Fall 2007 EE 380L Midterm: NAME:________________________

1. (6 pts) Assume I have the following Base and Derived classes as shown. Indicate what happens when the following program is run:

```
class Base {
public:
  virtual void doit(void) const { cout << "Base\n"; }
  template <typename V>
  static void examQ(const V& x) {
    x[0].doit();
    x[1].doit();
  }
};
class Derived : public Base {
public:
  virtual void doit(void) const { cout << "Derived\n"; }
};

int main(void) {
  vector<Base> x;
  x.push_back(Base());
  x.push_back(Derived());
  Base::examQ(x);
}
```

2. (6 pts) Using the same Base and Derived classes from question 1, what happens when the following program is run:

```
int main(void) {
  Derived x[2];
  Base::examQ(x);
}
```

3. (9 pts) Complete the poly_vector<T> class so that the following program prints "Base" when x[0].doit() is called, and prints "Derived" when x[1].doit() is called. Hint: it's OK if your program has a memory leak – you can call **new** and you don't need to worry about calling **delete** for this question. Hint: you might need to override the push_back method from the vector class. Hint: you might need to override push_back with a "member template".

```
int main(void) {
  poly_vector<Base> x ;
  x.push_back(Base());
  x.push_back(Derived());
  Base::examQ(x);
}

template <typename T>
class poly_vector : public vector<T*> {
public:








  const T& operator[](unsigned k) const {
     return *(vector<T*>::operator[](k));
  }
};
```

4. (12 pts) It is widely known that the operators for **new** and **delete** can be overloaded (replaced) with custom functions. That is, it is possible to write our own new and delete operators. It is widely believed that it is impossible, or at least impractical, to use garbage collection with C++. Why is it difficult to replace **new** and **delete** with a garbage collector? (answer TRUE or FALSE for each of the following).

   a. The garbage collector must be run as a privileged thread by the operating system

   b. There is no way to determine when garbage should be collected

   c. The garbage collector will slow down the program significantly

   d. There is no way to know the size (in bytes) of the objects once they have been allocated

   e. There is no way to know for certain if an object is reachable

   f. There is no way to know for certain that an object is garbage

5. (10 pts) Write a template function *has_vft* that takes an argument x of generic type T. The function should return true if T has at least one virtual function and should return false if T has no virtual functions. Hint: You should assume that T is a class (i.e., x will not be a base type, like int. x will be an object of some class). Hint: you might want to look at the sizeof(T). Hint: You might want to use inheritance and create another class.

6. (30 pts) Following the conventions of the C++ standard library (i.e., the STL), write a function or functions that return the smallest element in a container. There are some easy parts and some hard parts to this problem, please be sure to get the easy parts correct.
   a. Easy Part: declare the arguments to the function so that I can pass either an array or a data structure using the standard conventions of the C++ STL.
   b. Hard Part: declare the return type of the function correctly
   c. Easy Part: compare the elements using the less than operator (i.e., "<") by default
   d. Hard Part: allow me to specify my own comparison operator. Be sure to use the conventions of the C++ STL as far as defining and using comparison operators
   e. Medium Part: do not change the elements in the container.
   f. Easy Part: You should assume that the container has at least one element (i.e., don't worry about the container being empty, that won't happen).

7. (5 pts) Why is it that virtual destructors do not cause recursion? Consider the following example.

```
class Base {
public:
  virtual ~Base() { cout << "Base Destroyed\n"; }
};
class Derived : public Base {
  virtual ~Derived() { cout << "Derived Destroyed\n"; }
};
int main(void) {
  Base* p = new Derived;
  delete p;
}
```

   a. Only the Base class destructor is called
   b. Only the Derived class destructor is called
   c. The compiler automatically generates two destructor calls when p is deleted. The first destructor call uses the virtual function table (running the Derived destructor). The second destructor call uses static binding and calls the Base destructor
   d. The Derived class destructor is called. However, the Derived class destructor in turn calls (with static binding) the destructor for its Base class.

8. (5 pts) Define a class T and a variable x such that OK(x) compiles without error, but notOK(x) is an error. Be sure to define any classes you use.

```
template <typename T>
void OK(T& x) {}

template <typename T>
void notOK(T x) {}

class T {
// your class


};






int main(void) {
  // declare x


  OK<T>(x); // compiles and runs (doesn't do anything)
  notOK<T>(x); // generates a compile-time error
}
```

9. (5 pts) Define a class T and a variable x such that OK(x) compiles without error, but notOK(x) is an error. Be sure to define any classes you use.

```
template <typename T>
void OK(T x) {}

template <typename T>
void notOK(T& x) {}

class T {
// your class


};




int main(void) {
  // declare x


  OK<T>(x); // compiles and runs (doesn't do anything)
  notOK<T>(x); // generates a compile-time error
}
```

10. (12 pts) I have the following classes:

```
class Base {
public:
  virtual void prompt(void) = 0;
  void doit(void) {
    prompt();
  }
};
class Polite : public Base {
public:
  virtual void prompt(void) {
    cout << "please: ";
  }
};
class Rude : public Base {
public:
  virtual void prompt(void) {
    cout << "gimme a number: ";
  }
};
class VeryRude : public Polite {
public:
  virtual void prompt(void) {
    cout << "gimme a number, dammit: ";
  }
  void doit(void) {
    Base::doit();
  }
}
```

What happens when the following code is run (indicate the output on the lines provided).

```
int main(void) {
  Polite p;
  Rude r;
  VeryRude vr;
  Base* b;
  p.doit();       a.___________________________


  r.doit();       b.___________________________


  vr.doit();      c.___________________________

  b = &vr;
  b->doit();      d.___________________________
}
```