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 1-1 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 — Nonprogramming Exercises

- 8.24 a b c [container selection]
- 8.26 [separation of construction/destruction and allocation/deallocation]
- 8.27 [array-based vs. node-based]
- 8.29 a b c [intrusive vs. nonintrusive containers]

6.4 Problems — Part B — Nonintrusive Set

B.1 *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 each of the elements in the set (i.e., the key type).

(b) `Compare`. The type of the callable entity (e.g., function or functor) used to test if one key is less than another. Since the elements of the container are ordered, a comparison predicate must be provided by the container user to define the sorting criterion to be employed by the container. A callable entity `f` of type `Compare` can have the function-call operator applied with exactly two function arguments `x` and `y` of type `Key` and yields a return type of **`bool`**. The value returned by `f(x,y)` is **`true`** if `x` is less than `y` and **`false`** otherwise (i.e., `f` is a less-than predicate). Only a less-than predicate is provided by the user of `sv_set`, since all other relational operators (e.g., equal, not-equal, greater-than, greater-than-or-equal, and less-than-or-equal) can be synthesized from this single predicate. The type `Compare` need not be default constructible.

The interface for the `sv_set` class template is given in Listing 8.

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 (node-based) tree. In contrast, `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 8: 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:

```

```
    // Note:
    // In the time complexity specifications of various functions below,
    // it is assumed that the following operations for the Compare type
    // are constant time: default construction, destruction, copy
    // construction and assignment, and move construction and assignment.

    // A dummy type used to indicate that elements in a range are
    // both ordered and unique.
    struct ordered_and_unique_range {};

    // The type of the elements held by the container. This is
    // simply an alias for the template parameter Key.
    using value_type = Key;
    using key_type = Key;

    // The type of the function/functor used to compare two keys.
    // This is simply an alias for the template parameter Compare.
    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.
    // This type must support all of the functionality associated
    // with a random-access iterator.
    using iterator = /* implementation defined */;

    // The non-mutable (random-access) const_iterator type for
    // the container.
    // This type must support all of the functionality associated
    // with a random-access iterator.
    using const_iterator = /* implementation-defined */;

    // Default construct a set.
    //
    // Creates an empty set (i.e., a set containing no elements)
    // with a capacity of zero (i.e., no allocated storage for
    // elements).
    //
    // Time complexity:
    // Constant.
    sv_set() noexcept(std::is_nothrow_default_constructible_v<
      key_compare>);

    // Construct a set with the specified comparison object.
    //
    // Creates an empty set (i.e., a set containing no elements)
    // with a capacity of zero (i.e., no allocated storage for
    // elements). The comparison object associated with the set is
    // set to comp.
    //
    // Time complexity:
    // Constant.
    explicit sv_set(const Compare& comp);

    // Construct a set from a range.
    //
```

```
    // Create a set consisting of the n elements in the range starting at
    // first, where the elements in the range must be both unique and
    // ordered with respect to the comparison operation embodied by the
    // key_compare type. If the specified range is not both ordered and
    // unique, the behavior of this function is undefined.
    // The comparison object associated with the set is set to comp.
    //
    // Time complexity:
    // Linear in n.
    //
    // Template constraints:
    // The type InputIterator must meet the requirements of an input
    // iterator.
    //
    // Note:
    // The parameter of type ordered_and_unique_range is always ignored.
    // This parameter is only present to allow for future expansion
    // (i.e., adding a constructor that does not require an ordered
    // and unique range).
    template <class InputIterator>
    sv_set(ordered_and_unique_range, InputIterator first,
      std::size_t n, const Compare& comp = Compare());

    // Move construct a set.
    //
    // Creates a new set by moving from the specified set other. After
    // construction, the source set (i.e., other) is guaranteed to be
    // empty.
    //
    // Time complexity:
    // Constant.
    sv_set(sv_set&& other) noexcept(
      std::is_nothrow_move_constructible_v<key_compare>);

    // Move assign a set.
    //
    // Assigns the value of the specified set other to *this via a move
    // operation. After the move operation, the source set (i.e., other)
    // is guaranteed to be empty.
    //
    // Iterator/reference invalidation:
    // Move assignment may invalidate iterators/references to elements
    // in the moved-from and moved-to containers.
    //
    // Time complexity:
    // Linear in size().
    //
    // Preconditions:
    // The objects *this and other are distinct.
    sv_set& operator=(sv_set&& other) noexcept(
      std::is_nothrow_move_assignable_v<key_compare>);

    // Copy construct a set.
    //
    // Creates a new set by copying from the specified set other.
    //
    // Time complexity:
```

```
	// Linear in other.size().
	sv_set(const sv_set& other);

	// Copy assign a set.
	//
	// Assigns the value of the specified set other to *this.
	//
	// Iterator/reference invalidation:
	// Copy assignment may invalidate iterators/references to elements in
	// the copied-to container.
	//
	// Time complexity:
	// Linear in size() and other.size().
	sv_set& operator=(const sv_set& other);

	// Destroy a set.
	//
	// Erases all elements in the container and destroys the container.
	//
	// Time complexity:
	// Linear in size().
	~sv_set();

	// Get the comparison object for the container.
	//
	// Return value:
	// Returns the comparison object for the container.
	//
	// Time complexity:
	// Constant.
	key_compare key_comp() const;

	// Get an iterator referring to the first element in a set.
	//
	// Return value:
	// Returns an iterator referring to the first element in the set if
	// the set is not empty and end() otherwise.
	//
	// Time complexity:
	// Constant.
	const_iterator begin() const noexcept;
	iterator begin() noexcept;

	// Get an iterator referring to the one-past-the-end position in a
	// set.
	//
	// Return value:
	// Returns an iterator referring to the fictitious one-past-the-end
	// element for the set.
	//
	// Time complexity:
	// Constant.
	const_iterator end() const noexcept;
	iterator end() noexcept;

	// Get the size of a set.
	//
```

```
    // Return value:
    // Returns the number of elements in the set (i.e., the size
    // of the set).
    //
    // Time complexity:
    // Constant.
    size_type size() const noexcept;

    // Get the capacity of a set.
    //
    // Return value:
    // Returns the number of elements for which storage is
    // available (i.e., the capacity of the set). This value is
    // always at least as great as size().
    //
    // Time complexity:
    // Constant.
    size_type capacity() const noexcept;

    // Reserve storage for use by a set.
    //
    // Reserves storage in the container for at least n elements.
    // After this function has been called with a value of n, it
    // is guaranteed that no memory-allocation is needed as long
    // as the size of the container does not exceed n.
    // Calling this function has no effect if the capacity of the
    // container is already at least n (i.e., the capacity of
    // the container is never reduced by this function).
    //
    // Iterator/reference invalidation:
    // The reserve member function may invalidate iterators/references
    // to elements in the container if the capacity of the container is
    // increased.
    //
    // Time complexity:
    // At most linear in size().
    void reserve(size_type n);

    // Minimize the amount of storage used for the elements in a set.
    //
    // Reduces the capacity of the container to the container size.
    // If the capacity of the container is greater than its size,
    // the capacity is reduced to the size of the container.
    // Calling this function has no effect if the capacity of the
    // container does not exceed its size.
    //
    // Iterator/reference invalidation:
    // The shrink_to_fit member function may invalidate
    // iterators/references to elements in the container if the capacity
    // of the container is decreased.
    //
    // Time complexity:
    // At most linear in size().
    void shrink_to_fit();

    // Insert an element in a set.
    //
```

```
    // Inserts the element x in the set.
    // If the element x is already in the set, no insertion is
    // performed (since a set cannot contain duplicate values).
    //
    // Return value:
    // The second (i.e., boolean) component of the returned pair
    // is true if and only if the insertion takes place; and the
    // first (i.e., iterator) component of the pair refers to
    // the element with key equivalent to the key of x
    // (i.e., the element inserted if insertion took place or
    // the element found with an equal key if insertion did not
    // take place).
    //
    // Iterator/reference invalidation:
    // The insert member function may invalidate iterators/references that
    // refer to elements in the container only if an insertion is actually
    // performed (i.e., the element to be inserted is not already in the
    // container). If an insertion is performed into a container whose
    // size is less than its capacity, insert may invalidate only the
    // iterators/references that refer to elements in the container with
    // a value greater than the inserted element.
    //
    // Time complexity:
    // Search logarithmic in size() plus insertion linear in either the
    // number of elements with larger keys than x (if size() < capacity())
    // or size() (if size() == capacity()).
    std::pair<iterator, bool> insert(const key_type& x);

    // Remove an element from a set.
    //
    // Erases the element referenced by pos from the container.
    // Returns an iterator referring to the element following the
    // erased one in the container if such an element exists or
    // end() otherwise.
    //
    // Iterator/reference invalidation:
    // The erase member function may invalidate iterators/references that
    // refer to elements in the container with a value greater than the
    // erased element.
    //
    // Time complexity:
    // Linear in number of elements with larger keys than x.
    iterator erase(const_iterator pos);

    // Swap the contents of two sets.
    //
    // Swaps the contents of the container with the contents of the
    // container x.
    //
    // Iterator/reference invalidation:
    // The swap member function may invalidate iterators/references to
    // elements in both of the containers being swapped.
    //
    // Time complexity: Constant.
    void swap(sv_set& x) noexcept(
      std::is_nothrow_swappable_v<key_compare>);

```

```
        // Clear the contents of the set.
        //
        // Erases any elements in the container, yielding an empty container.
        //
        // Time complexity:
        // Linear in size().
        void clear() noexcept;

        // Find an element in a set.
        //
        // Searches the container for an element with the key k.
        // If an element is found, an iterator referencing the element
        // is returned; otherwise, end() is returned.
        //
        // Time complexity:
        // Logarithmic.
        iterator find(const key_type& k);
        const_iterator find(const key_type& k) const;

        // Additional Remarks
        //
        // Iterator/reference invalidation:
        // Each nonmutating (public) member function of sv_set is guaranteed
        // not to invalidate iterators/references to elements in the
        // container. The mutating (public) member functions of sv_set can
        // only invalidate iterators/references as documented herein.
        // Clearly, if an element is removed from the container, all
        // iterators/references that refer to it will be invalidated.

    };
}
```

The term "input iterator" is used in the above interface specification. If you are unsure as to what exactly an input iterator is, refer to the section of the lecture slides on containers, iterators, and algorithms. This section discusses iterator categories as well as input iterators specifically.

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`.

Constructors and testing. Since the code that tests the `sv_set` class template does not have access to the internal (i.e., private) state used to implement the class template, test code must rely heavily on the constructors of the class in order to place data in `sv_set` objects in order to perform testing. If any of the constructors of the class have bugs, this could easily result in every single test case failing. So, it is critically important that the constructors be very well tested. For example, the constructor that takes an `ordered_and_unique_range` and iterator range is used heavily by the instructor's test code to place data in `sv_set` objects for testing. If this constructor were to work incorrectly, every test using this constructor would likely fail, due to placing the wrong data inside `sv_set` objects during part of the test.

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`.

It is very strongly recommended that the ASan and LSan code sanitizers be employed during the testing of the code for this exercise. ASan is helpful for detecting bugs related to bad pointers, while LSan is helpful for detecting memory leaks. An option for enabling ASan and/or LSan can be placed in a file called `Sanitizers.cmake` and then included in the `CMakeLists.txt` file.

6.5 Problems — Part C — Intrusive List

C.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 9.

Listing 9: Interface for class template `list`

```
namespace ra::intrusive {

    // Per-node list management information class.
    // This type contains per-node list management information (i.e., the
    // successor and predecessor in the list). This class has the list class
    // template as a friend. This type must contain pointers (of type
    // list_hook*) to the next and previous node in the list.
    class list_hook {
    public:

        // Default construct a list hook.
        // This constructor creates a list hook that does not belong to any
        // list.
        list_hook();

        // Copy construct a list hook.
        // This constructor creates a list hook that does not belong to any
        // list. The argument to the constructor is ignored. The copy
        // construction operation is defined only so that types with list hooks
        // are copy constructible. The list class itself never copies (or
        // moves) a list hook.
        list_hook(const list_hook&);

        // Copy assign a list hook.
        // The copy assignment operator is defined as a no-op. The argument to
        // the operator is ignored. The copy assignment operation is defined
        // only so that types with list hooks are copy assignable. The list
        // class itself never copies (or moves) a list hook.
        list_hook& operator=(const list_hook&);

        // Destroy a list hook.
        // The list hook being destroyed must not belong to a list. If the
        // list hook belongs to a list, the resulting behavior is undefined.
        ~list_hook();
    };

    // Intrusive doubly-linked list (with sentinel node).
    template <class T, list_hook T::* Hook>
```

```
    class list {
    public:

        // The type of the elements in the list.
        using value_type = T;

        // The pointer-to-member associated with the list hook object.
        static constexpr list_hook T::* hook_ptr = Hook;

        // The type of a mutating reference to a node in the list.
        using reference = T&;

        // The type of a non-mutating reference to a node in the list.
        using const_reference = const T&;

        // The mutating (bidirectional) iterator type for the list. This type
        // must provide all of the functionality of a bidirectional iterator.
        // If desired, the Boost Iterator library may be used to implement
        // this type.
        using iterator = /* implementation defined */;

        // The non-mutating (bidirectional) iterator type for the list. This
        // type must provide all of the functionality of a bidirectional
        // iterator. If desired, the Boost Iterator library may be used to
        // implement this type.
        using const_iterator = /* implementation defined */;

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

        // Default construct a list.
        //
        // Creates an empty list.
        //
        // Time complexity:
        // Constant.
        list();

        // Destroy a list.
        //
        // Erases any elements from the list and then destroys the list.
        //
        // Time complexity:
        // Either linear or constant.
        ~list();

        // Move construct a list.
        //
        // The elements in the source list (i.e., other) are moved from the
        // source list to the destination list (i.e., *this), preserving their
        // relative order. After the move, the source list is empty.
        //
        // Time complexity:
        // Constant.
        list(list&& other);

        // Move assign a list.
```

```
        //
        // The elements of the source list (i.e., other) are swapped with the
        // elements of the destination list (i.e., *this). The relative order
        // of the elements in each list is preserved.
        //
        // Precondition:
        // The objects *this and other are distinct.
        //
        // Time complexity:
        // Constant.
        list& operator=(list&& other);

        // Do not allow the copying of lists.
        list(const list&) = delete;
        list& operator=(const list&) = delete;

        // Swap the elements of two lists.
        //
        // Swaps the elements of *this and x.
        // Swapping the elements of a list with itself has no effect.
        //
        // Time complexity:
        // Constant.
        void swap(list& x);

        // Returns the number of elements in the list.
        //
        // Time complexity:
        // Constant.
        size_type size() const;

        // Inserts an element in the list before the element referred to
        // by the iterator pos.
        // An iterator that refers to the inserted element is returned.
        //
        // Time complexity:
        // Constant.
        iterator insert(iterator pos, value_type& value);

        // Erases the element in the list at the position specified by the
        // iterator pos.
        // An iterator that refers to the element following the erased element
        // is returned if such an element exists; otherwise, end() is
        // returned.
        //
        // Time complexity:
        // Constant.
        iterator erase(iterator pos);

        // Inserts the element with the value x at the end of the list.
        //
        // Time complexity:
        // Constant.
        void push_back(value_type& x);

        // Erases the last element in the list.
        //
```

```
        // Precondition:
        // The list is not empty.
        //
        // Time complexity:
        // Constant.
        void pop_back();

        // Returns a reference to the last element in the list.
        //
        // Precondition:
        // The list is not empty.
        //
        // Time complexity:
        // Constant.
        reference back();
        const_reference back() const;

        // Erases any elements from the list, yielding an empty list.
        //
        // Time complexity:
        // Either linear or constant.
        void clear();

        // Returns an iterator referring to the first element in the list
        // if the list is not empty and end() otherwise.
        //
        // Time complexity:
        // Constant.
        const_iterator begin() const;
        iterator begin();

        // Returns an iterator referring to the fictitious one-past-the-end
        // element.
        //
        // Time complexity:
        // Constant.
        const_iterator end() const;
        iterator end();

    };
}
```

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 may not work with other compilers, such as 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 10.

Listing 10: 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`.

It is very strongly recommended that the ASan and LSan code sanitizers be employed during the testing of the code for this exercise. ASan is helpful for detecting bugs related to bad pointers, while LSan is helpful for detecting memory leaks. An option for enabling ASan and/or LSan can be placed in a file called `Sanitizers.cmake` and then included in the `CMakeLists.txt` file.