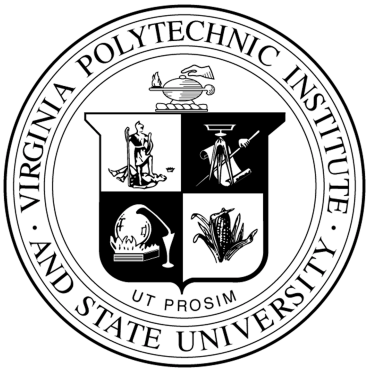





***READ THIS NOW!***

- Print your name in the space provided below.
- Print your name and ID number on the Opscan form; be sure to code your ID number on the Opscan form. Code **Form A** on the Opscan.
- Choose the single best answer for each question — some answers may be partially correct. If you mark more than one answer, it will be counted wrong.
- Unless a question involves determining whether given C++ code is syntactically correct, assume that it is valid. The given code has been compiled and tested, except where there are deliberate errors. Unless a question specifically deals with compiler `#include` directives, you should assume the necessary header files have been included.
- Be careful to distinguish integer values from floating point (real) values (containing a decimal point). In questions / answers which require a distinction between integer and real values, integers will be represented without a decimal point, whereas real values will have a decimal point, [ 1704 (integer), 1704.0 (real)].
- The answers you mark on the Opscan form will be considered your official answers.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- This is a closed-book, closed-notes examination. No calculators or other electronic devices may be used during this examination. You may not discuss (in any form: written, verbal or electronic) the content of this examination with any student who has not taken it. You must return this test form when you complete the examination. Failure to adhere to any of these restrictions is an Honor Code violation.
- There are 20 questions, equally weighted. The maximum score on this test is 100 points.

**Do not start the test until instructed to do so!**

**Print Name (Last, First)** **Solution**

**Pledge:** On my honor, I have neither given nor received unauthorized aid on this examination.

**N. D. Barnette**
signature

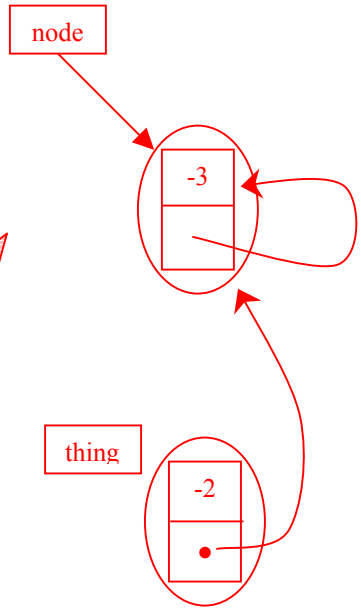
## I. Class Pointers

For the following 2 questions, assume the following declarations:

```
class   Element {
private:
    int       value;
    Element*  pointer;
public:
        void bar();
};
```

```
//inside the Element member function bar()
Element* node;
node = new Element;
node->value = -1;
node->pointer = NULL;
Element thing = *node;
thing.pointer = node;
thing.value = -2;
thing.pointer->value = -3;
thing.pointer->pointer = thing.pointer;
```

**Code Execution Results**




KEY

1. From inside the same member function as the above code, what is the data type of the expression at the right:

```
thing.pointer->pointer
```

1) Element
2) Pointer
3) Element*
4) Pointer*
5) class*
6) None of the above

2. From inside the same member function as the above code, which of the following statements could be used to connect, (i.e., point), the `Element` object containing the integer `-2` to the `Element` object containing the integer `-2`, (i.e. to itself)?

1) `node->pointer = node;`
2) `thing->pointer = thing;`
3) `node.pointer = &node;`
4) `thing.pointer = &thing;`
5) `thing->pointer = &thing;`
6) None of the above

3. From inside the same member function as the above code, what is the data type of the expression at the right?

```
*node
```

1) Element
2) Pointer
3) Element*
4) Pointer*
5) Element->
6) None of the above

4. Assuming the default (language supplied) destructor (i.e. no destructor has been explicitly implemented), consider just the code above, after the member function would complete execution, how many memory leaked `Element` objects would still exist in memory?

1) 1
2) 2
3) 3
4) 4
5) 0
6) None of the above

Only 1 Element object was allocated dynamically, default language destructor do NOT automatically de-allocate dynamic memory.

**II. Linked List Class Manipulation**

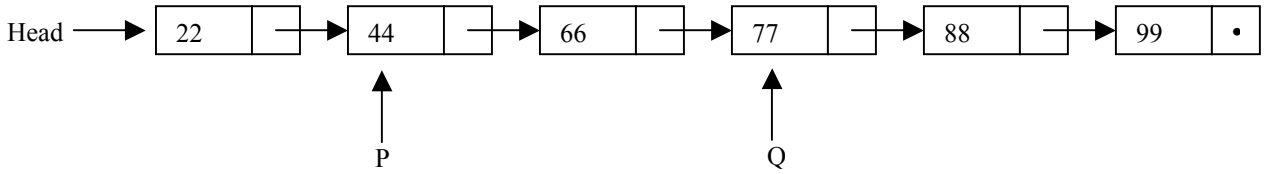
Consider the linked list class and list node declarations given below:

```
class ItemType {
private:
   int Value;
public:
   ItemType();
   ItemType(int newValue);
   void setValue(int newValue);
   int  getValue() const
         {return Value;}
};
```

```
class LinkNode {
private:
   ItemType  Data;
   LinkNode* Next;
public:
   LinkNode();
   LinkNode(ItemType newData);
   bool setNext(LinkNode* newNext);
   bool setData(ItemType  newData);
   ItemType  getData() const;
   LinkNode* getNext() const;
};

LinkNode *Head, *P, *Q;
```

Assume that the member functions above have been implemented correctly to carry out their intended task. Given the initial list structure:




For the next 4 questions, determine what the execution of the given code fragment would do, assuming the list structure above as your starting point (for each question). Note – dangling references into the heap should be ignored, indicated by "??". Choose from the possible answers given on the following page.

5.
```
LinkNode     *Y, *X=Head;
for (int i=0; i<=0; i++) {
  Y = X->getNext()->getNext();
  delete X->getNext();
  X->setNext(Y);
  X = Y;
}
```
**4**

6.
```
LinkNode     *T=P;
while (T->getNext() != NULL) T = T->getNext();
Head->setNext(P->getNext());
P->setNext(T);
T->setNext(Head->getNext());
```
**6**

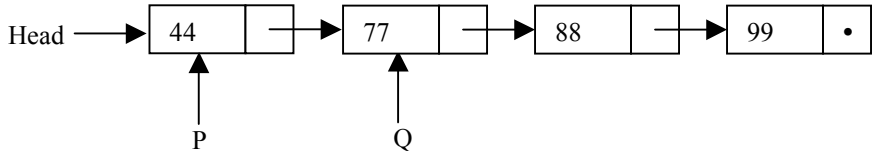
7.
```
LinkNode     *S=Head;
P->getNext()->setNext(Q->getNext());
delete Q;
S->setNext(P->getNext());
delete P;
```
**2**

8.
```
LinkNode     *R;
R = P->getNext()->getNext()->getNext();
R->setNext(Head->getNext());
```
**8**

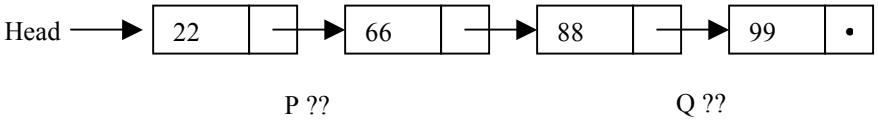
**II. Linked List Class Manipulation** (*continued*)

Select from the possible answers for the 4 questions given on the previous page. Question marks (??) indicate the pointer has an unknown, or invalid, value.

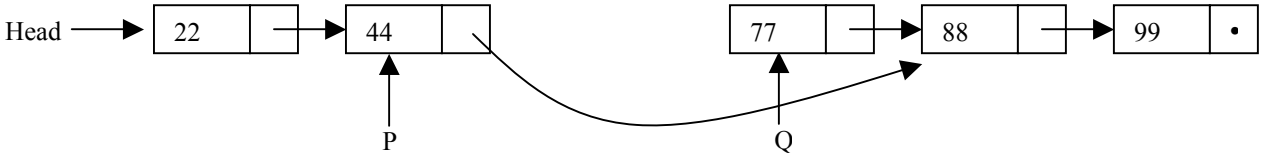
1)



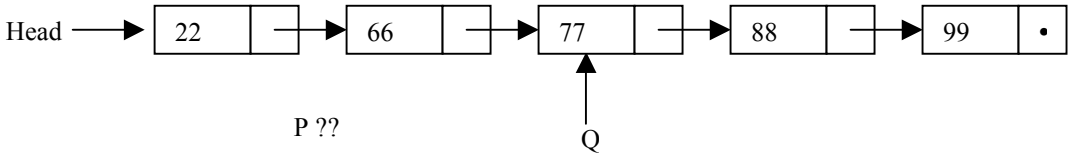

2)



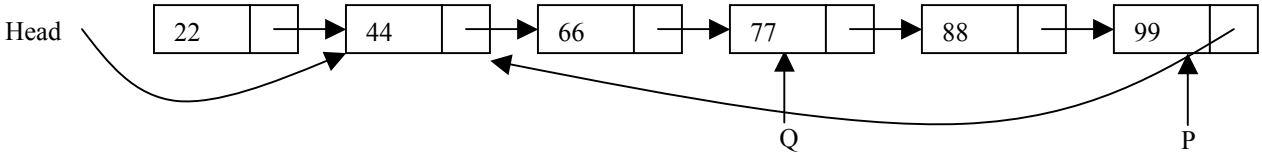

3)



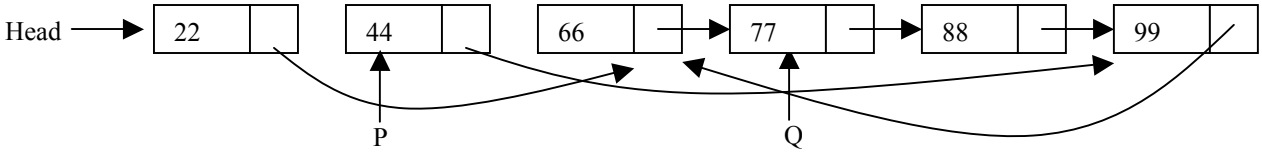

4)



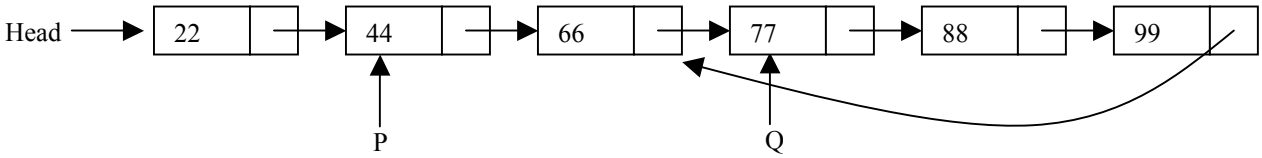

5)



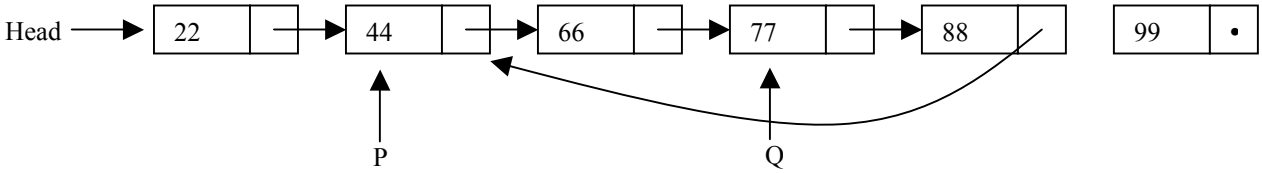

6)




7)




8)



**III. Command Line Arguments**

Consider the P2 LAMS program that provides for zero command line arguments, one or two:

```
LAMS <InitialMovieDataFileName>
```

or

```
LAMS <DatabaseFileName> <DatabaseActionsFileName>
```

For the next question, choose from the following possible answers:

| | | |
|---|---|---|
| 1) arg | 4) argc[0] | 7) argv[0] |
| 2) argc | 5) argc[1] | 8) argv[1] |
| 3) argv | 6) argc[2] | 9) argv[2] |

9. The incomplete code below checks for the existence of the optional command line arguments and calls functions to process them. Select the answer to fill in the blank to carry out the indicated task correctly. All of the blanks are to be filled in by the same choice.

**8**

The first program argument is the program's name, the second argument is stored in the second argv array location.

```
switch(argc) {
   case 0: break;
   case 1: break;
   case 2: //call ConvertToDatabase passing Initial Movie Data File Name
           ConvertToDatabase(___________); break;
   case 3: //call ReadDatabase passing Database File Name
           ReadDatabase(___________);
           //call ProcessActions passing Database Actions File Name
           ProcessActions(arg?????????); break;
   default://invalid number of CLAs
           cout <<"usage: LAMS  <InitialMovieDataFile>"<<endl;
           cout <<"       LAMS  <DatabaseFile> <DatabaseActionsFile>"<<endl;
} //end switch
```

10. In the code above one of the switch cases can never logically occur, which one?

| | | |
|---|---|---|
| 1) Case 1 | 3) Case 3 | 5) Case 0 |
| 2) Case 2 | 4) default | 6) None of the above |

Every C++ program receives at least one command line, the program's own name.

## IV. Object Manipulations

Assume the following class declaration and implementation:

```
class Info {
private:
   double* Dinfo;
public:
   Info();
   Info(double Dinit);
   double getDinfo()const;
   void setDinfo(double Dset);
   bool operator<(const Info& Info2);
   ~Info();
};

Info::Info() {
   this->Dinfo = new double(0.0);
}

Info::Info(double Dinit) {
   Dinfo = new double(Dinit);
}
```

```
double Info::getDinfo() const {
   return (*Dinfo);
}

void Info::setDinfo(double Dset) {
   *Dinfo = Dset;
}

bool Info::operator<
           (const Info& Info2) {
   return ( (*Dinfo) <
           *(Info2.Dinfo) );
}

Info:: ~ Info() {
    delete Dinfo;
}
```

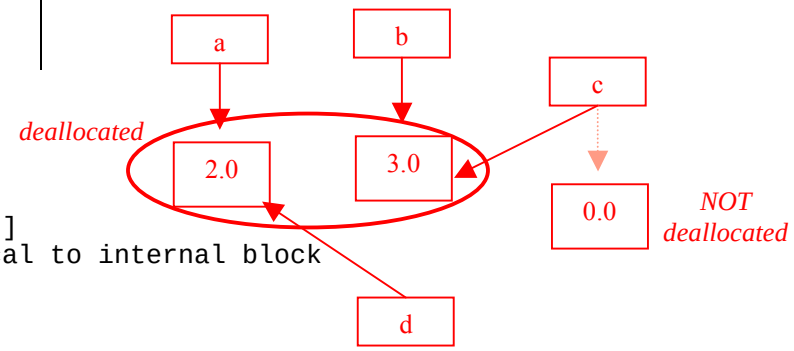
Given the following code:

```
void main() {
Info a, b(1.0);

{ // [(| Internal Block in main() |)]
   Info c, d = a; //Info objects local to internal block
   d.setDinfo(2.0);
   c = b;
   b.setDinfo(3.0);
   cout << "Info of a is:" << a.getDinfo() << endl;          //LINE 1
   cout << "Info of c is:" << c.getDinfo() << endl;          //LINE 2
} // [(| End of Internal block |)]

    cout << "Info of b = " << b.getDinfo() << endl;          //LINE 3
}
```

**Code Execution Results**




11. In the above code, after execution, how many dynamically allocated double, (not bytes), memory storage locations are not deleted, (that is, how many memory leaks occurs in the above code)?

1) 1
2) 2
3) 3
4) 4
5) 0 (all are deleted)
6) None of these

When the default memberwise copy sets c = b; the double memory dynamically allocated to c by the constructor becomes a memory leak. The Dinfo pointer in b becomes a dangling pointer.

**IV. Object Manipulations** *(continued)*

Considering the previous code, for the next 3 questions, select your answers from the following:

1) `1.0`
2) `2.0`
3) `3.0`
4) `0.0`
5) Execution Error
6) None of these

12. What is output by the call `a.getDinfo()` in `LINE 1` above? **2**
13. What is outputby the call `c.getDinfo()` in `LINE 2` above? **3**
14. What is outputby the call `b.getDinfo()` in `LINE 3` above? **5**

See code execution results above.

---

15. When a copy constructor function is implemented in a class which contains dynamic data, then the copy constructor would be automatically invoked in each of the following described execution points in a program except one. Identify at which instance the copy constructor function would **NOT** be automatically executed?

```
1) When an object is returned by a function.
2) When an object is initialized to another object in a defi
3) When an object is passed by value.
4) When an object is assigned to an existing object of the same class.
5) None of the above, (all of the above situations would execute the copy
   constructor).
```

In this case the over-loaded assignment operator function would be invoked.

16. (True or False) Copy constructor and assignment operator overload functions should be implemented for all classes. Even for classes that do not contain dynamic data.

`1) True` `2) False`

For classes that do not contain dynamic memory, the default language supplied memberwise copy constructor and assignment operator will usually suffice.

**V. Recursion**

Consider the recursive function given below:

```
int SigmaSqr(int a, int b) {
   if ( a == b) return( a * a );
   else return( SigmaSqr(((a+b)/2+1), b) + SigmaSqr(a, ((a+b)/2)) );
}
```

17. What is the value returned by the call `cout << SigmaSqr(2,6);`

1) None of these
2) 9
3) 16
4) 25
5) 29
6) 40
7) 51
8) 80
9) 90

See recursive execution trace on following page.

18. The previous function is an example of what type of recursive problem solving strategy?

1) going up (tail) recursion
2) going down (head) recursion
3) middle decomposition recursion

The SigmaSqr() decomposes the work of computing by breaking the range up into an upper and lower part and recursively applying the

---

19. The following recursive function call, `Sum(IntArray, 0, Size)`, sums an array of integers:

```
int Sum(const int Array[], int Begin, int Dim ) {

    if ( Begin >= Dim ) return 0;
    else if (____________________)
       return Array[Dim-1];
    else
       return ( Array[Begin] + Sum(Array, Begin+1, Dim) );
}
```

What should the missing condition in the `if` statement be?

The return of the Dim-1 value as a base case requires a check for when the end of the array is reached.

1) Begin != Dim
2) Array[Begin] == Array[Dim]
3) Array[Begin-1] == Array[Dim-1]
4) Begin == Dim-1
5) Begin-1 == Dim-1
6) None of the above

20. The previous function is an example of what type of recursive problem solving strategy?

1) going up (tail) recursion
2) going down (head) recursion
3) middle decomposition recursion
4) edges & center decomposition
5) backtracking
6) None of the above

The recursive code works upon the current data (Array[Begin-1]) and then progresses toward the end of the data.

**Code Execution Results for # 17.**