Ministry of Science and Education of Russian Federation
Peter the Great St. Petersburg Polytechnic University
—
Institute of Computer Science and Technology
Department «Information security of computer systems»

L A B № 2

STL Container
course «OOP»

Student
Gr. 33508/3 Evgeniy G. Protsenko

Instructor Andrey Y. Chernov

Saint-Petersburg
2016

CONTENTS

1 TASK

The main goal is to develop C++ STL container as good as possible. Container is a bidirectional constantly sorted list. One also need to develop iterator and reverse iterator.

In report must be present description of requirements to container, reasons of Inability to satisfy some of requirements, source code of developed container.

2 INTRODUCTION

The containers are class templates; when you declare a container variable, you specify the type of the elements that the container will hold. Containers can be constructed with initializer lists. They have member functions for adding and removing elements and performing other operations. You iterate over the elements in a container, and access the individual elements by using iterators. Iterators for all STL containers have a common interface but each container defines its own specialized iterators.

Standard requirements:

```
void                    assign (size_type n, const value_type& val);
reference               back();
iterator                begin();
void                    clear();
bool                    empty() const;
iterator                end();
iterator                erase (iterator position);
reference               front();
void                    merge (list& x);
list &                  operator= (const list& x);
void                    pop_back();
void                    pop_front();
reverse_iterator        rbegin();
void                    remove (const value_type& val);
reverse_iterator        rend();
size_type               size() const;
void                    swap (list& x);
void                    unique();
```

Requirements that have not been met:

```
iterator insert (iterator position, const value_type& val);
//The container is extended by inserting new elements before the element at the
specified position.
void splice (iterator position, list& x);
//Transfers elements from x into the container, inserting them at position.

void push_back (const value_type& val);
void push_front (const value_type& val);

//We cant insert items in random places, because list is sorted.

void reverse();
//we cant reverse constantly sorted list

void sort();
//this operation has no sence for this type of container
```

3 CONCLUSION

At this lab I created STL container. Read some staff about this data type and know requirements of creating containers. Now I am familiar with basics of STL containers development.

REFERENCES

1. http://www.cplusplus.com/reference/list/list/
2. http://www.cplusplus.com/reference/iterator/BidirectionalIterator/
3. https://msdn.microsoft.com/en-us/library/cscc687y.aspx
4. http://cpp.com.ru/stl/

APPENDIX A

```
#include <cassert>
#include <iostream>

template <typename T>
class slist
{
private:
          struct list_elem{
                    T data;
                    list_elem * next;
                    list_elem * prev;

                    list_elem();
                    list_elem(T _data);
          };

          list_elem * head;
          list_elem * tail;

          size_t _size;
public:
          //*********ITERATOR DECLARATION*********//
          class iterator
          {
          public:
                    friend class slist<T>;

                    iterator();
                    iterator(list_elem * p_node);

                    T & operator *() const;
                    iterator & operator ++();
                    iterator operator ++(int);
                    bool operator !=(const iterator & rhs) const;
                    bool operator ==(const iterator & rhs) const;
                    iterator & operator --();
                    iterator operator --(int);

          private:
                    list_elem * current;
          };

          //*********REVERSE_ITERATOR DECLARATION*********//
          class reverse_iterator
                    : public iterator
          {
          public:
                    reverse_iterator();
                    reverse_iterator(list_elem * p_node);

                    reverse_iterator & operator ++();
                    reverse_iterator operator ++(int);
                    reverse_iterator & operator --();
                    reverse_iterator operator --(int);
          };

          //*********CONST_ITERATOR DECLARATION*********//
```

```
class const_iterator
          : iterator
{
public:
          T operator *() const;

          const_iterator();
          const_iterator(list_elem * p_node);

          const_iterator & operator ++();
          const_iterator operator ++(int);
          const_iterator & operator --();
          const_iterator operator --(int);

private:
          list_elem * current;
};

//*********CONST_REVERSE_ITERATOR DECLARATION*********//
class const_reverse_iterator
          : reverse_iterator
{
public:
          T operator *() const;

          const_reverse_iterator();
          const_reverse_iterator(list_elem * p_node);

          const_reverse_iterator & operator ++();
          const_reverse_iterator operator ++(int);
          const_reverse_iterator & operator --();
          const_reverse_iterator operator --(int);
};

//*********SLIST DECLARATION*********//
slist();
slist(slist<T> & x);
~slist();

iterator begin() const; //+
iterator end() const; //+
reverse_iterator rbegin() const; //+
reverse_iterator rend() const; //+
void push(T _data); //+
size_t size() const; //+
bool empty() const; //+
T & front() const; //+
T & back() const; //+
void pop_front(); //+
void pop_back(); //+
void clear(); //+
slist<T> & operator =(const slist<T> & rhs); //+
void remove(const T & val); //+
void unique(); //+
void merge(slist & x); //+
void assign(size_t n, const T & val); //+
void assign(iterator first, iterator last); //+
void assign(reverse_iterator first, reverse_iterator last); //+
iterator erase(iterator & position); //+
```

```
        reverse_iterator erase(reverse_iterator & position); //+
        void swap(slist & x); //+

        const_iterator cbegin() const;
        const_iterator cend() const;
        const_reverse_iterator crbegin() const;
        const_reverse_iterator crend() const;
};

//*********SLIST IMPLEMENTATION*********//
template <typename T>
slist<T>::slist()
: _size(0)
{
        head = new list_elem();
        tail = new list_elem();
        head->prev = NULL;
        head->next = tail;
        tail->prev = head;
        tail->next = NULL;
};

template <typename T>
slist<T>::slist(slist<T> & x)
: _size(0)
{
        head = new list_elem();
        tail = new list_elem();
        head->prev = NULL;
        head->next = tail;
        tail->prev = head;
        tail->next = NULL;

        merge(x);
};

template <typename T>
slist<T>::~slist()
{
        list_elem * curr = head->next;
        while (curr != tail)
        {
                list_elem * temp = curr;
                temp->next->prev = temp->prev;
                temp->prev->next = temp->next;

                curr = curr->next;
                delete temp;
        }
};

template <typename T>
size_t slist<T>::size() const
{
        return _size;
};

template <typename T>
void slist<T>::push(T _data)
```

```
{
	list_elem * elem = new list_elem(_data);

	list_elem * temp = head->next;
	while (temp != tail && elem->data < temp->data)
	{
		temp = temp->next;
	}

	elem->prev = temp->prev;
	elem->next = temp;
	temp->prev->next = elem;
	temp->prev = elem;

	_size++;
};

template <typename T>
typename slist<T>::iterator slist<T>::erase(iterator & position)
{
	assert(position.current != this->tail);
	assert(position.current != this->head);

	list_elem * curr = this->head->next;
	while (curr != this->tail)
	{
		if (curr == position.current)
		{
			++position;

			curr->next->prev = curr->prev;
			curr->prev->next = curr->next;

			delete curr;

			break;
		}
		curr = curr->next;
	}
	return position;
};

template <typename T>
typename slist<T>::reverse_iterator slist<T>::erase(reverse_iterator & position)
{
	assert(position.current != this->tail);
	assert(position.current != this->head);

	list_elem * curr = this->head->next;
	while (curr != this->tail)
	{
		if (curr == position.current)
		{
			++position;

			curr->next->prev = curr->prev;
			curr->prev->next = curr->next;

			delete curr;
```

```
                        break;
                }
                curr = curr->next;
        }
        return position;
};

template <typename T>
typename slist<T>::iterator slist<T>::begin() const
{
        return iterator(head->next);
};

template <typename T>
typename slist<T>::iterator slist<T>::end() const
{
        return iterator(tail);
};

template <typename T>
typename slist<T>::reverse_iterator slist<T>::rbegin() const
{
        return reverse_iterator(tail->prev);
};

template <typename T>
typename slist<T>::reverse_iterator slist<T>::rend() const
{
        return reverse_iterator(head);
};

template <typename T>
typename slist<T>::const_iterator slist<T>::cbegin() const
{
        return const_iterator(head->next);
};

template <typename T>
typename slist<T>::const_iterator slist<T>::cend() const
{
        return const_iterator(tail);
};

template <typename T>
typename slist<T>::const_reverse_iterator slist<T>::crbegin() const
{
        return const_reverse_iterator(tail->prev);
};

template <typename T>
typename slist<T>::const_reverse_iterator slist<T>::crend() const
{
        return const_reverse_iterator(head);
};

template <typename T>
bool slist<T>::empty() const
{
```

```
        return (_size == 0);
}

template <typename T>
T & slist<T>::front() const
{
        assert(head->next != tail);
        return head->next->data;
}

template <typename T>
T & slist<T>::back() const
{
        assert(head != tail->prev);
        return tail->prev->data;
}

template <typename T>
void slist<T>::pop_front()
{
        assert(_size != 0);

        list_elem * curr = head->next;

        curr->next->prev = curr->prev;
        curr->prev->next = curr->next;
        delete curr;

        _size--;
};

template <typename T>
void slist<T>::pop_back()
{
        assert(_size != 0);

        list_elem * curr = tail->prev;

        curr->next->prev = curr->prev;
        curr->prev->next = curr->next;
        delete curr;

        _size--;
};

template <typename T>
void slist<T>::clear()
{
        list_elem * curr = head->next;

        while (curr != tail)
        {
                curr->next->prev = curr->prev;
                curr->prev->next = curr->next;

                list_elem * tmp = curr;
                curr = curr->next;
                delete tmp;
        }
```

```
        _size = 0;
};

template <typename T>
slist<T> & slist<T>::operator =(const slist<T> & rhs)
{
        clear();

        for (slist<T>::iterator it = rhs.begin(); it != rhs.end(); ++it)
        {
                push(*it);
                _size++;
        }

        return *this;
};

template <typename T>
void slist<T>::remove(const T & val)
{
        list_elem * curr = head->next;

        while (curr != tail)
        {
                if (curr->data == val)
                {
                        curr->next->prev = curr->prev;
                        curr->prev->next = curr->next;

                        list_elem * tmp = curr;
                        curr = curr->next;
                        delete tmp;

                        _size--;
                }
                else curr = curr->next;
        }
}

template <typename T>
void slist<T>::unique()
{
        if (size() == 0) return;

        list_elem * curr = head->next;
        T curr_val = curr->data;

        curr = curr->next;
        while (curr != tail)
        {
                if (curr->data == curr_val)
                {
                        curr->next->prev = curr->prev;
                        curr->prev->next = curr->next;

                        list_elem * tmp = curr;
                        curr = curr->next;
                        delete tmp;
```

```
                        _size--;
                }
                else
                {
                        curr_val = curr->data;
                        curr = curr->next;
                }
        }
}

template <typename T>
void slist<T>::merge(slist<T> & x)
{
        for (slist<T>::iterator it = x.begin(); it != x.end(); ++it)
        {
                push(*it);
                _size++;
        }

        x.clear();
}

template <typename T>
void slist<T>::assign(size_t n, const T & val)
{
        clear();

        for (size_t i = 0; i < n; i++)
        {
                push(val);
                _size++;
        }
}

template <typename T>
void slist<T>::assign(iterator first, iterator last)
{
        clear();

        while (first != last)
        {
                push(*first);
                _size++;
                ++first;
        }
}

template <typename T>
void slist<T>::assign(reverse_iterator first, reverse_iterator last)
{
        clear();

        while (first != last)
        {
                push(*first);
                _size++;
                ++first;
        }
```

```
}

template <typename T>
void slist<T>::swap(slist<T> & x)
{
          slist<T> tmp = *this;
          *this = x;
          x = tmp;
}

//*********LIST_ELEM IMPLEMENTATION*********//
template <typename T>
slist<T>::list_elem::list_elem()
: next(NULL), prev(NULL)
{};

template <typename T>
slist<T>::list_elem::list_elem(T _data)
: next(NULL), prev(NULL), data(_data)
{};


//*********ITERATOR IMPLEMENTATION*********//
template <typename T>
slist<T>::iterator::iterator()
: current(NULL)
{};

template <typename T>
slist<T>::iterator::iterator(list_elem * p)
: current(p)
{};

template <typename T>
T & slist<T>::iterator::operator *() const
{
          assert(current->prev != NULL);
          assert(current->next != NULL);

          return current->data;
};

template <typename T>
typename slist<T>::iterator & slist<T>::iterator::operator ++()
{
          assert(current->next != NULL);

          current = current->next;
          return *this;
};

template <typename T>
typename slist<T>::iterator slist<T>::iterator::operator ++(int)
{
          assert(current->next != NULL);

          iterator temp = *this;
          ++(*this);
          return temp;
```

```
};

template <typename T>
bool slist<T>::iterator::operator !=(const iterator & rhs) const
{
          return current != rhs.current;
};

template <typename T>
bool slist<T>::iterator::operator ==(const iterator & rhs) const
{
          return current == rhs.current;
};


template <typename T>
typename slist<T>::iterator & slist<T>::iterator::operator --()
{
          assert(current->prev != NULL);

          current = current->prev;
          return *this;
};

template <typename T>
typename slist<T>::iterator slist<T>::iterator::operator --(int)
{
          assert(current->prev != NULL);

          iterator temp = *this;
          --(*this);
          return temp;
};


//*********REVERSE_ITERATOR IMPLEMENTATION*********//
template <typename T>
typename slist<T>::reverse_iterator & slist<T>::reverse_iterator::operator ++()
{
          assert(current->prev != NULL);

          this->current = this->current->prev;
          return *this;
};

template <typename T>
typename slist<T>::reverse_iterator slist<T>::reverse_iterator::operator ++(int)
{
          assert(current->prev != NULL);

          reverse_iterator temp = *this;
          --(*this);
          return temp;
};

template <typename T>
slist<T>::reverse_iterator::reverse_iterator()
: iterator()
{};
```

```
template <typename T>
slist<T>::reverse_iterator::reverse_iterator(list_elem * p)
: iterator(p)
{};

template <typename T>
typename slist<T>::reverse_iterator & slist<T>::reverse_iterator::operator --()
{
          assert(this != reverse_iterator(tail));

          current = current->next;
          return *this;
};

template <typename T>
typename slist<T>::reverse_iterator slist<T>::reverse_iterator::operator --(int)
{
          assert(current->next != NULL);

          reverse_iterator temp = *this;
          ++(*this);
          return temp;
};

//*********CONST_ITERATOR IMPLEMENTATION*********//
template <typename T>
T slist<T>::const_iterator::operator *() const
{
          assert(current->prev != NULL);
          assert(current->next != NULL);

          return current->data;
};

template <typename T>
slist<T>::const_iterator::const_iterator()
: current(NULL)
{};

template <typename T>
slist<T>::const_iterator::const_iterator(list_elem * p)
: current(p)
{};

template <typename T>
typename slist<T>::const_iterator & slist<T>::const_iterator::operator ++()
{
          assert(current->prev != NULL);

          this->current = this->current->next;
          return *this;
};

template <typename T>
typename slist<T>::const_iterator slist<T>::const_iterator::operator ++(int)
{
          assert(current->prev != NULL);
```

```
        const_iterator temp = *this;
        ++(*this);
        return temp;
};

template <typename T>
typename slist<T>::const_iterator & slist<T>::const_iterator::operator --()
{
        assert(current->prev != NULL);

        current = current->prev;
        return *this;
};

template <typename T>
typename slist<T>::const_iterator slist<T>::const_iterator::operator --(int)
{
        iterator temp = *this;
        --(*this);
        return temp;
};

//*********CONST_REVERSE_ITERATOR IMPLEMENTATION*********//

template <typename T>
T slist<T>::const_reverse_iterator::operator *() const
{
        assert(current->prev != NULL);
        assert(current->next != NULL);

        return current->data;
};

template <typename T>
slist<T>::const_reverse_iterator::const_reverse_iterator()
: current(NULL)
{};

template <typename T>
slist<T>::const_reverse_iterator::const_reverse_iterator(list_elem * p)
: current(p)
{};

template <typename T>
typename slist<T>::const_reverse_iterator & slist<T>::const_reverse_iterator::operator ++()
{
        assert(current->prev != NULL);

        this->current = this->current->prev;
        return *this;
};

template <typename T>
typename slist<T>::const_reverse_iterator slist<T>::const_reverse_iterator::operator ++(int)
{
        const_iterator temp = *this;
        --(*this);
        return temp;
};
```

```
template <typename T>
typename slist<T>::const_reverse_iterator & slist<T>::const_reverse_iterator::operator --()
{
          current = current->next;
          return *this;
};

template <typename T>
typename slist<T>::const_reverse_iterator slist<T>::const_reverse_iterator::operator --(int)
{
          iterator temp = *this;
          ++(*this);
          return temp;
};
```