CSCE 622: Generic Programming -- Assignment 2

Peihong Guo UIN: 421003404

Problem 1

The simple definition of `array` class below misses many important things to make it work sensibly in generic algorithms.

```
template <typename T>
class array {
  T* data;
  int n;

public:
  array (int n_) : data(new T[n_]), n(n_) {}
  ~array () { delete [] data; }

  friend bool operator==(const array<T>& v1, const array<T>& v2) {
    return v1.data == v2.data;
  }
  T& operator[](int i) { return data[i]; }
  int size() { return n; }
};
```

By default, C++ compiler will generate the following member functions for this class:

1. Copy constructor
2. Copy assignment operator

The move constructor and move assignment operator are not generated because of the user-defined destructor.

Issue 1: Default constructor

A default constructor is missing in the class definition, which greatly restricts the

application of this class.

For example, the following code fragment would not compile:

```
array<array<int>> array_of_array(3);
```

See `p1_default_ctr.cpp` for a complete demonstration.

To fix it, we need to add a well behaved default constructor:

```
array() : data(nullptr), n(0) {}
```

Issue 2: Copy constructor

The default copy constructor generated by compiler is something equivalent to the following code fragment:

```
array(const array& other) : data(other.data), n(other.n) {}
```

This will cause undefined behavior in runtime:

```
{
  array<int> a1(3);

  // use a nested scope to make sure a2 is destroyed at the end of
this scope.
  // when a2 is destroyed, the memory block it holds is deallocated
.
  {
    // invoke the copy constructor to make a copy of a1
    array<int> a2(a1);
  }

  // a1 will be destroyed here, but it will crash the program becau
se of its
  // data pointer is dangling at this point.
}
```

This code fragment will crash the program because the memory block will be deallocated twice. See `p1_copy_ctor.cpp` for a complete demonstration.

To fix it, add the following copy constructor:

```
array (const array& other) : data(new T[other.n]), n(other.n) {
  cout << "invoking copy assignment operator ..." << endl;
  memcpy(data, other.data, sizeof(T)*n);
}
```

The memory copy operation is added to make sure the new copy has exactly the same elements as the source array.

Issue 3: Copy assignment operator

The default copy assignment operator generated by the compiler is something equivalent to the following code fragment:

```
array& operator=(const array& other) {
  data = other.data;
  n = other.data;
}
```

Similar to the copy constructor, this generated copy assignment operator will cause undefined behavior in runtime:

```
{
  array<int> a1(3);

  // use a nested scope to make sure a2 is destroyed at the end of
this scope.
  // when a2 is destroyed, the memory block it holds is deallocated
.
  {
    // invoke the copy constructor to make a copy of a1
    array<int> a2(1);
    a2 = a1;
  }

  // a1 will be destroyed here, but it will crash the program becau
se of its
  // data pointer is dangling at this point.
}
```

This code fragment will crash the program because the memory block will be deallocated twice. See `p1_copy_assign_op.cpp` for a complete demonstration.

To fix it, add the following copy constructor:

```
array& operator=(const array& other) {
    if(&other != this) {
      delete [] data;
      n = other.n;
      data = new T[n];
      memcpy(data, other.data, sizeof(T)*n);
    }
    return (*this);
}
```

The memory copy operation is added to make sure the new copy has exactly the same elements as the source array.

Issue 4: Equality operator

The definition of of the equality operator only compares the memory address of the data held by the array, but fails to check the whether the sizes of both arrays are the same. This could be problematic because it is possible two arrays share the same memory address while having different sizes.

The more important issue with this definition is that it allows two arrays sharing the same memory block, which is very dangerous because it could lead to undefined behaviour when either one of them is destroyed and the data pointer could become a dangling pointer. See `p1_equality_op.cpp` for a complete demonstration.

A better design compatible with proper copy constructor/copy-assignment operator would be comparing the elements inside the arrays:

```
friend bool operator==(const array<T>& v1, const array<T>& v2) {
  if( v1.n != v2.n ) return false;
  for(int i=0;i<v1.n;++i) {
    if( v1.data[i] != v2.data[i] ) return false;
  }
  return true;
}
```

Issue 5: Inequality operator

This operator is missing in the class definition. It must be added to make the class usable with various generic algorithms:

```
friend bool operator!=(const array<T>& v1, const array<T>& v2) {
  return !(v1 == v2);
}
```

See `p1_inequality_op.cpp` for a complete demonstration.

Issue 6: Element accessor and `size()` function

Only reference version of element accessor is provided in the class definition; however, a const-reference version must be included for many generic algorithms to work properly.

Consider the following function:

```
bool contains(const array<T>& A, const T& element) {
  for(int i=0;i<A.size();++i) {
    if( element == A[i] ) return true;
  }
  return false;
}
```

The following const-reference element accessor must be added for the above code fragment to compile.
`const T& operator[](int i) const { return data[i]; }`

Similarly, the `size()` function must be changed to `int size() const { return n; }` as well for the same reason. See `p1_accessor.cpp` for a complete demonstration.

Problem 2

`func1` implements the algorithm to sort an input sequence. For example, the following code fragment will output `abcde`:

```
string s("acedb");
func(s.begin(), s.end());
cout << s << endl;
```

See `p2_test.cpp` for demonstration.

The complexity of this algorithm is $O(n!)$ because it will try all possible permutations before reaching the solution.

Problem 3

To satisfy the pre and post conditions of `dragdrop`, the movement of the elements must be stable, i.e. the relative order of elements maintains the same in both the set of elements that satisfy s and those do not. `std::stable sort` is well suited for this purpose, as long as proper comparer is provided. Turned out the following comparer

works well in this case

```
template <typename UnaryPredicate>
struct Comparer {
  Comparer(UnaryPredicate s) : s(s) {}

  template <typename T>
  bool operator()(const T& a, const T&b) {
    return !s(a) && s(b);
  }
  UnaryPredicate s;
};
```

When sorting a sequence with this comparer, the elements satisfy the predicate become the postfix of the output sequence, and other become the prefix. Combined with `std::stable_sort`, the relative order of the elements will be maintained in the output sequence.

Using this comparer, `dragdrop` can be written as

```
template <typename BidirectionalIterator, typename UnaryPredicate>
void dragdrop(BidirectionalIterator f, BidirectionalIterator l,
              BidirectionalIterator p, UnaryPredicate s) {
  // sort the elements before p, so the elements satisfy s will be
postfix of [f, p]
  std::stable_sort(f, p, Comparer<UnaryPredicate>(s));
  // sort the elements before p, so the elements satisfy s will be
prefix of [p, l]
  std::stable_sort(p, l, Comparer<Negate<UnaryPredicate>>(s));
}
```

where `Negate` is a functor that negates the unary predicate:

```
template <typename UnaryPredicate>
struct Negate {
  Negate(UnaryPredicate s) : s(s) {}

  template <typename T>
  bool operator()(const T& t) {
    return !s(t);
  }
  UnaryPredicate s;
};
```

See `p3.cpp` for a complete demonstration.

Problem 4

Using `std::iterator_traits`, this function can be written as

```
template <class Iterator>
void print_category(Iterator x) {
  print_category_dispatch<typename iterator_traits<Iterator>::itera
tor_category>();
}
```

where `print_category_dispatch` is a templated helper function with 5 specializations:

```
template <typename iterator_category>
void print_category_dispatch();

template <>
void print_category_dispatch<input_iterator_tag>() {
  cout << "Input iterator" << endl;
}

template <>
void print_category_dispatch<output_iterator_tag>() {
  cout << "Output Iterator" << endl;
}

template <>
void print_category_dispatch<forward_iterator_tag>() {
  cout << "Forward Iterator" << endl;
}

template <>
void print_category_dispatch<bidirectional_iterator_tag>() {
  cout << "Bidirectional Iterator" << endl;
}

template <>
void print_category_dispatch<random_access_iterator_tag>() {
  cout << "Random Asscess Iterator" << endl;
}
```

See `p4.cpp` for a complete demonstration.