This homework assignment focuses primarily on some of the more complex concepts in C, including pointers, arrays, and dynamic memory allocation, as well as some number representation. The answers to the following questions can be determined by consulting a C language reference, writing short C programs to test your logic, and/or sitting down with a pen and paper. For the purpose of this assignment, we will stipulate that answers are expected to be consistent with the behavior of the `gcc` C compiler, used with `-std=c99`. The given code does compile, possibly with warnings.

After you have analyzed the questions and decided what answers you believe are correct, you may find it useful to write some short programs to test your logic. Submit your answers to the Curator quiz collection point `OQ02`.

For questions 1 through 4, select the datatype of the given C variable given the following code:

```
int     a = 2, b = 5;
int*    p1 = &a, p2 = NULL;
```

1. `p1`

1) **`int**`**
2) **`int*`**
3) **`int`**
4) None of these

2. `*p1`

1) **`int**`**
2) **`int*`**
3) **`int`**
4) None of these

3. `&p1`

1) **`int**`**
2) **`int*`**
3) **`int`**
4) None of these

4. `p2`

1) **`int**`**
2) **`int*`**
3) **`int`**
4) None of these

For questions 5 through 7, consider the following code:

```
1   char  myString[] = "CS2505 is so much fun\n";
2   char* partway = &myString[13];
3   partway -= 3;
4   printf("%s\n",partway);
```

5. What is the dimension of `myString`?

1) `20`
2) `21`
3) `22`
4) `23`
5) None of these

6. What is the error on line 2?

1) No `&` operator necessary
2) `partway` should be `char**`
3) Can't reference an array index like this
4) There is no [13]
5) There is no error

7. After line 2 has been corrected (if necessary), what is the output of line 4?

1) Another error will still prevent compilation
2) **`"so much fun\n"`**
3) **`" so much fun\n"`**
4) **`"o much fun\n"`**
5) **`%s`** is the wrong format specifier

For questions 8-15, consider the following questions regarding numeric representation.

8. What is the binary representation of the decimal number `12345`?

1) `11000000111001`
2) `11000001111001`
3) `11100000111001`
4) `11100001111001`
5) None of these

9. What is the hexadecimal representation of the binary number `110010100101010`?

1) `CA52`
2) `DB52`
3) `652A`
4) `652B`
5) None of these

10. What is the smallest (most negative) value that can be saved in an 128-bit signed integer?

1) `-2^128-1`
2) `-2^127-1`
3) `-2^(128-1)`
4) `-2^(127-1)`
5) None of these

11. What is the largest (most positive) value that can be saved in an 128-bit signed integer?

1) `2^128-1`
2) `2^127-1`
3) `2^(128-1)`
4) `2^(127-1)`
5) None of these

12. What is the 8-bit signed binary representation of the signed decimal number 123?

1) `01011011`
2) `11011011`
3) `01111011`
4) `11111011`
5) None of these

13. What is the 8-bit signed binary representation of the signed decimal number -65?

1) `10111111`
2) `11000001`
3) `00111111`
4) `01000001`
5) None of these

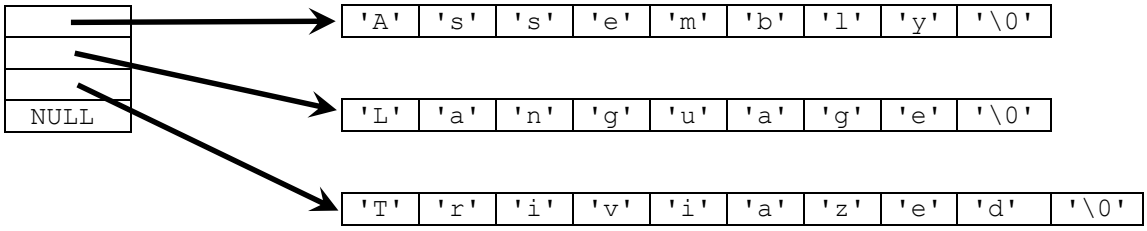
14. What is the sum of the 8-bit signed binary values 01101010 and 10010101?

1) `11111111`
2) `11011111`
3) `10000001`
4) The sum will overflow
5) None of these

15. What is the sum of the 8-bit unsigned binary values 01101010 and 10010101?

1) `11111111`
2) `11011111`
3) `10000001`
4) The sum will overflow
5) None of these

The function below takes a C-string `source` as input, and creates an array of C-strings corresponding to substrings of `source`, using space characters as separators. For example, if `source` is `"Assembly Language Trivialized"`, the function creates the following structure:



The array on the left uses a `NULL` in the first empty cell to indicate the end of the substrings. The implementation shown below omits some important error-checking in order to fit on the page. The allocation of the array, in line 3, is sufficient to guarantee enough space for the worst case; we accept that it may waste memory.

```
char** split(const char* const source) {                          //  1

   int sourceLength = strlen(source);                             //  2

   char** pieces = calloc(sourceLength / 2 + 2, sizeof(char*));   //  3
   int numPieces = 0;                                             //  4

   int pos = 0;                                                   //  5
   while ( pos < sourceLength ) {                                 //  6

      // find start of current piece
      while ( pos < sourceLength && source[pos] == ' ' )          //  7
         pos++;                                                   //  8
      if ( pos == sourceLength ) break;                           //  9
      int start = pos;                                            // 10

      // find end of current piece
      while ( pos < sourceLength && source[pos] != ' ' )          // 11
         pos++;                                                   // 12
      int end = pos - 1;                                          // 13

      // create holder for current piece
      int pieceLength = _________________ ;                       // 14
      char* currPiece = calloc(pieceLength + 1, sizeof(char));    // 15

      // copy current piece
      memcpy(currPiece, &source[start], pieceLength);             // 16

      // store piece
      pieces[numPieces] = currPiece;                              // 17

      // count current piece
      numPieces++;                                                // 18
   }

   return pieces;                                                 // 19
}
```

16. Line 14 is supposed to calculate the length of the current substring. How should the blank there be filled?

1) `end`
2) `end - start`
3) `end - start + 1`
4) `pos - start`
5) None of these

17. Which statement guarantees that each of the substrings of `source` will have a terminator after the last character of the substring?

1) Line 3
2) Line 15
3) Line 16
4) There is no guarantee of a terminator
5) None of these

The structures created by `split()` need to be deallocated. The following function is supposed to accomplish the deallocation of such a structure, assuming `list` points to the array of pointers:

```
void dealloc(char** list) {          //  1

   int pos = 0;                      //  2

   while ( ___________________ ) {   //  3

      _______________;               //  4

      pos++;                         //  5
   }

   _______________;                  //  6
}
```

18. How should the blank in line 3 be completed?

1) `pos < strlen(list)`
2) `pos < strlen(list[0])`
3) `list[pos] != '\0'`
4) `list[pos] != NULL`
5) None of these

19. How should the blank in line 4 be completed?

1) `free( list )`
2) `free( list[pos] )`
3) `free( pos )`
4) `free( *list )`
5) `list[pos] = NULL`
6) None of these

20. How should the blank in line 6 be completed?

1) It should be left blank
2) `list = NULL`
3) `*list = NULL`
4) `free( list )`
5) `free( *list )`
6) None of these