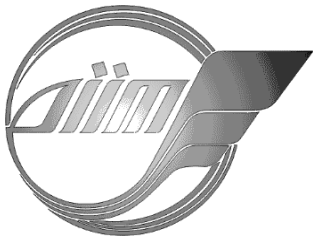
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ



Дніпровський національний університет
залізничного транспорту імені академіка В. Лазаряна

Кафедра «Комп'ютерні інформаційні технології»

Лабораторна робота №6

з дисципліни «Об'єктно-орієнтоване програмування»

на тему: «Спадкування»

Виконав:
студент гр.ПЗ1911
Сафонов Д.Є.
Прийняла:
Демидович І. М.

Дніпро, 2020

Тема. Спадкування

1 Постановка задачі згідно загального та індивідуального завдання.

Варіант	Клас	Нащадок
6	Список	Двозв’язний список, стек

2 Текст програми.

“main.cpp”

```
#include "forward_list.h"
#include "list.h"
#include "stack.h"
#include <iostream>

int main() {
	SequenceContainers::Stack<int> s;
	std::cout << "push 0:4 ";
	for(int i = 0; i < 5; i++) {
		s.push(i);
		std::cout << s.top() << " ";
	}
	std::cout << "\n";

	std::cout << "s.size() " << s.size() << "\n";
	std::cout << "s.empty() " << s.empty() << "\n";

	std::cout << "pop 5times ";
	for(int i = 0; i < 5; i++)
		std::cout << s.pop() << " ";
```

```
    std::cout << "\n";
//list test
    SequenceContainers::List<int> l = {1, 2, 3};
    for (auto i : l)
        std::cout << i << " ";
    std::cout << "\n";

    std::cout << "push_back(4) ";
    l.push_back(4);
    for (auto i : l)
        std::cout << i << " ";
    std::cout << "\n";

    std::cout << "pop_back " << l.pop_back() << "\n";
    std::cout << " l after pop_back ";
    for (auto i : l)
        std::cout << i << " ";
    std::cout << "\n";

    std::cout << "l.back " << l.back() << "\n";
    std::cout << "l.size " << l.size() << "\n";

    std::cout << "l.rbegin() : l.rend() ";
    for (auto i = l.rbegin(); i != l.rend(); i++)
        std::cout << *i << " ";
    std::cout << "\n";

    l.reverse();
```

```
	std::cout << "l.reverse() ";
	for (auto i : l)
		std::cout << i << " ";
	std::cout << "\n";
//fl tes
	//SequenceContainers::ForwardList<int> fl0 {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
	/*while(1) {
		fl0.insert(fl0.begin(), 10);
		fl0.erase(++fl0.begin());


		for (auto i : fl0)
			std::cout << i << " ";
		std::cout << "\n";
	}*/

	auto il = {10, 20, 30, 40, 50, 60, 70, 80, 90, 0};
	//SequenceContainers::List<int> fl(il);//ForwardList
	SequenceContainers::List<int> fl{0, 90, 80, 70, 60, 50, 40, 30, 20, 10};
	fl.reverse();//test
	std::cout << "fl: ";
	for (auto i : fl)
		std::cout << i << " ";
	std::cout << "\n";

	std::cout << "il: ";
	for (auto i : il)
		std::cout << i << " ";
```

```
	std::cout << "\n";

	std::cout << "pop: " << fl.pop_front() << "\n";
	std::cout << "after pop: ";
	for (auto i : fl)
		std::cout << i << " ";
	std::cout << "\n";

	fl.push_front(10);
	std::cout << "after push(10): ";
	for (auto i : fl)
		std::cout << i << " ";
	std::cout << "\n";

	auto fl_begin_pp = ++fl.begin();
	std::cout << "++fl.begin(): " << *fl_begin_pp << "\n";

	fl.erase(fl_begin_pp);
	std::cout << "after fl.erase(fl_begin_pp): ";
	for (auto i : fl)
		std::cout << i << " ";
	std::cout << "\n";



	for (; *fl_begin_pp != 70; fl_begin_pp++) {}
	std::cout << "*fl_begin_pp after <<for (; *fl_begin_pp != 70; fl_begin++) {}>>: " <<
*fl_begin_pp << "\n";
```

```
    fl.erase(fl_begin_pp);
    std::cout << "after fl.erase(fl_begin_pp): ";
for (auto i : fl)
    std::cout << i << " ";
std::cout << "\n";

    fl.insert(++fl.begin(), 35);
    std::cout << "after fl.insert(++fl.begin(), 35): ";
    for (auto i : fl)
            std::cout << i << " ";
    std::cout << "\n";

    fl.clear();
    std::cout << "fl.empty(): " << std::boolalpha << fl.empty() << "\n";

    fl = {1, 2, 3};

    std::cout << "fl = {1, 2, 3}: ";
    for (auto i : fl)
            std::cout << i << " ";
    std::cout << "\n";


    SequenceContainers::List<int> fl2 = fl;
    std::cout << "fl2 = fl: ";
    for (auto i : fl2)
            std::cout << i << " ";
    std::cout << "\n";
```

```
	fl.clear();
	std::cout << "fl.clear(), fl2: ";
	for (auto i : fl2)
		std::cout << i << " ";
	std::cout << "\n";

	fl = {1, 2, 3};
	fl.reverse();
	std::cout << "fl.front() on reversed 1,2,3:" << fl.front() << "\n";

	fl.push_front(99);
	std::cout << "fl.front() after fl.push_front(99): " << fl.front() << "\n";

	//fl.erase(fl.end());
	//std::cout << "fl.erase(fl.end()): " << fl.front() << "\n";

	//fl.insert(fl.end(), 101);
	//std::cout << "fl.insert(fl.end(), 101): " << *++fl.begin() << "\n";
}
```

“sequence_containers.h”

```
#include <initializer_list>
#include <cstddef>//size_t

namespace SequenceContainers {
	template<class T> class ForwardList;
	template<class T> class Stack;
	template<class T> class List;
}
```

“forward_list.h”

```
#include "sequence_containers.h"
#include <initializer_list>
#include <stdexcept>
#include <optional>

using SequenceContainers::ForwardList;

template<class T>
class ForwardList {
public:
	class Iterator;
protected:
	Iterator* _front = NULL;
private:
	template<class InputIterator>
	void assign(InputIterator first, InputIterator last);
	std::optional<Iterator*> _find_prev(const Iterator& position);

public:
	ForwardList(std::initializer_list<T> il) {assign(il.begin(), il.end());}
	ForwardList(ForwardList& fl) {assign(fl.begin(), fl.end());}
	ForwardList() {}
	~ForwardList() {clear();}
	ForwardList& operator= (const ForwardList& fl);
	ForwardList& operator= (std::initializer_list<T> il);
	virtual Iterator begin() {return _front == NULL ? end() : *_front;}
	static Iterator end() {return Iterator();}
```

```
        bool empty() {return begin() == end();}
        T& front();
        void push_front(const T& val) {_front = new Iterator(val, _front);}
        T pop_front();
        Iterator insert(const Iterator position, const T& val);
        Iterator erase(const Iterator position);
        void clear() {while(!empty()) pop_front();}
};

template<class T>
class ForwardList<T>::Iterator {
protected:
        std::optional<T> data;
        ForwardList<T>::Iterator* next = NULL;
        virtual void insert_after(const T& val) {next = new ForwardList<T>::Iterator(val, next);}
public:
        Iterator() {}
        Iterator(const T& data, Iterator* next);
        Iterator(const Iterator& other);
        virtual ~Iterator() {}
        Iterator& operator= (const Iterator& other);
        bool operator== (const Iterator& other) {return next == other.next && data == other.data;}
        friend class ForwardList<T>;
        friend SequenceContainers::Stack<T>;
        T& operator* ();
        Iterator operator++ (int);//postfix
        Iterator& operator++ () {return *this = next == NULL ? end() : *next;}
};
```

```
template<class T>
T& ForwardList<T>::front() {
	if (empty()) throw(std::out_of_range("front call on empty ForwardList"));
	return **_front;
}

template<class T>
ForwardList<T>& ForwardList<T>::operator= (const ForwardList& fl) {
	assign(fl.begin(), fl.end());
	return *this;
}

template<class T>
ForwardList<T>& ForwardList<T>::operator= (std::initializer_list<T> il) {
	assign(il.begin(), il.end());
	return *this;
}

template<class T> template<class InputIterator>
void ForwardList<T>::assign (InputIterator first, InputIterator last) {
	if(!empty()) clear();
	_front = new ForwardList<T>::Iterator(*first++, NULL);
	for(auto i = _front; first != last; first++, i = i->next)
		i->insert_after(*first);
}

template<class T>
```

```
T ForwardList<T>::pop_front() {
	ForwardList<T>::Iterator* tmp = _front;
	_front = _front->next;
	T val = **tmp;
	delete tmp;
	return val;
}

template<class T>
typename ForwardList<T>::Iterator ForwardList<T>::insert(ForwardList<T>::Iterator position, const T& val) {
	if (position == end()) throw(std::out_of_range("inserting after past the end iterator"));
	std::optional<ForwardList<T>::Iterator*> it;
	if (!empty() && position == *_front) it = _front;
	else if (auto prev = _find_prev(position)) it = (*prev)->next;
	if(it) (*it)->next = new ForwardList<T>::Iterator(val, position.next);
	else throw(std::bad_exception("bad iterator to insert after"));
	return *((*it)->next);
}

template<class T>
typename std::optional<typename ForwardList<T>::Iterator*> ForwardList<T>::_find_prev(const ForwardList<T>::Iterator& position) {
	std::optional<ForwardList<T>::Iterator*> prev;
	for(auto i = _front; i && !prev; i = i->next) {
		if (i->next && *(i->next) == position) prev = i;
	}
	return prev;
}
```

```
template<class T>
typename ForwardList<T>::Iterator ForwardList<T>::erase(const ForwardList<T>::Iterator position) {
	std::optional<ForwardList<T>::Iterator*> prev = _find_prev(position);
	if (prev) {
		delete (*prev)->next;
		(*prev)->next = position.next;
	}
	return prev ? **prev : begin();
}

template<class T>
ForwardList<T>::Iterator::Iterator(const ForwardList<T>::Iterator& other) {
	if (other.data) data = *(other.data);
	next = other.next;
}

template<class T>
ForwardList<T>::Iterator::Iterator(const T& data, ForwardList<T>::Iterator* next) {
	this->data = data;
	this->next = next;
}

template<class T>
typename ForwardList<T>::Iterator& ForwardList<T>::Iterator::operator= (const
ForwardList<T>::Iterator& other) {
	data = other.data;
	next = other.next;
	return *this;
```

```
}

template<class T>
T& ForwardList<T>::Iterator::operator* () {
        if (!data) throw(std::out_of_range("dereferencing past the end iterator"));
        return *data;
}

template<class T>
typename ForwardList<T>::Iterator ForwardList<T>::Iterator::operator++ (int) {
        ForwardList<T>::Iterator tmp = *this;
        operator++();
        return tmp;
}
```

“list.h”

```
#include "sequence_containers.h"



using SequenceContainers::List;


template<class T>
class SequenceContainers::List : public SequenceContainers::ForwardList<T> {
public:
	class Iterator;
private:
	T pop(bool side);
	Iterator begin_or_end(bool side) const;
	T& front_or_back(bool side);
	void push(const T& val, bool side);
	bool _reversed = false;
	Iterator* _front = NULL;
	Iterator* _back = NULL;
	template<class InputIterator>
	void assign(InputIterator first, InputIterator last);
public:
	List() : ForwardList<T>::ForwardList() {}
	List(std::initializer_list<T> il) : ForwardList<T>::ForwardList() {assign(il.begin(), il.end());}
	List(const List& other) : ForwardList<T>::ForwardList() {assign(other.begin(), other.end());}
	bool empty() const {return _front == NULL;}
	List& operator=(const List& other) {assign(other.begin(), other.end()); return *this;}
	Iterator rbegin() const {return begin_or_end(!_reversed);}
	static Iterator end() {return Iterator();}
```

```
        static Iterator rend() {return end();}
        Iterator begin() const {return begin_or_end(_reversed);}
        void reverse() {_reversed ^= 1;}
        size_t size();
    T& back() {return front_or_back(_reversed);}
        T& front() {return front_or_back(!_reversed);}
        void push_front(const T& val) {push(val, !_reversed);}
        T pop_front() {return pop(!_reversed);}
        void push_back(const T& val) {push(val, _reversed);}
        T pop_back() {return pop(_reversed);}
        Iterator erase(const Iterator position);
        Iterator insert(const Iterator position, const T& val);
};

template<class T>
class List<T>::Iterator : public ForwardList<T>::Iterator {
        List<T>::Iterator& next_or_prev(bool side);
        bool reversed = false;
        List<T>::Iterator* prev = NULL;
        List<T>::Iterator* next = NULL;
        void insert_after(const T& val) override {next = new Iterator(val, next, this, false);}//reversed =
false!!!!!!!!!!!!!!
public:
        Iterator() {}
        Iterator(const Iterator& other) : ForwardList<T>::Iterator::Iterator(other),
reversed(other.reversed) {operator=(other);}
        Iterator(const T& data, Iterator* next, Iterator* prev, bool reversed) :
ForwardList<T>::Iterator::Iterator(data, next), reversed(reversed), prev(prev), next(next) {}
        Iterator operator--(int) {Iterator tmp = *this; operator--(); return tmp;}
```

```
        Iterator& operator--() {return next_or_prev(reversed);}
        Iterator& operator=(const Iterator& other) {ForwardList<T>::Iterator::operator=(other); prev = other.prev; next = other.next; reversed = other.reversed; return *this;}
        Iterator& operator++ () {return next_or_prev(!reversed);}
        Iterator operator++ (int) {Iterator tmp = *this; operator++(); return tmp;}

        bool operator== (const Iterator& other) {return ForwardList<T>::Iterator::operator==(other) && prev == other.prev;}
        friend class List<T>;
};

template<class T> template<class InputIterator>
void List<T>::assign(InputIterator first, InputIterator last) {
        if(!empty()) ForwardList<T>::clear();
        _reversed = false;
        _front = new List<T>::Iterator(*first++, NULL, NULL, false);
        _back = _front;
        for(; first != last; first++, _back = _back->next)
                _back->insert_after(*first);
}

template<class T>
List<T>::Iterator& List<T>::Iterator::next_or_prev(bool side) {
        auto t = side ? next : prev;
        bool tmp = reversed;
        *this = t ? *t : end();
        reversed = tmp;
        return *this;
}
```

```
template<class T>
size_t List<T>::size() {
	size_t ret = 0;
	if (!empty()) {
		auto i = _front;
		auto j = _back;
		for(; i != j && j->next != i; i = i->next, j = j->prev)
			ret += 2;
		ret += i == j;
	}
	return ret;
}

template<class T>
List<T>::Iterator List<T>::begin_or_end(bool side) const {
	if(empty()) return end();
	List<T>::Iterator t;
	t = *(side ? _back : _front);
	t.reversed = side;
	return t;
}

template<class T>
T& List<T>::front_or_back(bool side) {
	if(empty()) throw(std::out_of_range("element access on empty List"));
	return **(side ? _front : _back);
}
```

```
template<class T>
void List<T>::push(const T& val, bool side) {
	if(side) {
		_front = new List<T>::Iterator(val, _front, NULL, false);
		_front->next->prev = _front;
	} else {
		_back = new List<T>::Iterator(val, NULL, _back, false);
		_back->prev->next = _back;
	}
}

template<class T>
T List<T>::pop(bool side) {
	List<T>::Iterator* tmp;
	if(side) {
		tmp = _front;
		_front = _front->next;
		_front->prev = NULL;
	} else {
		tmp = _back;
		_back = _back->prev;
		_back->next = NULL;
	}
	T val = **tmp;
	delete tmp;
	return val;
}
```

```
template<class T>
typename List<T>::Iterator List<T>::erase(const List<T>::Iterator position) {
	if(empty())
		throw(std::out_of_range("erase on empty List"));
	List<T>::Iterator* it = NULL;
	for (auto i = _front; i != NULL; i = i->next)
		if (*i == position) it = i;
	if(!it) throw(std::out_of_range("bad iterator to erase"));
	if (position.prev) position.prev->next = position.next;
	if (position.next) position.next->prev = position.prev;
	delete it;
	return position;
}

template<class T>
typename List<T>::Iterator List<T>::insert(const List<T>::Iterator position, const T& val) {
	if (position == end()) throw(std::out_of_range("inserting after past the end iterator"));
	List<T>::Iterator* it = NULL;
	for (auto i = _front; i != NULL; i = i->next)
		if (*i == position) it = i;
	if(!it) throw(std::out_of_range("bad iterator to insert"));
	if(_reversed)
		return *(it->prev = new List<T>::Iterator(val, it, it->prev, false));
	else
		return *(it->next = new List<T>::Iterator(val, it->next, it, false));
}
```

“stack.h”

```
#include "sequence_containers.h"


template<class T>
class SequenceContainers::Stack : private SequenceContainers::ForwardList<T> {
public:
	size_t size() {size_t size = 0; for(auto i = this->_front; i != NULL; i = i->next) size++; return size;}
	using ForwardList<T>::ForwardList;
	T& top() {return ForwardList<T>::front();}
	void push(const T& val) {ForwardList<T>::push_front(val);}
	T pop() {return ForwardList<T>::pop_front();}
	using ForwardList<T>::empty;
};
```

3 Результати виконання.

```
push 0:4 0 1 2 3 4
s.size() 5
s.empty() 0
pop 5times 4 3 2 1 0
1 2 3
push_back(4) 1 2 3 4
pop_back 4
 l after pop_back 1 2 3
l.back 3
l.size 3
l.rbegin() : l.rend() 3 2 1
l.reverse() 3 2 1
fl: 10 20 30 40 50 60 70 80 90 0
il: 10 20 30 40 50 60 70 80 90 0
pop: 10
after pop: 20 30 40 50 60 70 80 90 0
after push(10): 10 20 30 40 50 60 70 80 90 0
++fl.begin(): 20
after fl.erase(fl_begin_pp): 10 30 40 50 60 70 80 90 0
*fl_begin_pp after <<for (; *fl_begin_pp != 70; fl_begin++) {}>>: 70
after fl.erase(fl_begin_pp): 10 30 40 50 60 80 90 0
after fl.insert(++fl.begin(), 35): 10 30 35 40 50 60 80 90 0
fl.empty(): false
fl = {1, 2, 3}: 1 2 3
fl2 = fl: 1 2 3
fl.clear(), fl2: 1 2 3
fl.front() on reversed 1,2,3:3
fl.front() after fl.push_front(99): 99
```

4 Висновок.

Спадкування — метод скорочення коду завдяки тому, що успадкованні класи та їх об’єкти мають спільний функціонал. У C++ це реалізовано так, що успадкований класс якюи має поле з ім’ям суперклассу, тож можна викликати функції, які були переписані.