1. (3 pts) In order to determine the correctness of a 10,000 line C++ program, you should (circle the letter of the best answer):

(a) Prove the correctness of the program by finding an invariant.
(b) Test the program on a several sets of test data.
(c) Remove all compile errors, and the program should be correct.

Answer: b

2. (12 pts) Under the “Meaning” column, write the words the acronym stands for, and under the “Match” column, write the letter of the corresponding acronym (use each letter once).

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Meaning</th>
<th>Match</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. LIFO</td>
<td>Last In First Out</td>
<td>c</td>
<td>List</td>
</tr>
<tr>
<td>b. FIFO</td>
<td>First In First Out</td>
<td>b</td>
<td>Queue</td>
</tr>
<tr>
<td>c. ADT</td>
<td>Abstract Data Type</td>
<td>a</td>
<td>Stack</td>
</tr>
</tbody>
</table>

3. (3 pts) In general, the coding phase of the life cycle of software is a longer period of time than the maintenance phase. (TRUE or FALSE?)

FALSE

4. (8 pts) Under the Match column, write the letter of the Method that best describes the format of the Expression (you may use letters more than once).

<table>
<thead>
<tr>
<th>Method</th>
<th>Match</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. prefix</td>
<td>b</td>
<td>468 * +</td>
</tr>
<tr>
<td>b. postfix</td>
<td>d</td>
<td>4 + 6 * 8</td>
</tr>
<tr>
<td>c. infix</td>
<td>c</td>
<td>4 + 6 * 8</td>
</tr>
<tr>
<td>d. none of the above</td>
<td>a</td>
<td>+4 * 68</td>
</tr>
</tbody>
</table>

5. (3 pts) Briefly explain how C++ allows a programmer to hide implementation details from a user.

The private section of classes hides details from the user.

Never. The destructor is automatically called when an object is out of scope.

7. (5 pts) What is the loop invariant of the following code segment? (Give the complete loop invariant). Assume n>0 and all variables are of type int.

```
i = 0; x = 0;
while (x < n*n) {
    x = x + n;
    i = i + 3;
}
```

0< x ≤ n^2, 0< i < 3n, x = n*i/3

8. (12 pts) See array A and the Move function on the first page of the Exam 1 Handout.

The Move function is an attempt to move through a maze looking for a pot of gold (G). What is the output and the tree of recursive calls for the following: (If Move is called more than 12 times, just show the output and tree up to the 12th call.)

```
if (Move(5,4))     // Starting position is indicated by 'o' in the array A
    cout << "Success! " << endl;
else
    cout << "Failure! " << endl;
```

Output:

row: 5 col: 4
row: 4 col: 4
row: 5 col: 5
row: 6 col: 4
row: 5 col: 4
row: 6 col: 5
row: 5 col: 5
row: 6 col: 6
row: 5 col: 6
Gold found!
Success!

Show the tree of recursive calls:
9. See the Exam 1 Handout for a description of a data structure to represent the number of votes for candidates in different voting regions.

(a) (5 pts) Write a function called NumRegions that has one argument that is a pointer to the data structure, and returns the number of regions in the data structure. (For example, in the picture on the Exam 1 Handout, there are 5 regions).

```c
int NumRegions(rnode * H)
{
    int cnt = 0;
    while (H != NULL) {
        cnt++;
        H = H->Next;
    }
    return cnt;
}
```

Alternative (recursive) solution:

```c
int NumRegions(rnode * H)
{
    if (H != NULL) {
        return NumRegions(H->Next)+1;
    }
    else
        return 0;
}
```

(b) (10 pts) Write a function called TotalVotes that takes two arguments, the name of a candidate and a pointer to the data structure and returns the total number of votes the candidate received (over all regions). For example, in the picture on the Exam 1 Handout, the candidate DT received $14+20+14=48$ votes.
int TotalVotes(String Name, struct rnode * H) {
    cnode * tmp;
    boolean done;
    int cnt = 0;
    while (H !=NULL) {
        tmp = H->List;
        done = FALSE;
        while (tmp !=NULL && (!done)) {
            if (tmp->Name == Name) {
                cnt += tmp->Votes;
                done = TRUE;
            }
        }
        tmp = tmp->Next;
    }
    H = H->Next;
}

    return cnt;
}

Alternative (recursive and modular) solution:

int FindVote(cnode * tmp, String Name) {
    if (tmp == NULL)
        return 0;
    else if (tmp->Name == Name)
        return tmp->Votes;
    else
        return FindVote(tmp->Next, Name);
}

int TotalVotes(String Name, struct rnode * H) {
    cnode * tmp;
    if (H !=NULL)
        return TotalVotes(Name, H->Next) + FindVote(H->List, Name);
    else
        return 0;
}
(c) (14 pts) Write a function called CombineRegions that takes three arguments (two region numbers and a pointer to the data structure) and combines the regions into one region with the same region number as the first argument. See the Exam 1 Handout for an example and a function you may use.

```c
void CombineRegions(rnode * & H, int reg1, int reg2)
{
    rnode * tmp1, * tmp2;
    cnode * c1, * c2, * tmp;
    int found;

    tmp1 = H; tmp2 = H;
    while (tmp1 != NULL && tmp1->Region != reg1)
        tmp1 = tmp1->Next;
    while (tmp2 != NULL && tmp2->Region != reg2)
        tmp2 = tmp2->Next;
    if (tmp2 == NULL)
        return;
    if (tmp1 == NULL) {
        tmp2->Region = reg1;
        return;
    }
    c2 = tmp2->List;
    while (c2 != NULL) {
        c1 = tmp1->List;
        found = 0;
        while (c1 != NULL) {
            if (c1->Name == c2->Name) {
                c1->Votes += c2->Votes;
                found = 1;
            }
        }
        c1 = c1->Next;
    }
    if (!found) {
        c1 = new cnode;
        c1->Name = c2->Name;
        c1->Votes = c2->Votes;
        c1->Next = tmp1->List;
        tmp1->List = c1;
    }
    c2 = c2->Next;
}
DeleteRegion(reg2, H);
```
Alternative Solution

rnode * FindRegion(rnode * H, int regno)
{
    if (H == NULL)
        return NULL;
    else if (H->Region == regno)
        return H;
    else
        return FindRegion(H->Next, regno);
}

int VoteCount(cnode * ctmp, String Name)
{
    if (ctmp == NULL)
        return 0;
    else if (ctmp->Name == Name) {
        int num = ctmp->Votes;
        ctmp->Votes = -1;
        return num;
    }
    else
        return VoteCount(ctmp->Next, Name);
}

void CombineRegions(rnode * & H, int reg1, int reg2)
{
    rnode * tmp1, * tmp2;
    cnode * c1, * c2, * c1prev, * c2prev;

    tmp1 = FindRegion(H, reg1);
    tmp2 = FindRegion(H, reg2);
    if (tmp2 == NULL) // no region 2
        return;
    if (tmp1 == NULL) { // no region 1
        tmp2->Region = reg1;
        return;
    }
    c1 = tmp1->List; c1prev = NULL;
    while (c1 != NULL) { // add in all votes from region 2
        c1->Votes += VoteCount(tmp2->List, c1->Name);
        c1prev = c1;
        c1 = c1->Next;
    }
    // cleanup region 2 - all nodes without -2 move over here.
c2 = tmp2->List; c2prev = NULL;
while (c2 != NULL) {
    if (c2->Votes != -1) {  // put node in region 1 list
        if (c1prev == NULL)
            tmp1->List = c2;
        else
            c1prev->Next = c2;
        if (c2prev == NULL)
            tmp2->List = c2->Next;
        else
            c2prev->Next = c2->Next;
        c1prev = c2;
        c1prev->Next = NULL;
        c2 = c2prev->Next;
    }
    else {
        c2prev = c2;
        c2 = c2->Next;
    }
}
DeleteRegion(reg2, H);
(d) (10 pts) Write a function called PrintAll consisting of one argument (a pointer to the data structure) that prints information about every region starting with the last region in the list. See the sample output on the Exam 1 Handout. You should use recursion.

```cpp
void PrintAll(rnode * H)
{
    cnode * tmp;
    if (H == NULL)
        return;
    else {
        PrintAll(H->Next);
        cout << "Region: " << H->Region << endl;
        tmp = H->List;
        while (tmp != NULL) {
            cout << tmp->Name << ": " << tmp->Votes << endl;
            tmp = tmp->Next;
        }
    }
}
```

**Alternative Solution**

```cpp
void PrintRegion(cnode * tmp)
{
    if (tmp != NULL) {
        cout << tmp->Name << ": " << tmp->Votes << endl;
        PrintRegion(tmp->Next);
    }
}

void PrintAll(rnode * H)
{
    if (H == NULL)
        return;
    else {
        PrintAll(H->Next);
        cout << "Region: " << H->Region << endl;
        PrintRegion(H->List);
    }
}
```
10. (12 pts) Use the StackClass on the Exam 1 Handout to write a nonrecursive function called IsBalanced that returns TRUE if a line of characters has ()’s and {}’s correctly matched, otherwise it returns FALSE (it is ok to return without processing all the characters). Each ( must have a ) to the right of it, and each { must have a } to the right of it. They cannot be interleaved. For example,

(a+b+(c*d)+e)*f)  valid
(a+b+(c*d)+e)*f)  invalid - interleaved ({})
(a+b+(c*d)+e)*f)  invalid - extra 

```java
boolean IsBalanced()
{
    char c;
    char top;
    boolean Success;
    stackClass Paren;

    while (cin >> c) {
        if (c == '(' || c == ')') {
            Paren.Push(c, Success);
        } else if (c == ' ') {
            Paren.GetStackTop(top, Success);
            if (!Success && top == '(') {
                return FALSE;
            }
            Paren.Pop(Success);
        } else if (c == '}') {
            Paren.GetStackTop(top, Success);
            if (!Success && top == '{') {
                return FALSE;
            }
            Paren.Pop(Success);
        }
    }

    return TRUE;
}
```