## Spring 2012 EE380L Midterm: NAME: Craig

1. (part 1 is 20 points) There are two parts to this question, and I'd like to score the two parts independently (i.e., you can get credit for one part even if you are unable to solve the other). Our goal is to pass parameter values to a function by name. As an example, we'd like to invoke the function with a syntax similar to the following – note I've added an extra pair of parentheses which make the problem easier as I'll explain in part 2:

```
doit((speed = 60, distance = 100));
```

For part 1 of this question, please assume that doit is a function that can take two parameters, both of type **int**. The two parameters are named *speed* and *distance*, and each parameter has a default value of 50. We want to build some infrastructure so that doit can be called by passing the argument for only one parameter, by name. So, both of the following should be legal invocations of doit.

```
doit(speed=100); // speed is 100, distance defaults to 50
doit(distance=20); // speed defaults to 50, distance is 20
```

I'd like you to solve this part of the problem by making "placeholder" global variables called "speed" and "distance". These variables can be any type you want and can have any operators defined for them that you think would be helpful. In addition, I want you to create a type called ParamProxy. The actual function doit will have one formal parameter of that type. i.e., the real doit function looks like this:

```
void doit(ParamProxy param) { ...
```

In the space that follows, define the global variables *speed* and *distance*, if those variables are classes, provide a class definition, including any constructors, destructors, methods or operators you need, and write the ParamProxy class. To keep things simple, please make the ParamProxy class have two public data members, one called spd and one called dst. In fact, I'll give you the starting point for ParamProxy. Be sure that when doit is invoked in the examples above, both spd and dst are set correctly.

**PLEASE TURN TO THE NEXT PAGE AND PLACE YOUR RESPONSE THERE**

```
// complete this class and/or change it as necessary
// also, define your placeholder global variables speed and distance along with any
// classes, operators, etc. you need
class ParamProxy {
public:
   int spd; // default value should be 50
   int dst; // default value should be 50
// write any constructors or any methods you need for this class
```

Notes:
I want the return type of op= for speed and distance to be a Param Proxy

```
ParamProxy::ParamProxy(void) { // constructor
    spd = dst = 50; // set defaults
}
Class SpeedType {
public
  ParamProxy operator=(int x) {
      ParamProxy result;
      result.spd = x;
      return result;
  };
  SpeedType speed; // speed placeholder
Class DistanceType {
public:
   ParamProxy operator=(int x) {
      ParamProxy result;
      result.dst = x;
      return result;
}; }
```

DistanceType distance;

2. (part 2 is 15 pts) For part 2 of this question, we're going to build some additional infrastructure so that both values speed and distance can be specified by name in a single call. However, the order in which the parameters are defined must not matter. So, both of the following should be legal invocations of doit, and should result in the same behavior!

```
doit((speed=100,distance=20));
doit((distance=20,speed=100));
```

We're going to solve this problem by overloading the comma operator and using techniques similar to expression templates. The extra pair of parenthesis ensures that the compiler understands that there is actually only one argument passed to doit, but that the argument is the return value from operator,(speed=100,distance=20). In this case, the type of that return value should be ParamProxy. Write the comma operator in the space provided below. Define any additional types you think you will need, and/or feel free to rewrite the types that you used for part 1 if those types are not appropriate for part 2 (I will grade parts 1 and 2 independently to the extent possible).

```
ParamProxy operator,(ParamProxy a, ParamProxy b) {
    ParamProxy defaults;
    if (a.spd == defaults.spd) {
        a.spd = b.spd;
    }
    if (a.dst == defaults.dst) {
        a.dst = b.dst;
    }
    return a;
}
```

Nothing changes from part 1.

3. (part 1 is 15 pts) One of the criticisms of the C++ standard library is that data structures like vector<T> cannot be "polymorphic". We'll have a chance to talk about why that's the case as we move forward into our object-oriented component of EPL. However, in the meantime, demonstrate your expertise with C++ templates to prove that this criticism is not entirely well founded. We'll start off simple. Imagine that we want to create a class Vector<T1, T2> with the following properties
   - The [] operator returns type T1& always (i.e., regardless of the type of object stored at the position)
   - The actual types T1 and T2 may be different sizes (sizeof(T1) != sizeof(T2)).
   - The Vector stores the array of objects using a pointer of type X* where X is either T1 or T2, whichever is larger.

   Write a typedef for X (using whatever template metaprogramming technique(s) you think are necessary). You do not need to write any of the other functionality, just the typedef for X. Note: you may want to define some additional classes and/or functions. Feel free to do so.

```
template<typename T1, typename T2>
class Vector {
  X* data;
  int length;
public:
  explicit Vector(int sz) {
     length = sz;
     data = new X[sz];
  }
   T1& operator[](int k) { return *((T1*) &data[k]); }

   /* put your typedef for X here */
```

need a sizeof template meta program

typedef typename Select<(sizeof(T1)<sizeof(T2)), T2, T1>::RET X;

};

```
template<bool B, typename T1, typename T2>
struct Select { typedef T1 RET; };
template<typename T1, typename T2>
Struct Select<false, T1, T2> {
    typedef T2 RET;
};
```

4. (part 2 is 10 pts) I want to build on the polymorphic vector from the previous question. Now, instead of limited the vector to operating on two types, I want to permit the Vector to work with a list of types. We can solve this problem using our template expertise as follows.
   - We'll use the same Vector<T1, T2> we used before. However, we'll create a specialization for when T2 is type List<H, Tail>.
   - The type List<H, Tail> can be used with two ordinary types, or can be used recursively to create arbitrarily long lists. List<E1, List<E2, List<E3, E4>>> is a list with four types E1, E2, E3 and E4.

   So, using this approach, if someone wanted to create a polymorphic Vector with T1 is Shape and the list of other types includes Circle, Square, and Triangle, they'd use:

```
Vector<Shape, List<Circle, List<Square, Triangle>>> myList(10);
```

   Write the typedef for X so that it correctly sets X to the type contained in the list that is the largest (e.g., if sizeof(Circle) is larger than sizeof(Square), sizeof(Triangle) and sizeof(Shape), then X would be set to Circle.

```
template <typename T1, typename H, typename Tail>
class Vector<T1, List<H, Tail>> {
   X* data;
// show me your typedef for X, write a class or struct for List<H, T> and write any
// other templates, classes, structs or functions you need to support your typedef.
```

typedef typename ListSelect<H, Tail>::RET ListX;

~~typedef typename Select<T1, ListX>::~~

typedef typename Select<(sizeof(T1) < sizeof(ListX)) ?

~~T1, ListX~~>::Ret X;

ListX, T1

same as Q3

}

template <typename H, typename T>

Struct ListSelect {

typedef typename Select<(sizeof(H) < sizeof(T)) ?

T, H>::~~RET~~ RET;

};

template <typename H, typename T1, ~~typename~~ T2>

struct ListSelect<H, List<~~H~~, T1, T2>>{

ListSelect on Back

rewriting List Select

```
template <typename H, typename T>
Struct ListSelect {
    typedef typename Select<(sizeof(H) < sizeof(T)) ?
                            T, H>::RET RET;
};

template <typename H, typename T1, typename T2>
Struct ListSelect<H, List<T1, T2>> {
    typedef typename ListSelect<T1, T2>::RET ListRET;
    typedef typename Select<(sizeof(H) < sizeof(ListRET)) ?
                            ListRET, H>::RET RET;
};
```

// List<T1, T2> is very boring

```
template<typename T1, typename T2>
Struct List {};
```

5. (10 pts) Some multiple guess questions related to memory management and garbage collection – for the questions below please assume that **malloc** and/or **free** refers to an ordinary implementation of memory management without garbage collection:
    a. In languages/run-time environments that permit the use of a copy-based garbage collector, which should I expect (choose one):
        i. The time required to call **new** is usually less than the time typically required for a C program to call **malloc** although some invocations of **new** will take far longer than **malloc**.
        ii. The time to call **new** is usually just a little slower than the time typically required for a C program to call **malloc** (i.e., **new** is not faster than **malloc**) and some invocations of **new** will take longer than **malloc**.
        iii. The time to call **new** is generally longer than the time typically required for a C program to call **malloc.**

    b. Reference counting suffers from which of the following (circle all correct statements).
        i. Reference counting can lead to deallocation errors where objects that are still reachable are incorrectly deallocated.
        ii. Reference counting can lead to memory leaks where garbage is not detected and therefore not reclaimed.
        iii. Reference counting requires the compiler to store additional information (i.e., the reference count) in each pointer, effectively doubling the size of a pointer.
        iv. Reference counting implementations are slow relative to other garbage collection techniques (i.e., slower than mark-and-sweep).

    c. When a memory heap becomes fragmented, performance can suffer. Which of the following is the main reason why we can't degragment the heap in a C++ program. For this question, let's assume that we have some perfect magical method for determining what is and what is not garbage. Even in that case, defragmenting the heap in C++ is impossible. Choose one reason why.
        i. Objects may be different sizes
        ii. A single object may have more than one pointer pointing at it
        iii. The heap is stored in virtual memory and some of that memory may be "paged out" to disk.
        iv. Root pointers cannot be relocated to the heap.

6. (10 pts) Write a template function ds_type(DS x) that takes a data structure as an argument (e.g., DS could be something like std::vector<T>). The ds_type function should print either "vector like" or "list like" depending on whether DS has random-access iterators. If DS has random-access iterators, print "vector like", in any other case, print "list like". You can use any function, type or convention found in the C++ standard library. You should also assume that DS is a data structure designed to follow the convections of the C++ standard.

note constructor call here

```
template<typename DS>
void ds_type(DS x) {
    if (is_random(std::iterator_traits<
                  DS::iterator>::iterator_category())) {
        cout << "vector like";
    } else {
        cout "list like";
    }
}

bool is_random(std::random_access_iterator_tag t) {
    return true;
}

template<typename T>
bool is_random(T t) {
    return false;
}
```

7. (10 pts) One of the examples of STL programming we did in class was the quicksort function and it's partition subroutine. Recall that partition needs to identify elements in a data structure that are smaller than some specific value. Recall also that the meaning of "smaller than" is something we'd like to determine using a parameter to the function. As a simplified version of partition, and to demonstrate your knowledge of the STL-style of programming define and write a template function that does the following

   - The template function should have parameter(s) that define a data structure
   - The template function should have parameter(s) that define a means for determining when an element of the data structure is "smaller than"
   - The template function should have a parameter *val*.
   - Write the function so that it prints (using cout) those elements in the data structure that are smaller than *val*.

```
template<typename I, typename F, typename T>
void doit(I b, I e, F f, T val) {
    while (b != e) {
        if (f(*b, val)) {
            cout << *b;
        }
        ++b;
    }
}
```

8. (10 pts) A few more short answer questions
   a. I'm writing a generic algorithm implementation of binary search. On random access iterators, I expect to get O(log N) time complexity for binary search. If a client program attempts to call binary search using iterators that are not random access what is most likely going to happen?
      i. The algorithm will run in O(log N) time, but will produce the wrong answer
      ii. The algorithm will run, but it will be slower than O(log N)
      iii. The algorithm will not compile

   b. C++ has this horrible feature inherited from C called pre-increment and post-increment operators (i.e., p++ and ++p are different). When writing an iterator, we usually give these two operators different return types. One operator returns type "iterator&" and the other operator returns type "iterator". Which operator (pre-increment or post-increment) returns type iterator& and why?

pre-increment returns iterator& 'cause it can. Post-increment cannot because it must make a copy (local variable) and returning a reference to a local is very bad.

   c. Here's a trivia puzzle for you. If we don't consider overloaded operators (e.g., operator++ doesn't count), what method can never be a member template in C++?

the destructor!

   d. Here's a tougher puzzle for you. Write a template function that has more template arguments than it has function arguments, but for which it is possible for the compiler to deduce all the template arguments (i.e., we can use implicit instantiation).

There are several solutions here's one ✓ two template args

```
template <typename Ret, typename Arg>
void doit (Ret f(Arg)) { // one param f
}
```