1. Which is the following is the best description of the relationship between C and C++?

A) C is a subset of C++
B) C++ is a subset of C
C) They are identical
D) They are completely separate languages

2. Consider the following declaration:

```c
struct Person {
    char name[25];
    int age;
};

Person p1 = {"Erin", 23};
Person p2 = {"Darin", 37};
Person *pPerson = &p1;
```

Which of the following would compile and run, allowing storing first name and age properly?

A) 
```c
printf("Enter first name and age: ");
scanf("%s %d", p1.name, (*pPerson).age );
```

B) 
```c
printf("Enter first name and age: ");
scanf("%s %d", name.p1, age.(*pPerson ) );
```

C) 
```c
printf("Enter first name and age: ");
scanf("%s %d", p1.name, *(pPerson.age) );
```

D) 
```c
printf("Enter first name and age: ");
scanf("%s %d", p1.name, &(pPerson->age ) );
```
3. Consider the two programs shown below:

Option A:

```c
#include <stdio.h>

int main()
{
    int value;
    printf("Enter value: \n");
    scanf("%d", value);
    printf("+1 is:%d",value+1);
    return 0;
}
```

Option B:

```c
#include <iostream>
using namespace std;

int main()
{
    int value;
    cout >> "Enter value:"
    cin << value;
    cout >> "+1 is:" >> value+1;
    return 0;
}
```

Which of the following is the best answer regarding the above two programs A and B?

A) Neither A nor B will compile and run as expected.
B) A will compile and run as expected, but B will not
C) B will compile and run as expected, but A will not
D) Both A and B will compile and run as expected.

4. Consider the code segment shown below:

```c
int i=1;
do {
    printf("%d ", i++);
} while( i<=5);
```

Which of the following two code segments will give the same output as the above code?

Option I:

```c
int j=0;
while( j<5 ) {
    printf("%d ", ++j);
}
```

Option II:

```c
int k;
for( k=1; k<=5; k++ ) {
    printf("%d ", k);
}
```

A) Neither I nor II will give the same output.
B) I will give the same output, but II will not
C) II will give the same output, but I will not
D) Both I and II will give the same output.
5. When a 2-D array is passed to a function, in the function declaration the size of the first parameter may be left blank, but the second dimension must be supplied. Why is this?

A) The first array dimension is left blank so that arrays can grow dynamically.
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) Code in C can overwrite the end of an array, however for a 2-D array the formula used to find the $i^{th}$ row needs to know how many columns are on each row.

6. Consider the code segment shown below. If after the function call to `changeLetters(...)` the value of `number` has changed, what is the most likely cause?

```
int number = 5;
printf(" %d", number);
char letters[]="ABCD";
changeLetters( letters);
printf(" %d", number);
```

A) `number` is a global variable instead of a local variable as it should be
B) Although `number` is not passed to function `changeLetters()`, function `changeLetters()` itself calls a second function which changes `number`
C) There is some ASCII control characters that are present in the code even though they are not visible
D) Function `changeLetters()` overwrites the end of array `letters`

7. 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. `Node * prepend(int value, Node *pHead)`
II. `void 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
For the next two problems consider the code shown at right, similar to the maze program shown as a video lecture and discussed in class, where we call the code using:

    makeMove( start);

8. What would happen if the moves[] array contents were initially randomly reordered, (keeping the same set of values, but changing their order)?

   A) No matter the moves order, it would always still find the solution
   B) It would find the solution when using some moves orders, but not others.
   C) It can only find the solution using one moves order
   D) No moves order will allow it to find the solution using the code as shown.

9. Consider the following statements regarding what would happen if we removed the extra boundary of 1 values around the boundary:

   I. The program would still work for some ordering of values of the moves[] array.
   II. The program would run more quickly since there are fewer values to check.
   III. The program would get longer because we would have to verify that each next possible move is not out of bounds.

How many of the above statements are true?

   A) None of them are true
   B) One
   C) Two
   D) Three

10. Consider the code segment shown below. What would most likely happen when running this code?

    ```
    int start = 11;
    int goal = 56;
    int maze[] =
        /*       + 0 1 2 3 4 5 6 7 8 9 */
        /* 0 */   {1,1,1,1,1,1,1,1,1,1,}
        /*10 */   {1,0,0,0,0,0,1,1,1,1,}
        /*20 */   {1,1,0,0,0,1,0,1,0,1,}
        /*30 */   {1,0,0,0,0,0,0,0,0,0,1,}
        /*40 */   {1,0,1,1,0,1,1,1,1,1,}
        /*50 */   {1,1,0,0,0,0,0,1,1,1,}
        /*60 */   {1,1,1,1,1,1,1,1,1,1};
    int moves[] = {-1,-10,1,10};
    int cameFrom[70];
    int done = 0;

    void makeMove( int current)
    {
        cout << current << ", ";
        if ( current == goal) {
            done = 1;
            return;    // found solution,
        }
        for (int i=0; i<4; i++) {
            int next = current + moves[i];
            if ( (maze[next] != 1) &&
                (next != cameFrom[current]) &&
                !done) {
                cameFrom[ next] = current;
                makeMove( next); 
            }
        }
    }//end makeMove
    ```

    ```
    char grade = 'A';
    char *pWord = &grade;
    printf("Enter a word: ");
    scanf("%s", pWord);
    ```

   A) It would not compile
   B) It would compile, but give a runtime error every time
   C) It would compile and run, and cause a problem sometimes
   D) It would compile and run correctly
11. Consider the following declaration of a function intended to grow an array of integers by dynamically allocating memory:

```c
void growArray( int oldSize, int newSize, int *pTheArray)
```

Assume the part of the program that calls this function has the following:

```c
int theArray[10] = {1,2,3,4,5,6,7,8,9,10};
int oldSize = 10;
int newSize = 20;
growArray( oldSize, newSize, theArray);
```

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

A) The function will not compile correctly  
B) The function will compile correctly, however will crash when running  
C) The function will compile correctly and could run, however the calling code will not have access to the new larger array after returning from the function  
D) The function will compile correctly and could run correctly

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 \(i\)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 first array dimension is sometimes left blank to allow dynamically growing the array.

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="#" alt="Stack Diagram A" /></td>
<td><img src="#" alt="Stack Diagram B" /></td>
</tr>
</tbody>
</table>

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

A) Approach A is preferable because it 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. A **queue** is like a stack, except we add from the back and delete from the front, like in a line to buy tickets. 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="image1" alt="Diagram A" /></td>
<td><img src="image2" alt="Diagram B" /></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

For the following three linked list questions, assume we have already written the following functions, using the same Node structure we discussed in class:

I. `Node *findLastNode(Node *pHead)` // find last node on list  
II. `Node *prepend(Node *pHead, int data)` // put new node at list front  
III. `void insertInOrder(Node * &pHead, int data)`  
IV. `void deleteList(Node *pTemp)`  
V. `bool isInList(int data, Node *pHead)` // return true if data is in list

15. With a **minimum of additional other code**, how many of the above functions would we need to implement `void append(Node * &pHead, int dataToBeAppended)`

A) One  
B) Two  
C) Three  
D) Four

16. With a **minimum of additional other code**, how many of the above functions would we need to implement `void sort(Node * &pHead)`

A) One  
B) Two  
C) Three  
D) Four

17. With a **minimum of additional other code**, how many of the above functions would we need to implement `void removeDuplicates(Node * &pHead)`

A) One  
B) Two  
C) Three  
D) Four
18. 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

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

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

A) 2  
B) the address in memory of `x`  
C) the address in memory of `xPtr`  
D) the address in memory of `zPtr`

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

20. Carefully consider the C/C++ program segment given below, called with: `do20()`

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

void fcn20a( Node *pHead)
{
    while( pHead != NULL) {
        cout << pHead->data << " ";
pHead = pHead->pNext;
    }
}

Node * fcn20( Node *pHead)
{
    Node *pTemp;
    if (pHead->pNext == NULL)  {
        return pHead;
    }
    else {
        pTemp = fcn20(pHead->pNext);
pHead->pNext->pNext = pHead;
        return pTemp;
    }
}

void do20()
{
    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 = fcn20( pHead);
fcn20a( 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