Osbaldo Gonzalez Jr. (Group 4)
Checked by:
Luwei Tan
Jerry Li (Group 2)

Template Worksheet

1. Why is the keyword typename preferable to the use of the keyword class in a template definition?

It is more obvious as to what you are trying to do, by conveying the idea that this type is subject to change. If you follow the convention to have class use the class keyword only, and templates use the typename keyword only, then it is also a lot easier to see where all of the templated functions and classes are with just a quick search (ctrl-f).

2. Identify the errors in the following template function headers

a. `template<typename T> f(int a)`

No return type. Although using the typename T is not required, probably meant to use it somewhere, either as return type or for the parameter a.

b. `template<typename T, typename T> f(T* a)`

Again, no return type. Can't have two type names with the same variable name; the compiler can no longer distinguish what to use.

c. `template<typename T1, typename T2> f(T1 a)`

No return type, but it is fine having another type just because. Won't be of any use (traditionally), but no compiler errors.

3. Why do you have to specify template parameters when you instantiate a class template, but not when you instantiate a function template?

Because there is no context as to what type you want to use for the class, you are just declaring a new object. When you call a templated function, the compiler can figure out basically which function to overload, and the type is implicitly deduced. Ofcourse, sometimes it is not obvious to the compiler, per say using the template parameter to change the return type of a function, in which case you would have to specify the template parameter. And the same goes for a class: if you have a default template argument then there is no need to specify a template; the compiler will just assume you want to use the default template.

4. How is a template class different from a template function?

In a template class, not only does the class definition need to be declared with a template argument, but the implementation of it's methods (functions) also need to be declared with a typename. With functions, you only need to do it once; there are no methods.

5. How is a non-typename template parameter different from a type name parameter? When might you use such a value?

Non-typename template parameters can not be used to change the type of anything in a function or class, and just serve to provide information. A typename parameter allows for the types to change. A good example of this is the std::array class, which requires you to define a type and a size for the array. std::array<int, 42> test; Here 42 is merely a parameter used to set array size, but int is used to define what kind of array we want to use.

Merge sort and Binary Search

6. Rewrite the merge sort algorithm as a template function.

```
//util.h
 #ifndef UTIL_H
#define UTIL_H

#include <vector>

using namespace std;

template<typename T>
void swap(T& x, T& y)
{
   T temp = x;
   x = y;
   y = temp;
}

template<typename T>
void print(vector<T> a)
{
   for (int i = 0; i < a.size(); i++)
      cout << a[i] << " ";
   cout << "\n";
```

```
}


/**
   Sets the seed of the random number generator.
*/
void rand_seed();

/**
   Computes a random integer in a range.
   @param a the bottom of the range
   @param b the top of the range
   @return a random integer x, a <= x and x <= b
*/
int rand_int(int a, int b);

#endif




//util.cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

#include "util.h"
/**
   Sets the seed of the random number generator.
*/
void rand_seed()
{
   int seed = static_cast<int>(time(0));
   srand(seed);
}

/**
   Computes a random integer in a range.
   @param a the bottom of the range
```

```
   @param b the top of the range
   @return a random integer x, a <= x and x <= b
*/
int rand_int(int a, int b)
{
   return a + rand() % (b - a + 1);
}
```

```
//mergesort.cpp
#include <iostream>

#include "util.h"

/**
   Merges two adjacent ranges in a vector
   @param a the vector with the elements to merge
   @param from the start of the first range
   @param mid the end of the first range
   @param to the end of the second range
*/
template<typename T>
void merge(vector<T>& a, int from, int mid, int to)
{
   int n = to - from + 1; // Size of the range to be merged
   // Merge both halves into a temporary vector b
   vector<int> b(n);

   int i1 = from;
      // Next element to consider in the first half
   int i2 = mid + 1;
      // Next element to consider in the second half
   int j = 0; // Next open position in b

   // As long as neither i1 nor i2 is past the end, move the smaller
   // element into b
```

```
   while (i1 <= mid && i2 <= to)
   {
      if (a[i1] < a[i2])
      {
         b[j] = a[i1];
         i1++;
      }
      else
      {
         b[j] = a[i2];
         i2++;
      }
      j++;
   }

   // Note that only one of the two while loops below is executed

   // Copy any remaining entries of the first half
   while (i1 <= mid)
   {
      b[j] = a[i1];
      i1++;
      j++;
   }
   // Copy any remaining entries of the second half
   while (i2 <= to)
   {
      b[j] = a[i2];
      i2++;
      j++;
   }

   // Copy back from the temporary vector
   for (j = 0; j < n; j++)
      a[from + j] = b[j];
}

/**
   Sorts the elements in a range of a vector.
```

```
   @param a the vector with the elements to sort
   @param from start of the range to sort
   @param to end of the range to sort
*/
template <typename T>
void merge_sort(vector<T>& a, int from, int to)
{
   if (from == to) return;
   int mid = (from + to) / 2;
   // Sort the first and the second half
   merge_sort(a, from, mid);
   merge_sort(a, mid + 1, to);
   merge(a, from, mid, to);
}

int main()
{
   rand_seed();
   vector<int> v(20);
   for (int i = 0; i < v.size(); i++)
      v[i] = rand_int(1, 100);
   print(v);
   merge_sort(v, 0, v.size() - 1);
   print(v);
   return 0;
}
```

7. Rewrite the binary search algorithm as a template function.

```
#include <iostream>

//bsearch.cpp
#include "util.h" //same util as insertion sort

/**
   Finds an element in a sorted vector.
   @param v the sorted vector with the elements to search
   @param from the start of the range to search
   @param to the end of the range to search
   @param value the value to search for
```

```
   @return the index of the first match, or -1 if not found
*/
template <typename T>
T binary_search(vector<T> v, int from, int to, int value)
{
   if (from > to)
      return -1;
   int mid = (from + to) / 2;
   if (v[mid] == value)
      return mid;
   else if (v[mid] < value)
      return binary_search(v, mid + 1, to, value);
   else
      return binary_search(v, from, mid - 1, value);
}

int main()
{
   rand_seed();
   vector<int> v(20);
   v[0] = 1;
   for (int i = 1; i < v.size(); i++)
      v[i] = v[i - 1] + rand_int(1, 10);

   print(v);
   cout << "Enter number to search for: ";
   int n;
   cin >> n;
   int j = binary_search(v, 0, v.size() - 1, n);
   cout << "Found in position " << j << "\n";
   return 0;
}
```