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»**

**LAB №2**

**STL Container**

course «OOP»

Student

Gr. 33508/3 Evgeniy G. Protsenko

Instructor Andrey Y. Chernov

Saint-Petersburg

2016

# contents

[1 Task 3](#_Toc463314312)

[2 Introduction 4](#_Toc463314313)

[3 Conclusion 5](#_Toc463314314)

[References 6](#_Toc463314315)

[Appendix A 7](#_Toc463314316)

# 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.

# 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

# 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;

};