Instructions: This homework assignment covers some of the basic C++ background you should have in order to take this course.

Opscan forms will be passed out in class. Write your name and code your ID number on the Opscan form. Turn in your completed Opscan at the time announced by your instructor. Opscans will not be accepted at any other time.

I. Pointers

For questions 1 through 6, assume the variable declarations and initial memory layout shown below:

```
int a = 17, b = 42;
int *p, *q;
```

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>006AFDF4</td>
</tr>
<tr>
<td>b</td>
<td>006AFDF0</td>
</tr>
<tr>
<td>p</td>
<td>006AFDEC</td>
</tr>
<tr>
<td>q</td>
<td>006AFDE8</td>
</tr>
</tbody>
</table>

Suppose the following statements are executed:

```
p = &b; // statement 1
q = &a; // 2
*p = *q; // 3
q = p; // 4
```

For questions 1 through 6, chose from the following answers:

1) 006AFDF4 4) 006AFDE8 7) Unknown
2) 006AFDF0 5) 17 8) None of these
3) 006AFDEC 6) 42

1. Immediately after the execution of statement 1, what is the value of: 
   p
   1) 006AFDF4 2) 006AFDF0 3) 006AFDEC 4) 006AFDE8 5) 17 6) 42 7) Unknown 8) None of these

2. Immediately after the execution of statement 2, what is the value of: 
   *q
   1) 006AFDF4 2) 006AFDF0 3) 006AFDEC 4) 006AFDE8 5) 17 6) 42 7) Unknown 8) None of these

3. Immediately after the execution of statement 2, what is the value of: 
   &q
   1) 006AFDF4 2) 006AFDF0 3) 006AFDEC 4) 006AFDE8 5) 17 6) 42 7) Unknown 8) None of these
1) 006AFDF4
2) 006AFDF0
3) 006AFDEC
4) 006AFDE8
5) 17
6) 42
7) Unknown
8) None of these

4. Immediately after the execution of statement 3, what is the value of: $p$

1) 006AFDF4
2) 006AFDF0
3) 006AFDEC
4) 006AFDE8
5) 17
6) 42
7) Unknown
8) None of these

5. Immediately after the execution of statement 3, what is the value of: $*p$

1) 006AFDF4
2) 006AFDF0
3) 006AFDEC
4) 006AFDE8
5) 17
6) 42
7) Unknown
8) None of these

6. Immediately after the execution of statement 4, what is the value of: $*q$

1) 006AFDF4
2) 006AFDF0
3) 006AFDEC
4) 006AFDE8
5) 17
6) 42
7) Unknown
8) None of these

7. Assume the variable declarations:

   int *Foo = new int(0);

Which of the following statements will increment the target of Foo?

1) Foo++;
2) (*Foo)++;
3) *Foo++;
4) 2 and 3 only
5) None of these
8. Both code fragments below will compile but the one on the right contains a logic error, and will probably cause a runtime error. Why?

```
int x = 5;
int *p = new int(x);
delete p;
```

```
int x = 5;
int *p = &x;
delete p;
```

1) Assigns an address to an int variable.
2) Assigns an int variable to a pointer.
3) Deletes a statically allocated variable.
4) None of these

---

For questions 9 through 11 assume that we have a dynamically allocated array $A$ of integers of dimension $Size$, with memory layout as shown:

```
const int Size = 5;
int *A = new int[Size];
```

<table>
<thead>
<tr>
<th>Index</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>007D0E80</td>
</tr>
<tr>
<td>1</td>
<td>007D0E7C</td>
</tr>
<tr>
<td>2</td>
<td>007D0E78</td>
</tr>
<tr>
<td>3</td>
<td>007D0E74</td>
</tr>
<tr>
<td>4</td>
<td>007D0E70</td>
</tr>
</tbody>
</table>

9. Which code fragment(s) could be inserted in the blank in order to safely initialize each element of $A$ to zero?

```
int* p = &A[0];
for (int Idx = 0; Idx < Size; Idx++, p++) {
    ______________;
}
```

1) $A[Idx] = 0$;
2) *$p = 0$;
3) *$A = 0$;
4) *$Idx = 0$;
5) All of the above
6) 1 and 2 only
7) 1 and 3 only
8) 1 and 4 only
9) None of these

10. What value will be printed by the code fragment:

```
for (int Idx = 0; Idx < Size; Idx++) {
    A[Idx] = int(&A[Idx]);  // typecast converts to int
}
```

1) 007D0E70
2) 007D0E74
3) 007D0E78
4) 007D0E7C
5) 007D0E80
6) Unknown
7) None of these

11. Assuming only the initial declarations given above, what logical error(s) would result if the following statement were executed: $A = new int[2*Size]$?

1) A dangling pointer would result (a pointer whose value is the address of memory that the program no longer owns).
2) A memory leak would result (the program would own memory that it could no longer access).
3) Both a dangling pointer and a memory leak would result.
4) Neither 1 nor 2, but some other logical error would result.
5) No logical error would result.
12. Consider implementing a function to dynamically allocate an array of integers and reset all its elements to zero:

```cpp
void ZeroIt(int*& A, const int Size) {
    A = new int[Size];
    for (int Idx = 0; Idx < Size; Idx++) {
        A[Idx] = 0;
    }
}
```

Which of the following choices for the blank preceding the formal parameter `A` is best?

1) `int*&`  
2) `int*`  
3) `const int*`  
4) `int* const`  
5) `const int* const`  
6) All of the above

For questions 13 and 14, assume the declarations:

```cpp
struct Node {
    float Volume;
    Node* Next;
};
Node* headPtr;
```

Also assume that `headPtr` is the head pointer to a linked list of many `Node`s.

13. Which statement renders the head node (ONLY the head node) inaccessible?

1) `*headPtr = headPtr->Next;`  
2) `headPtr->Next = headPtr->Next->Next;`  
3) `headPtr = headPtr->Next->Next;`  
4) `headPtr->(*Next) = headPtr->Next->Next;`  
5) None of these

14. Which statement changes the data element of the first Node in the list?

1) `headPtr->Next = 42;`  
2) `headPtr->Next->Volume = 42;`  
3) `*(headPtr->Next->Volume) = 42;`  
4) `*(headPtr->Next) = 42;`  
5) None of these

II. Recursion

15. Given the recursive function below, what is the value of the expression `Sum(5)`?

```cpp
int Sum( int n ) {
    if ( n < 8 )
        return ( n + Sum(n - 1) );
    else
        return 2;
}
```

Due: TBA
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>5) None--the result is infinite recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>5</td>
<td>3)</td>
<td>20</td>
</tr>
<tr>
<td>2)</td>
<td>13</td>
<td>4)</td>
<td>28</td>
</tr>
</tbody>
</table>

6) None of these
For questions 16 through 20, consider the following function, `countEm()`, and the associated recursive helper function `rCount()`, which are intended to determine how many times a specified `Value` occurs in a given array holding `Size` elements:

```cpp
int countEm(int Value, const int Array[], int Size) {
    return rCount(Value, Array, Size);
}

int rCount(int Value, const int Array[], int Size) {
    if (Size <= __________)    // Line 1
        return __________;    // Line 2
    else if (Array[__________] == Value)  // Line 3
        return __________;  // Line 4
    else
        return __________;  // Line 5
}
```

16. How should the blank in Line 1 be filled?
   1) 0  
   2) `Size`  
   3) `-1`  
   4) `Value`  
   5) `INT_MAX`  
   6) None of these

17. How should the blank in Line 2 be filled?
   1) 0  
   2) 1  
   3) `Size`  
   4) `INT_MAX`  
   5) `Value`  
   6) None of these

18. How should the blank in Line 3 be filled?
   1) `Size`  
   2) `Size - 1`  
   3) 0  
   4) 1  
   5) `INT_MAX`  
   6) None of these

19. How should the blank in Line 4 be filled?
   1) `rCount(Value, Array, Size - 1)`  
   2) `rCount(Value, Array, Size)`  
   3) `1 + rCount(Value, Array, Size - 1)`  
   4) `1 + rCount(Value, Array, Size)`  
   5) 1  
   6) None of these

20. How should the blank in Line 5 be filled?
   1) `rCount(Value, Array, Size - 1)`  
   2) `rCount(Value, Array, Size)`  
   3) `1 + rCount(Value, Array, Size - 1)`  
   4) `1 + rCount(Value, Array, Size)`  
   5) 1  
   6) None of these
### III. Lists

For questions 21 through 25, consider the following declaration and implementation for a circular array-based Queue. **Note:** In this implementation the Front pointer is the index of the first element and the Rear pointer is the index of the next available cell (not of the last element in the Queue).

```cpp
class Queue {
private:
    int Size;                      // dimension of queue array
    int Front;                     // index for next deletion
    int Rear;                      // index for next insertion
    ItemType *Q;                   // queue array
public:
    Queue(int Sz)
        : Size(Sz)
        , Q(new ItemType[Size])
        , Front(0)
        , Rear(0)
    {
    }
    Queue(const Queue& Source)
        : Size(Source.Size)
        , Q(new ItemType[Size])
        , Front(Source.Front)
        , Rear(Source.Rear)
    {
    }
    Queue& operator=(const Queue& Source)
    {
        Size = Source.Size;
        Front = Source.Front;
        Rear = Source.Rear;
        return *this;
    }
    bool Enqueue(ItemType Item)
    {
        if (isFull())
            return false;
        Rear = Rear == Size - 1 ? 0 : Rear + 1;
        return true;
    }
    bool Dequeue(ItemType& Item)
    {
        if (isEmpty())
            return false;
        Item = Front == 0 ? Size - 1 : Front - 1;
        Front = Front == 0 ? Size - 1 : Front - 1;
        return true;
    }
    bool isEmpty() const
    {
        return (Rear == Front);
    }
    bool isFull() const
    {
        return (Front == Size - Rear);
    }
    void Display()
    {
        for (int i = Front; i != Rear; i = (i + 1) % Size)
            std::cout << Q[i] << ' ';  
        std::cout << std::endl;
    }
    int getSize() const
    {
        return Size;
    }
    void Clear()
    {
        Front = Rear = 0;
    }
    ~Queue()
    {
        delete[] Q;
    }
};
```

21. How should the blank in Line 1 be filled? (Remember that the states "full" and "empty" are different.)

1) \((\text{Rear} + 1) \mod \text{Size}\)
2) \(\text{Size} - \text{Rear}\)
3) \(\text{Rear}\)
4) \(\text{Rear} + 1\)
5) None of these

22. How should the blank in Line 2 be filled?

```cpp
bool Queue::Enqueue(ItemType Item) {
    if (isFull())
        return false;
    Rear = _____________;  // Line 2
    return true;
}
```
1) Q[Front]  
2) Q[Size]  
3) Q[Rear]  
4) Q[Rear + 1]  
5) None of these
23. How should the blank in Line 3 be filled?

1) Rear - 1  
2) Rear + 1  
3) Front  
4) (Rear + 1) % Size  
5) None of these

24. How should the blank in Line 4 be filled?

1) Q[Front - 1]  
2) Q[Front + 1]  
3) Q[Rear]  
4) Q[Rear + 1]  
5) None of these

25. How should the blank in Line 5 be filled?

1) Front - 1  
2) (Front - 1) % Size  
3) Front + 1  
4) (Front + 1) % Size  
5) None of these

For questions 26 through 30 we consider implementing a function (NOT a member function) to sort the entries of a Queue object into ascending order, where Queue is declared on page 2. Given the shell below, which is a variant of selection sort, and assuming that int has been typedef'd to ItemType:

```cpp
void Sort(Queue& Q) {
    int numItems = Q.getSize();
    int SmallSoFar, Look;
    Queue Temp(numItems);    // 1: holds unsorted values from Q
    Queue Sorted(numItems);   // 2: holds sorted values

    while ( !Q.isEmpty() ) {
        SmallSoFar = 0;    // 3: first is smallest seen yet
        while ( !Q.isEmpty() ) {
            Q.Dequeue(Look);    // 4: grab next element from Q
            if (Look < SmallSoFar) {  // 5: if it's a new minimum
                SmallSoFar = Look;    // 6: save old minimum for future
            } else {    // 7: reset minimum
                SmallSoFar = Look;    // 8: save current value for future
            }
            Sorted.Enqueue(SmallSoFar);  // 9: put smallest in results queue
        }   // 10: restore unsorted values to Q
    }   // 11: clear temporary queue
    Q = Sorted;    // 12: put sorted values into Q
}
```

26. How should the blank in line 3 be filled?

1) SmallSoFar = 0  
2) SmallSoFar = Q[0]  
3) Q.Dequeue(Look)  
4) Q.Dequeue(SmallSoFar)  
5) None of these
27. How should the blank in line 6 be filled?

1) It should be blank
2) Q.Enqueue(SmallSoFar)
3) Temp.Enqueue(SmallSoFar)
4) Temp.Enqueue(Look)
5) None of these

28. How should the blank in line 8 be filled?

1) Temp.Enqueue(SmallSoFar)
2) Temp.Enqueue(Look)
3) Q.Enqueue(SmallSoFar)
4) Q.Enqueue(Look)
5) None of these

29. How should the blank in line 10 be filled?

1) Temp = Q
2) Q = Sorted
3) Q = Temp
4) Sorted = Temp
5) None of these

30. How should the blank in line 11 be filled?

1) Temp = NULL
2) delete Temp
3) Temp.Clear()
4) Sorted.Clear()
5) None of these

IV. Inheritance and Polymorphism

For questions 31 through 34, assume that Foo and Bar are C++ classes, and that the class Foo is derived, using public inheritance, from the class Bar.

31. If X is an object of type Bar, then the member functions of X:

1) can directly access only the added public members of class Foo.
2) cannot directly access any of the added members of class Foo.
3) can directly access the added public, protected, and private members of class Foo.
4) can directly access only the added public and protected members of class Foo.
5) None of these

32. Suppose that an object X of type Foo is declared. Then:

1) a constructor for the base class, Bar, will be executed after any constructor for the derived class, Foo.
2) no constructor for the base class, Bar, will be executed at all.
3) a constructor for the base class, Bar, will be executed before any constructor for the derived class, Foo.
4) constructors for both classes will be executed at the same time.
5) a constructor for the base class, Bar, may (or may not) be executed, and that may take place either before or after the execution of a constructor for the derived class, Foo.
6) None of these
33. Suppose that an object $X$ of type $\text{Foo}$ is declared. Then when the lifetime of that object ends:

1) destructors for both classes will be executed at the same time.
2) the destructor for the base class, $\text{Bar}$, will be executed after the destructor for the derived class, $\text{Foo}$.
3) the destructor for the base class, $\text{Bar}$, will not be executed at all.
4) the destructor for the base class, $\text{Bar}$, will be executed before the destructor for the derived class, $\text{Foo}$.
5) the destructor for the base class, $\text{Bar}$, may (or may not) be executed, and that may take place either before or after the execution of the destructor for the derived class, $\text{Foo}$.
6) None of these

34. Which of the following is true?

1) Public members of $\text{Foo}$ become public members of $\text{Bar}$.
2) Public members of $\text{Bar}$ become public members of $\text{Foo}$.
3) Public members of $\text{Foo}$ become private members of $\text{Bar}$.
4) Public members of $\text{Bar}$ become private members of $\text{Foo}$.
5) 2 and 3 only
6) None of these

For questions 35 and 36, suppose that a C++ class $D$ is derived from a base class $B$, that class $B$ has a public member function $F()$, and class $D$ redefines its own version of $F()$. At execution time, suppose that a pointer to a $D$ object is passed to the following function:

```cpp
void Foo(B* x) {  
    x->F();  
}
```

35. If $F()$ is declared to be virtual in class $B$, whose version of $F()$ is called?

1) class $D$'s version  
2) class $B$'s version  
3) Both versions are called.  
4) Neither version is called.

36. If $F()$ is not declared to be virtual in class $B$, whose version of $F()$ is called?

1) class $D$'s version  
2) class $B$'s version  
3) Both versions are called.  
4) Neither version is called.

For questions 37 and 38, suppose that a C++ class $D$ is derived from a base class $B$, that class $B$ has a public member function $F()$, and class $D$ redefines its own version of $F()$. At execution time, suppose that a $D$ object is passed to the following function:

```cpp
void Foo(B x) {  
    x. F();  
}
```

37. If $F()$ is declared to be virtual in class $B$, whose version of $F()$ is called?

1) class $D$'s version  
2) class $B$'s version  
3) Both versions are called.  
4) Neither version is called.
38. If \( F() \) is not declared to be virtual in class B, whose version of \( F() \) is called?

1) class D's version
2) class B's version
3) Both versions are called.
4) Neither version is called.

For questions 39 through 45, assume the following class declarations:

```cpp
class Base {
public:
  virtual void F();
  virtual void G() = 0;
  void H();
};

class D : public Base {
public:
  void F();
  void G();
  void H();
};

class E : public D {
public:
  void F();
  void G();
};
```

Suppose corresponding implementations are given for each class, and consider the following `main()`:

```cpp
void main() {
  D* pD = new D;
  Base* pB = (Base*) pD;

  pB->F();               // call 1
  pB->G();               // call 2
  pB->H();               // call 3

  E* pE = new E;
  pB = (Base*) pE;
  pD = (D*) pE;

  pB->F();               // call 4
  pD->F();               // call 5
  pB->G();               // call 6
  pE->H();               // call 7
}
```

39. Which function is called in the statement labeled call 1?

1) Base::F()
2) D::F()
3) E::F()
4) Two or more of the above.
5) None of these.

40. Which function is called in the statement labeled call 2?

1) Base::G()
2) D::G()
3) E::G()
4) Two or more of the above.
5) None of these.
41. Which function is called in the statement labeled call 3?

1) Base::H ()
2) D::H ()
3) E::H ()
4) Two or more of the above.
5) None of these.

42. Which function is called in the statement labeled call 4?

1) Base::F ()
2) D::F ()
3) E::F ()
4) Two or more of the above.
5) None of these.

43. Which function is called in the statement labeled call 5?

1) Base::F ()
2) D::F ()
3) E::F ()
4) Two or more of the above.
5) None of these.

44. Which function is called in the statement labeled call 6?

1) Base::G ()
2) D::G ()
3) E::G ()
4) Two or more of the above.
5) None of these.

45. Which function is called in the statement labeled call 7?

1) Base::H ()
2) D::H ()
3) E::H ()
4) Two or more of the above.
5) None of these.

V. Other Class Issues

For questions 46 through 50, consider the classes Track and Album:

```c++
class Track {
private:
    string Title;
    int    Length;
public:
    Track(string T, int L);
    string getTitle() const;
    int    getLength() const;
};

class Album {
private:
    string Title;
    string Artist;
    int    numTracks;
    Track* PlayList;
public:
    Album(string T, string A, int numTracks);
    bool AddTrack(const Track& T);
    Track getTrack(int Position) const;
    int getNumTracks() const;
    ~Album();
};

Album::~Album() {
    ...;
    Album::~Album() {
```
Consider the following function, which computes the sum of the lengths of all the tracks on an album:

```cpp
int Length(Album CD) {
    int totalLength = 0;
    int Idx;
    for (Idx = 0; Idx < CD.getNumTracks(); Idx++) {
        totalLength += CD.getTrack(Idx).getLength();
    }
    return totalLength;
}
```

46. The call `Length(myAlbum)` will have an unfortunate side effect (even though the body of the function is correct). What is that effect?

1) `myAlbum.numTracks` is changed.
2) The array of tracks, `myAlbum.PlayList []`, is deleted.
3) The destructor for `myAlbum` is invoked.
4) All of the above.
5) 2 and 3 only
6) None of these

47. Assuming that `myAlbum` is declared in the calling function, will the call `Length(myAlbum)` also cause a runtime exception at the end of the calling function?

1) Perhaps yes, perhaps no, depending on factors not specified in the question.
2) Yes, definitely.
3) No, definitely not.

48. Given that the interface and implementation of `Length()` cannot be changed, and that the use of global variables is unacceptable, which of the following actions is/are necessary to eliminate the difficulties cited in questions 46 and 47?

1) A deep assignment operator overload should be implemented for the class `Album`.
2) The destructor for the class `Album` should be removed from the class.
3) A deep copy constructor should be implemented for the class `Album`.
4) All of the above.
5) 1 and 2 only
6) None of these

49. Aside from the action(s) you chose in question 48, which of those actions should also be done in order to produce a robust, general implementation?

1) A deep assignment operator overload should be implemented for the class `Album`.
2) The destructor for the class `Album` should be removed from the class.
3) A deep copy constructor should be implemented for the class `Album`.
4) All of the above.
5) 1 and 2 only
6) None of these
50. What feature of the class *Album* or of the class *Track* should have immediately tipped the implementer of that class to take the actions referred to in questions 48 and 49?

1) *Track* does not implement a destructor.
2) *Album* contains an aggregation of *Track* objects.
3) *Album* contains a pointer to dynamically allocated storage.
4) *Track* contains a string object.
5) None of these