Question 1. True or False

Circle **T** if the statement is true, otherwise circle **F** if the statement is false.

1. Generic programming is a subset of metaprogramming

T (F

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

T) F

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

(F)

4. Default template arguments must be compile-time constants.

r) F

5. An iterator always requires an associated iterable.

r (F

Question 2. Short Answers

a) List 3 differences between Rust generics and C++ templates.

C++ template parameters can be anything, Rust type parameters can only be types.

Rust generics have type bounds, C++ templates do not.

C++ templates can have default arguments, Rust generics do not.

C++ templates allow specializations to have completely different interface and implementation. When you implement a Rust trait, it must implement a common interface (i.e. the trait methods).

b) Explain the difference between iter(), iter_mut(), and into_iter().

iter() creates an iterator that iterates by immutable borrow.

iter_mut() creates an iterator that iterates by mutable borrow.

into_iter() creates an iterator that moves the iterable into the iterator (i.e, takes ownership).

Question 3. Programming Questions

1. In Rust, create a generic struct named Factorial and implement the iterator trait for it.

```
struct Factorial<T> {
    curr: T,
    index: usize,
}

impl<T> Iterator for Factorial<T>
    where T: Copy + AddAssign + MulAssign + From<usize> {
    type Item = T;

    fn next(&mut self) -> Option<T> {
        let ret = self.curr;
        self.index += 1;
        self.curr *= T::from(self.index);
        Some(ret)
    }
}
```

2. Implement the binary search algorithm using a C++ function template, assume the array is sorted and return -1 upon not found.

```
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) {</pre>
          int mid = (top + bot)/2;
          if (array[mid] == val)
               return mid;
          else if (array[mid] < val)</pre>
               bot = mid+1;
          else
                top = mid-1;
    }
    return -1;
```

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

```
template<typename T>
struct Triple {
  T a, b, c;
   Triple() : a(0), b(0), c(0) {}
   Triple(const T & a, const T & b, const T & c)
     : a(a), b(b), c(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 )</pre>
          return true;
     else if ( b > rhs.b )
          return false;
     else if ( c < rhs.c )</pre>
          return true;
     /* c >= rhs.c */
     return false;
};
```

4. Write a C++ template class, Pair, which accepts one template parameter, T, and creates a class which has two fields, x and y, both of type T. Implement a constructor that takes two parameters to initialize both x and y, and implement a member function named get which returns a pointer to x if the single integer argument is 0, pointer to y if the argument is 1, and nullptr for all other argument values. You are required to implement the member function get outside of the body of the template class (i.e. you may not inline the member function inside the class).

```
template<typename T>
class Pair {
   T x, y;
public:
   Pair(T x, T y) : x(x), y(y) {}
   T * get(int i);
};

template<typename T>
T * Pair<T>::get(int i) {
   if (i == 0) return &x;
    else if (i == 1) return &y;
    return nullptr;
}
```

5. Create a generic Queue class 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.

```
See queue.cpp
```

6. Write a recursive template named SumSquare that will calculate the sum of squares of consecutive integers from 1 to N. e.g. SumSquare<5>::value should be 55.

```
template<int F>
struct SumSquare {
    enum { value = F * F + SumSquare<F - 1>::value };
};

template<>
struct SumSquare<1> {
    enum { value = 1 };
};
```