Spring 2013 EE380L Midterm: NAME:________________________

1. (12 pts) A container in C++ normally stores values. However, as long as we have a method to access the values, we don't really need to store anything. For this problem, I want to create a pseudo-container in C++ that produces the elements in a Fibonacci sequence (1, 1, 2, 3, 5, 8, 13, etc.). I'll provide more details later, but the primary goal is for the following program to print the first 20 elements in the Fibonacci sequence.

```
int main(void) {
   for (int64_t v : Fibonacci(20)) {
      cout << v << endl;
   }
}
```

As I'm sure you've surmised, the key to a pseudo container is the iterator. Write a bidirectional iterator with the minimum functions (simple dereference, pre-increment, pre-decrement, and equivalence), and a class Fibonacci so that the example above works. Your Fibonacci class will need a constructor that has an int argument, and the standard support required for iterators. The Fibonacci class and its iterator must have a constant amount of storage (no arrays or dynamically allocated storage).

2. (18 pts) NOTE, some of the functionality I'm asking you to create in this example is available through the typeinfo and run-time type identification support of C++-11. However, I want to test your problem solving and expertise with C++, so don't use typeid, typeinfo, etc, and solve this problem from scratch. I want to be able to extract the name of a type. This will possibly help me debug some template programs I write. As an examples of what I'm looking for, consider the following very simple template function. This function should print the name of type T, so if T is **int**, then the function should print "x is type: int".

```
template <typename T>
void fun(const T& x) {
   string s = typeString(x);
   cout << "x is type: " << s << endl;
}
```

This problem is in multiple parts, and the first part I've already completed for you. The first part deals with the common base types in C++, and for that, I've overloaded the typeString function in all the obvious ways. The overloads for int and double are shown below (assume there are overloads for all the basic types in C++).

```
string typeString(int) { return string("int"); }
string typeString(double) { return string("double"); }
// etc, etc.
```

a. (5 pts) Extend my method so that when T is int*, the typeString function returns a string "pointer to int", and when T is double*, the typeString function returns the string "pointer to double". In fact, extend my approach so that when T is any pointer type T*, the typeString function returns the string, "pointer to <type>" where <type> is replaced with the name of type T (note: your approach should work for T** types as well, "pointer to pointer to <type>").

b. (5 pts) I want the typeString function to work with class types, so if x is an object of type Foo, then typeString(x) should return the string “Foo”. The way I’m thinking about making this work is by adopting a convention that all classes will have a static function called myType. Assume that every class has this static method (shown below for class Foo). Extend my typeString function so that it works for class types (in addition to the basic types and pointer types).

```
class Foo {
public:
    static string myType(void) { return string(“Foo”); }
};
```

c. (3 pts) What happens if I have inheritance and a have a pointer type? Specifically, if I invoke typeString(p) where the pointer p is type Base* but p points to an object of type Derived (i.e., p points to a subclass). What will the return value be?

d. (5 pts) Write a companion function isTemp that I can call that will return true if its argument is a temporary, and will return false, if its argument is a real value.

```
bool shouldBeFalse, shouldBeTrue;
int x, y;
int *p;
```

In the following examples, isTemp would return true;

```
shouldBeFalse = isTemp(42);
shouldBeFalse = isTemp(x + y);
shouldBeFalse(new int);
```

In the following examples, isTemp would return false;

```
shouldBeTrue = isTemp(x);
shouldBeTrue = isTemp(p);
```

3. (21 pts) This is a true/false question. Indicate which of the following will correctly print “true”, by putting “TRUE” next to the cases that print “true” and “false” next to any case that does not print “TRUE”.

```
class Base {
public:
  virtual void doit(void) { cout << "TRUE\n"; }
  void foo(void) { doit(); }
};

class Derived : public Base {
   int x;
public:
  virtual void doit(void) { cout << "FALSE\n"; }
};

Base* array = new Derived[10];

template <typename T> void tfun1(T x) { x.doit(); }
template <typename T> void tfun2(T x) { x.foo(); }
```

a. array->doit(); ______________________

b. array->foo();______________________

c. (array + 1) ->doit();______________________

d. array[0].doit();______________________

e. array[1].doit();______________________

f. tfun1(*array); ______________________

g. tfun2(*array); ______________________

4. (15 pts) Using the conventions for generic functions in the Standard Template Library (STL), write a function that returns the smallest element in a data structure.
    a. (3 pts) What do you think the return type should be for your function? Explain

    b. Be sure that your solution works for most data structures, including Vectors or singly-linked lists.
    c. Be sure that your solution is generic, and allows the meaning of "smallest" to be defined by the user.
    d. Be sure that your solution provides a default meaning of smallest that is based on the less operator (i.e., x is smaller than y if "x < y" evaluates to true). The function should allow me to get this definition of "smaller" by default, and also allow me to create my own definition (please use the conventions of the STL).

Write your function here, and please satisfy the constraints for parts (a-d) in your solution.

5. (10 pts) In the code below, I want the message “burned” to be displayed exactly one time, and for this message to be displayed only after the loop has completed executing. With the code as written, I do not get that behavior (in fact, I have a double-delete bug and the program crashes). Add one or more new functions to class Container so that I get the desired behavior (do not change the pass function).

```
class HotPotato {
public:
   ~HotPotato(void) { cout << “burned\n”; }
}

class Container {
   HotPotato* pot;
public:
   Container(void) { pot = new HotPotato; }
};

Container pass(Container& c) { return std::move(c); }

main(void) {
   Container c;
   for (int k = 0; k < 10; k += 1) {
      c = pass(p);
   }
}
```

6. (12 pts) For each of the following cases, declare two pointers (e.g., “int* p; char* q) that will satisfy the constraint(s) given. If the type of your pointer depends on a class, please define the class(es) you use (e.g., “class Foo { }; Foo* p; int* q;”).
   a. Declare two pointers, p and q so that the assignment “p = q;” is legal, but “q = p;” is not a legal C++ statement (q = p must create a compile-time error.)

   b. Declare two pointers p and q so that the assignment “p = q;” is legal, and that after performing that assignment, the address stored in p is not the same as the address in q – i.e., “(void*) p != (void*)q”

   c. Declare two pointers p and q where *p = *q is a legal expression but sizeof(*p) != sizeoof(*q)

7. (4 pts) The goal of object-oriented programming is that you can write programs that work with objects without knowing the actual type of the object. The C++ language places constraints on the return types of virtual functions. Assume I have a virtual function defined in the Base class, and that function returns a pointer to some type T. I want to override this function in derived class. In the derived class, the function will return a pointer to type S. What relationship must exist between types S and T? (select one)
   a. S and T must be exactly the same type.
   b. S (the type in the derived class) must be a subtype of type T (from the base class)
   c. T (the type in the base class) must be a subtype of type S (from the derived class)
   d. S can be any type, if I override the function, I can return any type I wish.

8. (5 pts) In C++, destructors can be declared virtual. Personally, I think all destructors should be virtual. In fact, even classes that don't have destructors (classes for which the destructor is empty, i.e,. "T::~T(void) { }") can need their (empty) destructor to be virtual. Demonstrate why. Create a class (or classes) and create an instance of the class (i.e., declare and initialize a variable) that demonstrates incorrect behavior when the destructor is not virtual but which would have correct behavior when the destructor is virtual.

9. (3 pts) I have a function named foo. The function has one parameter named x. When the parameter x is declared with type const T& the program compiles and produces one behavior. When the parameter x is declared with type T, the function has a different behavior. Which of the following is a true statement about type T.
    a. T is an abstract type
    b. T is a base type for some other type S
    c. T is a subtype of some other type S
    d. T is a pointer