1. Consider the struct declaration and code shown at right below intended for setting and displaying employee information for a single employee. Which of the following best describes the correctness of this code segment?

   a) The code will compile and run correctly as shown
   b) The code will not compile and has one error
   c) The code will not compile and has two errors
   d) The code will not compile and has three or more errors

   ```cpp
   struct Employee {
       int id;
       float hourlyWage;
       char code;
   };

   Employee anEmployee;
   Employee *pEmployee = &anEmployee;
   anEmployee.id = 3.1415; // should be int
   anEmployee.hourlyWage = 8.0; // should be 'D'
   (*pEmployee).code = 'D';
   cout << anEmployee->code; // should be '
   cout << pEmployee.id; // should be ""
   ```

2. Consider a large array of Employees, using the struct declaration from the previous problem. Which of the following is the best implementation so that we can both access the employee information in order according to the id field, and in addition also access employee information in order according to the code field?

   a) When accessing by id, ensure the array is first sorted by id. Do this similarly for code
   b) Create duplicate copies of the array, where one is always kept sorted by id, and the other is always kept sorted by code
   c) Keep the array sorted by id, and access it sequentially when searching by code
   d) Have separate arrays of pointers for id and for code, with the pointers implementing ordering of what they point to
3. Consider the struct declaration (shown at right) used to implement a list which has a *sentinel node*, so there is always at least that one node on the list. Consider the following function declarations used to prepend nodes at the beginning of the list:

I. `void prepend(int value, Node **pHead)`
II. `Node * prepend(int value, Node *pHead)`
III. `void prepend(int value, Node * &pHead)`

How many of the above functions could be used to prepend nodes at the beginning of the list?

a) None  
b) One  
c) Two  
d) Three

4. Which of the following is the best description of how long it will take to sort elements of an array using the *bubble sort* algorithm as described in class?

a) It always takes the same amount of time  
b) It is proportional to the square of the number of elements  
c) It depends on the original order of the elements  
d) It depends on both the number of elements and the initial order of the elements

5. Consider a situation where items are already mostly sorted. Which of the following sort algorithm(s) would be preferable?

a) Bubble Sort  
b) Insertion Sort  
c) Selection Sort  
d) Insertion and Selection Sort are both equally good

6. Consider using an array to store id numbers for a class of 100 students, where the array index is the student id number. When searching for a particular id number, on average how many id numbers will need to be examined? Choose the closest number if the exact number is not shown.

a) 1  
b) 10  
c) 20  
d) 50

7. Consider storing student last names alphabetically in an array for a class of 100 students. When searching for a particular last name using binary search, how many last names will need to be examined before we are guaranteed to find the one we are looking for? Choose the closest number if the exact number is not shown.

a) 2  
b) 7  
c) 50  
d) 64
8. Consider the function shown at right below. What output is displayed when it is called using:

\[
\text{problem8( 12358, 1, } x) ;
\]

\[\text{a) 1} \]
\[\text{b) 2} \]
\[\text{c) 3} \]
\[\text{d) 8} \]

9. Consider the following declaration of a function intended to grow an array of integers by dynamically allocating memory:

\[
\text{void growArray( int oldSize, int newSize, int** pTheArray)}
\]

What is the best description of whether or not this function declaration will work?

\[\text{a) The function will not compile correctly} \]
\[\text{b) The function could compile correctly, however will crash when running} \]
\[\text{c) The function could compile correctly and run, however the calling code will not have access to the new larger array.} \]
\[\text{d) The function could compile correctly and run correctly} \]

10. Consider the following declaration of a function intended to grow an array of integers by dynamically allocating memory:

\[
\text{void growArray( int oldSize, int newSize, int* &pTheArray)}
\]

What is the best description of whether or not this function declaration will work?

\[\text{a) The function will not compile correctly} \]
\[\text{b) The function could compile correctly, however will crash when running} \]
\[\text{c) The function could compile correctly and run, however the calling code will not have access to the new larger array.} \]
\[\text{d) The function could compile correctly and run correctly} \]

11. Consider reading in product labels from a file into an array called \textit{allWords}, where you know ahead of time that each label is always 10 or more characters long, however you don’t know how many labels there will be. The declaration for this array will likely be:

\[\text{a) char allWords[ 10][ 10];} \]
\[\text{b) char allWords[ n][ 10];} \]
\[\text{c) char *allWords[ 10];} \]
\[\text{d) char **allWords;} \]
12. Consider the following function declaration where a 2-d array has been passed:

```c
int countOnes( int numbers[ ][ 100])
```

In the above function declaration the value in the first set of braces may be left blank as shown, but the second dimension must be supplied. Why is this?

a) Code in C can overwrite the end of an array, however for a 2-D array the formula used to find the i\textsuperscript{th} row needs to know how many columns there are in each row.

b) The size of the first dimension is always automatically supplied, even when not specified by the user, since it is included as part of the definition of every array.

c) A NULL character is always inserted at the end of every row so that the compiler can tell where one row ends and the next begins, so the size of the first dimension is not necessary.

d) The total size of the array divided by the size given in the second dimension is used internally to calculate the number for the first dimension, so the user need not supply it.

13. Consider the two approaches shown below to implement a stack with push() and pop() operations:

<table>
<thead>
<tr>
<th>Approach A:</th>
<th>Approach B:</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Stack Diagram" /></td>
<td><img src="image" alt="Stack Diagram" /></td>
</tr>
</tbody>
</table>

Which is the best description of which of the two above approaches is necessary?

a) Approach A is all you need to implement a stack

b) Approach B is preferable because both pTop and pBottom are needed for implementing a stack

c) Neither one is preferable, they are both equally well suited for implementing a stack

d) It depends on what kinds of stack operations will be implemented

14. Consider the two front-and-back pointer approaches shown below to implement a queue where nodes are added using pBack and nodes are removed using pFront:

<table>
<thead>
<tr>
<th>Approach A:</th>
<th>Approach B:</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Queue Diagram" /></td>
<td><img src="image" alt="Queue Diagram" /></td>
</tr>
</tbody>
</table>

Which is the best description of which of these two approaches is preferable?

a) Approach A is preferable for implementing a queue

b) Approach B is preferable for implementing a queue

c) Neither one is preferable, they are both equally well suited for implementing a queue

d) It depends on what kinds of queue operations will be implemented
15. Consider the function shown at right below used to traverse and display the values on a linked list. What is the best description of this function when called with `pHead` pointing to a non-empty list?

a) It works correctly  

b) It traverses and displays the list, however it destroys the list head pointer in the process  

c) It does not compile  

d) It compiles but gives a run-time error

```cpp
void displayList( Node *pHead)
{
    while( pHead != NULL) {
        cout << pHead->data << " ";
        pHead = pHead->pNext;
    }
    cout << "\n\n";
}
```

16. Consider the struct declaration at left below that is used to implement a doubly-linked list. If there are *multiple nodes* on the list and we want to insert a new node *before* the current node in the middle of the list, which of the following is true about the `insertNode(...)` function?

```cpp
struct Node2 {
    Node2* pPrev;
    int data;
    Node2* pNext;
};

void insertNode( Node2 * &pCurrent, int input)
{  Node2 *pTemp = new Node2;
    pTemp->data = input;
    pTemp->pNext->pPrev->pNext = pTemp;
    pTemp->pNext = pCurrent;
    pTemp->pNext->pPrev = pTemp;
}
```

a) It could be fixed by reordering the current instructions.  

b) To fix it would require modifying one instruction.  

c) To fix it would require adding an additional instruction.  

d) To fix it would require modifying or adding more than one instruction.

17. Consider the section of code shown at right below. What is displayed in the output?

a) 2  

b) the address in memory of `xPtr`  

c) the address in memory of `xPtr`  

d) the address in memory of `zPtr`  

```cpp
int x, y, z;  
int *xPtr, *yPtr, **zPtr;  
x = 2; y = 7; z = 9;  
xPtr = &x;  
yPtr = &y;  
zPtr = &xPtr;  
cout << *zPtr;
```


<table>
<thead>
<tr>
<th>variable</th>
<th>Memory content</th>
<th>Make-up Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2</td>
<td>204A</td>
</tr>
<tr>
<td>y</td>
<td>7</td>
<td>2048</td>
</tr>
<tr>
<td>z</td>
<td>9</td>
<td>2046</td>
</tr>
<tr>
<td>xPtr</td>
<td>204A</td>
<td>2046</td>
</tr>
<tr>
<td>yPtr</td>
<td>2048</td>
<td>2044</td>
</tr>
<tr>
<td>zPtr</td>
<td>2044</td>
<td>2040</td>
</tr>
</tbody>
</table>
Consider a maze program segment shown at right. It is similar, though not exactly the same, as what we discussed in class. Note that there are a total of 20 array values of zero. This function will be called as:

```
makeMove( start);
```

18. What is the output from running this code?

a) A solution path in reverse order, displayed once.

b) A solution path in order, along with numbers of extra squares visited along the way.

c) A solution path in reverse order, along with numbers of extra squares visited along the way.

d) It goes into an infinite loop

19. If the values of start and goal are swapped, how many unique array positions containing 0 are visited by this algorithm, including the start and goal squares?

a) 10

b) 15

c) 16

d) It goes into an infinite loop

20. If the start and goal values remain as originally shown, however the moves array initial values are set to {10,1,-1,-10}, How many unique array positions containing 0 are visited by this algorithm, including the start and goal squares?

a) 10

b) 11

c) 14

d) 15

e) It goes into an infinite loop
Consider the two versions of function `add(...)` shown below:

**Option A:**

```c
void add( int newNumber,
          int *&pArray,
          int &size)
{
    int *pNewArray= new int[size+1];
    for( int i=0; i<size; i++) {
        pNewArray[ i] = pArray[ i];
    }
    pNewArray[ size] = newNumber;
    delete( pArray);
    size++;    
    pArray = pNewArray;
}
```

**Option B:**

```c
void add( int newNumber,
          int * *pArray,
          int *size)
{
    int *pNewArray=new int[*size +1];
    for( int i=0; i<*size; i++) {
        pNewArray[ i] = (*pArray)[ i];
    }
    pNewArray[ *size] = newNumber;
    delete( *pArray);
    (*size)++;    
    *pArray = pNewArray;
}
```

21. Given the following code that could be used to call one of the functions above to grow the array:

```c
int *pNumbers = new int[ 3];
pNumbers[ 0] = 1; pNumbers[ 1] = 3; pNumbers[ 2] = 5;
int size = 3;
add( 7, pNumbers, size);
```

which of the following are true about the use of code to call Option A and B above?

a) Both A and B will compile and run with this code
b) A will not work, but B will compile and run with this code

c) A will compile and run, but B will not work

d) Neither A nor B will compile and run with this code

22. What is the output from the following code when calling function `pointerIncrementProblem()`?

```c
void foo( int *pArray)
{
    for( int i=0; i<8; i++) {
        (*pArray)++;    
        *(pArray++);
    }
}

void pointerIncrementProblem()
{
    int size = 8;
    int theArray[ ] = {1,3,5,7,9,11,13,15};
    foo( theArray);
    for( int i=0; i<size; i++) {
        printf("%3d", theArray[ i]);
    }
}
```
23. Carefully consider the C/C++ program segment given below, called with:  

```c++
void do23() {
    int number = 0;
    Node *pHead = NULL;
    Node *pTemp;
    cout << "Enter numbers, then -1: ";
    while (number != -1) {
        cin >> number;
        if (number != -1) {
            pTemp = new Node;
            pTemp->data = number;
            pTemp->pNext = pHead;
            pHead = pTemp;
        }
    }
    pTemp = pHead;
    pHead = fcn23( pHead);
    fcn23a( pHead);
}
```

If the input is:

```
23 7 62 4 17 -1
```

then what is the output?

a) The reverse of the input, excluding -1
b) The same as the input, excluding -1
c) The same as the input, including -1
d) It compiles and runs, but goes into an infinite loop

```
struct Node {
    int data;
    Node *pNext;
};

void fcn23a( Node *pHead)
{
    while( pHead != NULL) {
        cout << pHead->data << " ";
        pHead = pHead->pNext;
    }
}

Node * fcn23( Node *pHead)
{
    Node *pTemp;
    if (pHead->pNext == NULL)  {
        return pHead;
    }
    else {
        pTemp = fcn23(pHead->pNext);
        pHead->pNext->pNext = pHead;
        return pTemp;
    }
}
```