CME212, Winter 2020
Advanced Software Development for Scientists and Engineers

Paper Exercise 2
Due Tuesday, February 11th by 4:30 P.M. PST via GradeScope

Chih-Hsuan (Carolyn) Kao chkao831@stanford.edu

Question 0: Write an implementation of `compute_median` below that computes and returns the median value of an array of `doubles`. The implementation should have an average time complexity no worse than $\mathcal{O}(n)$ and are only allowed to use functionality available in the header files `<algorithm>` and `<vector>`. For an even number of values, return the mean of the two middle values. HINT: Find a more general function in `<algorithm>` that you can map this `compute_median` onto.

```cpp
#include <algorithm>
#include <vector>

/** Compute the median of an array of doubles.
 * @param[in] values  The array of values.
 * @param[in]      n  The number of values in the array.
 * @result The median value of the array.
 *
 * @pre @a n > 0
 * @post Let sorted_v be @a values sorted by op<.
 *         If @a n odd,  result ==  sorted_v[n/2]
 *         If @a n even, result == (sorted_v[n/2] + sorted_v[n/2-1])/2
 */
double compute_median(const double* values, unsigned n)
{
    //copy the array of values into a new vector of doubles
    std::vector<double> vec(values, values+n);
    //random access iterator defining the sort partition point
    auto iter = vec.begin() + n/2;
    //defining the range sort
    std::nth_element(vec.begin(), iter, vec.end());
    //distinguish odd and even cases
    if (n % 2)
    {
        return *iter;
    } else {
        std::nth_element(vec.begin(), iter - 1, iter);
        return (*iter + *(iter - 1))/2;
    }
}
```

Question 1: Lecture 4 Consider a data type U with abstract type $U = \{u_0, \ldots, u_{n-1}\}$, where each u_i is an integer. U's iterator iterates over the values in *any* order. Assume U has been implemented with a `std::vector<int> value_` that contains all the elements.

Write an implementation of `U::iterator U::erase(U::iterator it)` that has $O(1)$ time complexity. The specification is not necessary and we emphasize that the returned iterator references an unordered collection of elements.

```
/** An implementation of U::iterator U::erase(U::iterator it)
 * that reduces the data type U size from its end.
 *
 * Complexity: O(1)
 */
U::iterator U::erase(U::iterator it) {
    if (it != end()){
        //copy the last element to the iterator it
        *it = value_.back();
        //at the last element, pop it off to erase
        value_.pop_back();
    }
    //next iterator--not invalidated
    //it references an unordered collection of elements
    return it;
}
```

Question 2: Lecture 4 Use the `erase` method you wrote above to write a method with the following specification. Never use an invalid iterator—all iterators to a `vector` become invalid after any `insert` or `erase`—and your method should work for other STL containers, such as `map` and `list`.

```
/** Erase all elements of @a x for which @a pred returns true.
 * @param[in,out] x Container of elements.
 * @param[in] pred Predicate that takes an element and returns a bool.
 *
 * @post new @a x.size() <= old @a x.size()
 * @post new @a x contains exactly those elements of e of old @a x for
                  which pred(e) returned false
 */
template <typename Container, typename Predicate>
void erase_if(Container& x, Predicate pred){
    auto iter = x.begin()
    while(iter != x.end()){
        if(pred(*iter)){
            //set iter to the return type
            //erasing the predicate and invalidating iterator
            iter = x.erase(iter);
        } else {
            ++iter;
        }
    }
}
```

Question 3: Lecture 6 Remember the `Book` class from Exercise 1?

```
class Book {
  public:
  Book(std::string filename)
  {
    std::string sent;
    std::ifstream book_file(filename);
    while(std::getline(book_file, sent))
    {
      book_.push_back(split(sent, ' '));
    }
    book_file.close();
  }
};
```

There is a problem with this class, it does not implement RAII. There are two points where we need to fix this.

a

In the constructor if anything goes wrong in the `while` loop the error will immediately propagate up the stack and the `close()` function will never be called. This can cause memory leak. Use exception handling to fix this issue.

```
#include <fstream>
#include <iostream>
#include <stdexcept>

class Book {
  public:
  Book(std::string filename)
  {
      std::vector<std::vector<std::string>> book_;
    std::string sent;
    std::ifstream book_file(filename);
    //use exception handling: try catch
    try{
        while(std::getline(book_file, sent))
        {
          book_.push_back(split(sent, ' '));
        }
        book_file.close();
    }
    catch(std::exception const& e){
        book_file.close();
        std::cout << "An error occurs: " << e.what() << std::endl;
        throw e;
    }
  }
};
```

b

Assume that we are reading in large books and are worried that we might run out of space on the stack so we are allocating `std::vector<std::vector<string>> book_` on the heap. this would change the class outline.

Now there is another problem. The default destructor does not clean up the heap allocated memory. This means that as soon as a book object goes out of scope we have a memory leak. Implement the `Book` class following RAII.

```
1
2  #include <fstream>
3  #include <iostream>
4  #include <stdexcept>
5
6  class Book {
7    public:
8    Book(std::string filename); //declare constructor
9    ~Book(); //declare destructor
10
11     private:
12     std::vector<std::vector<std::string>> *book_;
13 };
14
15 /** Constructor */
16 Book::Book(std::string filename)
17 {
18   book_ = new std::vector<std::vector<std::string>>;
19   std::string sent;
20   std::ifstream book_file(filename);
21   try{
22     while(std::getline(book_file, sent))
23     {
24       book_->push_back(split(sent, ' '));
25     }
26     book_file.close();
27   }
28   catch(std::exception const& e){
29     book_file.close();
30     std::cout << "An error occurs: " << e.what() << std::endl;
31     throw;
32   }
33 }
34
35 /** Destructor */
36 Book::~Book()
37 {
38     // De-allocate the memory that was previously reserved
39     // for the vector of vector
40     //book_.clear();
41     delete book_;
42 }
```

Question 4: Lectures 6 & 7 We have to be careful with resource allocation when it comes to inheritance. Consider the following polymorphism example.

```
#include <string>
#include <fstream>
#include <iostream>
#include <vector>

class Student
    {
        protected:
            std::string name_;
            std::string uni_;
        public:
            Student(std::string name, std::string uni ): name_(name), uni_(uni) {};
            virtual std::string print()
            {
                std::cout<<"Hello my name is "<< name_ <<std::endl;
                std::cout<<"I am an student at "<< uni_ <<std::endl;
            }

    };

    class GradStudent: public Student
    {
        protected:
        std::fstream thesis_;

        public:

        GradStudent(std::string name, std::string uni, std::string thesis): Student(n
        {
            thesis_.open(thesis);
        };

        virtual std::string print()
        {
            std::cout<<"Hello my name is "<< name_ <<std::endl;
            std::cout<<"I am a grad student at "<< uni_ <<std::endl;
        };

        ~GradStudent(){
            thesis_.close();
        };
    };

    int main()
    {
        std::vector<Student*> *students = new std::vector<Student*>;
        GradStudent a("Amy","Stanford", "Amys_thesis.txt");
        Student b("Karl","Berkley");
        Student c("Steve", "UCLA");
        students->push_back(&a);
```

```
51          students->push_back(&b);
52          students->push_back(&c);
53
54
55          for(auto student= students->begin(); student != students->end(); student++)
56          {
57              (*student)->print();
58          }
59          delete students;
60      }
```

What goes wrong in the code above and how do you fix it?

In this problem, I have fixed the following few problems in the original code:
Firstly, I have revised the return type of both print() from std::string to void to avoid compiler warnings while executing the code.
Secondly, and most importantly, is about memory allocation. I have identified that once we did `std::vector<Student*> *students = new std::vector<Student*>;`, we have elements that are pointers. If we naively use `std::vector<T,Allocator>:: vector` from the standard library, the pointed-to objects are not destroyed. Hense, I use a for loop to assign those pointers to `nullptr`.

```
1   #include <string>
2   #include <fstream>
3   #include <iostream>
4   #include <vector>
5
6   class Student
7   {
8       protected:
9           std::string name_;
10          std::string uni_;
11      public:
12          Student(std::string name, std::string uni ): name_(name), uni_(uni) {};
13          virtual void print()
14          {
15              std::cout<<"Hello my name is "<< name_ <<std::endl;
16              std::cout<<"I am an student at "<< uni_ <<std::endl;
17          };
18  };
19
20  class GradStudent: public Student
21  {
22
23      protected:
24          std::fstream thesis_;
25
26      public:
27
28          GradStudent(std::string name, std::string uni, std::string thesis)
29              : Student(name, uni){
30                  thesis_.open(thesis);
```

```
31              };
32
33          virtual void print()
34          {
35              std::cout<<"Hello my name is "<< name_ <<std::endl;
36              std::cout<<"I am a grad student at "<< uni_ <<std::endl;
37          };
38
39          ~GradStudent(){
40              thesis_.close();
41              std::cout << "DELETE GRAD" << std::endl;
42          };
43  };
44
45  int main()
46  {
47      std::vector<Student*> *students = new std::vector<Student*>;
48      GradStudent a("Amy","Stanford", "Amys_thesis.txt");
49      Student b("Karl","Berkley");
50      Student c("Steve", "UCLA");
51      students->push_back(&a);
52      students->push_back(&b);
53      students->push_back(&c);
54
55
56      for(auto student= students->begin(); student != students->end(); student++)
57      {
58          (*student)->print();
59      }
60
61      for (int i = 0; i < (*students).size(); ++i) {
62          (*students)[i] = nullptr;
63      }
64      delete students;
65  }
```