Лабораторная работа № 7 (БДЗ №2)

**Применение стандартной библиотеки шаблонов (STL)**

**Цель работы:** Формирование навыка применения контейнеров , обобщенных алгоритмов, итераторов. Закрепление навыков работы с классами.

Литература по самостоятельному изучению стандартной библиотеки шаблонов (STL) :

1. **[Дердж Ж.](http://mexalib.com/author/%D0%94%D0%B5%D1%80%D0%B4%D0%B6%20%D0%96." \o "Дердж Ж.) [Сейни А.](http://mexalib.com/author/%D0%A1%D0%B5%D0%B9%D0%BD%D0%B8%20%D0%90." \o "Сейни А.) «C++ и STL. Справочное руководство»**

Данная книга представляет собой полное справочное руководство по данной теме. Она включает небольшой учебный курс, подробное описание каждого элемента библиотеки и большое количество примеров. В книге вы найдете подробное описание итераторов, обобщенных алгоритмов, контейнеров и многое другое

1. **Скотт Мейерс** - "Эффективное использование STL"
2. **Рэй Лишнер STL. Карманный справочник**   
   Краткий справочник по библиотеке STL в своем современном виде - алгоритмы, итераторы и контейнеры

**Теория**

**Основные концепции STL**

STL – Standard Template Library, стандартная библиотека шаблонов состоит из двух основных частей: на­бора контейнерных классов и набора обобщенных алгоритмов.

Контейнеры — это объекты, содержащие другие однотипные объекты. Контейнерные классы являются шаблонными, поэтому хранимые в них объекты могут быть как встроенных, так и пользовательских типов. Эти объекты должны допускать *копирование*и *присваивание.*Встроенные типы этим требованиям удовлетворяют; то же самое относится к классам, если конструктор копирования или операция присваи­вания не объявлены в них закрытыми или защищенными. Контейнеры STL реализуют основные структуры данных, используемые при написании программ.

Обобщенные алгоритмы реализуют большое количество процедур, применимых к контейнерам: поиск, сортировку, слияние и т. п. Однако они не являются методами контейнерных классов. Алгоритмы представлены в STL в форме глобальных шаблонных функций. Благодаря этому достигается универсальность: эти функции можно применять не только к объектам различных контейнерных классов, но также и к массивам. Независимость от типов кон­тейнеров достигается за счет косвенной связи функции с контейнером: в функцию передается не сам контейнер, а пара адресов first, last, задающая диапазон обрабатываемых элементов.

Реализация указанного механизма взаимодействия базируется на использовании так называемых итераторов.

Итераторы – это обобщение концепции указателей: они ссылаются на элементы контейнера. Их можно инкрементировать, как обычные указатели, для последовательного продвижения по контейнеру, а также разыменовывать для получения или изменения значения элемента.

**Контейнеры**

Контейнеры STL можно разделить на два типа: последовательные и ассоциа­тивные. В данной лабораторной работе мы будем рассматривать только последовательные контейнеры, которые обеспечивают хранение конечного количества однотипных объектов в виде непрерывной последовательности. К базовым после­довательным контейнерам относятся векторы (vector), списки (list) и двусто­ронние очереди (deque).

Для использования контейнера в программе необходимо включить в нее соответствующий заголовочный файл. Тип объектов, сохраняемых в контейнере, задается с помощью аргумента шаблона, например:

#include<vector>

#include<list>

#include"person.h"

…..

vector<int> v;

list<person> l;

## Итераторы

Рассмотрим, как можно реализовать шаблон функции для поиска элементов в массиве, который хранит объекты типа Data:

emplate **<**class **Data>**

Data\* Find(Data\*mas, int **n,** const **Data& key)**

{

for(int **i=0;i<n;i++)**

if **(\*(mas + i) == key)**

return **mas + i;**

return **0;**

}

Функция возвращает адрес найденного элемента или 0, если элемент с заданным значением не найден.

Эту функцию можно использовать для поиска элементов в массиве любого типа, но использовать ее для списка нельзя, поэтому авторы STL ввели понятие итератора. Итератор более общее понятие, чем указатель. Тип iterator определен для *всех*контейнерных классов STL, однако, реализация его в разных классах разная.

К основным операциям, выполняемым с любыми итераторами, относятся:

* Разыменование итератора: если р — итератор, то \*р — значение объекта, на ко­торый он ссылается.
* Присваивание одного итератора другому.
* Сравнение итераторов на равенство и неравенство (== и !=).
* Перемещение его по всем элементам контейнера с помощью префиксного (++р) или постфиксного (р++) инкремента.

Так как реализация итератора специфична для каждого класса, то при объявле­нии объектов типа итератор всегда указывается область видимости в форме имя\_шаблона::, например:

**vector<int>::iterator iterl;**

**List<person>::iterator iter2;**

Организация циклов просмотра элементов контейнеров тоже имеет некоторую специфику. Так, если i — некоторый итератор, то вместо привычной формы

for **(i =0; i < n; ++i)**

используется следующая:

for **(i = first; i != last; ++i),**

где first - значение итератора, указывающее на первый элемент в контейнере, a last — значение итератора, указывающее на воображаемый элемент, который следует *за*последним элементом контейнера.

Операция сравнения < заме­нена на операцию ! =, т. к. операции < и > для итераторов в общем случае не поддерживаются.

Для всех контейнерных классов определены унифицированные методы begin() и end(), возвращающие адреса first и last соответственно.

В STL существуют следующие типы итераторов: вход­ные, выходные, прямые, двунаправленные итераторы и итераторы произвольного доступа.

Входные итераторы (Inputlterator) используются алгоритмами STL для чтения значений из контейнера, аналогично тому, как программа может вводить данные из потока сin.

Выходные итераторы (Outputlterator) используются алгоритмами для записи значений в контейнер, аналогично тому, как программа может выводить данные в поток cout.

Прямые итераторы (Forwardlterator) используются алгоритмами для навигации по контейнеру только в прямом направлении, причем, они позволяют и читать, и изменять данные в контейнере.

Двунаправленные итераторы (Bidirectionallterator) имеют все свойства прямых итераторов, но позволяют осуществлять навигацию по контейнеру и в прямом, и в обратном направлениях (для них дополнительно реализованы операции префиксного и постфиксного декремента).

Итераторы произвольного доступа (RandomAccessIterator) имеют все свойства двунаправленных итераторов плюс операции (наподобие сложения указателей) для доступа к произвольному элементу контейнера.

Для двунаправленных итераторов и итераторов произвольного доступа определе­ны разновидности, называемые адаптерами итераторов. Адаптер, просматриваю­щий последовательность в обратном направлении, называется reverse\_iterator.

### Общие свойства контейнеров

Унифицированные типы, определенные в STL

|  |  |
| --- | --- |
| **Поле** | **Пояснение** |
| size\_type | Тип индексов, счетчиков элементов и т. д. |
| iterator | Итератор |
| const\_iterator | Константный итератор (значения элементов изменять запрещено) |
| reference | Ссылка на элемент |
| const\_reference | Константная ссылка на элемент (значение элемента изменять запрещено) |

Операции и методы, общие для всех контейнеров

|  |  |
| --- | --- |
| **Операция или метод** | **Пояснение** |
| Операции равенства (==) и неравенства (!=) | Возвращают значение true или false |
| Операция присваивания (=) | Копирует один контейнер в другой |
| clear | Удаляет все элементы |
| insert | Добавляет один элемент или диапазон элементов |
| erase | Удаляет один элемент или диапазон элементов |
| size\_type size() const | Возвращает число элементов |
| size\_type max\_size() const | Возвращает максимально допустимый размер контейнера |
| bool empty0 const | Возвращает true, если контейнер пуст |
| iterator begin() | Возвращают итератор на начало контейнера (итерации будут производиться в прямом направлении) |
| iterator end() | Возвращают итератор на конец контейнера (итерации в прямом направлении будут закончены) |
| reverse\_iterator begin() | Возвращают реверсивный итератор на конец контейнера (итерации будут производиться в обратном направлении) |
| reverse\_iterator end() | Возвращают реверсивный итератор на начало контейнера (итерации в обратном направлении будут закончены |

### Использование последовательных контейнеров

К основным последовательным контейнерам относятся *вектор*(vector), *список*(list) и *двусторонняя очередь*(deque).

Чтобы использовать последовательный контейнер, нужно включить в програм­му соответствующий заголовочный файл:

#include<vector>

#include<list>

#include<deque>

usingnamespace **std;**

Контейнер *вектор*является аналогом обычного массива, за исключением того, что он автоматически выделяет и освобождает память по мере необходимости. Контейнер эффективно обрабатывает произвольную выборку элементов с по­мощью операции индексации [] или метода at. Однако вставка элемента в лю­бую позицию, кроме конца вектора, неэффективна. Для этого потребуется сдви­нуть все последующие элементы путем копирования их значений. По этой же причине неэффективным является удаление любого элемента, кроме последнего.

Контейнер *список*организует хранение объектов в виде двусвязного списка. Каждый элемент списка содержит три поля: значение элемента, указатель на предше­ствующий и указатель на последующий элементы списка. Вставка и удаление работают эффективно для любой позиции элемента в списке. Однако список не поддерживает произвольный доступ к своим элементам: например, для выбор­ки n-го элемента нужно последовательно выбрать предыдущие п-1 элементов.

Контейнер *двусторонняя очередь*во многом аналогичен вектору, элементы хранятся в непрерывной области памяти. Но в отличие от вектора двусторонняя очередь эффективно поддерживает вставку и удаление первого элемента (так же, как и последнего).

Существует пять способов определить объект для последовательного контейнера.

1. Создать пустой контейнер:

**vector<int> vecl;**

**list<double> listl;**

2. Создать контейнер заданного размера и инициализировать его элементы зна­чениями по умолчанию:

**vector<int> vecl(100);**

**list<double> listl(20);**

3. Создать контейнер заданного размера и инициализировать его элементы ука­занным значением:

**vector<int> vecl(100, 0);**

**deque<float> decl(300, 0.0);**

4. Создать контейнер и инициализировать его элементы значениями диапазона (first, last) элементов другого контейнера:

**int arr[7] = (15, 2, 19, -3, 28, 6, 8);**

**vector<int> vl(arr, arr + 7);**

**list<int> lst(vl.beg() + 2, vl.end());**

5. Создать контейнер и инициализировать его элементы значениями элементов другого однотипного контейнера:

**vector<int> vl;**

**// добавить в vl элементы**

**vector<int> v2(vl);**

**Методы, которые поддерживают последовательные контейнеры**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Vector** | | **Deque** | | **List** | |
| push\_back() | добавление в конец | push\_back(T&key) | добавление в конец | push\_back(T&key) | добавление в конец |
| pop\_back() | удаление из конца | pop\_back() | удаление из конца | pop\_back() | удаление из конца |
| insert | Вставка в произвольное место | push\_front(T&key) | добавление в начало | push\_front(T&key) | добавление в начало |
| erase | удаление из произволь-  ного места | pop\_front() | удаление из начала | pop\_front() | удаление из начала |
| []  at | доступ к произвольному элементу | insert | Вставка в произвольное место | insert | Вставка в произвольное место |
|  |  | erase | удаление из произвольног оместа | erase | удаление из произвольног оместа |
|  |  | []  at | доступ к произвольному элементу |  |  |
| swap | обмен векторов |  |  | swap | обмен списков |
| clear() | очистить вектор |  |  | clear() | очистить вектор |
|  |  |  |  | splice | сцепка списков |

**Метод insert имеет следующие реализации:**

1. Вставляет элемент key в позицию, на которую указывает итератор pos, возвращает итератор на вставленный элемент

**iterator insert(iterator pos,** const **T&key);**

1. Вставляет n элементов key начиная с позиции, на которую указывает итератор pos

void **insert(iterator pos, size\_type n,** const **T&key);**

1. Вставляет элементы диапазона first..last, начиная с позиции, на которую указывает итератор pos

emplate **<**class **InputIter>**

void **insert(iterator pos, InputIter first, InputIter last);**

Пример

#include"stdafx.h"

#include<iostream>

#include<vector>

usingnamespace **std;**

int **\_tmain(**int **argc, \_TCHAR\* argv[])**

{vector <int>v1(5,0);

int **m[5]={1,2,3,4,5};**

v1.insert(v1.begin(),100); //+1

v1.insert(v1.begin()+1,2,200); //+2

v1.insert(v1.begin()+3,m,m+5); //+5

v1.insert(v1.end(),100); //+1

for(int **i=0;i<v1.size();i++)**

cout<<v1[i]<<" ";

**cout<<endl; system (**"pause");

return **0;**

}

Результат:

100 200 200 1 2 3 4 5 0 0 0 0 0 100

**Метод erase имеет следующие реализации:**

1. Удаляет элемент, на который указывает итератор pos

**iterator erase(iterator pos);**

1. Удаляет диапазон элементов

**iterator erase(iterator first,iterator last);**

**Пример. Удаление элемента, на который указывает итератор**

#include"stdafx.h"

#include<iostream>

#include<vector>

usingnamespace **std;**

int **\_tmain(**int **argc, \_TCHAR\* argv[])**

{

vector <int>v1;

int **n,a;**

cout<<"\nn(v1)="; cin>>n; // размер

for(int **i=0;i<n;i++)**

{cout<<"v1="; cin>>a;

v1.push\_back(a);

}

for(int **i=0;i<v1.size();i++)**

cout<<v1[i]<<'\t';

cout<<"\n--------------- erase --------------\n";

vector<int>::iterator iv=v1.erase(v1.begin());

for( int **i=0;i<v1.size();i++)**

cout<<v1[i]<<'\t';

**cout<<endl; system (**"pause");

return **0;**

}

**Пример. Удаление диапазона элементов (с 2 по 5)**

int **\_tmain(**int **argc, \_TCHAR\* argv[])**

{ . . .

cout<<"\n--------------- erase (2-5) --------------\n";

vector<int>::iterator iv=v1.erase(v1.begin()+1,v1.begin()+5);

. . .

return **0;**

}

**Пример. Два способа удаления последнего элемента**

int **\_tmain(**int **argc, \_TCHAR\* argv[])**

{ . . .

vector<int>::iterator iv=v1.erase(v1.end()-1); // удалить последний

v1.pop\_back(); // удалить последний

. . .

return **0;**

}

**Пример.** Работа с использова­нием методов swap(), empty (), back(), pop\_back()

Метод swap(v) служит для обмена элементов одного типа, но не обязательно одного размера: v1.swap(v2);

back() – получить доступ к последнему элементу

#include"stdafx.h"

#include<iostream>

#include<vector>

usingnamespace **std;**

int **\_tmain(**int **argc, \_TCHAR\* argv[])**

{double **arr[] = {1.1, 2.2 ,3.3, 4.4 };**

// Инициализация вектора массивом

int **n =** sizeof(arr)/sizeof(double);

vector<double> vl(arr, arr + n);

vector<double> v2; // пустой вектор

vl.swap(v2); // обменять содержимое vl и v2

while **(!v2.empty())**

{

cout << v2.back() <<' ';// вывести последний элемент

v2.pop\_back(); // и удалить его

}

**cout<<endl; system (**"pause");

return **0;**

}

К векторам применяются операции сравнения(==,!=,<,<=)

**Пример. Шаблонная функция print() для вывода содержимого контейнера**

В процессе работы над программами, использующими контейнеры, часто приходится выводить на экран их текущее содержимое. Приведем шаблон функции, решающей эту задачу для любого типа контейнера:

template<class **T>**

void **print(**char\*String,T&C)

{

T:: iterator p=C.begin();

cout<<String<<endl;

if(C.empty())

cout<<"\nEmpty!!!\n";

else

for(;p!=C.end();p++)cout<<\*p<<" ";

cout<<"\n";

}

**Пример. Работа со списком**

#include"stdafx.h"

#include<iostream>

#include<list>

usingnamespace **std;**

//функция для печати последовательного контейнера

template<class **T>**

void **print(**char\*String,T&C)

{

T:: iterator p=C.begin();

cout<<String<<endl;

if(C.empty())

cout<<"\nEmpty!!!\n";

else

for(;p!=C.end();p++)cout<<\*p<<" ";

cout<<"\n";

}

int **\_tmain(**int **argc, \_TCHAR\* argv[])**

{setlocale(0,"Russian");

list<int>l;

list<int>::iterator i;

int **x=1;**

cout<<"<<ввод данных, пока не 0>>"<<endl;

while(x!=0)//заполнение списка

{

cout<<"l:";cin>>x;

l.push\_back(x);

}

print("list:",l);

cout<<"<<добавить элемент>>"<<endl;

i=l.begin();

l.insert(++i,100); //добавление

print("list:",l);

cout<<"<<добавить элемент>>"<<endl;

i=l.end();

l.insert(--i,100);//добавление

print("list:",l);

cout<<"<<удалить последний элемент>>"<<endl;

l.pop\_back();//удаление

print("list:",l);

cout<<"<<удалить первый элемент>>"<<endl;

l.pop\_front();//удаление

print("list:",l);

**cout<<endl; system (**"pause");

return **0;**

}

Результат:

list:

1 2 3 4 5 0

list:

1 100 2 3 4 5 100 0

list:

100 2 3 4 5 100 0

list:

2 3 4 5 100

**Алгоритмы**

Алгоритм — это функция, которая производит некоторые действия над элемен­тами контейнера (контейнеров). Чтобы использовать обобщенные алгоритмы, нужно подключить к программе заголовочный файл <algorithm>.

Часто используемые алгоритмы STL

|  |  |
| --- | --- |
| Алгоритм | Назначение |
| accumulate | Вычисление суммы элементов в заданном диапазоне |
| сору | Копирование последовательности, начиная с первого элемента |
| count | Подсчет количества вхождений значения в последовательность |
| count\_if | Подсчет количества выполнений условия в последовательности |
| equal | Попарное равенство элементов двух последовательностей |
| fill | Замена всех элементов заданным значением |
| find | Нахождение первого вхождения значения в последовательность |
| find\_first\_of | Нахождение первого значения из одной последовательности в другой |
| find\_if | Нахождение первого соответствия условию в последовательности |
| for\_each | Вызов функции для каждого элемента последовательности |
| merge | Слияние отсортированных последовательностей |
| remove | Перемещение элементов с заданным значением |
| replace | Замена элементов с заданным значением |
| search | Нахождение первого вхождения в первую последовательность второй последовательности |
| sort | Сортировка |
| swap | Обмен двух элементов |
| transform | Выполнение заданной операции над каждым элементом последовательности |

В списках параметров всех алгоритмов первые два параметра задают диапазон обрабатываемых элементов в виде полуинтервала [first, last), где first — ите­ратор, указывающий на начало диапазона, a last — итератор, указывающий на выход за границы диапазона.

Например, если имеется массив

int arr[7] = (15. 2, 19, -3, 28, 6. 8};

то его можно отсортировать с помощью алгоритма sort:

sort(arr, arr + 7);

Пример: Программа считывает числа в вектор, сортирует по возрастанию и выводит на экран:

#include"stdafx.h"

#include<iostream>

#include<fstream>

#include<vector>

#include<algorithm>

usingnamespace **std;**

int **\_tmain(**int **argc, \_TCHAR\* argv[])**

{setlocale (LC\_ALL,"Russian");

int **n;**

cout<<"\nвведите количество: ";

cin>>n;

vector<int> v;

int **x;**

cout<<"\nвведите данные: "<<endl;

for(int **i=0;i<n;i++)**

{ cout<<"v: "; cin>>x;

**v.push\_back(x);**

}

sort(v.begin(), v.end());

vector<int>::const\_iterator i; //print(“Sorted vector:”,v);

for **(i = v.begin(); i != v.end(); ++i)**

cout<< \*i<< " ";

**cout<<endl; system (**"pause");

return **0;**

}

**Алгоритм sort имеет две сигнатуры:**

1. Обеспечивает сортировку элементов из диапазона [first, last), причем для упорядочения по умолчанию используется операция <, котораядолжна быть определена для типа Т. Таким образом, сортировка по умолча­нию — это сортировка по возрастанию значений.

tempiate<class **RandomAccessIt>**

void **sort(RandomAccessIt first, RandomAccessIt last);**

1. Позволяет задать произвольный критерий упоря­дочения. Для этого нужно передать через третий аргумент соответствующий предикат, то есть функцию или функциональный объект, возвращающие значе­ние типа bool.

tempiate<class **RandomAccessIt>**

void **sort(RandomAccessIt first, RandomAccessIt last, Compare comp);**

**Функциональным объектом** называется объект некоторого класса, для которого определена единственная операция вызова функции operator().

В стандартной библиотеке определены шаблоны функциональных объектов для операций сравнения, встроенных в язык C++, они возвращают значение типа bool.

**equal\_to ==**

**not\_equa1\_to !=**

**greater >**

**less <**

**greater\_equa1 >=**

**less\_equal <=**

При подстановке в качестве аргумента алгоритма требуется инстанцирование (создание экземпляров) этих шаблонов, например: **equal\_to<int>()**

Если заменить в преддущей программе вызов функции sort на следующий:

**sort(v.begin(). v.end(),greater<int>());**

то вектор будет отсортирован по убыванию значений его элементов.

Если сортировка выполняется для контейне­ра с объектами пользовательского класса, то программисту нужно са­мому позаботиться о наличии в классе предиката (функции, возвращающей значение типа bool), задающего сортировку по умолчанию, а также (при необходимости) определить функциональные классы, объекты которых позволяют изменять настройку алгоритма sort.

Пример: Использование функциональных объектов.

#include"stdafx.h"

#include<iostream>

#include<fstream>

#include<vector>

#include<algorithm>

usingnamespace **std;**

class **person{**

char **name[20];**

int **age;**

public:

person(){name[0]='\0';age=0;} //конструктор без параметров

person( char[20],int **);** //конструктор с параметрами

friend **istream&** operator>>(istream&,person&);//функция для ввода объекта из потока

friend **ostream&** operator<<(ostream&,const **person&);**//функця для вывода объекта в поток

char\* get\_name(){return **name;}**//получить имя

int **get\_age(){**return **age;}**

booloperator **== (**constint **&x)** const{if **(age==x)** returntrue;returnfalse;};

booloperator **< (**const **person & x)** const{if **(age < x.age)** returntrue; elsereturnfalse;};

booloperator **< (**constint **& x)** const{if **(age < x)** returntrue; elsereturnfalse;};

booloperator **> (**constint **& x)** const{if **(age > x)** returntrue; elsereturnfalse;};

}; //--------------------------- end person

// функциональные объекты

class **Comp1** //предикат для сортировки по возрастанию

{

public:

booloperator()(person&p1,person&p2)

{

if(strcmp(p1.get\_name(),p2.get\_name())<0)return **1;**

elsereturn **0;**

}

};

class **Comp2** //предикат для сортировки по убыванию

{public:

booloperator()(person&p1,person&p2)

{

if(strcmp(p1.get\_name(),p2.get\_name())>0)return **1;**

elsereturn **0;**

}

//предикат для поиска по заданному условию <

template **<**class **t>**class **pred1**

{

public:

booloperator()(t& x)

{

if(x<year) returntrue;

elsereturnfalse;

}

};

//предикат для поиска по заданному условию ==

template **<**class **t>**class **pred2**

{

public:

booloperator()(t& x)

{

if(x==year)returntrue;elsereturnfalse;

}

};

……..

vector<person>temp;

sort(temp.begin(),temp.end(),Comp2()); //сортировка по убыванию

//найти всеx людей, у которых возраст меньше 18

vector<person>::iterator tv1=temp.begin(); //итератор поставили на начало вектора

vector<person>::iterator tv2=temp.end(); //итератор поставили на конец вектора

pred1<person> pp; //поиск значения, соответствующего предикату pp

cout<<"Персоны моложе 18 лет:\n";

if((tv1=find\_if(temp.begin(),tv2,pp))!=temp.end())

{

cout<<\*tv1;

}

else

{

cout<<"\nНичего не найдено\n";

}

cout<<"\n";

//отсортировать контейнер по возрстанию

sort(temp.begin(),temp.end(),Comp1());

//подсчитать сколько элементов, удовлетворяющих условию содержит контейнер

tv1=temp.begin(); //итератор поставили на начало вектора

tv2=temp.end(); //итератор поставили на конец вектора

pred2<person> pp2;

cout<<"Персоны в возрасте 18 лет:\n";

int **count=count\_if(tv1,tv2,pp2);**

cout<<"Count="<<count<<endl;

cout<<"\n";

**Пример. Работа с объектами класса student и контейнером vector**

#include"stdafx.h"

#include<iostream>

#include<vector>// подключаем необходимую библиотеку

usingnamespace **std;**

struct **student {**

char **fio[80];**

short **age;**

// конструктор объекта

**student(**constchar\* fio, short **age) : age(age) {** // присваиваем полю age,значение параметра age

// в поле fio структуры копируем параметр fio, который указывается при создании объекта

**strncpy\_s(**this->fio, fio, sizeof(this->fio));

**}**

// ...

};

// перегружаем оператор для форматирования вывода объектов

inline **ostream&** operator **<<(ostream& out,** const **student& obj) {**

return **out << obj.fio <<** "\t: " **<< obj.age;**

}

int **\_tmain(**int **argc, \_TCHAR\* argv[])**

{setlocale(LC\_ALL, "rus");

**vector<student> group;**

// метод push\_back() добавляет новую запись в конец вектора

**group.push\_back(student(**"Иванов И.И.", 20));

**group.push\_back(student(**"Петров П.П.", 21));

**group.push\_back(student(**"Сидоров С.С.", 19));

for **(**int **i = 0; i < group.size(); i++)**

**cout << group[i] << endl;** // работает перегруженный оператор <<

cout<<endl; system ("pause");

return **0;**

}

**Пример. Работа с контейнером list и итераторами**

#include"stdafx.h"

#include<list>

#include<iostream>

usingnamespace **std;**

int **\_tmain(**int **argc, \_TCHAR\* argv[])**

{ int **s,c,m,e;**

**setlocale (LC\_ALL,**"Russian");

**list<**int> l;

**list<**int>::iterator it; // Объявляем итераторы it , it2

**list<**int>::iterator it2;

**cout <<** "Количество элементов=" **<< l.size()<<** "\n";

**cout <<** "-------------------------------"<<endl<< "Наполняем список"<<endl;

for **(**int **i=0;i<20;i++)**

**{**

**l.push\_back(i);**

**}**

for **(it=l.begin();it!=l.end();++it)** // печать списка

**cout << \*it <<** '\t' **;**

cout<<endl;

cout << "Количество элементов=" **<< l.size()<<** "\n";

cout << "-------------------------------"<<endl<< "Удаляем из списка 5 элементов"<<endl;

for **(**int **i=0;i<5;i++) {** // удаляем из списка 5 элементов

**l.pop\_back();**

**}**

for **(it=l.begin();it!=l.end();++it)** // печать списка

**cout << \*it <<** '\t' **;**

cout<<endl;

cout << "Количество элементов=" **<< l.size()<<** "\n";

cout << "-------------------------------"<<endl<< "вставляем в середину списка новый элемент 77"<<endl;

for **(it=l.begin();it!=l.end();++it) {** // проходим по списку с помощью итераторов

if **(\*it==6) {**

**it2=l.insert(it,77);** // вставляем в "середину списка" новый элемент

**}**

**}**

for **(it=l.begin();it!=l.end();++it)** // печать списка

**cout << \*it <<** '\t' **;**

cout<<endl;

cout << "Количество элементов=" **<< l.size()<<** "\n";

cout<< "-------------------------------"<<endl << "вставляем в середину еще 3 новых элемена 99"<<endl;

for **(it=l.begin();it!=l.end();++it) {** // проходим по списку с помощью итераторов

if **(\*it==6) {**

**it2=l.insert(it2,99);** // вставляем новый элемент

**it2=l.insert(it2,99);** // вставляем новый элемент

**it2=l.insert(it2,99);** // вставляем новый элемент

**}**

**}**

for **(it=l.begin();it!=l.end();++it)** // печать списка

**cout << \*it <<** '\t' **;**

cout<<endl;

cout << "Количество элементов=" **<< l.size()<<** "\n";

cout << "-------------------------------"<<endl<<"Удаляем элемент из списка"<<endl;

**it2=l.erase(it2);**

**s=l.size();**

**m=l.max\_size();**

**e=l.empty();**

**cout <<** "Количество элементов=" **<< s <<** "\n";

**cout <<** "максимальный размер=" **<< m <<** "\n";

**cout <<** "Empty=" **<< e <<** "\n";

for **(it=l.begin();it!=l.end();++it)** // печать списка

**cout << \*it <<** '\t' **;**

cout<<endl;

**cout <<** "\n";

**cout <<** "it2=" **<< \*it2 ;**

**cout<<endl; system (**"pause");

return **0;**

}

**Общие правила и требования**

1. Использовать элементы класса, реализованные в лабораторной работе №3 для компоновки программы в стиле STL, используя контейнеры, алгоритмы, итераторы.
2. Перегрузить операторы, указанные в задании в стиле STL
3. Использовать HDD для загрузки и сохранения данных
4. Смотрите общие требования к предыдущим лабораторным работам, а именно

* Общие требования к выполнению заданий
* Пример распечатки данных в табличном виде
* Пример диалогового интерфейса
* Обязательные элементы для всех вариантов:
  + загрузка данных с диска при запуске программы
  + распечатка данных в табличном виде
  + выход из программы с сохранением данных на диске

1. В задании указаны методы, которые создаются обязательно, но как правило, для работы требуются и другие методы, какие – решать вам.
2. Распечатывать как исходную информацию, так и результат.
3. Оценка лабораторной работы (всего 14 баллов) складывается из трех частей:

* Проект базы данных (5 баллов)
* Интерфейс пользователя 4 балла)
* Выполнение программы и защита работы (5 баллов)

|  |  |
| --- | --- |
| **Номер компьютера** | **Варианты заданий** |
| 3,13,  23 | **При выполнении задания используйте общие требования**  Из программы, реализованной в Л.Р.№3 скомпонуйте базу данных в стиле STL: Склад (товары). Для хранения и обработки данных использовать контейнер **list**  Класс  **storage** имеет следующие элементы:  **Элементы-данные класса (все данные pri)**:  Название товара : текстовая строка  Цена : дробное число  Количество : целое число  Дата прихода : вложенный объект класса **date**  **Элементы-функции класса**  1. Создать обязательные функции, указанные в общих требованиях  2. Создать функции для данного варианта:   * добавить новые элементы в начало и конец списка * удалить элементы из заданной позиции списка * поиск товара по полю **date** * сортировка по убыванию количества   **Перегрузить следующие операторы:**  **> (сравнение)** объекта типа **storage** собъектом типа **date**  **>>**  ввод объекта класса **storage** из входного потока **cin** (с клавиатуры)  **Тестовая программа (функция main):**   1. При запуске программы данные ввести с клавиатуры (первый запуск программы) или загрузить с диска (все последующие запуски) 2. При реализации действий использовать перегруженные операторы 3. Выполнить все реализованные действия (добавление, удаление, поиск, сортировка) 4. При выходе из программы запомнить измененную базу на диске | |