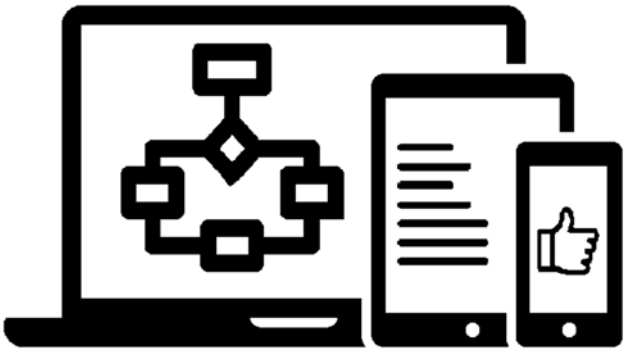
МЕЖГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Программное обеспечение информационных технологий»

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ И ПРОЕКТИРОВАНИЕ

Методические рекомендации к лабораторным работам для студентов специальности 1-53 01 02 «Автоматизированные системы обработки информации» дневной и заочной форм обучения

Часть 2



Могилев 2021

УДК 621.01
ББК 36.4
О87

Рекомендовано к изданию
учебно-методическим отделом
Белорусско-Российского университета

Одобрено кафедрой ПОИТ «1» сентября 2021 г., протокол № 1

Составители: ст. преподаватель Ю. В. Вайнилович;
канд. техн. наук, доц. Н. Н. Горбатенко;
ст. преподаватель О. В. Сергиенко
Рецензент канд. техн. наук, доц. В. М. Ковальчук

Методические рекомендации разработаны на основе рабочей программы по дисциплине «Объектно-ориентированное программирование и проектирование» для студентов направлений подготовки 1-53 01 02 «Автоматизированные системы обработки информации» и предназначены для использования при проведении лабораторных работ.

Учебно-методическое издание

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ И ПРОЕКТИРОВАНИЕ

Часть 2

Ответственный за выпуск В. В. Кутузов

Корректор Т. А. Рыжикова

Компьютерная верстка Е. В. Ковалевская

Подписано в печать 30.11.2021 . Формат 60×84/16. Бумага офсетная. Гарнитура Таймс. Печать трафаретная. Усл. печ. л. 2,79 . Уч.-изд. л. 3,0 . Тираж 26 экз. Заказ № 856.

Издатель и полиграфическое исполнение:
Межгосударственное образовательное учреждение высшего образования
«Белорусско-Российский университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/156 от 07.03.2019.
Пр-т Мира, 43, 212022, г. Могилев.

Содержание

1 Лабораторная работа № 11. Последовательные контейнеры библиотеки STL

Цель работы: освоение приемов программирования с использованием последовательных контейнеров библиотеки STL.

1.2 Теоретические сведения

1.2.1 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.

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 (*).

1.2.2 Контейнеры.

Контейнеры STL можно разделить на два типа: последовательные и ассоциативные (рисунок 1.1).

Рисунок 1.1 – Контейнерные классы

Последовательные контейнеры обеспечивают хранение конечного количества однотипных объектов в виде непрерывной последовательности. К базовым последовательным контейнерам относятся:

– векторы (vector);

– списки (list);

– двусторонние очереди (deque).

Есть еще специализированные контейнеры (или адаптеры контейнеров), реализованные на основе базовых:

– стеки (stack);

– очереди (queue);

– очереди с приоритетами (priority_queue).

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;
```

Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Эти контейнеры построены на основе сбалансированных деревьев. Существует пять типов ассоциативных контейнеров:

1) словари (map);

2) словари с дубликатами (multimap);

3) множества (set);

4) множества с дубликатами (multiset);

5) битовые множества (bitset).

1.2.3 Итераторы.

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

```
template <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 p – итератор, то *p – 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 (++p) или постфиксного (p++) 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;

last – значение итератора, указывающее на воображаемый элемент, который следует за последним элементом контейнера (рисунок 1.2).

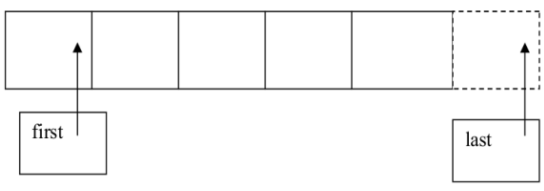


Рисунок 1.2 – Итераторы на первый и последний элементы контейнера

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;

– итераторы произвольного доступа.

1.2.4 Общие свойства контейнеров.

В таблице 1.1 представлены типы, определенные в STL.

Таблица 1.1 – Унифицированные типы, определенные в STL

Поле	Описание
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)
key_type	Тип ключа (для ассоциативных контейнеров)
key compare	Тип критерия сравнения (для ассоциативных контейнеров)

В таблице 1.2 представлены некоторые общие для всех контейнеров операции.

Таблица 1.2 – 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 empty() 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)

1.2.5 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>
using namespace 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 произвольного доступа к своим элементам: например, для выборки n-го элемента нужно последовательно выбрать предыдущие n-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);

Метод insert вставляет элемент в произвольное место. Он имеет следующие реализации:

– iterator insert(iterator pos, const T&key).

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

– void insert(iterator pos, size_type n, const T&key).

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

– template <class InputIter>

void insert(iterator pos, InputIter first, InputIter last).

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

Пример использования метода insert():

```
void main()
{
    /*создать вектор из 5 элементов, проинициализировать элементы
    нулями*/
    vector <int>v1(5,0);
    int m[5]={1,2,3,4,5};//массив из 5 элементов
    //вставить элемент со значением 100 в начало вектора
    v1.insert(v1.begin(),100);
    /*вставить два элемента со значением 200 после первого элемента вектора*/
    v1.insert(v1.begin()+1,2,200);
    //вставить элементы из массива m после третьего элемента
    v1.insert(v1.begin()+3,m,m+5);
    //вставить элемент 100 в конец вектора
    v1.insert(v1.end(),100);//
    //вывести вектор на печать
    for(int i=0;i<v1.size();i++)
    cout<<v1[i]<<" ";
}
```

Результат работы программы будет следующим:

100 200 200 1 2 3 4 5 0 0 0 0 0 100

Метод erase контейнера имеет следующие реализации:

– iterator erase(iterator pos); удаляет элемент, на который указывает итератор pos.

– iterator erase(iterator first,iterator last); удаляет диапазон элементов.

Пример использования метода erase():

```
void main()
{
   vector <int>v1;//создать пустой вектор
   int m[5]={1,2,3,4,5};
   int n,a;
   cout<<"\nn?";
   cin>>n;
   for(int i=0;i<n;i++)
   {
      cout<<"?";
      cin>>a;
      //добавить в конец вектора элемент со значением a
      v1.push_back(a);
   }
   //вывод вектора
   for( i=0;i<v1.size();i++)
      cout<<v1[i]<<" ";
   cout<<"\n";
   /*удалить элемент из начала вектора и итератор поставить на
   начало вектора*/
   vector<int>::iterator iv=v1.erase(v1.begin());
   cout<<(*iv)<<"\n";//вывод первого элемента
   //вывод вектора
   for( i=0;i<v1.size();i++)
      cout<<v1[i]<<" ";
}
```

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:

```
//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";
}
```

1.2.6 Адаптеры контейнеров.

Специализированные последовательные контейнеры – стек, очередь и очередь с приоритетами – не являются самостоятельными контейнерными классами, а реализованы на основе рассмотренных выше классов, поэтому они называются адаптерами контейнеров.

По умолчанию для стека прототипом является класс deque.

Объявление stack<int> s создает стек на базе двусторонней очереди (по умолчанию). Если по каким-то причинам нас это не устраивает, и мы хотим создать стек на базе списка, то объявление будет выглядеть следующим образом:

stack<int, list<int> > s;

Смысл такой реализации заключается в том, что специализированный класс просто переопределяет интерфейс класса-прототипа, ограничивая его только теми методами, которые нужны новому классу. Стек не позволяет выполнить произвольный доступ к своим элементам, а также не дает возможности пошагового перемещения, т. е. итераторы в стеке не поддерживаются.

Методы класса stack:

– push () – добавление в конец;

– pop () – удаление из конца;

– top () – получение текущего элемента стека;

– empty() – проверка пустой стек или нет;

– size () – получение размера стека.

Шаблонный класс queue (заголовочный файл <queue>) является адаптером, который может быть реализован на основе двусторонней очереди (реализация по умолчанию) или списка. Класс vector в качестве класса-прототипа не подходит, поскольку в нем нет выборки из начала контейнера. Очередь использует для проталкивания данных один конец, а для выталкивания – другой.

Методы класса queue:

– push () – добавление в конец очереди;

– pop () – удаление из начала очереди;

– front () – получение первого элемента очереди;

– back() – получение последнего элемента очереди;

– empty () – проверка пустая очередь или нет;

– size() – получение размера очереди.

Шаблонный класс priority_queue (заголовочный файл <queue>) поддерживает такие же операции, как и класс queue, но реализация класса возможна либо на основе вектора (реализация по умолчанию), либо на основе списка. Очередь с приоритетами отличается от обычной очереди тем, что для извлечения выбирается максимальный элемент из хранимых в контейнере. Поэтому после каждого изменения состояния очереди максимальный элемент из оставшихся сдвигается в начало контейнера.

Методы класса priority_queue:

– push () – добавление в конец очереди;

– pop () – удаление из начала очереди;

– front () – получение первого элемента очереди;

– back() – получение последнего элемента очереди;
– empty () – проверка пустая очередь или нет;
– size() – получение размера очереди.
Рассмотрим пример использования очереди с приоритетами:

```
#include <queue>
void main()
{
   priority_queue <int> P;//очередь с приоритетами
   P.push(17); //добавить элементы
   P.push(5);
   P.push(400);
   P.push(2500);
   P.push(1);
   //пока очередь не пустая
   while (!P.empty())
   {
      cout<<P.top()<<' ' ;//вывести первый элемент
      P.pop();//удалить элемент из начала
   }
}
```

Результат работы программы:
2500 400 17 5 1

1.2 Индивидуальные задания

Задача 1

1 Создать последовательный контейнер.
2 Заполнить его элементами стандартного типа (тип указан в варианте).
3 Добавить элементы в соответствии с заданием.
4 Удалить элементы в соответствии с заданием.
5 Выполнить задание варианта для полученного контейнера.
6 Выполнение всех заданий оформить в виде глобальных функций.

Задача 2

1 Создать последовательный контейнер.
2 Заполнить его элементами пользовательского типа (тип указан в варианте). Для пользовательского типа перегрузить необходимые операции.
3 Добавить элементы в соответствии с заданием.
4 Удалить элементы в соответствии с заданием.
5 Выполнить задание варианта для полученного контейнера.
6 Выполнение всех заданий оформить в виде глобальных функций.

Задача 3

1 Создать параметризированный класс, используя в качестве контейнера последовательный контейнер.

2 Заполнить контейнер элементами.

3 Добавить элементы в соответствии с заданием.

4 Удалить элементы в соответствии с заданием.

5 Выполнить задание варианта для полученного контейнера.

6 Выполнение всех заданий оформить в виде методов параметризированного класса.

Задача 4

1 Создать адаптер контейнера.

2 Заполнить его элементами пользовательского типа (тип указан в варианте). Для пользовательского типа перегрузить необходимые операции.

3 Добавить элементы в соответствии с заданием.

4 Удалить элементы в соответствии с заданием.

5 Выполнить задание варианта для полученного контейнера.

6 Выполнение всех заданий оформить в виде глобальных функций.

Задача 5

1 Создать параметризированный класс, используя в качестве контейнера адаптер контейнера.

2 Заполнить его элементами.

3 Добавить элементы в соответствии с заданием.

4 Удалить элементы в соответствии с заданием.

5 Выполнить задание варианта для полученного контейнера.

6 Выполнение всех заданий оформить в виде методов параметризированного класса.

Задачи и задания по вариантам представлены в таблице 1.3.

Таблица 1.3 – Варианты заданий

<table>
<tr><th>Номер варианта</th><th colspan="3">Задание</th></tr>
<tr><td rowspan="3">1</td><td colspan="3">Задача 1
1 Контейнер – вектор.
2 Тип элементов – double.
Задача 2
Тип элементов – Time.
Задача 3
Параметризированный класс – вектор.
Задача 4
Адаптер контейнера – стек.
Задача 5
Параметризированный класс – вектор.
Адаптер контейнера – стек</td></tr>
<tr><td>Задание 1</td><td>Задание 2</td><td>Задание 3</td></tr>
<tr><td>Найти максимальный элемент и добавить его в начало контейнера</td><td>Найти минимальный элемент и удалить его из контейнера</td><td>К каждому элементу добавить среднее арифметическое контейнера</td></tr>
</table>

Продолжение таблицы 1.3

<table>
<tr><th>Номер варианта</th><th colspan="3">Задание</th></tr>
<tr><td rowspan="3">2</td><td colspan="3">Задача 1
1 Контейнер – список.
2 Тип элементов – float.
Задача 2
Тип элементов – Time.
Задача 3
Параметризированный класс – вектор.
Задача 4
Адаптер контейнера – очередь.
Задача 5
Параметризированный класс – вектор.
Адаптер контейнера – очередь</td></tr>
<tr><td>Задание 1</td><td>Задание 2</td><td>Задание 3</td></tr>
<tr><td>Найти минимальный элемент и добавить его в конец контейнера</td><td>Найти элемент с заданным ключом и удалить его из контейнера</td><td>К каждому элементу добавить сумму минимального и максимального элементов контейнера</td></tr>
<tr><td rowspan="3">3</td><td colspan="3">Задача 1
1 Контейнер – двунаправленная очередь.
2 Тип элементов – double.
Задача 2
Тип элементов – Time.
Задача 3
Параметризированный класс – вектор.
Задача 4
Адаптер контейнера – очередь с приоритетами.
Задача 5
Параметризированный класс – вектор.
Адаптер контейнера – очередь с приоритетами</td></tr>
<tr><td>Задание 1</td><td>Задание 2</td><td>Задание 3</td></tr>
<tr><td>Найти элемент с заданным ключом и добавить его на заданную позицию контейнера</td><td>Найти элемент с заданным ключом и удалить его из контейнера</td><td>Найти разницу между максимальным и минимальным элементами контейнера и вычесть ее из каждого элемента контейнера</td></tr>
<tr><td rowspan="3">4</td><td colspan="3">Задача 1
1 Контейнер – двунаправленная очередь.
2 Тип элементов – int.
Задача 2
Тип элементов – Time.
Задача 3
Параметризированный класс – вектор.
Задача 4
Адаптер контейнера – очередь.
Задача 5
Параметризированный класс – вектор.
Адаптер контейнера – очередь</td></tr>
<tr><td>Задание 1</td><td>Задание 2</td><td>Задание 3</td></tr>
<tr><td>Найти максимальный элемент и добавить его в конец контейнера</td><td>Найти элемент с заданным ключом и удалить его из контейнера</td><td>К каждому элементу добавить среднее арифметическое элементов контейнера</td></tr>
</table>

Продолжение таблицы 1.3

<table>
<tr><th>Номер варианта</th><th colspan="3">Задание</th></tr>
<tr><td rowspan="3">5</td><td colspan="3">Задача 1
1 Контейнер – список.
2 Тип элементов – float.
Задача 2
Тип элементов – Time.
Задача 3
Параметризированный класс – вектор.
Задача 4
Адаптер контейнера – вектор.
Задача 5
Параметризированный класс – вектор.
Адаптер контейнера – стек</td></tr>
<tr><td>Задание 1</td><td>Задание 2</td><td>Задание 3</td></tr>
<tr><td>Найти минимальный элемент и добавить его на заданную позицию контейнера</td><td>Найти элементы большие среднего арифметического и удалить их из контейнера</td><td>Каждый элемент домножить на максимальный элемент контейнера</td></tr>
<tr><td rowspan="3">6</td><td colspan="3">Задача 1
1 Контейнер – вектор.
2 Тип элементов – double.
Задача 2
Тип элементов – Money.
Задача 3
Параметризированный класс – множество.
Задача 4
Адаптер контейнера – стек.
Задача 5
Параметризированный класс – множество.
Адаптер контейнера – стек</td></tr>
<tr><td>Задание 1</td><td>Задание 2</td><td>Задание 3</td></tr>
<tr><td>Найти максимальный элемент и добавить его в начало контейнера</td><td>Найти минимальный элемент и удалить его из контейнера</td><td>К каждому элементу добавить среднее арифметическое контейнера</td></tr>
<tr><td rowspan="3">7</td><td colspan="3">Задача 1
1 Контейнер – вектор.
2 Тип элементов – float.
Задача 2
Тип элементов – Money.
Задача 3
Параметризированный класс – вектор.
Задача 4
Адаптер контейнера – очередь.
Задача 5
Параметризированный класс – вектор.
Адаптер контейнера – очередь</td></tr>
<tr><td>Задание 1</td><td>Задание 2</td><td>Задание 3</td></tr>
<tr><td>Найти минимальный элемент и добавить его в конец контейнера</td><td>Найти элемент с заданным ключом и удалить его из контейнера</td><td>К каждому элементу добавить сумму минимального и максимального элементов контейнера</td></tr>
</table>

Продолжение таблицы 1.3

<table>
<tr><th>Номер варианта</th><th colspan="3">Задание</th></tr>
<tr><td rowspan="3">8</td><td colspan="3">Задача 1
1 Контейнер – список.
2 Тип элементов – double.
Задача 2
Тип элементов – Money.
Задача 3
Параметризированный класс – вектор.
Задача 4
Адаптер контейнера – очередь с приоритетами.
Задача 5
Параметризированный класс – вектор.
Адаптер контейнера – очередь с приоритетами</td></tr>
<tr><td>Задание 1</td><td>Задание 2</td><td>Задание 3</td></tr>
<tr><td>Найти элемент с заданным ключом и добавить его на заданную позицию контейнера</td><td>Найти элемент с заданным ключом и удалить его из контейнера</td><td>Найти разницу между максимальным и минимальным элементами контейнера и вычесть ее из каждого элемента контейнера</td></tr>
<tr><td rowspan="3">9</td><td colspan="3">Задача 1
1 Контейнер – двунаправленная очередь.
2 Тип элементов – int.
Задача 2
Тип элементов – Money.
Задача 3
Параметризированный класс – вектор.
Задача 4
Адаптер контейнера – стек.
Задача 5
Параметризированный класс – вектор.
Адаптер контейнера – стек</td></tr>
<tr><td>Задание 1</td><td>Задание 2</td><td>Задание 3</td></tr>
<tr><td>Найти максимальный элемент и добавить его в конец контейнера</td><td>Найти элемент с заданным ключом и удалить его из контейнера</td><td>К каждому элементу добавить среднее арифметическое элементов контейнера</td></tr>
<tr><td rowspan="3">10</td><td colspan="3">Задача 1
1 Контейнер – вектор.
2 Тип элементов – float.
Задача 2
Тип элементов – Money.
Задача 3
Параметризированный класс – вектор.
Задача 4
Адаптер контейнера – очередь с приоритетами.
Задача 5
Параметризированный класс – вектор.
Адаптер контейнера – очередь с приоритетами</td></tr>
<tr><td>Задание 1</td><td>Задание 2</td><td>Задание 3</td></tr>
<tr><td>Найти минимальный элемент и добавить его на заданную позицию контейнера</td><td>Найти элементы большие среднего арифметического и удалить их из контейнера</td><td>Каждый элемент домножить на максимальный элемент контейнера</td></tr>
</table>

Продолжение таблицы 1.3

<table>
<tr><th>Номер варианта</th><th colspan="3">Задание</th></tr>
<tr><td rowspan="3">11</td><td colspan="3">Задача 1
1 Контейнер – вектор.
2 Тип элементов – float.
Задача 2
Тип элементов – Money.
Задача 3
Параметризированный класс – список.
Задача 4
Адаптер контейнера – очередь.
Задача 5
Параметризированный класс – список.
Адаптер контейнера – очередь</td></tr>
<tr><td>Задание 1</td><td>Задание 2</td><td>Задание 3</td></tr>
<tr><td>Найти среднее арифметическое и добавить его в начало контейнера</td><td>Найти элемент с заданным ключом и удалить их из контейнера</td><td>Из каждого элемента вычесть минимальный элемент контейнера</td></tr>
<tr><td rowspan="3">12</td><td colspan="3">Задача 1
1 Контейнер – список.
2 Тип элементов – int.
Задача 2
Тип элементов – Pair.
Задача 3
Параметризированный класс – список.
Задача 4
Адаптер контейнера – очередь с приоритетами.
Задача 5
Параметризированный класс – список.
Адаптер контейнера – очередь с приоритетами</td></tr>
<tr><td>Задание 1</td><td>Задание 2</td><td>Задание 3</td></tr>
<tr><td>Найти среднее арифметическое и добавить его на заданную позицию контейнера.</td><td>Найти элементы ключами из заданного диапазона и удалить их из контейнера.</td><td>Из каждого элемента вычесть среднее арифметическое контейнера.</td></tr>
<tr><td rowspan="3">13</td><td colspan="3">Задача 1
1 Контейнер – двунаправленная очередь.
2 Тип элементов – double.
Задача 2
Тип элементов – Pair.
Задача 3
Параметризированный класс – список.
Задача 4
Адаптер контейнера – стек.
Задача 5
Параметризированный класс – список.
Адаптер контейнера – стек</td></tr>
<tr><td>Задание 1</td><td>Задание 2</td><td>Задание 3</td></tr>
<tr><td>Найти максимальный элемент и добавить его в конец контейнера</td><td>Найти элементы ключами из заданного диапазона и удалить их из контейнера</td><td>К каждому элементу добавить среднее арифметическое контейнера.</td></tr>
</table>

Окончание таблицы 1.3

<table>
<tr><td>Номер варианта</td><td colspan="3">Задание</td></tr>
<tr><td>14</td><td colspan="3">Задача 1
1 Контейнер – вектор.
2 Тип элементов – float.
Задача 2
Тип элементов – Pair.
Задача 3
Параметризированный класс – список.
Задача 4
Адаптер контейнера – очередь.
Задача 5
Параметризированный класс – список.
Адаптер контейнера – очередь</td></tr>
<tr><td></td><td>Задание 1</td><td>Задание 2</td><td>Задание 3</td></tr>
<tr><td></td><td>Найти минимальный элемент и добавить его на заданную позицию контейнера</td><td>Найти меньше среднего арифметического и удалить их из контейнера</td><td>Каждый элемент разделить на максимальный элемент контейнера.</td></tr>
<tr><td>15</td><td colspan="3">Задача 1
1 Контейнер – список.
2 Тип элементов – double.
Задача 2
Тип элементов – Pair.
Задача 3
Параметризированный класс – список.
Задача 4
Адаптер контейнера – очередь с приоритетами.
Задача 5
Параметризированный класс – список.
Адаптер контейнера – очередь с приоритетами</td></tr>
<tr><td></td><td>Задание 1</td><td>Задание 2</td><td>Задание 3</td></tr>
<tr><td></td><td>Найти среднее арифметическое и добавить его в конец контейнера</td><td>Найти элементы ключами из заданного диапазона и удалить их из контейнера</td><td>К каждому элементу добавить сумму минимального и максимального элементов контейнера.</td></tr>
</table>

Контрольные вопросы

1 Из каких частей состоит библиотека STL?

2 Какие типы контейнеров существуют в STL?

3 Что нужно сделать для использования контейнера STL в своей программе?

4 Что представляет собой итератор?

5 Какие операции можно выполнять над итераторами?

6 Каким образом можно организовать цикл для перебора контейнера с использованием итератора?

7 Какие типы итераторов существуют?

8 Перечислить операции и методы, общие для всех контейнеров.

9 Какие операции являются эффективными для контейнера vector? Почему?

10 Какие операции являются эффективными для контейнера list? Почему?

11 Какие операции являются эффективными для контейнера deque? Почему?

12 Перечислить методы, которые поддерживает последовательный контейнер vector.

13 Перечислить методы, которые поддерживает последовательный контейнер list.

14 Перечислить методы, которые поддерживает последовательный контейнер deque.

2 Лабораторная работа № 12. Абстрактные типы данных. Контейнеры. Итераторы

Цель работы: создание консольного приложения, состоящего из нескольких файлов в системе программирования Visual Studio; реализация класса-контейнера.

2.1 Теоретические сведения

2.1.1 Абстрактные типы данных. Контейнеры.

Абстактные типы данных (АТД) – тип данных, определяемый только через операции, которые могут выполняться над соответствующими объектами безотносительно к способу представления этих объектов.

АТД включает в себя абстракцию как через параметризацию, так и через спецификацию.

Абстракция через параметризацию может быть осуществлена так же, как и для процедур (функций) с использованием параметров там, где это имеет смысл.

Абстракция через спецификацию достигается за счет того, что операции представляются как часть типа.

Для реализации АТД необходимо, во-первых, выбрать представление памяти для объектов и, во-вторых, реализовать операции в терминах выбранного представления.

Примером абстрактного типа данных является класс в языке C++.

В реальных задачах требуется обрабатывать группы данных большого объема, поэтому в любом языке программирования существуют средства, позволяющие объединять данные в группы. В первую очередь это массивы. В C++ массивы – это очень простые конструкции. В ООП этого недостаточно для работы с группами однородных данных. Поэтому была выработана более общая концепция объединения однородных данных в группу – контейнер.

Контейнер – набор однотипных элементов. Встроенные массивы в C++ – частный случай контейнера.

Контейнер – это объект. Имя контейнера – это имя переменной. Контейнер, так же как и другие объекты, обладает временем жизни. Время жизни контейнера в общем случае не зависит от времени жизни его элементов. Элементами контейнера могут любые объекты, в том числе и другие контейнеры.

Контейнеры могут быть фиксированного и переменного размера. В контейнере фиксированного размера число элементов постоянное, оно обычно задается при создании контейнера. Примером такого контейнера является массив. Для контейнера переменного размера количество элементов при объявлении обычно не задается. Элементы в таком контейнере добавляются и удаляются во время работы программы. Примером является список.

Если элементы контейнера не упорядочены, то добавление и удаление элементов обычно выполняется в начале и в конце контейнера. Способ вставки и удаления определяет вид контейнера. Если вставка и удаление осуществляются на одном конце, то такой контейнер называется стеком (`Last In First Out` – последним пришел, первым ушел). Если элементы добавляются на одном конце контейнера, а удаляются на другом, то такой контейнер называется очередью (`First In First Out` – первым пришел, последним ушел). Можно выполнять вставку и удаление на обоих концах контейнера, тогда такой контейнер будет называться двусторонней очередью (`deque – double ended queue`).

Если контейнер каким-то образом упорядочен, то операция вставки работает в соответствии с порядком элементов в контейнере. Операция удаления может выполняться по-разному: в начале, в конце или удаление заданного элемента.

2.1.2 Операции контейнера.

Среди всех операций контейнера можно выделить несколько типовых групп:

– операции доступа к элементам, которые обеспечивают и операцию замены значений элементов;

– операции добавления и удаления элементов или групп элементов;

– операции поиска элементов и групп элементов;

– операции объединения контейнеров;

– специальные операции, которые зависят от вида контейнера.

Доступ к элементам

Доступ к элементам контейнера бывает последовательный, прямой и ассоциативный.

Прямой доступ – это доступ по индексу. Например, `a[10]` – требуется найти элемент контейнера с номером 10. В C++ нумерацию элементов контейнера принято начинать с нуля.

Ассоциативный доступ также выполняется по индексу, но индексом будет являться не номер элемента, а его содержимое. Пусть имеется контейнер словарь, в котором хранится информация состоящая как минимум из двух полей: слово и его перевод.

Индексом может служить слово, например, `a["word"]`. С этим словом будет связано слово-перевод. Поле, с содержимым которого ассоциируется элемент контейнера, называется ключом или полем доступа. Элемент, с которым ассоциируется ключ, называется значением. Контейнер, который представляет ассоциативный доступ, состоит из пар «ключ – значение». Ассоциативный контейнер каким-то образом должен быть упорядочен по ключу. Например, в словаре упорядочение выполняется по алфавиту.

При последовательном доступе осуществляется перемещение от элемента к элементу контейнера. Набор операций последовательного доступа включает следующие:

– перейти к первому элементу;
– перейти к последнему элементу;
– перейти к следующему элементу;
– перейти к предыдущему элементу;
– перейти на n элементов вперед;
– перейти на n элементов назад;
– получить текущий элемент.

Если контейнером является массив, то мы можем осуществлять последовательный доступ к его элементам с помощью указателя. И все перечисленные операции можно реализовать, используя указатели. В более общем случае, объект, который перебирает элементы контейнера, называется итератором. Итератор – это объект, который обеспечивает последовательный доступ к элементам контейнера. Итератор может быть реализован как часть класса-контейнера в виде набора методов:

– v.first() – перейти к первому элементу;
– v.last() – перейти к последнему элементу;
– v.next() – перейти к следующему элементу;
– v.prev() – перейти к предыдущему элементу;
– v.skip(n) – перейти на n элементов вперед;
– v.skip(-n) – перейти на n элементов назад;
– v.current() – получить текущий элемент.

Итератор можно реализовать как класс, представляющий такой же набор операций.

В C++ итератор реализуется как класс, который имеет такой же интерфейс, как и указатель для совместимости с массивами.

Если объект-итератор имеет имя `iterv`, то операции могут быть представлены следующим образом:

– iterv=v.first() – перейти к первому элементу;
– iterv=v.last() – перейти к последнему элементу;
– iterv++ – перейти к следующему элементу;

– iterv-- – перейти к предыдущему элементу;
– iterv+=n – перейти на n элементов вперед;
– v.skip-=n – перейти на n элементов назад;
– *iterv – получить текущий элемент.

При наличии последовательного доступа с помощью итератора обычными операциями являются:

– вставка элемента перед текущим элементом (после текущего);
– изменение значения текущего элемента;
– удаление текущего элемента;
– поиск элемента с заданным значением (возвращает итератор на текущий элемент).

Операции объединения контейнеров

Наиболее часто используется операция объединения двух контейнеров с получением нового контейнера. Она может быть реализована в разных вариантах:

– простое сцепление двух контейнеров: в новый контейнер попадают сначала элементы первого контейнера, потом второго, операция не коммутативна;
– объединение упорядоченных контейнеров, новый контейнер тоже будет упорядочен, операция коммутативна;
– объединение контейнеров как объединение множеств, в новый контейнер попадают только те элементы, которые есть хотя бы в одном контейнере, операция коммутативна;
– объединение контейнеров как пересечение множеств, в новый контейнер попадают только те элементы, которые есть в обоих контейнерах, операция коммутативна;
– для контейнеров-множеств может быть еще реализована операция вычитания, в контейнер попадают только те элементы первого контейнера, которых нет во втором, операция не коммутативна;
– извлечение части элементов из контейнера и создание нового контейнера. Эта операция может быть выполнена с помощью конструктора, а часть контейнера задается двумя итераторами.

2.2 Постановка задачи

Для указанного варианта индивидуального задания выполнить следующее.

1 Определить класс-контейнер.

2 Реализовать конструкторы, деструктор, операции ввода-вывода, операцию присваивания.

3 Перегрузить операции, указанные в варианте.

4 Реализовать класс-итератор. Реализовать с его помощью операции последовательного доступа.

5 Написать тестирующую программу, иллюстрирующую выполнение операций.

2.3 Пример выполнения работы

Условие задачи. Класс-контейнер вектор с элементами типа int. Реализовать операции: [] – доступ по индексу; () – определение размера вектора; + число – добавляет константу ко всем элементам вектора; ++ – переход к следующему элементу (с помощью класса-итератора).

1 Создать пустой проект.
2 Добавить в него класс Vector.
3 В файл Vector.h добавить описание класса Вектор:

```
class Vector
{
public:
//конструктор с параметрами: выделяет память под s элементов и
//заполняет их значением k
        Vector(int s,int k=0);
//конструктор с параметрами
        Vector(const Vector&a);
//деструктор
        ~Vector();
//оператор присваивания
        Vector&operator=(const Vector&a);
//операция доступа по индексу
        int&operator[](int index);
//операция для добавления константы
        Vector operator+(const int k);
//операция, возвращающая длину вектора
        int operator()();
//перегруженные операции ввода-вывода
      friend ostream& operator<<(ostream& out, const Vector&a); friend
      istream& operator>>(istream& in, Vector&a); private:
      int size;//размер вектора
      int*data;//указатель на динамический массив значений вектора
};
```

4 В файл Vector.cpp добавить определение методов класса Вектор:

```
//конструктор с параметрами
Vector::Vector(int s,int k)
{
        size=s;
```

```
        data=new int[size];
        for(int i=0;i<size;i++)
                data[i]=k;
}

//конструктор копирования
Vector::Vector(const Vector&a)
{
        size=a.size;
        data=new int[size];
        for(int i=0;i<size;i++)
                data[i]=a.data[i];
}

//деструктор
Vector::~Vector()
{
        delete[]data; data=0;
}

//операция присваивания
Vector&Vector::operator=(const Vector&a)
{
        if(this==&a)return *this;
        size=a.size;
        if (data!=0) delete[]data;
        data=new int[size];
        for(int i=0;i<size;i++)
                data[i]=a.data[i];
        return *this;
}

//операция доступа по индексу
int&Vector::operator[](int index)
{
       if (index<size) return data[index];
       else cout<<"\nError! Index>size";
}

//операция для добавления константы
Vector Vector::operator+(const int k)//+k
{
```

```
        Vector temp(size);
        for (int i=0;i<size;++i)
                temp.data[i]+=data[i]+k;
        return temp;
}

//операция для получения длины вектора
int Vector::operator ()()
{
        return len();
}

//операции для ввода-выода
ostream&operator<<(ostream&out,const Vector&a)
{
        for(int i=0;i<a.len();++i)
                out<<a.data[i]<<" ";
        return out;
}

istream&operator>>(istream&in,Vector&a)
{
for(int i=0;i<a.len();++i) in>>a.data[i];
return in;
}
```

5 В функции main() выполнить тестирование класса вектор:

```
void main()
{
        Vector a(5);//создали вектор из 5 элементов,
                //заполненный нулями
        cout<<a<<"\n";//вывели значения элементов вектора
        cin>>a; //ввели с клавиатуры значения элементов вектора
        cout<<a<<"\n";//вывели значения элементов вектора
        a[2]=100;//используя операцию [], присвоили новое значение
              //элементу
        cout<<a<<"\n";//вывели значения элементов вектора
        Vector b(10);//создали вектор b из 10 элементов,
                 //заполненный нулями
```

```
        cout<<b<<"\n";//вывели значения элементов вектора b=a;//присвоили
        вектору b значения вектора a cout<<b<<"\n";//вывели значения эле-
        ментов вектора
        Vector c(10); //создали вектор с из 10 элементов,
                //заполненный нулями
        c=b+100;//увеличили значения вектора b на 100 и
                //присвоили вектору c
        cout<<c<<"\n";//вывели значения элементов вектора c
        cout<<"\nthe length of a="<<a()<<endl;//вывели длину
                                                //вектора a
}
```

6 В файл Vector.h добавить описание класса итератор (перед классом вектор):

```
class Iterator
{
        friend class Vector;//дружественный класс
public:
        Iterator(){elem=0;}//конструктор без параметров
        Iterator(const Iterator&it){elem=it.elem;}//конструктор
                                                    // копирования
//перегруженные операции сравнения
        bool operator==(const Iterator&it){return elem==it.elem;} bool opera-
        tor!=(const Iterator&it){return lem!=it.elem;};
//перегруженная операция инкремент
        void operator++(){ ++elem;};
//перегруженная операция декремент
        void operator--(){--elem;}
//перегруженная операция разыменования
        int& operator *() const{ return*elem;}
private:
        int*elem;//указатель на элемент типа int
};
```

7 В класс Vector добавить атрибуты и методы для работы с итератором:

```
class Vector
{
public:

. . . . . .
        Iterator first(){return beg;}//возвращает указатель на
                        //первый элемент
```

```
        Iterator last(){return end;}//возвращает указатель на
                                    //элемент, следующий за последним
private:
        int size; int*data;
        Iterator beg;//указатель на первый элемент вектора
        1Iterator end;//указатель на элемент, следующий
                      //за последним
};
```

8 В конструкторы добавить операторы, инициализирующие значения beg и end:

```
Vector::Vector(int s,int k)
{
        . . . . . .
        beg.elem=&data[0];
        end.elem=&data[size];
}

Vector::Vector(const Vector&a)
{
        . . . . . .
        beg=a.beg;
        end=a.end;
}
Vector&Vector::operator=(const Vector&a)
{
        . . . . . .
        beg=a.beg;
        end=a.end;
        return *this;
}
```

9 В функцию main() добавить операторы для тестирования операций итератора:

```
void main()
{
        . . . . . .
//разыменовываем значение, которое возвращает a.first()
//и выводим его
        cout<<*(a.first())<<endl;
//переменную типа Iterator устанавливаем на первый элемент
//вектора a с помощью метода first
```

```
        Iterator i=a.first();
//операция инкремент
        i++;
//разыменовываем итератор и выводим его значение
        cout<<*i<<endl;
//выводим значения элементов вектора с помощью итератора
        for( i=a.first();i!=a.last();i++)cout<<*i<<endl;
}
```

2.4 Индивидуальные задания

Задания по вариантам представлены в таблице 2.1.

Таблица 2.1 Задания по вариантам

Номер варианта	Задание
1	Класс-контейнер вектор с элементами типа int. Реализовать операции: [] – доступ по индексу; () – определение размера вектора; + число – добавление константы ко всем элементам вектора; ++ – переход к следующему элементу (с помощью класса-итератора)
2	Класс- контейнер вектор с элементами типа int. Реализовать операции: []– доступ по индексу; int() – определение размера вектора; + вектор – сложение элементов векторов a[i]+b[i]; +n – переход вправо к элементу с номером n (с помощью класса-итератора)
3	Класс- контейнер вектор с элементами типа int. Реализовать операции: [] – доступ по индексу; + вектор – сложение элементов векторов a[i]+b[i]; + число – добавление константы ко всем элементам вектора; -- – переход к предыдущему элементу (с помощью класса-итератора)
4	Класс-контейнер вектор с элементами типа int. Реализовать операции: [] – доступ по индексу; () – определение размера вектора; * число – умножение всех элементов вектора на число; –n – переход влево к элементу с номером n (с помощью класса-итератора)
5	Класс- контейнер вектор с элементами типа int. Реализовать операции: [] – доступ по индексу; int() – определение размера вектора; * вектор – умножение элементов векторов a[i]*b[i]; + n – переход вправо к элементу с номером n (с помощью класса-итератора)

Продолжение таблицы 2.1

Номер варианта	Задание
6	Класс-контейнер множество с элементами типа int. Реализовать операции: [] – доступ по индексу; () – определение размера множества; + – объединение множеств; ++ – переход к следующему элементу (с помощью класса-итератора)
7	Класс-контейнер множество с элементами типа int. Реализовать операции: [] – доступ по индексу; int() – определение размера вектора; * – пересечение множеств; -- – переход к предыдущему элементу (с помощью класса-итератора)
8	Класс-контейнер множество с элементами типа int. Реализовать операции: [] – доступ по индексу; == – проверка на равенство; > число – принадлежность числа множеству; - n – переход влево к элементу с номером n (с помощью класса-итератора)
9	Класс-контейнер множество с элементами типа int. Реализовать операции: [] – доступа по индексу; != – проверка на неравенство; < число – принадлежность числа множеству; + n – переход вправо к элементу с номером n (с помощью класса-итератора)
10	Класс-контейнер множество с элементами типа int. Реализовать операции: [] – доступ по индексу; () – определение размера вектора; - – разность множеств; -- – переход к предыдущему элементу (с помощью класса-итератора)
11	Класс-контейнер список с ключевыми значениями типа int. Реализовать операции: [] – доступ по индексу; int() – определение размера списка; + вектор – сложение элементов списков a[i]+b[i]; - n – переход влево к элементу с номером n (с помощью класса-итератора)
12	Класс-контейнер список с ключевыми значениями типа int. Реализовать операции: [] – доступ по индексу; () – определение размера вектора; + число – добавляет константу ко всем элементам вектора; ++ – переход к следующему элементу (с помощью класса-итератора)

Окончание таблицы 2.1

Номер варианта	Задание
13	Класс-контейнер список с ключевыми значениями типа int. Реализовать операции: [] – доступ по индексу; + вектор – сложение элементов списков a[i]+b[i]; + число – добавление константы ко всем элементам списка; -- – переход к предыдущему элементу (с помощью класса-итератора)
14	Класс-контейнер список с ключевыми значениями типа int. Реализовать операции: [] – доступ по индексу; () – определение размера списка; * число – умножение всех элементов списка на число; - n – переход влево к элементу с номером n (с помощью класса-итератора)
15	Класс-контейнер список с ключевыми значениями типа int. Реализовать операции: [] – доступ по индексу; int() – определение размера списка; * вектор – умножение элементов списков a[i]*b[i]; + n – переход вправо к элементу с номером n (с помощью класса-итератора)

Контрольные вопросы

1 Что такое абстрактный тип данных? Привести примеры АТД.
2 Привести примеры абстракции через параметризацию.
3 Привести примеры абстракции через спецификацию.
4 Что такое контейнер? Привести примеры.
5 Какие группы операций выделяют в контейнерах?
6 Какие виды доступа к элементам контейнера существуют? Привести примеры.
7 Что такое итератор?
8 Каким образом может быть реализован итератор?
9 Каким образом можно организовать объединение контейнеров?
10 Какой доступ к элементам предоставляет контейнер, состоящий из элементов «ключ – значение»?
11 Как называется контейнер, в котором вставка и удаление элементов выполняется на одном конце контейнера?

3 Лабораторная работа № 13. STL-алгоритмы

Цель работы: применение обобщенных алгоритмов вместе с контейнерами библиотеки STL.

3.1 Основные концепции STL

STL – Standard Template Library – в дополнение к контейнерным классам включает набор обобщенных алгоритмов. 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 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.

3.2 Алгоритмы

Включение заголовка <algorithm> определяет набор функций, специально разработанных для применения совместно с контейнерами. Диапазон любой коллекции объектов доступен с помощью итераторов либо указателей, например, в массиве или в STL-контейнере. Следует отметить, что алгоритмы работают с элементами через итераторы, ничего не зная о контейнере, его структуре и размере. Таким образом, алгоритм может работать с очень разными контейнерами, содержащими значения разнообразных типов. Алгоритмы, которые возвращают итератор, как правило, для сообщения о неудаче используют конец входной последовательности. Алгоритмы не выполняют проверки диапазона на их входе и выходе.

Когда алгоритм возвращает итератор, это будет итератор того же типа, что и был на входе. Алгоритмы в STL реализуют большинство распространенных универсальных операций с контейнерами, такие как просмотр, сортировка, поиск, вставка и удаление элементов. Подробнее о функциях: http://www.cplusplus.com/reference/algorithm/.

Ниже приведены наиболее часто используемые функций-алгоритмов STL.

Операции, которые не изменяют элементы контейнера:

for_earch() – вызывает заданную функцию для каждого элемента последовательности;

find() – находит первое вхождение значения в последовательность;

find_if() – находит первое соответствие предикату в последовательности;

count() – подсчитывает количество вхождений значения в последовательность;

count_if() – подсчитывает количество выполнений предиката в последовательности;

search() – находит первое вхождение последовательности как подпоследовательности;

search_n() – находит n-е вхождение значения в последовательность.

Операции, которые меняют содержимое контейнера:

copy() – копирует последовательность, начиная с первого элемента;

swap() – меняет местами два элемента;

replace() – заменяет элементы с указанным значением;

replace_if() – заменяет элементы при выполнении предиката;

replace_copy() – копирует последовательность, заменяя элементы с указанным значением;

replace_copy_if() – копирует последовательность, заменяя элементы при выполнении предиката;

fill() – заменяет все элементы данным значением;

remove() – удаляет элементы с данным значением;

remove_if() – удаляет элементы при выполнении предиката;

remove_copy() – копирует последовательность, удаляя элементы с указанным значением;

remove_copy_if() – копирует последовательность, удаляя элементы при выполнении предиката;

reverse() – меняет порядок следования элементов на обратный;

random_shuffle() – перемещает элементы согласно случайному равномерному распределению («тасует» последовательность);

transform() – выполняет заданную операцию над каждым элементом последовательности;

unique() – удаляет равные соседние элементы;

unique_copy() – копирует последовательность, удаляя равные соседние элементы.

Операции сортировки:

sort() – сортирует последовательность с хорошей средней эффективностью;

partial_sort() – сортирует часть последовательности;

stable_sort() – сортирует последовательность, сохраняя порядок следования равных элементов;

lower_bound() – находит первое вхождение значения в отсортированной последовательности;

upper_bound() – находит первый элемент, больший, чем заданное значение;

binary_search() – определяет, есть ли данный элемент в отсортированной последовательности;

merge() – сливает две отсортированные последовательности.

Операции работы с множествами:

includes() – проверка на вхождение;

set_union() – объединение множеств;

set_intersection() – пересечение множеств;

set_difference() – разность множеств.

Операции поиска минимумов и максимумов:

min() – меньшее из двух;

max() – большее из двух;

min_element() – наименьшее значение в последовательности;

max_element() – наибольшее значение в последовательности.

Операции перестановок:

next_permutation() – следующая перестановка в лексикографическом порядке;

pred_permutation() – предыдущая перестановка в лексикографическом порядке.

3.3 Функциональные объекты

Функциональные объекты (ФО или предикат) – это объекты, использующие синтаксис такой же, как обычный вызов функции. Это возможно при определении в классе метода operator().

Например, следующим образом:

```
struct myclass {
int operator()(int a) {return a;}
} myobject;
int x = myobject (0); // function-like syntax with object myobject
```

ФО обычно передают как аргументы функции, например, предикат сравнения – это функция, передаваемая стандартному алгоритму. ФО важны для эффективного использования библиотеки. Разработчик может использовать и указатель на функцию, и обьекты классов, в которых определен operator(). В свою очередь, это позволяет алгоритмическим шаблонам работать как с указателями на функции, так и с функциональными объектами.

Ниже приведен пример использования алгоритма for_each() с ФО.

```
#include <iostream> // std::cout
#include <algorithm> // std::for_each
#include <vector> // std::vector
void myfunction (int i) { // function:
std::cout << ' ' << i;
}
// определение функционального объекта:
struct myclass { // function object type:
   void operator() (int i) {std::cout << ' ' << i;}
} myobject;

int main () {
   std::vector<int> myvector;
   myvector.push_back(10);
   myvector.push_back(20);
   myvector.push_back(30);
   std::cout << "myvector contains:";
   for_each (myvector.begin(), myvector.end(), myfunction);
   std::cout << '\n';
   // or:
```

```
    std::cout << "myvector contains:";
    for_each (myvector.begin(), myvector.end(), myobject);
    std::cout << '\n';
    return 0;
}
```

Например, если мы хотим поэлементно сложить два вектора a и b, содержащие double, и поместить результат в a, мы можем сделать это так:

```
transform(a.begin(), a.end(), b.begin(), a.begin(), plus‹double›());
```

Если мы хотим отрицать каждый элемент a, мы можем сделать это так:

```
transform(a.begin(), a.end(), a.begin(), negate‹double›());
```

Соответствующие функции вставят сложение и отрицание. Чтобы позволить адаптерам и другим компонентам манипулировать функциональными объектами, которые используют один или два параметра, требуется, чтобы они, соответственно, обеспечили определение типов (typedefs) argument_type и result_type для функциональных объектов, которые используют один параметр, и first_argument_type, second_argument_type и result_type для функциональных объектов, которые используют два параметра.

3.4 Индивидуальные задания

Задача 1

1 Создать последовательный контейнер.

2 Заполнить контейнер элементами стандартного типа (тип указан в варианте). Для пользовательского типа перегрузить необходимые операции.

3 Заменить элементы в соответствии с заданием (использовать алгоритмы replace_if(), replace_copy(), replace_copy_if(), fill()).

4 Удалить элементы в соответствии с заданием (использовать алгоритмы remove(), remove_if(), replace_copy(), replace_copy_if()).

5 Отсортировать контейнер по убыванию и по возрастанию ключевого поля (использовать алгоритм sort()).

6 Найти в контейнере заданный элемент (использовать алгоритмы find(), find_if(), count(), count_if()).

7 Выполнить задание варианта для полученного контейнера (использовать алгоритм for_each()).

8 Для выполнения всех заданий использовать стандартные алгоритмы библиотеки STL.

Задача 2

1 Создать несортированный ассоциативный контейнер.

2 Заполнить контейнер элементами стандартного типа (тип указан в варианте). Для пользовательского типа перегрузить необходимые операции.

3 Заменить элементы в соответствии с заданием (использовать алгоритмы replace_if(), replace_copy(), replace_copy_if(), fill()).

4 Удалить элементы в соответствии с заданием (использовать алгоритмы remove(), remove_if(), replace_copy(), replace_copy_if()).

5 Отсортировать контейнер по убыванию и по возрастанию ключевого поля (использовать алгоритм sort()).

6 Найти в контейнере заданный элемент (использовать аалгоритмы find(), ind_if(), count(), count_if()).

7 Выполнить задание варианта для полученного контейнера (использовать алгоритм for_each()).

8 Для выполнения всех заданий использовать стандартные алгоритмы библиотеки STL.

Задача 3

1 Создать ассоциативный контейнер.

2 Заполнить контейнер элементами стандартного типа (тип указан в варианте). Для пользовательского типа перегрузить необходимые операции.

3 Заменить элементы в соответствии с заданием (использовать алгоритмы replace_if(), replace_copy(), replace_copy_if(), fill()).

4 Удалить элементы в соответствии с заданием (использовать алгоритмы remove(), remove_if(), replace_copy(), replace_copy_if()).

5 Отсортировать контейнер по убыванию и по возрастанию ключевого поля (использовать алгоритм sort()).

6 Найти в контейнере заданный элемент (использовать аалгоритмы find(), find_if(), count(), count_if()).

7 Выполнить задание варианта для полученного контейнера (использовать алгоритм for_each()).

8 Для выполнения всех заданий использовать стандартные алгоритмы библиотеки STL.

Варианты заданий представлены на рисунке 3.1.

Вариант	номер задачи	тип контейнера	тип элементов	Что нужно сделать к каждой задаче варианта
1	1	*вектор*	*АТД — time*	Пункт 3 Задача X(X = 1...3) Заменить максимальный элемент на заданное значение
	2	*Не сортированный ассоциативный контейнер - словарь с дубликатами.*	*АТД — time*	Пункт 4 Задача X(X = 1..3) Найти минимальный элемент и удалить его из контейнера
	3	*Ассоциативный контейнер - множество.*	*АТД — time*	Пункт 5 Задача X(X = 1...3)К каждому элементу добавить среднее арифметическое контейнера.
2	1	*список*	*АТД — time*	Пункт 3 Задача X(X = 1...3) Найти минимальный элемент и добавить его в конец контейнера.
	2	*Не сортированный ассоциативный контейнер — словарь.*	*АТД — time*	Пункт 4 Задача X(X = 1...3) Найти элемент с заданным ключом и удалить его из контейнера.
	3	*Ассоциативный контейнер – множество с дубликатами.*	*АТД — time*	Пункт 5 Задача X(X = 1...3) К каждому элементу добавить сумму минимального и максимального элементов контейнера.
3	1	*двунаправленная очередь*	*АТД — time*	Пункт 3 Задача X(X = 1...3) Найти элемент с заданным ключом и добавить его на заданную позицию контейнера
	2	*Не сортированный ассоциативный контейнер – множество с дупликатами.*	*АТД — time*	Пункт 4 Задача X(X = 1..3) Найти элемент с заданным ключом и удалить его из контейнера
	3	*Ассоциативный контейнер – словарь*	*АТД — time*	Пункт 5 Задача X(X = 1...3) Найти разницу между максимальным и минимальным элементами контейнера и вычесть ее из каждого элемента контейнера.
4	1	*двунаправленная очередь*	*АТД — time*	Пункт 3 Задача X(X = 1...3) Найти максимальный элемент и добавить его в конец контейнера.
	2	*Не сортированный ассоциативный контейнер - множество.*	*АТД — time*	Пункт 4 Задача X(X = 1...3) Найти элемент с заданным ключом и удалить его из контейнера.
	3	*Ассоциативный контейнер – словарь с дубликатами.*	*АТД — time*	Пункт 5 Задача X(X = 1...3) К каждому элементу добавить среднее арифметическое элементов контейнера.
5	1	*список*	*АТД — time*	Пункт 3 Задача X(X = 1...3) Найти минимальный элемент и добавить его на заданную позицию контейнера.
	2	*Не сортированный ассоциативный контейнер – словарь с дубликатами.*	*АТД — time*	Пункт 4 Задача X(X = 1..3) Найти элементы большие среднего арифметического и удалить их из контейнера.
	3	*Ассоциативный контейнер — множество.*	*АТД — time*	Пункт 5 Задача X(X = 1...3) Каждый элемент домножить на максимальный элемент контейнера.
6	1	*вектор*	*АТД — money*	Пункт 3 Задача X(X = 1...3) Найти максимальный элемент и добавить его в начало контейнера.
	2	*Не сортированный ассоциативный контейнер – словарь.*	*АТД — money*	Пункт 4 Задача X(X = 1...3) Найти минимальный элемент и удалить его из контейнера.
	3	*Ассоциативный контейнер – множество с дубликатами.*	*АТД — money*	Пункт 5 Задача X(X = 1...3) К каждому элементу добавить среднее арифметическое контейнера.

Рисунок 3.1 – Варианты заданий

Вариант	номер задачи	тип контейнера	тип элементов	Что нужно сделать к каждой задаче варианта
7	1	*список*	*АТД — money*	Пункт 3 Задача X(X = 1...3) Найти минимальный элемент и добавить его в конец контейнера.
	2	*Не сортированный ассоциативный контейнер – множество с дубликатами.*	*АТД — money*	Пункт 4 Задача X(X = 1..3) Найти элемент с заданным ключом и удалить его из контейнера.
	3	*Ассоциативный контейнер — словарь.*	*АТД — money*	Пункт 5 Задача X(X = 1...3) К каждому элементу добавить сумму минимального и максимального элементов контейнера.
8	1	*список*	*АТД — money*	Пункт 3 Задача X(X = 1...3) Найти элемент с заданным ключом и добавить его на заданную позицию контейнера.
	2	*Не сортированный ассоциативный контейнер – множество.*	*АТД — money*	Пункт 4 Задача X(X = 1...3) Найти элемент с заданным ключом и удалить его из контейнера.
	3	*Ассоциативный контейнер – словарь с дубликатами.*	*АТД — money*	Пункт 5 Задача X(X = 1...3) Найти разницу между максимальным и минимальным элементами контейнера и вычесть ее из каждого элемента контейнера.
9	1	*двунаправленная очередь*	*АТД — money*	Пункт 3 Задача X(X = 1...3) Найти максимальный элемент и добавить его в конец контейнера.
	2	*Не сортированный ассоциативный контейнер – словарь с дубликатами.*	*АТД — money*	Пункт 4 Задача X(X = 1..3) Найти элемент с заданным ключом и удалить его из контейнера.
	3	*Ассоциативный контейнер — множество.*	*АТД — money*	Пункт 5 Задача X(X = 1...3) К каждому элементу добавить среднее арифметическое элементов контейнера.
10	1	*вектор*	*АТД — money*	Пункт 3 Задача X(X = 1...3) Найти минимальный элемент и добавить его на заданную позицию контейнера.
	2	*Не сортированный ассоциативный контейнер – словарь.*	*АТД — money*	Пункт 4 Задача X(X = 1...3) Найти элементы большие среднего арифметического и удалить их из контейнера.
	3	*Ассоциативный контейнер – множество с дубликатами.*	*АТД — money*	Пункт 5 Задача X(X = 1...3) Каждый элемент домножить на максимальный элемент контейнера.
11	1	*вектор*	*АТД — money*	Пункт 3 Задача X(X = 1...3) Найти среднее арифметическое и добавить его в начало контейнера.
	2	*Не сортированный ассоциативный контейнер – множество с дубликатами.*	*АТД — money*	Пункт 4 Задача X(X = 1..3) Найти элемент с заданным ключом и удалить их из контейнера.
	3	*Ассоциативный контейнер — словарь.*	*АТД — money*	Пункт 5 Задача X(X = 1...3) Из каждого элемента вычесть минимальный элемент контейнера.
12	1	*список*	*АТД — point*	Пункт 3 Задача X(X = 1...3) Найти среднее арифметическое и добавить его на заданную позицию контейнера.
	2	*Не сортированный ассоциативный контейнер – множество.*	*АТД — point*	Пункт 4 Задача X(X = 1...3) Найти элементы с ключами из заданного диапазона и удалить их из контейнера.
	3	*Ассоциативный контейнер – словарь с дубликатами.*	*АТД — point*	Пункт 5 Задача X(X = 1...3) Из каждого элемента вычесть среднее арифметическое контейнера.
13	1	*двунаправленная очередь*	*АТД — point*	Пункт 3 Задача X(X = 1...3) Найти максимальный элемент и добавить его в конец контейнера.
	2	*Не сортированный ассоциативный контейнер – словарь с дубликатами.*	*АТД — point*	Пункт 4 Задача X(X = 1..3) Найти элементы с ключами из заданного диапазона и удалить их из контейнера.
	3	*Ассоциативный контейнер — множество.*	*АТД — point*	Пункт 5 Задача X(X = 1...3) К каждому элементу добавить среднее арифметическое контейнера.

Продолжение рисунка 3.1

Вариант	номер задачи	тип контейнера	тип элементов	Что нужно сделать к каждой задаче варианта
14	1	*вектор*	*АТД — point*	Пункт 3 Задача X(X = 1...3) Найти минимальный элемент и добавить его на заданную позицию контейнере.
	2	*Не сортированный ассоциативный контейнер – словарь.*	*АТД — point*	Пункт 4 Задача X(X = 1..3) Найти меньше среднего арифметического и удалить их из контейнера.
	3	*Ассоциативный контейнер — множество с дубликатами.*	*АТД — point*	Пункт 5 Задача X(X = 1...3) Каждый элемент разделить на максимальный элемент контейнера.
15	1	*список*	*АТД — point*	Пункт 3 Задача X(X = 1...3) Найти среднее арифметическое и добавить его в конец контейнера.
	2	*Не сортированный ассоциативный контейнер – множество с дубликатами.*	*АТД — point*	Пункт 4 Задача X(X = 1...3) Найти элементы сключами из заданного диапазона и удалить их из контейнера.
	3	*Ассоциативный контейнер – словарь.*	*АТД — point*	Пункт 5 Задача X(X = 1...3) К каждому элементу добавить сумму минимального и максимального элементов контейнера.

Окончание рисунка 3.1

Контрольные вопросы

1 Из каких частей состоит библиотека STL?

2 Какие типы контейнеров существуют в STL?

3 Что нужно сделать для использования контейнера STL в своей программе?

4 Что представляет собой итератор?

5 Какие операции можно выполнять над итераторами?

6 Каким образом можно организовать цикл для перебора контейнера с использованием итератора?

7 Какие типы итераторов существуют?

8 Перечислить операции и методы общие для всех контейнеров.

9 Какие операции являются эффективными для контейнера vector? Почему?

10 Какие операции являются эффективными для контейнера list? Почему?

11 Какие операции являются эффективными для контейнера deque? Почему?

12 Перечислить методы, которые поддерживает последовательный контейнер vector.

13 Перечислить методы, которые поддерживает последовательный контейнер list.

14 Перечислить методы, которые поддерживает последовательный контейнер deque.

4 Лабораторная работа № 14. Создание приложения «Калькулятор» в Qt

Цель работы: изучить основные принципы построения приложений, имеющих графический интерфейс пользователя с использованием кросс-платформенной библиотеки Qt; научиться создавать оконные приложения с помощью библиотеки Qt.

Рассмотрим создание приложения «Калькулятор», наглядно иллюстрирующего возможности библиотеки Qt в части компактного кода программы и динамического создания элементов управления.

Разработаем калькулятор, внешний вид которого представлен на рисунке 4.1.

Рисунок 4.1 – Приложение «Калькулятор»

Форму приложения реализуем на основе класса Qdialog. При этом воспользуемся классом QSignalMapper, позволяющим в данном случае унифицировать процесс обработки нажатий кнопок, направив сигналы от всех кнопок на единственный слот void clicked(int id), параметром которого будет идентификатор кнопки. Число будем отображать в элементе m_pLineEdit, имеющем тип QLineEdit*.

Заголовочный файл calcDialog.h с описанием класса CalcDialog представлен далее:

```
#ifndef _CALC_DIALOG_H_
#define _CALC_DIALOG_H_
#include <QDialog>
#include <QLineEdit>
#include <QSignalMapper>

/// Класс, реализующий калькулятор
class CalcDialog: public QDialog
{
        Q_OBJECT
```

```
public:
        CalcDialog( QWidget * parent = 0);
        virtual ~CalcDialog(){};
protected:
        QSignalMapper * m_pSignalMapper;
        QLineEdit * m_pLineEdit;
        double m_Val; ///< Значение, с которым будет выполнена операция
        int m_Op; ///< Код нажатой операции
        bool m_bPerf;///< Операция была выполнена. Надо очистить поле
                     ///< ввода
        void initNum();///< Инициализировать переменные, связанные с
                       ///<вычислениями
        double getNumEdit(); ///< Получить число из m_pLineEdit
        void setNumEdit( double ); ///< Отобразить число в m_pLineEdit
        /// Вычислить предыдущую операцию
        ///(в бинарных операциях был введен второй операнд)
        void calcPrevOp( int curOp );
        /// Проверить, была ли выполнена операция при нажатии на
        /// цифровую клавишу
        /// Если операция выполнена, значит m_pLineEdit необходимо
        /// очистить
        void checkOpPerf();
private slots:
        /// Слот для обработки нажатий всех кнопок
       void clicked(int id);
};
#endif
```

При создании кнопок необходимо обеспечить их правильное размещение. Воспользуемся средствами Qt и создадим схемы размещения виджетов в соответствии с таблицей 4.1 и рисунком 4.2.

Таблица 4.1 – Назначение схем выравнивания

Имя объекта	Класс	Назначение
gridLayout	QGridLayout	Цифровые кнопки и кнопки операций
bccKeysLayout	QHBoxLayout	Кнопки удаления последней цифры, сброс текущего значения и сброс операции
mainKeysLayout	QHBoxLayout	Цифровые кнопки + кнопка выполнения «=»
dlgLayout	QVBoxLayout	Вертикальное размещение элемента QLineEdit и групп кнопок bccKeysLayout и mainKeysLayout

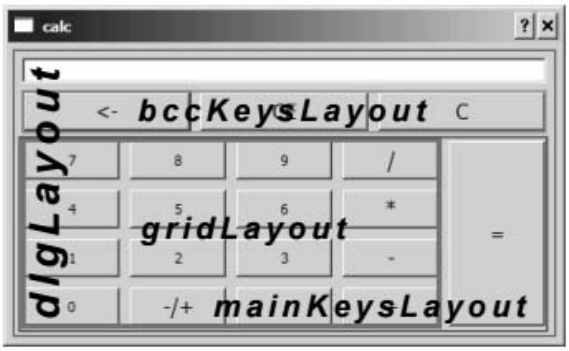


Рисунок 4.2 – Схемы выравнивания кнопок калькулятора

Реализуем основную программу – файл calcDialog.cpp. Ниже приведен текст программы, снабженный комментариями.

```
#include <QtGui>
#include <QVector>
#include <QGridLayout>
#include <QHBoxLayout>
#include <QVBoxLayout>
#include ”calcDialog.h”

// Идентификаторы кнопок
// Для цифровых кнопок идентификатор является соответствующая цифра
#define DIV 10
#define MUL 11
#define MINUS 12
#define PLUS 13
#define INVERSE 15
#define DOT 16
#define EQ 20
#define BKSP 30
#define CLR 31
#define CLR_ALL 32

// количество кнопок в группе, отображаемой в виде сетки
#define GRID_KEYS 16

/// Описатель кнопки
struct BtnDescr{
        QString text; ///< Отображаемый на кнопке текст
        int id; ///< Идентификатор кнопки
        BtnDescr() { id=0;}; ///< Конструктор по умолчанию
        ///< Конструктор для инициализации
```

```
        BtnDescr( const QString & str, int i)
        { text = str; id = i; };
};

/// Динамический массив-вектор элементов описателей кнопок
QVector<BtnDescr> _btnDescr;

/// Инициализация массива _btnDescr всеми отображаемыми кнопками
void InitBtnDescrArray()
{
        _btnDescr.push_back( BtnDescr("7", 7) );
        _btnDescr.push_back( BtnDescr("8", 8) );
        _btnDescr.push_back( BtnDescr("9", 9) );
        _btnDescr.push_back( BtnDescr("/", DIV) );
        _btnDescr.push_back( BtnDescr("4", 4) );
        _btnDescr.push_back( BtnDescr("5", 5) );
        _btnDescr.push_back( BtnDescr("6", 6) );
        _btnDescr.push_back( BtnDescr("*", MUL) );
        _btnDescr.push_back( BtnDescr("1", 1) );
        _btnDescr.push_back( BtnDescr("2", 2) );
        _btnDescr.push_back( BtnDescr("3", 3) );
        _btnDescr.push_back( BtnDescr("-", MINUS) );
        _btnDescr.push_back( BtnDescr("0", 0) );
        _btnDescr.push_back( BtnDescr("-/+", INVERSE) );
        _btnDescr.push_back( BtnDescr(".", DOT) );
        _btnDescr.push_back( BtnDescr("+", PLUS) );
        _btnDescr.push_back( BtnDescr("<-",BKSP) );
        _btnDescr.push_back( BtnDescr("CE",CLR) );
        _btnDescr.push_back( BtnDescr("C", CLR_ALL) );
        _btnDescr.push_back( BtnDescr("=", EQ) );
}

/// конструктор класса калькулятора
CalcDialog::CalcDialog( QWidget * parent)
{
        initNum(); // инициализируем счетные переменные
        InitBtnDescrArray(); // инициализируем массив с описанием кнопок

        // создаем форму
        m_pLineEdit = new QLineEdit(this);

        // устанавливаем режим только чтения – разрешаем ввод только
        // с нарисованных кнопок
        m_pLineEdit->setReadOnly ( true );
```

```
m_pSignalMapper = new QSignalMapper(this);
// создаем схемы выравнивания
QGridLayout *gridLayout = new QGridLayout();
QHBoxLayout *bccKeysLayout = new QHBoxLayout();
QHBoxLayout *mainKeysLayout = new QHBoxLayout();
QVBoxLayout *dlgLayout = new QVBoxLayout();

// Заполняем форму кнопками из _btnDescr
for (int i = 0; i < _btnDescr.size(); i++) {
        // Создаем кнопку с текстом из очередного описателя
        QPushButton *button = new QPushButton(_btnDescr[i].text);

        // если кнопка в основном блоке цифровых или "=" -
        // разрешаем изменение всех размеров
        if( i >= GRID_KEYS + 3 || i < GRID_KEYS)
                button->setSizePolicy ( QSizePolicy::Expanding,
                                        QSizePolicy::Expanding);

        // если кнопка не цифровая – увеличиваем шрифт надписи на
        // 4 пункта
        if( _btnDescr[i].id >= 10 ){
                QFont fnt = button->font();
                fnt.setPointSize( fnt.pointSize () + 4 );
                button->setFont( fnt );
        }

        // связываем сигнал нажатия кнопки с объектом
        // m_pSignalMapper
        connect(button, SIGNAL(clicked()), m_pSignalMapper,
                  SLOT(map()));
        // обеспечиваем соответствие кнопки её идентификатору
        m_pSignalMapper->setMapping(button, _btnDescr[i].id);
        if(i<GRID_KEYS) // Если кнопка из центрального блока –
                                //помещаем в сетку
                gridLayout->addWidget(button, i / 4, i % 4);
        else if( i < GRID_KEYS + 3) // кнопка из верхнего блока –
                                        // в bccKeysLayout
                bccKeysLayout->addWidget(button);

        else { // кнопка "=" – помещаем в блок mainKeysLayout
                // после gridLayout
                mainKeysLayout->addLayout(gridLayout);
                mainKeysLayout->addWidget(button);
        }
```

```
        }
        // связываем сигнал из m_pSignalMapper о нажатии со слотом clicked
        // нашего класса
        connect(m_pSignalMapper, SIGNAL(mapped(int)),
        this, SLOT(clicked(int)));

        // добавляем блоки кнопок в схему выравнивания всей формы
        dlgLayout->addWidget(m_pLineEdit);
        dlgLayout->addLayout(bccKeysLayout);
        dlgLayout->addLayout(mainKeysLayout);

        // связываем схему выравнивания dlgLayout с формой
        setLayout(dlgLayout);
        // отображаем "0" в поле ввода чисел m_pLineEdit
        setNumEdit( 0 );
};

// Обработка нажатия клавиш
void CalcDialog::clicked(int id)
{ // по идентификатору кнопки ищем действие для выполнения
        switch(id){
                case INVERSE: // унарная операция +/-
                {
                        setNumEdit( getNumEdit() * -1.0 ); break;
                 };
                case DOT: // добавление десятичной точки
                {
                        // если на экране результат предыдущей операции –
                        // сбросить
                         checkOpPerf();
                         QString str = m_pLineEdit->text ();
                         str.append( "." ); // добавляем точку к строке
                         bool ok = false;
                         // проверяем, является ли результат числом (исключаем
                        // 0.1. )
                         str.toDouble(&ok);
                         // если строка является числом – помещаем результат в  ..
                         // m_pLineEdit
                         if( ok ) m_pLineEdit->setText ( str );
                         break;
                };
                case DIV: // бинарные арифметические операции
                case MUL:
                case PLUS:
```

```
                case MINUS:
                case EQ:{
                        calcPrevOp( id );
                        break;
                }
                case CLR_ALL: initNum();// удалить всё
                case CLR:{
                         setNumEdit( 0 ); // записать в m_pLineEdit число 0
                         break;
                 }
                case BKSP:{ // удалить последний символ
                 // если на экране результат предыдущей операции – сбросить
                 checkOpPerf();
                 QString str = m_pLineEdit->text ();
                 if( str.length() ){
                         // если строка в m_pLineEdit ненулевая – удалить символ
                         str.remove( str.length()-1, 1 );
                         m_pLineEdit->setText ( str );
                 }
                 break;
                }
                default:{ // обработка цифровых клавиш
                        // если на экране результат предыдущей операции –
                        // сбросить
                         checkOpPerf();
                         QString sId;
                         // сформировать строку по идентификатору нажатой
                         // клавиши
                         sId.setNum( id );
                         QString str = m_pLineEdit->text ();
                         if( str == "0" )
                                 str = sId; // затираем незначащий нуль
                         else
                                 str.append( sId ); // добавить в m_pLineEdit
                                                    //нажатую цифру
                         m_pLineEdit->setText ( str );
        }
 };};

// Получить число из m_pLineEdit
double CalcDialog::getNumEdit()
{
        double result;
        QString str = m_pLineEdit->text ();
```

```
        result = str.toDouble(); // преобразовать строку в число
        return result;
};

// записать число в m_pLineEdit
void CalcDialog::setNumEdit( double num )
{
        QString str;
        str.setNum ( num, 'g', 25 ); // преобразовать вещественное число
                                     // в строку
        m_pLineEdit->setText ( str );
};

// Выполнить предыдущую бинарную операцию
void CalcDialog::calcPrevOp( int curOp )
{
        // получить число на экране
        // m_Val хранит число, введенное до нажатия кнопки операции
        double num = getNumEdit();
        switch( m_Op )
        {
                case DIV:{
                        if ( num != 0) m_Val /= num;
                        else m_Val = 0;
                        break;
                }
                case MUL:{
                        m_Val *= num;
                        break;
                }
                case PLUS:{
                        m_Val += num;
                        break;
                }
                case MINUS:{
                        m_Val -= num;
                        break;
                }
                case EQ: { // если была нажата кнопка "=" – не делать ничего
                        m_Val = num;
                        break;
                }
        }
```

```
        m_Op = curOp; // запомнить результат текущей операции
        setNumEdit( m_Val ); // отобразить результат
        m_bPerf = true; // поставить флаг выполнения операции
};

void CalcDialog::checkOpPerf()
{
        if( m_bPerf ){
                // если что-то выполнялось – очистить m_pLineEdit
                m_pLineEdit->clear();
                m_bPerf = false;
        };
};

void CalcDialog::initNum()
{
        m_bPerf = false; m_Val = 0; m_Op = EQ;
};
```

В отдельном файле calc.cpp реализуем запуск приложения и создание формы CalcDialog.

```
#include <QApplication>
#include "calcDialog.h"
int main(int argc, char *argv[])
{
        QApplication app(argc, argv);
        CalcDialog * dialog = new CalcDialog();

        dialog->show(); // отображаем окно
        return app.exec(); // запускаем цикл обработки сообщений
}
```

Последний этап – создание проекта для сборки приложения – файл calc.pro.

```
TEMPLATE = app
TARGET = calc
CONFIG += release
# Input
HEADERS += calcDialog.h
SOURCES += calc.cpp calcDialog.cpp
```

Сборка проекта производится командами qmake calc.pro и nmake.

Задание для самостоятельной работы. Добавьте меню, которое позволяет прейти к «инженерному» виду калькулятора. В расширенной версии добавьте кнопки, выполняющие: бинарные операции (по аналогии с операциями +,-,/,*), а также унарные sin(x) и cos(x) (по аналогии с операцией -/+), и разместите этот ряд кнопок вертикально, слева от цифровых кнопок с использованием нового объекта выравнивания (Layout).

Список литературы

1 **Павловская, Т. А.** C/C++. Программирование на языке высокого уровня: учебник для вузов / Т. А. Павловская. – Санкт-Петербург: Питер, 2004. – 461 с.

2 **Павловская, Т. А.** C++. Объектно-ориентированное программирование: практикум / Т. А. Павловская, Ю. А. Щупак. – Санкт-Петербург: Питер, 2006. – 265 с.

3 **Ашарина, И. В.** Язык C++ и объектно-ориентированное программирование в C++. Лабораторный практикум: учебное пособие для вузов / И. В. Ашарина, Ж. Ф. Крупская. – Москва: Горячая линия – Телеком, 2016. – 232 с.

4 **Прата, С.** Язык программирования C++. Лекции и упражнения / С. Прата. – 6-е изд. – Москва: Вильяме, 2012. – 1248 с.