CME212, Winter 2020
Advanced Software Development for Scientists and Engineers

Paper Exercise 3
Due Tuesday, February 25th at 4:30 P.M. P.S.T.
Name: Chih-Hsuan (Carolyn) Kao
SID: 06366163; chkao831@stanford.edu

Question 0: Lecture 5 Our graph is undirected, but we could use it to build a *directed* graph or *digraph*. A digraph's edges are *ordered* pairs of nodes (n_i, n_j), rather than unordered pairs $\{n_i, n_j\}$; all operations look like their undirected counterparts and have similar complexity.

Here's one digraph representation using our graph class.

```
template <typename V>
class Digraph {
  ...
 private:
  Graph<V> g_;
};
```

Explain the representation by giving an abstraction function and representation invariant. Note that `Digraph` is not `Graph`'s friend and can only use `Graph`'s public interface.

Note: To be concrete, we are asking: How can you use an undirected graph to construct (or represent) a directed graph? This alludes to the representation. Are there "any assumptions" used during your construction? If so, these may be necessary invariants.

Note: The representational invariants and the abstraction functions should not refer to the way you implemented your graph class but only to the abstract concept of an undirected graph.

Solution:

For a directed graph (aka digraph), I would redefine the concept of nodes such that each original node now "splits" to two things–an outgoing one with its edge coming out; an incoming one with its edges coming in. This applies to every node in the new digraph implementation. To specify further, `num_nodes()` doubles its returned value and would certainly yield even value–which is a critical assumption that serves as an invariant as I would mention below.
AF, the abstraction function, maps the representation of a class to the corresponding abstract value, allowing us to bridge between the more abstract specifications provided by the comments and what actually happens in the code.
AF(Digraph) = $< Node, Edge >$

Node is a vector of the even nodes which represents/captures `node_value`.
Edge is a set of all ordered pairs of nodes in the graph.

Specifically,
AF(Directed_Graph) = $< Node, Edge >$
where
Node = $[n_0, n_1, ..., n_{k-1}]$, k = `ptr_to_graph_.num_nodes()/2`;
in which n_i represents node info:
`ptr_to_graph_.node(2*i).position()` //node position of Point type
`ptr_to_graph_.node(2*i).index()` //node index of size_type
`ptr_to_graph_.node(2*i).value()` //node value of node_value_type

Edge = set of all pairs (n_i, n_j) s.t.
`ptr_to_graph_.has_edge(ptr_to_graph_.node(i*2),ptr_to_graph_.node(j*2+1))`

RI, the Representation Invariant, which helps prove that data structure operations are correct, entails our assumption here. As mentioned,
RI = `ptr_to_graph_.num_nodes()` % 2 == 0 //enforced to be even

Question 1: Lecture 5 The `vector<T>::resize` operation extends a vector with *memory initialized* elements. (For example, after `vector<int> v; v.resize(20)`, `v` contains 20 zeroes in memory.) This is almost always what we want, but it can be useful to extend a vector with *uninitialized memory* so that we do not take up unnecessary space. Here's an example: uninitialized memory helps build a *sparse vector*, in which elements are initialized only when first referenced. We assume a special `garbage_vector` type whose `resize` method does not initialize new memory. **See next page**
Note: This question will not completely make sense until you understand what the code is doing below!

```
template <typename T>
class sparse_vector {
  garbage_vector<size_t> position_;
  vector<size_t> check_;
  vector<T> value_;
 public:
  // Construct a sparse vector with all elements equal to T().
  sparse_vector() {}
  // Return a reference to the element at position @a i.
  T& operator[](size_t i) {
    // If out of bounds, we must re-size our array and
    //add (uninitialized) memory!
    if (i >= position_.size())
      position_.resize(i + 1);
    // If we haven't initialized the memory yet, go ahead and do so now.
    if (position_[i] >= check_.size() || check_[position_[i]] != i) {
      position_[i] = check_.size(); // 1st ref to the element at position i.
```

```
18        check_.push_back(i);           // We associate an index...
19        value_.push_back(T());         // ...alongside a value.
20      }
21      return value_[position_[i]];
22    }
23  };
```

An abstract `sparse_vector` value is an **infinite** vector.

Write an abstraction function and representation invariant for `sparse_vector`.

Solution:

From the code above, we see that it's basically the abstraction that bridges the representation of the `sparse_vector` and the `sparse_vector` value. Also, we also know that this abstract `sparse_vector` is an infinite vector, whose elements (i.e. values of T type) are all abstract elements.
AF, allowing us to bridge between the more abstract specifications provided by the comments and what actually happens in the code, is simply the infinite vector that represents value. Specifically speaking,
AF $= [v_0, v_1, ...]$, where
1. $v_i =$ `value_[position_[i]]` IF `i < position_.size()`;//line 20
AND IF `[position_[i]] < check_.size()` //line 14
AND IF `check_[position_[i]] = i` //line 14
2. $v_i =$ `T()` otherwise //line 18

RI = `value_.size() == check_.size()` //line 17 and 18
Also, for every element e within `check_`, i.e. for $e \in [0,$`check_.size()`$]$,
as seen from line 16, we would need to have
`check_[k] < position_.size()` AND `position_[check_[k]] = k`

Question 2: Lecture 12 The `CME212/BoundingBox.hpp` is kind of interesting. The following code will create a bounding box that encloses our entire graph:

```
template <typename V, typename E>
Box3D graph_bounding_box(const Graph<V,E>& g) {
  auto first = graph.node_begin();
  auto last  = graph.node_end();
  assert(first != last);
  Box3D box = Box3D((*first).position());
  for (++first; first != last; ++first)
    box |= (*first).position();
  return box;
}
```

This will be very important for homework 4. Interestingly, `Box3D` also provides a constructor that takes a range of `Points` and performs the above operation for us. So, instead, we could write

```
template <typename V, typename E>
Box3D graph_bounding_box(const Graph<V,E>& g) {
  return Box3D(g.node_begin(), g.node_end());
}
```

except that this doesn't work because we're passing it `Node` iterators instead of `Point` iterators. The Boost and Thrust libraries both provide "fancy iterators", one of which is the `transform_iterator`. See https://github.com/thrust/thrust/tree/master/thrust/iterator for documentation.

Use `thrust::transform_iterator` to correct the above code that uses `Box3D`'s range constructor.

Solution:

```
#include <thrust/iterator/make_transform_iterator.h>

/**
  * @brief A functor that gives us node's position.
  *
  * This struct should inherit from unary_function as instructed
  * from thrust::make_transform_iterator template documentation.
  */
template <typename V, typename E>
struct node2position : public thrust::unary_function<Node,Point>{
    Point operator()(const Node& n){
        return n.position();
    }
};//end functor struct

/* An implementation that returns Box3D that takes a range of Node
position and perform equivalent operation */
template <typename V, typename E>
Box3D graph_bounding_box(const Graph<V,E>& g) {
    using Node = typename Graph<V,E>::node_type;
    return Box3D(
    thrust::make_transform_iterator(g.node_begin(), node2position()),
    thrust::make_transform_iterator(g.node_end(),node2position()));
}//end implementation
```

Question 3: Lecture 12 In fact, it is possible to implement your `Graph::node_iterator` as a `thrust::transform_iterator`. That is, we could delete `NodeIterator` entirely and write

```
using node_iterator = thrust::transform_iterator<__, __, Node>;
```

The third parameter simply explicitly states that the `value_type` should be `Node`. Fill in the two blanks and explain your answer.

Solution:

In the implementation below, `node_iterator` transforms every element of `uid_type` into a `Node`. I firstly construct a struct functor; then, I implement `thrust::transform_iterator` as instructed. Finally, the begin() and end() iterators are implemented for post-transformation item. The codes would roughly look like

```
1  #include <thrust/iterator/transform_iterator.h>
2
3  /**
4    * @brief A functor that gives us node object by uid.
5    *
6    * As instructed from thrust::transform_iterator template documentation,
7    * using an optional template argument by specifying the result_type
8    * of the function, this functor does not need to inherit from
9    * unary_function as I did above.
10   *
11   * Also, I assume the whole implementation to be occur within
12   * Graph.hpp scope, so I use "this" to specify graph on line 31 and 34.
13   *
14   * Note that the type of my uid in Graph.hpp is size_type.
15   */
16 struct uid2nodeobj {
17     Node operator() (size_type uid){
18         return Node(ptr_to_graph_,uid);
19     }
20     /* attribute: pointer to graph */
21     const graph_type* ptr_to_graph_;
22 }
23
24 /* Implement node_iterator using thrust::transform_iterator */
25 using node_iterator =
26 thrust::transform_iterator<uid2nodeobj, //function obj
27                            std::vector<size_type>::const_iterator, //iterator
28                            Node> //Node is the resulting_value_type
29
30 /* Specify iterator range after transformation w/graph pointer */
31 node_iterator node_begin() const {
32     return node_iterator(vec_activenodes_.begin(), uid2nodeobj{this});
33 }
34 node_iterator node_end() const {
35     return node_iterator(vec_activenodes_.end(), uid2nodeobj{this});
36 }
```

Question 4: Lecture 10 We may want to have multiple initialization methods for a class that basically do the same thing but have slightly different interfaces. It would be inefficient to duplicate the code in both constructors. Therefore since C++11 you can use delegate constructors, where one constructor calls another.

Imagine we want to write a class to contain the scores of a test or assignment. The class will be constructed from a vector containing the scores. However depending on the assignment this might be a vector of letter grades or a vector of percentages. Instead of having to write two entire constructors we want to have one constructor delegate to another.

(a) Write a functor that takes in a letter grade from the list [A,B,C,D,E,F] and returns an int with the equivalent percentage of that grade.

Solution:

```
class char2int_grade_conversion {
    public:
        /* Operator */
        int operator() (char c) {
            if (c >= 'A' && c <= 'F'){
                switch(c){
                    case 'A':
                        numbergrade = 90;
                        break;
                    case 'B':
                        numbergrade = 80;
                        break;
                    case 'C':
                        numbergrade = 70;
                        break;
                    case 'D':
                        numbergrade = 60;
                        break;
                    case 'E':
                        numbergrade = 50;
                        break;
                    case 'F':
                        numbergrade = 40;
                        break;
                    default:
                        numbergraph = 0;
                        break;
                }//end switch
            } else {
                std::cout << "Invalid lettergrade input: " << c << std::endl;
                numbergrade = 0;
            }
            return numbergrade;
        }

    private:
```

```
        /* private attribute */
        int numbergrade;
};
```

(b) We want to delegate to the constructor that takes in the start and end of a vector of percentages with the following interface.

```
using intIter = std::vector<int>::iterator
using charIter = std::vector<char>::iterator
class Grades
{
  public:
  Grades(intIter start, intIter end){
    // normal construction - pretend this is already implemented
  }
}
```

Write a delegating constructor that takes in a `std::vector<char>` and your functor and delegates to the constructor above.

Solution:

```
#include <thrust/iterator/make_transform_iterator.h>

using intIter = std::vector<int>::iterator;
using charIter = std::vector<char>::iterator;

...//Here, we have the functor from part (a)

class Grades {
    public:
        /**
         * @brief An ordinary constructor that takes in start/end
         * vector of percentage (int type)
         */
        Grades(intIter start, intIter end){
            // normal construction - pretend this is already implemented
        }

        /**
         * @brief A delegating constructor that  takes  in  a
         * std::vector<char> and the functor in part (a)  then delegates to
         * the constructor above.
         *
         * Note: Here, I preliminarily assume that the functor f passed-in
         * below has already been initialized elsewhere (e.g. in main())
         * outside of the scope of this class.
         */
        Grades(std::vector<char>& v,
```

```
                 char2int_grade_conversion f)
               : Grades{
                   thrust::make_transform_iterator(v.begin(),f),
                   thrust::make_transform_iterator(v.end(),f)
                   } {};
};
```

(c) You could also get rid of duplicate code with an `init()` method that is called by multiple constructors. What is a potential issue with this implementation?

Solution:

When we define an `init()` function, having a constructor followed by an Initialize call is error prone, which is generally not ideal. Another factor lies in the dynamically allocated memory – `init()` functions can be called by anyone at any time, such that the dynamically allocated memory may or may not have already been allocated when `init()` is called. This would introduce confusion.