МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Объектно-ориентированное программирование»

Тема: Создание шаблонных контейнеров и итератора

Студент гр. 6382 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Кацер И.С.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Филатов А.Ю.

Санкт-Петербург

2018

**Цель работы.**

Создание шаблонных контейнеров, создание собственных внешних итераторов. Научиться перегружать операторы.

**Постановка задачи.**

Необходимо создать свой контейнер (двусвязный список) и написать для него внешний итератор. В итераторе перегрузить основные операторы. Собрать проект с помощью Makefile.

**Описание программы.**

В программе описан шаблонный двусвязный список, который представлен классом List, в котором находятся указатели на голову (\*Head) и на хвост (\*Tail) списка.

Внутри класса List располагается класс Iterator, который содержит следующие функции:

const T& operator \*() const

T& operator \*()

T& operator [](int val)

const T& operator [](int val) const

Iterator operator ++()

Iterator operator ++(int)

Iterator operator --()

Iterator operator --(int)

friend bool operator !=(const Iterator& x, const Iterator& y)

friend bool operator ==(const Iterator& x, const Iterator& y)

В классе List представлены следующие функции:

1. void Show() – Вывод списока на экран.

2. void AddEnd(const T& data) - Добавление эл-та в конец списка.

3. void DeleteLast() - Удаление последнего эл-та.

4. void DeleteFirst() - Удаление первого эл-та.

5. void AddBegin(const T& data) - Добавление эл-та в начало списка.

6. void Clear() - Удаляет все эл-ты из списка.

7. void Delete(const int nomer) - Удаляет эл-т, на который указывает итератор номер.Если если список меньше, то эл-т удаляется с конца списка.

8. void Insert(const int nomer, const T& data) - Вставляет элемент в список перед элементом, на который указывает номер. Если список меньше, то эл-т добавляется в конец списка.

9. Iterator End() - Возвращает итератор на "конец" (для не константного списка).

10. const Iterator End() const - Возвращает константный итератор на "конец" (для константного списка).

11. Iterator Last() - Возвращает итератор на последний элемент (для не константного списка).

12. const Iterator Last() const - Возвращает константный итератор на последний элемент (для константного списка).

13. Iterator Begin() - Возвращает итератор на первый элемент (для не константного списка).

14. const Iterator First() const - Возвращает константный итератор на первый элемент (для константного списка).

15. ~List() - Деструктор (освобождает память).

16. List() - Конструктор пустого списка

**Выводы.**

В ходе работы был создан собственный контейнер (шаблонный двусвязный список), также для этого контейнера создан внешний итератор, в котором перегружены основные операторы. Также был создан Makefile, с помощью которого собирался проект.

**ПРИЛОЖЕНИЕ A (ИСХОДНЫЙ КОД)**

**MAKEFILE**

all: main.o main.cpp

g++ main.o -o out -static -std=c++11

main.o: main.cpp list.h

g++ main.cpp -c -o main.o -static -std=c++11

**LIST.H**

//---------------------------------------------------------------------------

#ifndef ListH

#define ListH

#include <iostream>

//---------------------------------------------------------------------------

/\*

Шаблон класса: связный список из элементов типа T.

\*/

template <class T>

class List {

private:

// Включаем определение структуры "узел списка".

// Узел списка.

struct Node

{

T Data; // данные (значение эл-та списка)

Node \*Prev, \*Next; // указатель на предыдущий и следующий узлы

// Создание отдельного узла.

Node(const T& data) // конструктор для удобства

: Data(data), // копируем данные;

Prev(NULL), // предыдущего и

Next(NULL) // последующего узлов нет

{

}

};

Node \*Head, // первый узел списка

\*Tail; // последний узел

// Делаем конструктор копии и операцию присваивания закрытыми.

List(const List&) { }

void operator =(const List&) { }

public:

// Включаем определение класса "итератор списка".

// Итератор списка.

class Iterator {

// Класс список объявляется дружественным для доступа к указателю на узел.

friend List<T>;

Node\* P; // ук-ль на узел списка, содержащий значение эл-та, на кот-й указывает итератор

public:

// Создает итератор, указывающий на эл-т списка, хранящийся в узле по адресу p.

// По умолчанию ни на что не указывает. Т. к. Node - закрытый тип класса List,

// то итератор на эл-т может быть создан только функциями-элементами List.

Iterator(Node\* p = NULL)

: P(p)

{

}

// Операция разыменовывания для константного итератора.

// Возвращает константую ссылку на значение эл-та.

const T& operator \*() const

{

return P->Data;

}

// Операция разыменовывания для не константного итератора.

// Возвращает не константного ссылку на значение эл-та, по которой

// оно может быть изменено.

T& operator \*()

{

return P->Data;

}

// Префиксная операция инкремента. Перемещает итератор к следующему эл-ту списка.

Iterator operator ++()

{

P = P->Next; // переходим к следующему эл-ту

return \*this; // возвращаем "увеличенный" итератор

}

// Постфиксная операция инкремента. Перемещает итератор к следующему эл-ту списка.

// Аргумент типа int указывает компилятору, что данная функция-операция operator ++

// является постфиксной.

Iterator operator ++(int)

{

Iterator t(\*this); // сохраняем "не увеличенное" значение итератора

P = P->Next; // переходим к следующему эл-ту

return t; // возвращаем "не увеличенное" значение итератора

}

// Префиксная операция декремента. Перемещает итератор к предыдущему эл-ту списка.

Iterator operator --()

{

P = P->Prev; // переходим к предыдущему эл-ту

return \*this; // возвращаем "уменьшенный" итератор

}

// Постфиксная операция декремента. Перемещает итератор к предыдущему эл-ту списка.

// Аргумент типа int указывает компилятору, что данная функция-операция operator --

// является постфиксной.

Iterator operator --(int)

{

Iterator t(\*this); // сохраняем "не уменьшенное" значение итератора

P = P->Prev; // переходим к предыдущему эл-ту

return t; // возвращаем "не уменьшенное" значение итератора

}

// Операция равенства.

friend bool operator ==(const Iterator& x,const Iterator& y)

{

return y.P == x.P;

}

// Операция неравенства.

friend bool operator !=(const Iterator& x,const Iterator& y)

{

return y.P != x.P;

}

};

// Конструктор пустого списка

List()

: Head(NULL), Tail(NULL)

{

}

// Деструктор (освобождает память).

~List()

{

Clear();

}

// Возвращает константный итератор на первый элемент (для константного списка).

const Iterator First() const

{

return Iterator(Head);

}

// Возвращает итератор на первый элемент (для не константного списка).

Iterator Begin()

{

return Iterator(Head);

}

// Возвращает константный итератор на последний элемент (для константного списка).

const Iterator Last() const

{

return Iterator(Tail);

}

// Возвращает итератор на последний элемент (для не константного списка).

Iterator Last()

{

return Iterator(Tail);

}

// Возвращает константный итератор на "конец" (для константного списка).

const Iterator End() const

{

return Iterator(NULL);

}

// Возвращает итератор на "конец" (для не константного списка).

Iterator End()

{

return Iterator(NULL);

}

// Проверка пустоты списка.

bool Empty() const

{

return !Head;

}

// Вставляет элемент в список перед элементом, на который указывает номер.

// Если если список меньше, то эл-т добавляется в конец списка.

void Insert(const int nomer, const T& data);

// Удаляет эл-т, на который указывает итератор номер.

// Если если список меньше, то эл-т удаляется с конца списка.

void Delete(const int nomer);

// Удаляет все эл-ты из списка.

void Clear();

// Добавление эл-та в начало списка.

void AddBegin(const T& data) ;

// Добавление эл-та в конец списка.

void AddEnd(const T& data) ;

// Удаление первого эл-та.

void DeleteFirst();

// Удаление последнего эл-та.

void DeleteLast();

// операция вывода списка

void show()

{

Iterator i;

std::cout << '[';

// Выводим все элементы кроме последнего, разделяя их запятой.

for (i = Begin(); i != Last(); i++)

std::cout << \*i << ", ";

// Выводим последний элемент (если список не пуст).

if (i != End()) std::cout << \*i;

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

}

};

//---------------------------------------------------------------------------

// Определение не встраиваемых функций-элементов

//---------------------------------------------------------------------------

template <class T>

void List<T>::Insert(const int nomer, const T& data)

{

// Если список не пуст, то

if ((Head) && (nomer!=0))

{

int m;

Iterator i;

m=0;

i=Begin();

// Ищем узел X, перед которым нужно вставить.

while( (m<nomer) && (i!= End()) )

{

m++;

i++;

}

// Если iter указывает на "конец", то добавляем эл-т в конец списка.

if (i==End())

{

AddEnd(data);

}

// Иначе вставляем эл-т в "середину" списка.

else

{

Node \*t,

\*n = new Node(data); // создаем отдельный узел

t = i.P->Prev; // в t заносим предшествующий X

t->Next = n; // связываем предшествующий X с новым

n->Prev = t; // связываем новый с предшествующим X

n->Next = i.P; // связываем новый с X

i.P->Prev = n; // связываем X с новым

}

}

// Иначе вставляем эл-т в начало

else

{

AddBegin(data);

}

}

//---------------------------------------------------------------------------

template <class T>

void List<T>::Delete(const int nomer)

{

// Если список не пуст, то

if (Head)

{

int m=0;

Iterator i;

i=Begin();

// Ищем узел X, который нужно удалить.

while( (m<nomer) && (i!= End()) )

{

m++;

i++;

}

// Если X первый.

if (i == Begin())

{

DeleteFirst();

}

// Иначе, если X последний.

else if ((i==Last()) || (i==End()))

{

void DeleteLast();

}

// Иначе X средний.

else

{

Node \*t, \*pred, \*sled;

t=i.P; //удаляемый элемент

pred=t->Prev; //предыдущий

sled=t->Next; //следующй

pred->Next = t->Next; // связываем предыдущий X со следующим за X,

sled->Prev = t->Prev; // отделяя X от списка

delete t; // Удаляем (уничтожаем и освобождаем память).

}

}

}

//---------------------------------------------------------------------------

template <class T>

void List<T>::Clear()

{

// Если список не пуст.

if (!Empty()) {

Node \*n;

Iterator i;

// Удаляем все узлы.

for (i = Begin(); i != End(); ) {

n = i.P; // сохраняем адрес узла

i++; // переходим к следующему

delete n; // и удаляем текущий

}

// Сбрасываем указатели на первый и последний.

Head = Tail = NULL;

}

}

//---------------------------------------------------------------------------

//Добавление эл-та в начало списка.

template <class T>

void List<T>::AddBegin(const T& data)

{

Node \*n=new Node(data);

if (Head)

// связываем первый узел с новым;

Head->Prev = n;

else

// иначе новый первый узел является и последним.

Tail = n;

// Связываем новый узел с первым,

n->Next = Head;

// и делаем его первым.

Head = n;

}

//---------------------------------------------------------------------------

//Добавление эл-та в конец списка.

template <class T>

void List<T>::AddEnd(const T& data)

{

Node \*n=new Node(data);

if (Head)

// связываем последний узел с новым;

Tail->Next = n;

else

// иначе новый последний узел является и первым.

Head = n;

// Связываем новый узел с последним,

n->Prev = Tail;

// и делаем новый последним.

Tail = n;

}

//---------------------------------------------------------------------------

// Удаление первого эл-та.

template <class T>

void List<T>::DeleteFirst()

{

Node \*t=Head;

Head = t->Next;

if (!Head) Tail = NULL;

delete t;

}

//---------------------------------------------------------------------------

// Удаление последнего эл-та.

template <class T>

void List<T>::DeleteLast()

{

Node \*t=Tail;

Tail = t->Prev;

if (!Tail) Head = NULL;

delete t;

}

//---------------------------------------------------------------------------

#endif

**MAIN.CPP**

#include <iostream>

#include <cstdlib>

#include <ctime>

#include "List.h"

using std::cout;

using std::cin;

using std::endl;

// Определяем синоним L для списка целых чисел.

typedef List<int> L;

int main()

{

// Пустые списки m и l.

List<int> l, m;

List<int>::Iterator i, j;

int c;

// Добавляем в конец l десять случайных чисел от 1 до 10.

srand( time( 0 ) ); // автоматическая рандомизация

for (c = 10; c--; )

l.PushBack(rand() % 10 + 1);

cout << "Push back 10 random el-ts to l.\nl = " ;

l.show();

cout << endl;

// Удваиваем элемнты l.

for (i = l.Begin(), c=0; // начинаем с первого

i != l.End(); // продолжаем, пока не "выскочим" за пределы списка

i++,c+=2) // затем переходим к следующему за i-м.

{

// Сначала вставляем элемент перед c-м эл-том его компию,

//l.Insert(c, \*i);

l.show();

//cout << endl;

}

cout << "Double each el-t in l.\nl = " ;

l.show();

cout << endl;

// Вставляем эл-ты l в m в обратном порядке.

for (i = l.Last(); // начинаем с последнего

i != l.End(); // продолжаем, пока не "выскочим" за пределы списка

i--) // и переходим к предшествующему.

m.PushBack(\*i); // Добавляем в конец m i-й эл-т из l,

cout << "Push back l's el-ts to m in reverse order.\nm = " ;

m.show();

cout << endl;

c=0;

// Удаляем каждый второй эл-т в m (начиная с первого).

for (i = m.Begin(); // начинаем с первого

i != m.End(); // продолжаем, пока не "выскочим" за пределы списка

i++) // (\*) затем пропускаем оставляемый элемент

{

m.Delete(c++); // Удаляем c-й эл-т, предварительно смещая c на один эл-т вперед.

i++; //смещаем i на один эл-т вперед, (\*)

}

cout << "Delete each second el-t from m.\nm = " ;

m.show();

cout << endl;

// Вставляем в начало m три нуля.

for (c = 3; c--; )

m.PushFront(0);

cout << "Push front 3 zeros to m.\nm = " ;

m.show();

cout << endl;

// Добавляем к эл-м l значения соответствующих эл-в m, пока один из списков не кончится.

for (i = l.Begin(), j = m.Begin(); i != l.End() && j != m.End(); i++, j++)

\*i += \*j;

cout << "Add m's el-ts to l's el-ts until one of the lists isn't done.\nl = " ;

l.show();

cout << endl;

// Удаляем из m три первых нуля.

for (c = 3; c--; )

m.PopFront();

cout << "Remove lead zeros form m.\nm = " ;

m.show();

cout << endl;

// Очищаем список m.

m.Clear();

cout << "Remove all el-ts from m.\nm = " ;

m.show();

cout << endl;

cin.ignore(100, 'q');

}