Assignment 1

Name: Arun Krishnakumar Rajagopalan
Date: 09/18/2015

1. Write a class template that implements a generic container triple that can hold three elements of different types.

    - The triple template must be default constructible, and have a 3-argument constructor, one argument for initializing each element.
    - There must be a mechanism to access each element (name elements first, second and third) and the type of each element (first_type, second_type, third_type).
    - Two triples must be copy constructible, assignable, and comparable with the equality (==) and inequality (!=) operators.

    Also, write a 3-argument function template make_triple, that creates a triple from its arguments. Write a program that tests your templates, compile, and run it. For example, the following code should work:

    ```
    triple<int, int, char> a;
    triple<int, int, char> b(1, 2, 'a');
    triple<int, int, char> c = b;
    a = b;
    assert(a.first == 1);
    assert(b.first == 1);
    assert(c.first == 1);
    typedef triple<int, int, char>::first_type t; // t is int
    assert(b == make_triple(1, 2, 'a'));
    ++b.first;
    assert(a != b);
    ```

    Are there any types that your triple template cannot handle, that is, types that cannot be used as type arguments to triple?

    Ans - *triple* supports the following operations on each of its template types

    - prefix ++ operator
    - == operator
    - ! = operator

    While you can construct an a triple with any set of types, if you want to perform any of the above mentioned operation, these operators will need to be overloaded for each of the types.

2. Add a zero-argument member function *is_zero* in the triple class (to familiarize yourself with template syntax, write only the declaration of *is_zero* in the class body, and the definition outside the body). The *is_zero* function should return true if all elements are equal to 0. Is adding such a function a good/bad idea? Is the triple template still usable with arbitrary types? If it is, to what extent?

    Ans - No it is not a good idea to add such a function. This is because triple can be constructed with any set of types and not all types can compare with '0'. Even if it can compare, it might not be what the user might expect. For example, checking if an Array object *is_zero* might mean different things - is the size zero or are all elements zero?

The triple template is still usable with arbitrary type. However, if you try to call the ‘is_zero’ function, and if any of the types don’t implement a comparison with ‘int’ (or if it can’t be implicitly converted), then it will result in a compiler error.

Using triple with ‘char’, ‘long’, ‘float’ etc will work correctly, but it is not guaranteed with all arbitrary types.

3. Consider the template function

```
template <class T>
inline T sum_all(T* first, T* last) {
T sum;
for (sum = 0; first != last; ++first)
    sum += *first;
    return sum;
}
```

Determine whether the template can be instantiated with:

- type int
- type std::string
- type void
- a pointer type

For each bullet, explain why or why not. List all the operations that must be supported on the parameter type T for instantiation to succeed.

Ans - The various operations that must be supported to correct instantiate the function are -

- ’=’ assignment operator
- ’++’ pre-increment operator
- ’+’ addition operator

There are two situations for each type that need to be checked - first, whether the code compiles or not, second if the expected behavior is achieved.

- **type int -** The program compiles fine, but the implementation does not work. When I attempted to run the compiled program, I encountered a segmentation fault. This is because when ++first is called, the pointer gets incremented and may point to an invalid address.
- **type std::string -** The program does not compile since the compiler cannot figure out whether 0 is to be interpreted as a char or an int in ‘$sum = 0$’. If std::string provided a specialization for ‘=’ that accepted either char or int, then this statement would be correct and the code would compile. However, at runtime we will still have the same error as for the previous case.
- **type void -** The program does not even compile since ‘T sum’ cannot be initialized as a void type.

- **a pointer type -** The program does not compile since the '+' operator is illegal for two pointers.

4. Write a template function

   ```
   template <class T> void exchange(T& x, T& y);
   ```

   that swaps the values of its arguments (we don't call the function swap so you don't have to worry about name clashes with std::swap). Write an exchange overload that swaps ints without creating a temporary variable (Hint: The code xˆ=yˆ=xˆ=y does the trick.) The std::vector class has a member function void swap(vector& other);. Provide an overload of exchange that accepts std::vectors, and performs the exchanging using std::vectors swap member. Remember that vector has two template parameters. Explain why the standard library provides, in addition to the generic swap function, a member function swap in the vector template. How would you overload exchange for triples to take advantage of your other exchange overloads? Write a test program that assures you that the correct exchange functions get called in each different case.

   Ans - The standard library provides '$std :: swap$'. $std :: vector$ provides a specialization of this standard swap because it is optimized for that specific container. Several containers provide specializations for swap for this very reason.

   Exchange can be overloaded for triples to take advantage of the other exchange overloads by calling exchange for each member. This will in turn get routed to the right overloaded exchange function.