6 Assignment 4 [Assignment ID: cpp_containers]

6.1 Preamble (Please Read Carefully)

Before starting work on this assignment, it is **critically important** that you **carefully** read Section 1 (titled "General Information") which starts on page 1 of this document.

6.2 Topics Covered

This assignment covers material primarily related to the following: memory management, intrusive and nonintrusive containers.

6.3 Problems — Part A

- 8.17 a b c [container selection]
- 8.18 [separation of construction/destruction and allocation/deallocation]
- 8.19 [array-based vs. node-based]
- 8.23 a b c [intrusive vs. nonintrusive containers]

6.4 Problems — Part B

1. *Intrusive doubly-linked list class template (*`list`*).* In this exercise, a class template called `list` is to be developed that represents an intrusive doubly-linked list with a sentinel node. The `list` class template relies on a (non-template) helper class called `list_hook`, which stores per-node list management information. The `list_hook` class (which is a non-template class) is used to store per-node information needed for list management (i.e., pointers to the successor and predecessor nodes). For a type `T` to be compatible with `list`, `T` must include a data member of type `list_hook`. The interfaces for the `list` class template and `list_hook` class are given in Listing 8.

Listing 8: Interface for class template `list`

```
1  namespace ra::intrusive {
2
3      // Per-node list management information class.
4      // The implementation-defined type that contains per-node list
5      // management information (i.e., successor and predecessor).
6      // All members of this class are private.
7      // This class has the list class template as a friend.
8      // This type must contain pointers (of type list_hook*) to the
9      // next and previous node in the list.
10     class list_hook {
11         implementation-defined
12     };
13
14     // Intrusive doubly-linked list (with sentinel node).
15     template <class T, list_hook T::* Hook>
16     class list {
17     public:
18
19         // The type of the nodes in the list.
20         using value_type = T;
```

```
21
22          // The pointer-to-member associated with the list hook object.
23          constexpr list_hook T::* hook_ptr = Hook;
24
25          // The type of a mutating reference to a node in the list.
26          using reference = T&;
27
28          // The type of a non-mutating reference to a node in the list.
29          using const_reference = const T&;
30
31          // The mutating (bidirectional) iterator type for the list.
32          using iterator = implementation-defined;
33
34          // The non-mutating (bidirectional) iterator type for the list.
35          using const_iterator = implementation-defined;
36
37          // The size type, an unsigned integral type.
38          using size_type = std::size_t;
39
40          // Creates an empty list.
41          list();
42
43          // Erases any elements from the list and then destroys the list.
44          ~list();
45
46          // Moves a list.
47          // Precondition: The destination list must be empty.
48          // The nodes in the source list are moved from the source list
49          // to the destination list (preserving their relative order).
50          // After the move, the source list is empty.
51          list(list&&);
52          list& operator=(list&&);
53
54          // Do not allow the copying of lists.
55          list(const list&) = delete;
56          list& operator=(const list&) = delete;
57
58          // Swaps the elements of *this and x.
59          void swap(list& x);
60
61          // Returns the number of elements in the list.
62          size_type size() const;
63
64          // Inserts an element in the list before the element referred to
65          // by the iterator pos.
66          // An iterator that refers to the inserted element is returned.
67          iterator insert(iterator pos, value_type& value);
68
69          // Erases the element in the list at the position specified by the
70          // iterator pos.
71          // An iterator that refers to element following the erased element
72          // is returned if such an element exists; otherwise, end() is
```

```
73          // returned.
74          iterator erase(iterator pos);
75
76          // Inserts the element with the value x at the end of the list.
77          void push_back(value_type& x);
78
79          // Erases the last element in the list.
80          // Precondition: The list is not empty.
81          void pop_back();
82
83          // Returns a reference to the last element in the list.
84          // Precondition: The list is not empty.
85          reference back();
86          const_reference back() const;
87
88          // Erases any elements from the list, yielding an empty list.
89          void clear();
90
91          // Returns an iterator referring to the first element in the list
92          // if the list is not empty and end() otherwise.
93          const_iterator begin() const;
94          iterator begin();
95
96          // Returns an iterator referring to the fictitious one-past-the-end
97          // element.
98          const_iterator end() const;
99          iterator end();
100
101     };
102 }
```

All of the necessary declarations and definitions for the `list` class template should be placed in a header file called `include/ra/intrusive_list.hpp`.

Note that `list` and `list_hook` are contained in the namespace `ra::intrusive`. The `iterator` and `const_iterator` types must provide all of the functionality of a bidirectional iterator. This includes, amongst other things, prefix and postfix increment, prefix and postfix decrement, dereference operators (both unary **`operator`**`*` and **`operator`**`->`). These iterator types must also behave in a const-correct manner. The code must be exception safe.

Determining the parent object from a pointer to one of its members requires nonportable (i.e., compiler-dependent) code. To simplify this exercise, the overloaded function `parent_from_member` is provided for making this determination. The relevant declarations are as follows:

```
namespace ra::util {
    template<class Parent, class Member>
    inline Parent *parent_from_member(Member *member,
      const Member Parent::* ptr_to_member);

    template<class Parent, class Member>
    inline const Parent *parent_from_member(const Member *member,
      const Member Parent::* ptr_to_member);
}
```

Given a pointer `member` to a subobject of some parent object (of type `Parent`) and a pointer-to-member `ptr_to_member` associated with that subobject, the function `parent_from_member` returns a pointer to the parent object of `*member`. The code for the above functions is provided in the file `parent_from_member.hpp`. This file must be placed in the directory `include/ra` and the contents of this file should not be modified. The code provided for `parent_from_member` is only guaranteed to work for the compilers (GCC and Clang) used in the course. (The code will likely not work for the MSVC compiler.)

The `list` class template should be tested with a variety of element types. A trivial example illustrating the use of the `list` class is given in Listing 9.

Listing 9: Example use of `list`

```
#include "ra/intrusive_list.hpp"

namespace ri = ra::intrusive;

struct Widget {
    Widget(int value_) : value(value_) {}
    int value;
    ri::list_hook hook;
};

int main()
{
    std::vector<Widget> storage;
    storage.push_back(Widget(42));
    ri::list<Widget, &Widget::hook> values;
    for (auto&& i : storage) {
        values.push_back(i);
    }
    values.clear();
}
```

The code used to test the `list` class template should be placed in a file called `app/test_intrusive_list.cpp`.

2. *Ordered set class template based on sorted array (`sv_set`).* In this exercise, a class template called `sv_set` is to be developed that represents an ordered set of unique elements. This template has two parameters:

   (a) `Key`. The type of the elements in the set.
   (b) `Compare`. The type of the callable entity (e.g., function or functor) used to test if one key is less than another. (For the sake of simplicity, the interface for `sv_set` assumes that `Compare` is a default-constructible type.)

   The interface for the `sv_set` class template is given in Listing 10.

   The `sv_set` class template is somewhat similar to `std::set`, except that the underlying data structure used to store container elements differs. In the case of `std::set`, container elements are stored in a balanced tree. In contrast, the `sv_set` uses a dynamically-resizable array as the underlying data structure for storing the elements of the set. In order to facilitate efficient searching for elements, the elements of the array are stored in sorted order, namely, ascending order by key.

The dynamically-resizable array used by `sv_set` is somewhat similar to `std::vector`. The `std::vector` class template cannot be used in this exercise, however. The code for `sv_set` must directly manage the storage of the container elements (i.e., it cannot delegate this responsibility to another class such as `std::vector`). Global operator new and operator delete must be used in order to allocate storage for the container elements.

Listing 10: Interface for class template `sv_set`

```
namespace ra::container {

    // A class representing a set of unique elements (which uses
    // a sorted array).
    template <class Key, class Compare = std::less<Key>>
    class sv_set {
    public:

        // The type of the elements held by the container.
        using value_type = Key;
        using key_type = Key;

        // The type of the function/functor used to compare two keys.
        using key_compare = Compare;

        // An unsigned integral type used to represent sizes.
        using size_type = std::size_t;

        // The mutable (random-access) iterator type for the
        // container.
        using iterator = implementation-defined;

        // The non-mutable (random-access) const_iterator type for
        // the container.
        using const_iterator = implementation-defined;

        // Creates an empty set (i.e., a set containing no elements).
        sv_set();

        // Propagates the value of a set to another set via a move
        // operation.  The elements of the source set are propagated
        // to the destination set.  After the move operation, the
        // source set is empty.
        sv_set(sv_set&&);
        sv_set& operator=(sv_set&&);

        // Propagates the value of a set to another set via a copy
        // operation.
        sv_set(const sv_set&);
        sv_set& operator=(const sv_set&);

        // Erases all elements in the container and destroys the
        // container.
        ~sv_set();
```

```
45
46          // Returns an iterator referring to the first element in the
47          // set if the set is not empty and end() otherwise.
48          const_iterator begin() const;
49          iterator begin();
50
51          // Returns an iterator referring to the fictitious
52          // one-past-the-end element.
53          const_iterator end() const;
54          iterator end();
55
56          // Returns the number of elements in the set (i.e., the size
57          // of the set).
58          size_type size() const;
59
60          // Returns the number of elements for which storage is
61          // available (i.e., the capacity of the set).  This value is
62          // always at least as great as size().
63          size_type capacity() const;
64
65          // Reserves storage in the container for at least n elements.
66          // After this function has been called with a value of n, it
67          // is guaranteed that no memory-allocation is needed as long
68          // as the size of the set does not exceed n.
69          // Calling this function has no effect if the capacity of the
70          // container is already at least n.
71          void reserve(size_type n);
72
73          // Reduces the capacity of the container to the container
74          // size.
75          // If the capacity of the container is greater than its size,
76          // the capacity is reduced to the size of the container.
77          // Calling this function has no effect if the capacity of the
78          // container does not exceed its size.
79          void shrink_to_fit();
80
81          // Inserts the element x in the set.  The iterator pos
82          // provides a hint as to where the search for the insertion
83          // position should begin.
84          std::pair<bool, iterator> insert(const key_type& x);
85
86          // Erases the element referenced by pos from the container.
87          // Returns an iterator referring to the element following the
88          // erased one in the container if such an elements exists or
89          // end() otherwise.
90          iterator erase(const_iterator pos);
91
92          // Swaps the contents of the container with the contents of the
93          // container x.
94          void swap(sv_set& x);
95
96          // Erases any elements in the container, yielding an empty
```

```
97            // container.
98            void clear();
99
100           // Searches the container for an element with the key k.
101           // If an element is found, an iterator referencing the element
102           // is returned; otherwise, end() is returned.
103           iterator find(const key_type& k);
104           const_iterator find(const key_type& k) const;
105
106       };
107   }
```

All of the necessary declarations and definitions for the `sv_set` class template should be placed in a header file called `include/ra/sv_set.hpp`.

Although the particular types to be used for the type members `iterator` and `const_iterator` are not specified, they must meet the requirements of a random access iterator. Raw pointer types may be used for these iterator types.

As indicated above, the `sv_set` class template must be placed in the namespace `ra::container`.

Some functionality of the standard library that may potentially prove useful in this exercise includes: `std::copy`, `std::copy_backward`, `std::move`, `std::move_backward`, `std::uninitialized_copy`, `std::uninitialized_copy_n`, `std::uninitialized_move`, `std::uninitialized_move_n`, `std::uninitialized_fill`, `std::uninitialized_fill_n`, `std::destroy_at`, and `std::destroy`.

The code used to test the `sv_set` class template should be placed in a file called `app/test_sv_set.cpp`.