMatch
//
int numMatches(const char** const A, const int Sz, const char* toMatch) {  // Line 1
    if ( A == NULL || Sz <= 0 ) return 0;                             //      2
    int Count = 0;                                                    //      3
    for (int Pos = 0; Pos < Sz; Pos++) {                              //      4
            Count++;                                                    //      6
        }
    }
    return Count;                                                     //      7
}
```

8. The parameter A is declared as: `const char** const A`. What is the effect of the first `const`?

1) The function is not allowed to modify the value of the pointer A.
2) The function is not allowed to modify the value of the contents of the array to which A points.
3) Both 1 and 2
4) None of these

9. The first part of the Boolean expression in the `if` statement in line 5 is: `A[Pos] != NULL`. Which of the following assertions are true?

1) This depends on the assumption that each cell of A[] either points to a C-string or is set to NULL.
2) This is intended to prevent runtime errors if some cell of A[] does not point to a C-string.
3) Both 1 and 2
4) None of these

10. Does the order of the two parts of the AND-expression in Line 5 matter?

1) No.
2) Yes, if the order is reversed then valid matches may not be detected.
3) Yes, if the order is reversed then matches may be reported when none occurred.
4) Yes, if the order is reversed then access violations could occur if the array did not contain the expected number of pointers that do have targets.
5) None of these