Fall 2008 EE 380L Final: NAME: _____

1. (15 pts) Write a template class Array<typename T, int size=128> that provides an operator[] (i.e., that looks like an array. There's no "push_back" or "pop_back" functions, and I don't need an iterator for this one. Just create a fixed size array of type T. Note that the size (the number of elements in the array) is part of the type. When size is equal to or smaller than 128, the elements should be allocated inside the object (i.e., sizeof(Array<T, 128>) should be exactly 128 * sizeof(T)). However, when size is larger than 128, the elements should be allocated on the heap, the array object should only contain a pointer to the first element, and sizeof(Array<T, 1024>) should be 4 bytes (for 32-bit machines, 8-bytes on 64-bit machines of course).

2. (15 pts) Write a template class Larger<typename X, typename Y> that has a nested type called “Answer”. Write your class such that Larger<X, Y>::Answer is always either X or Y. If sizeof(Y) is larger than sizeof(X) then Larger<X, Y>::Answer should be Y, otherwise it should be X.

3. (15 pts) Write an operator<< for ostream and class Foo such that the output is displayed in reverse order. So, for example, the following program should print CBA. Hint, use expression templates, and you can assume "endl" is type **char**.

```
class Foo {
  char let;
public:
  Foo(char v) { let = v; }
};

int main(void) {
  Foo a('A'); Foo b('B'); Foo c('C');
  cout << a << b << c << endl;
}
```

4. (15 pts) Design a class Ptr<T> that uses "reference counting" (similar to Project 1) to keep track of when the object it points to should be deallocated. Ptr<T> should have the following characteristics (for some arbitrary type T, e.g., T could be an **int** or a class).

```
Ptr<T> p; // p is a null pointer
if (p.isNull()) { // returns true, since p is null
  p = T(); // creates a new object of type T on the heap
}
p = T(); // creates a second object, the first object is deleted
Ptr<T> q;
q = p; // p and q both point to the same object
cout << *q; // prints the object
*q = T(); // assigns to the object (does not allocate a new one)
// when the destructors for both p and q have been run, the second object is deleted
```

5. (10 pts) Please assume that you have a correct Ptr<T> (just in case your solution was not quite perfect, you should assume that you use one that is perfect)… write a program that:
   - Uses one or more Ptr<T> pointers
   - Does not call **new** or **delete** explicitly
   - Allocates memory that is not deallocated (i.e., contains a memory leak).

   If you feel you need to, you may write templates, classes or functions as part of your solution.

6. (10 pts) I wrote a program that dynamically allocated an array of objects. The array could either be an array of Base objects or an array of Derived objects (Derived is a subtype of Base). Assume that class *Base* has no data members.

```
class Derived : public Base {
  int* p;
public:
  Derived(void) { p = new int; }
  ~Derived() { delete p; }
};
int main(void) {
  Base* b;
  if (random() % 2 == 1) { // coin toss
    b = new Base[5];
  } else {
    b = new Derived[5];
  }
  // how do I delete b?
}
```

The problem I have is trying to delete the array. I want to avoid any memory leaks (i.e., I need to make sure that every byte of memory allocated by **new** is deallocated by **delete**). I see two issues, each issue with two choices. What combination of the following two choices (if any) will solve my problem:

Choice 1: The *Base* destructor can be **virtual** or **not virtual**
Choice 2: I can deallocate b using "**`delete b`**" or "**`delete[] b`**"

Can the problem be solved using those options? If so, how?

7. (20 pts, short answer)
   a) Member templates in C++ cannot be virtual functions. Explain why not.

   b) What relationship must hold between types S and T (note A and B are used in the next question, assume they are the same type for purposes of this question)

```
class Base {
public:
  virtual S doit(const A&) { … }
};
class Derived : public Base {
public:
  virtual T doit(const B&) {… }
};
```

      i. S and T must be the same type
      ii. S must be a supertype of T (or the same)
      iii. S must be a subtype of T (or the same)

   c) Same as b) except now I want to know what relationship you think should hold between types A and B
      i. A and B must be the same type
      ii. A must be a supertype of B (or the same)
      iii. A must be a subtype of B (or the same)

   d) The following program assumes the use of a garbage collector. When is a copy based collector likely to provide faster performance than a mark-and-sweep collector?

```
int main(void) {
  typedef int* IntPtr;
  IntPtr ptrs[N];
  while (true) {
    int k = random() % N; // choose a random number 0..N-1
    ptrs[k] = new int;
  }
}
```

      i. When N is small (e.g., equal to or close to 1)
      ii. When N is large (e.g.., more than 1000)
      iii. For all values of N (copy collectors are always faster)
      iv. For no values of N (copy collectors are never faster)