ECE326 – Fall 2019: Week 5 Exercise Questions

1. True or False [1 mark each]

Circle T is true, otherwise circle F for false.

1. Generic programming is a subset of metaprogramming. **T** **(F)**

   Neither is a subset of each other, but they do have overlaps.

2. If no deep copy is required (e.g. class has no pointer), move semantics performs no better than copy semantics. **(T)** **F**

3. If template specialization is not used (i.e. not instantiated), its code is not generated for the final executable. **(T)** **F**

4. For template `T foo()`, you can write `int a = foo()` to instantiate the function template `foo` with an `int` parameter. **T** **(F)**

   Have to write foo<int>() because C++ does not do type inference based on return type.

5. The new operator in C++ couples heap allocation and constructor invocation. **(T)** **F**

2. Short Answers

1. Use container_of to return a pointer to the parent object of member field base. **[2 marks]**

```
struct base {
  int x, y, z;
};

struct derived {
    int a;
    struct base b;
    char c[10];
};

struct derived * get_derived(struct base * ptr) {

    return container_of(ptr, struct derived, b);


}
```

2. Implement binary search algorithm using a function template, assume the array is sorted and return -1 upon not found. **[5 marks]**

```
template<typename T>  /* find index of val in array of size n */
int binary_search(const T & val, T * array, int n) {

     int top = n-1;
     int bot = 0;

     while (bot <= top) {
          int mid = (top + bot)/2;

          if (array[mid] == val)
               return mid;
          else if (array[mid] < val)
               bot = mid+1;
          else
               top = mid-1;
     }

     return -1;

}
```

3. Implement a template class named Triple that is a tuple of 3 elements of the same type. Overload enough operators so that binary search template you implemented above can be instantiated for Triple. Use lexicographical order. **[8 marks]**

```
template<typename T>
struct Triple {
     T a, b, c;

     Triple() : a(0), b(0), c(0) {}

     Triple(T && a, T && b, T && c)
       : a(std::move(a))
       , b(std::move(b))
       , c(std::move(c))
     {}

     bool operator==(const Triple<T> & rhs) {
          return a == rhs.a && b == rhs.b && c == rhs.c;
     }

     bool operator<(const Triple<T> & rhs) {
          if ( a < rhs.a )
               return true;
          else if ( a > rhs.a )
               return false;
          else if ( b < rhs.b )
               return true;
          else if ( b > rhs.b )
               return false;
          else if ( c < rhs.c )
               return true;
          /* c >= rhs.c */
          return false;
     }
};
```

3. Generic Programming **[10 marks]**

Create a generic Queue class without using templates. Implement the Queue using a singly linked list, with the member functions, `push_back`, that pushes new elements to end of the queue, `front`, which returns the first element of the queue, and `pop_front`, which removes the first element of the queue.

```
class Queue {
     struct Node {
          Node * next;
          void * data;

          Node(void * data, Node * next=nullptr)
               : next(next) , data(data) {}
          ~Node() { /* managed by Queue */ }
     } * head, * tail;
     void (* dest_f)(void *);

public:
     Queue(void (* destroy)(void *))
          : head(nullptr)
          , tail(nullptr)
          , dest_f(destroy)
     {}

     ~Queue() {
          Node * curr = head;
          while (curr != nullptr) {
               Node * temp = curr;
               curr = curr->next;
               dest_f(temp->data);
               delete temp;
          }
     }


     void * front() {
          if (head == nullptr) {
               return nullptr;
          }
          return head->data;
     }
```

```
     bool push_back(void * data) {
          Node * node = new Node(data);

          if (node == nullptr) {
               return false;
          }

          if (head == nullptr) {
               head = tail = node;
          }
          else {
               tail->next = node;
               tail = node;
          }

          return true;
     }

     bool pop_front() {
          Node * node;

          if (head == nullptr) {
               return false;
          }

          node = head;
          head = head->next;
          if (head == nullptr) {
               tail = nullptr;
          }

          delete node;
          return true;
     }
};
```

Note that you are not expected to write this entire class during a midterm. This was more for you to write some code.

4. Template Programming **[10 marks]**

Using the generic Queue made in Question 3, write a FIFO class template, which allows type-safe use of the generic Queue class for any parameterized type. Use move semantics for `push_back` instead of copy semantics.

```
template<typename T>
class Fifo : private Queue {
     static void destroy(void * ptr) {
          delete (T *)ptr;
     }

public:
     Fifo() : Queue(&Fifo<T>::destroy) {}

     bool push_back(T && elem) {
          return Queue::push_back(new T(std::move(elem)));
     }

     T * front() {
          return (T *)Queue::front();
     }

     bool pop_front() {
          return Queue::pop_front();
     }
};
```

You would be expected to do something like this during the midterm, where the line of code you have to write is about 2 lines per 1 mark (assuming you write efficient code). You can write this same template class using composition instead of inheritance. Give that a try.