

Курс C++

Работа со стандартной библиотекой "C++"

Цель: получить общее представление о назначении и особенностях STL

Вопросы для актуализации ранее полученных знаний:

1. Что такое шаблонная функция, как ее объявить и использовать?
2. Что такое шаблонный класс?
3. Какие виды коллекций Вам приходилось использовать?
4. Приведите примеры алгоритмов, которые использовались этими коллекциями?

Стандартная библиотека шаблонов (Standard Template Library) это обобщенная библиотека, которая содержит решения для обработки коллекций (массивов) данных с использованием эффективных алгоритмов.

Особенности STL:

- STL содержит ряд классов и алгоритмы, которые работают с этими классами
- Классы STL являются обобщенными (шаблонными), т.е. могут использоваться с различными типами данных

Разработчики: Алекс Степанов, Мен Ли. 1-ая версия вышла в 1994 г. Библиотека STL входит в стандарт ISO "Information Technology – Programming Languages – C++" (был принят в 1998г.)

Основные понятия STL

STL состоит из следующих компонентов:

- контейнеры
- итераторы
- алгоритмы
- функторы
- предикаты
- аллокаторы

Контейнеры (Containers)

Назначение класса контейнера – содержать в себе объекты. Классы контейнеров – обобщенные классы, т.е. контейнер может содержать элементы произвольного типа. Контейнеры STL являются однородными, т.е. могут содержать значения только одного и того же типа.

Примеры контейнеров: vector, list, deque, set, multiset, map, multimap

2 основных типа контейнеров:

- последовательные
- ассоциативные

Последовательные контейнеры – упорядоченная коллекция, в которой каждый элемент имеет определенную позицию. Позиция зависит от места вставки, но не зависит от значения элемента.

Примеры: vector, deque, list

Ассоциативные контейнеры – позиция элемента зависит от его значения и выбранного критерия сортировки. Контейнер всегда находится в отсортированном состоянии, что ускоряет выполнение поиска (т.к. используется алгоритм бинарного поиска).

Примеры: set, multiset, map, multimap

Особенности:

- обеспечивают эффективное получение значения на основании ключа.
- не поддерживают вставки элементов в заданную позицию

Итераторы (Iterators)

Итератор – это обобщение указателя. Итератор используется для перемещения по контейнеру и для манипулирования с объектами, находящимися в контейнере. Итераторы обеспечивают однотипный интерфейс для различного типа контейнеров. Например, использование операторов ++ и – для перемещения итератора, оператора * для разыменования.

Каждый тип контейнера имеет собственный класс итератора.

Алгоритмы (Algorithms)

Используются для обработки элементов в контейнере. Например, сортировка, поиск.

Алгоритмы используют итераторы. Так как каждый класс контейнера имеет итератор, один и тот же алгоритм может работать с различными контейнерами.

Данные и операции над данными в STL разобщены. Классы контейнеров хранят данные, обработка данных выполняется алгоритмами. Итераторы являются связью между этими двумя компонентами, они позволяют алгоритмам перебирать элементы контейнеров.

Функторы (Function object or Functor)

Функторы – это объекты, действующие как функции, они могут быть объектами класса или указателями на функцию. Функторы позволяют выполнять конфигурирование алгоритмов для специального использования.

Предикаты (Predicate)

Функции, которые проверяют, удовлетворяет ли объект заданным критериям, и возвращают значение типа bool. Используются, например, в поисковых алгоритмах для задания более сложного критерия поиска.

Аллокаторы (Allocator)

Используются для нестандартного выделения памяти.

Детальный анализ итератора и реализация собственной версии итератора

Цель:

- познакомиться с понятием итератора
- изучить основные категории итераторов
- изучить вспомогательные функции для работы с итераторами
- изучить итераторные адаптеры
- ознакомиться с понятием трактовки итератора
- научиться создавать собственные итераторы

Вопросы для актуализации ранее полученных знаний:

5. Как можно использовать указатели для работы с массивами?
6. Как с помощью указателя перемещаться по элементам массива?
7. Как с помощью указателя получать значения элементов массива?

Понятие итератора

Итератор – это объект, который используется для перебора элементов контейнера. Для любого типа контейнера перебор осуществляется через единый интерфейс, аналогичный интерфейсу обычного указателя. Использование итераторов в алгоритмах позволяет использовать один и тот же алгоритм для любого типа контейнера.

Пример 1. Сравнения указателя и итератора

```
#include <vector>
#include <iostream>

using namespace std;
int main()
{
    int a[3] = {1,2,3};
    vector<int> b;

    //использование указателя для вывода элементов массива
    int * pa;
    for(pa = a; pa < a + 3; pa++)
        cout << *pa << " ";
    cout << endl;

    for(int i = 0; i < 3; i++) b.push_back(i);

    //использование итератора для вывода элементов вектора
    vector<int>::iterator pb;
    for(pb = b.begin(); pb != b.end(); pb++)
        cout << *pb << " ";
    cout << endl;

    return 0;
}
```

Категории итераторов

В зависимости от операций, которые позволяет выполнять итератор, итераторы делятся на 5 категорий.

Итератор ввода (InputIterator) перемещается только вперед и поддерживает только чтение. Пример: чтение данных из стандартного потока ввода (с клавиатуры). Может использоваться для однократных алгоритмов, использующих только чтение (например, алгоритм линейного поиска).

Итератор вывода (OutputIterator) перемещается только вперед и поддерживает только запись. Пример: запись данных в стандартный поток вывода (на монитор). Итератор не может использоваться для повторного перебора интервала, новые данные могут быть записаны не поверх старых данных, а следом за ними. Может

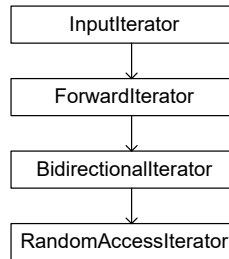
использоваться для однократных алгоритмов, использующих только запись (например, запись данных в контейнер).

Прямой итератор (ForwardIterator) перемещается только вперед, позволяет выполнять чтение и запись. Последовательность значений каждый раз проходится в одном и том же порядке (в отличие от итераторов ввода и вывода). Может использоваться в многопроходных алгоритмах.

Двусторонний итератор (BidirectionalIterator) прямой итератор, который поддерживает перебор элементов в обратном порядке.

Итератор произвольного доступа (RandomAccessIterator) имеет все свойства двустороннего итератора, поддерживает произвольный доступ к элементам. Имеется возможность складывать итератор с числом – смещать его на заданное количество элементов, сравнивать итераторы при помощи операторов отношения (таких как < >), вычислять «расстояние» между двумя итераторами.

При объявлении алгоритмов обычно указывается категория итератора, который используется алгоритмом.



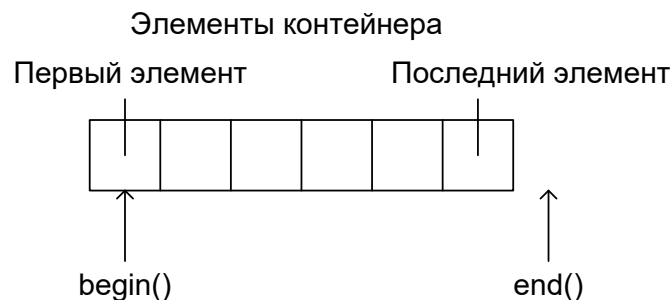
Иерархия (наследование) итераторов

Вопрос: К какому типу контейнеров относится указатель?

Каждый контейнер имеет свой собственный итератор. Каждый контейнер имеет 2 функции:

begin() – возвращает итератор, который указывает на первый элемент контейнера

end() – возвращает итератор, который указывает на элемент, следующий за последним элементом контейнера



Пример 2. Сравнение двунаправленного итератора и итератора произвольного доступа. Итератор контейнера list относится к категории двунаправленного итератора, итератор контейнера vector является итератором произвольного доступа.

```
#include <vector>
#include <iostream>

using namespace std;
int main()
{
    vector<int> b;
    vector<int>::iterator pb;

    for(int i = 0; i < 3; i++) b.push_back(i);

    //итератор произвольного доступа
    //позволяет обращаться к элементами контейнера по индексу
    for(int i = 0; i < (int)b.size(); i++) cout << b[i] << " ";
    cout << endl;

    //итератор поддерживает сравнение, сложение с числом
    for(pb = b.begin(); pb < b.end(); pb+=1) cout << *pb << " ";
    cout << endl;
```

```

//вывод количества элементов как расстояние между итераторами
cout << "Vecot contains " << (int)(b.end() - b.begin()) << " elements" << endl;

return 0;

}

```

```

0 1 2
0 1 2
Vecot contains 3 elements

```

При замене

vector<int> b; vector<int>::iterator pb;	на	list<int> b; list<int>::iterator pb;
---	----	---

будут сообщения об ошибках:

- итератор не поддерживает []
- итератор не поддерживает оператор +=
- итератор не поддерживает оператор –

Для корректной работы со списком этот пример надо изменить следующим образом:

```

#include <list>
#include <iostream>

using namespace std;
int main()
{
    list<int> b;

    for(int i = 0; i < 3; i++) b.push_back(i);

    list<int>::iterator pb;

    //итератор поддерживает сравнение != и оператор ++
    for(pb = b.begin(); pb != b.end(); pb++) cout << *pb << " ";
    cout << endl;

    return 0;
}

```

Вопрос (для совместного обсуждения) Почему list не поддерживает итератор произвольного доступа? При обсуждении обратить внимание, что list реализован как двусвязный список.

Вспомогательные функции для работы с итераторами

Предоставляют для любого типа итератора возможности, которыми обладает только итератор произвольного доступа.

Подключить заголовочный файл <iterator>

Функция advance(InputIterator &it, Dist n) смещает итератор it на заданное (n) количество элементов. Dist-тип шаблона, обычно является целочисленным типом.

Выход за пределы контейнера не контролируется.

Сложность: для итераторов произвольного доступа – постоянная
для всех остальных - линейная

Пример 3: Использование функции advance() – вывод произвольных элементов списка

```

#include <list>
#include <iterator>
#include <iostream>

```

```

using namespace std;
int main()
{
    list<int> b;

    for(int i = 0; i < 10; i++) b.push_back(i);

    list<int>::iterator pb = b.begin();

    //вывод первого элемента
    cout << "First element: " <<*pb << endl;

    //перемещение на 3 элемента вперед
    advance(pb, 3);
    cout << "Move forward on 3 elements: " << *pb << endl;

    //перемещение на 1 элемент назад
    advance(pb, -1);
    cout << "Move backward on 1 element: " << *pb << endl;
    return 0;
}

```

```

First element: 0
Move forward on 3 elements: 3
Move backward on 1 element: 2

```

Функция `Dist distance(InputIterator it1, InputIterator it2)` возвращает расстояние между итераторами ввода. Оба итератора должны ссылаться на элементы одного и того же контейнера.

Сложность: для итераторов произвольного доступа – постоянная
для всех остальных – линейная

Пример 4: Использование функции `distance()` – вычисление расстояния от начала списка до элемента с заданным значением.

```

#include <list>
#include <iterator>
#include <algorithm>
#include <iostream>
#include <ctime>

using namespace std;
int main()
{
    list<int> b;
    int val;
    list<int>::iterator pb;

    srand(time(NULL));
    for(int i = 0; i < 10; i++) b.push_back(rand()%10);

    for(pb = b.begin(); pb != b.end(); pb++) cout << *pb << " ";
    cout << endl;

    //ввод значения для поиска
    cout << "Element to find - ";
    cin >> val;

    //поиск элемента с заданным значением
    pb = find(b.begin(), b.end(), val);
    if (pb != b.end())
    {
        //вычисление расстояния от начала списка до найденного элемента
        cout << "Position of element is " << (int)distance(b.begin(), pb) << endl;
    }
    else
    {
        cout << "Element not found" << endl;
    }
}

```

```
    return 0;
}
```

```
1 1 6 1 0 3 2 4 8 8
Element to find - 4
Position of element is 7
```

Совет: Чтобы свободно менять типы контейнеров можно использовать функции `advance()` и `distance()`. Однако, для контейнеров, которые не поддерживают произвольного доступа, использование этих функций может приводить к значительному снижению быстродействия.

Итераторные адаптеры

Итераторные адаптеры - специализированные итераторы, позволяющие выполнять алгоритмы, которые требуют перебор данных в обратном порядке, режим вставки и потоки данных.

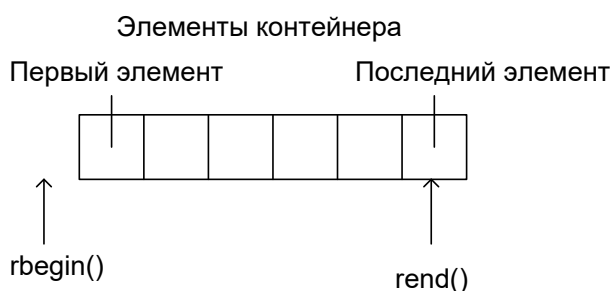
Обратные итераторы (BackwordIterator)

Позволяют выполнять перебор элементов контейнера в обратном направлении. Т.е. оператор `++` смещает итератор к началу контейнера, а оператор `--` к концу.

Для получения обратного итератора каждый контейнер имеет 2 функции:

`rbegin()` – возвращает позицию первого элемента при обратном переборе (т.е. позицию последнего элемента контейнера)

`rend()` – возвращает позицию за последним элементом при обратном переборе (т.е. позицию за первым элементом контейнера)



Пример 5. Вывод элементов списка с помощью прямого и обратного итераторов.

```
#include <list>
#include <iterator>
#include <iostream>

using namespace std;
int main()
{
    list<int> b;
    list<int>::iterator pb;
    list<int>::reverse_iterator rpb;

    for(int i = 0; i < 10; i++) b.push_back(i);

    //вывод элементов в прямом направлении
    cout << "Forward iterator: " << endl;
    for(pb = b.begin(); pb != b.end(); pb++) cout << *pb << " ";
    cout << endl;

    //вывод элементов в обратном направлении (с использованием обратного итератора)
    cout << "Backword iterator: " << endl;
    for(rpb = b.rbegin(); rpb != b.rend(); rpb++) cout << *rpb << " ";
    cout << endl;
    return 0;
}
```

```

Forward iterator:
0 1 2 3 4 5 6 7 8 9
Backward iterator:
9 8 7 6 5 4 3 2 1 0

```

Итераторы вставки (InsertIterator)

Представляют собой итераторный адаптер, который преобразует присвоение нового значения во вставку нового значения. С помощью этого итератора алгоритмы могут вставлять новые значения вместо того чтобы записывать их поверх старых. При необходимости контейнера автоматически увеличивается.

Типы итераторов вставки

Класс итераторы	Описание	Функция	Контейнеры, для которых этот итератор поддерживается
back_insert_iterator	выполняет вставку в конец, использует	push_back()	vector, deque, string, list
front_insert_iterator	выполняет вставку в начало	push_front()	deque, list
insert_iterator	выполняет вставку, начиная с заданной позиции	insert()	все стандартные контейнеры

Пример 5. Использование итератора вставки в конец контейнера:

- для начального заполнения контейнера
- в алгоритме copy()
- использование функции back_inserter() вместо явного создания итератора

```

#include <list>
#include <iterator>
#include <iostream>

using namespace std;
int main()
{
    list<int> b;
    list<int> b1;
    list<int>::iterator pb;

    //создание итератора вставки в конец контейнера
    //в конструктор передается контейнер
    back_insert_iterator<list<int> > bi(b);
    back_insert_iterator<list<int> > bi1(b1);

    //вставка элементов в конец контейнера
    for(int i = 0; i < 10; i++, bi++) *bi = i;

    cout << "Initial list: " << endl;
    for(pb = b.begin(); pb != b.end(); pb++) cout << *pb << " ";
    cout << endl;

    //copy(b.begin(),b.end(),b1.end());
    //Этот вызов приведет к ошибке времени выполнения, т.к. обычный итератор
    //не позволяет расширить контейнер
    //это можно сделать с помощью итератора вставки:
    copy(b.begin(),b.end(),bi1);

    cout << "Copy of list: " << endl;
    for(pb = b1.begin(); pb != b1.end(); pb++) cout << *pb << " ";
    cout << endl;

    //добавление 2 элементов с использованием функции back_inserter
    back_inserter(b1) = 11;
    back_inserter(b1) = 12;
    cout << "Add some new data: " << endl;
    for(pb = b1.begin(); pb != b1.end(); pb++) cout << *pb << " ";

    return 0;
}

```



```
Initial list:
0 1 2 3 4 5 6 7 8 9
Copy of list:
0 1 2 3 4 5 6 7 8 9
Add some new data:
0 1 2 3 4 5 6 7 8 9 11 12
```

Пример 6. Использование итератора вставки в начало

- для начального заполнения контейнера
- в алгоритме copy()
- использование функции front_inserter() вместо явного создания итератора

```
#include <list>
#include <iterator>
#include <iostream>

using namespace std;
int main()
{
    list<int> b;
    list<int> b1;
    list<int>::iterator pb;

    //создание итератора вставки в начало контейнера
    //в конструктор передается контейнер
    front_inserter_iterator<list<int> > bi(b);
    front_inserter_iterator<list<int> > bi1(b1);

    //вставка элементов в начало контейнера
    for(int i = 0; i < 10; i++, bi++) *bi = i;

    cout << "Initial list: " << endl;
    for(pb = b.begin(); pb != b.end(); pb++) cout << *pb << " ";
    cout << endl;

    //копирование данных с помощью итератора вставки:
    copy(b.begin(), b.end(), bi1);

    cout << "Copy of list: " << endl;
    for(pb = b1.begin(); pb != b1.end(); pb++) cout << *pb << " ";
    cout << endl;

    //добавление 2 элементов с использованием функции front_inserter
    front_inserter(b1) = 11;
    front_inserter(b1) = 12;
    cout << "Add some new data: " << endl;
    for(pb = b1.begin(); pb != b1.end(); pb++) cout << *pb << " ";

    return 0;
}
```

```
Initial list:
9 8 7 6 5 4 3 2 1 0
Copy of list:
0 1 2 3 4 5 6 7 8 9
Add some new data:
12 11 0 1 2 3 4 5 6 7 8 9
```

Пример 7. Использование итератора вставки в произвольную позицию:

- для начального заполнения вектора
- для вставки массива поле 2 элемента вектора
- использование функции inserter() вместо явного создания итератора

```
#include <vector>
#include <iterator>
#include <iostream>

using namespace std;
int main()
{
```

```

vector<char> b;
vector<char>::iterator pb;
char ar[] = {'a','b','c'};

//создание итератора вставки в конец контейнера
//в конструктор передается контейнер и начальная позиция вставки
insert_iterator<vector<char> > bi(b,b.begin());

//вставка элементов в контейнер
*bi = 'A'; bi++;
*bi = 'B'; bi++;
*bi = 'C';

cout << "Initial list: " << endl;
for(pb = b.begin(); pb != b.end(); pb++) cout << *pb << " ";
cout << endl;

//копирование данных с помощью итератора вставки:
copy(ar,ar+3,inserter(b,b.begin() + 2));

cout << "After insert: " << endl;
for(pb = b.begin(); pb != b.end(); pb++) cout << *pb << " ";
cout << endl;

//добавление 2 элементов с использованием функции front_inserter
inserter(b,b.begin()) = '1';
inserter(b,b.begin() + 4) = '2';
cout << "Add some new data: " << endl;
for(pb = b.begin(); pb != b.end(); pb++) cout << *pb << " ";
cout << endl;

return 0;
}

```

```

Initial list:
A B C
After insert:
A B a b c C
Add some new data:
1 A B a 2 b c C

```

Потоковые итераторы

Используют поток данных в качестве источника или приемника алгоритма.

Потоковый итератор ввода читает данные из входного потока (клавиатуры, файла). Для получения нового значения используется оператор >>.

```
istream_iterator(istream_type istr)
```

итератор инициализируется потоком вывода

Потоковый итератор вывода записывает данные в выходной поток (на экран, в файл). Для записи нового значения используется оператор <<.

```
ostream_iterator(ostream_type ostr, const char_type* delim)
```

итератор инициализируется выходным потоком ostr и строкой, которая будет использоваться в качестве разделителя.

Пример 8. Использование потокового итератора вывода для вывода данных на экран

```

#include <iterator>
#include <iostream>

using namespace std;
int main()
{
    char a[] = {'a','b','c'};
    //вывод данных на экран с использованием итератора вывода

```

```

//в конструктор передается поток, в который надо
//осуществлять вывод и разделитель

//без разделителя
ostream_iterator<char> out(cout);
cout << "Without delimiter: " << endl;
copy(a,a+3,out);
cout << endl;
//в качестве разделителя используется табуляция
ostream_iterator<char> out1(cout,"\t");
cout << "Without tab delimiter: " << endl;
copy(a,a+3,out1);
return 0;
}

```

```

Without delimiter:
abc
Without tab delimiter:
a      b      c

```

Пример 9. Использование потокового итератора ввода для ввода данных с клавиатуры

```

#include <vector>
#include <iterator>
#include <iostream>

using namespace std;
int main()
{
    vector<int> b;
    cout << "Input: " << endl;

    //итератор чтения из потока
    istream_iterator<int> in(cin);

    //итератор, установленный на конец потока - завершение ввода
    istream_iterator<int> inEnd;

    //ввод продолжается до тех пор, пока не произойдет ошибка,
    //например, при введении символа вместо числа
    copy(in,inEnd,back_inserter(b));

    //вывод данных из вектора
    cout << "Data in vector: " << endl;
    ostream_iterator<int> out(cout," ");
    copy(b.begin(),b.end(),out);
    cout << endl;
    return 0;
}

```

```

Input:
1
2
3
4
Data in vector:
1 2 3

```

Трактовка итераторов

В C++ имеется специальный шаблон структуры, которая позволяет определить трактовки итераторов (iterator traits). Структура содержит наиболее существенную информацию об итераторе и определяет общий интерфейс ко всем определениям типов, связанных с итератором.

```

template<class Iterator>
struct iterator_traits {
    typedef typename Iterator::iterator_category iterator_category; //категория итератора

```

```

typedef typename Iterator::value_type value_type; //тип данных контейнера
typedef typename Iterator::difference_type difference_type; //тип данных разности итераторов
typedef typename Iterator::pointer pointer; //тип данных указателя
typedef typename Iterator::reference reference; //тип данных ссылки
};

```

Эта структура, используется алгоритмами для проверки категории итератора и для выбора наиболее эффективного метода обработки в зависимости от категории итератора.

Для задания категории используются следующие теги:

```

output_iterator_tag – итератор вывода
input_iterator_tag – итератор ввода
forward_iterator_tag – прямой итератор
bidirectional_iterator_tag – двунаправленный итератор
random_access_iterator_tag – итератор произвольного доступа

```

Создание собственного класса итератора.

Как было отмечено ранее, каждый контейнер имеет собственный класс итератора. Следовательно, для разработки итератора сначала надо иметь контейнер, с которым этот итератор будет использоваться.

Создадим простой контейнер, который будет содержать динамический массив объектов произвольного типа.

Краткое описание контейнера: в конструкторе создается массив заданного размера, данные добавляются с помощью метода `push_back()`, при необходимости происходит дополнительное выделение памяти. Имеется метод `GetSize()` для получения размера массива.

Предположим, что для этого контейнера необходимо разработать прямой итератор.

В классе итератора должны быть перегружены следующие операторы:

Название оператора	Обозначение	Пояснение
оператор разыменования	<code>operator *()</code>	Возвращает ссылку на текущий элемент контейнера
оператор инкремента	<code>operator ++()</code>	Перемещает итератор на следующий элемент контейнера
операторы сравнения	<code>operator ==()</code> , <code>operator !=()</code>	Выполняет сравнение итераторов, итераторы равны, если они указывают на один и тот же элемент контейнера

Для использования итератора контейнер имеет методы `begin()` и `end()` для возвращения итераторов, указывающих на первый элемент контейнера и на элемент контейнера, следующий за последним.

Пример 10. Создание собственного итератора

```

#include <iterator>
#include <algorithm>
#include <iostream>

using namespace std;

//Обобщенный класс динамического массива
template <class T>
class DynArray
{
public:
    //класс итератора
    class iterator : public std::iterator<std::forward_iterator_tag,T>
    {
    private:
        //ссылка на контейнер
        DynArray& da;
        //текущая позиция в контейнере
        T* pos;
    public:
        //конструктор: принимает контейнер и текущую позицию
        iterator(DynArray& dal, T* pos1): da(dal),pos(pos1)
        {
        }
        //оператор инкремента: перемещает итератор на следующий элемент контейнера
        //префиксная форма инкремента
        iterator& operator ++( )
        {
            pos++;
            return *this;
        }
    };
};

```

```

    }
    //постфиксная форма инкремента:
    iterator& operator ++(int )
    {
        iterator* tmp = this;
        pos++;
        return *tmp;
    }
    //оператор присвоения
    iterator& operator =(const iterator& it)
    {
        da = it.da;
        pos = it.pos;
        return *this;
    }
    //оператор сравнения на равенство
    bool operator ==(const iterator& it)
    {
        return (pos == it.pos);
    }
    //оператор сравнения на равенство
    bool operator !=(const iterator& it)
    {
        return (pos != it.pos);
    }
    //оператор разыменования: возвращает ссылку на текущий элемент контейнера
    T& operator *()
    {
        return *pos;
    }
};

private:
    T* a; //динамический массив
    int size; //текущий размер массива
    int capacity; //емкость массива
public:
    //конструктор: создание массива заданной емкости
    DynArray(int capacity = 1)
    {
        this->capacity = capacity;
        a = new T[capacity];
        size = 0;
    }
    ~DynArray()
    {
        delete [] a;
    }
    //помещение нового элемента в конец массива,
    //при нехватке места массив расширяется
    void push_back(const T& val)
    {
        if (size >= capacity)
        {
            T* a1 = new T[capacity + 10];
            for(int i = 0; i < capacity; i++) a1[i] = a[i];
            delete[] a;
            a = a1;
        }
        a[size] = val;
        size++;
    }

    //определение текущего размера массива
    int getSize()
    {
        return size;
    }

```

```

//возвращает итератор, указывающий на начало массива
iterator begin()
{
    iterator b(*this, &a[0]);
    return b;
}
//возвращает итератор, указывающий элемент, следующий за концом массива
iterator end()
{
    iterator e(*this, &a[size]);
    return e;
}
};

int main()
{
    //создание динамического массива с начальной емкостью 10 элементов
    DynArray<int> da(10);
    //добавление 3 элементов
    da.push_back(0);
    da.push_back(1);
    da.push_back(2);

    //объявление итератора
    DynArray<int>::iterator it;
    //вывод массива с использованием итератора
    for(it = da.begin(); it != da.end(); it++) cout << *it << " ";

    return 0;
}

```

Хотя разработанный итератор реализует все операторы, которые должны быть реализованы для итератора перехода, он не может быть использован в алгоритмах, работающих с итератором перехода.

Рассмотрим, например, алгоритм `copy()`.

```

template<class InputIterator, class OutputIterator>
OutputIterator copy(
    InputIterator _First,
    InputIterator _Last,
    OutputIterator _DestBeg
);

```

Этот алгоритм предполагает, что диапазон копирования задается с помощью двух прямых итераторов (`_First`, `_Last`). Однако, следующий код

```

ostream_iterator<int> out(cout, " ");
copy(da.begin(), da.end(), out);

```

приведет к ошибке компиляции. Это связано с тем, что в реализации алгоритмов выполняется проверка, является ли переданный итератор объектом заданного класса. Например, для алгоритма `copy()` проверяется, что итераторы `_First`, `_Last` являются объектами класса `forward_iterator_tag` или производного от него. Для обеспечения этого требования надо задать структуру `iterator_traits` для своего итератора. Проще всего это сделать путем наследования своего итератора от базовой шаблонной структуры `iterator`.

```

template<class Category, class Type, class Distance = ptrdiff_t
    class Pointer = Type*, class Reference = Type&>
struct iterator {
    typedef Category iterator_category;
    typedef Type value_type;
    typedef Distance difference_type;
    typedef Pointer pointer;
    typedef Reference reference;
};

```

Как видно из определения этой структуры, она используется для задания типов, которые определяют трактовку итератора. Обязательными параметрами являются категория итератора и тип данных контейнера.

Пример 11. Наследования итератора от базовой структуры iterator

```
class iterator : public std::iterator<std::forward_iterator_tag,T>
//в параметрах шаблона указывается категория итератора: std::forward_iterator_tag и
//тип данных T
{
    ...
}
```

После выполнения наследования можно использовать этот итератор в любых алгоритмах, которые предполагают использование прямого итератора.

Анализ и использование класса string

Цель: изучить возможности класс string и научиться применять его в своих приложениях

Вопросы для актуализации ранее полученных знаний:

8. Как представляется строка в стандарте C?
9. В чем неудобство работы с таким представлением строки?
10. Типы данных char и wchar_t.

Заголовочный файл <string> содержит обобщенный класс контейнер basic_string и некоторые вспомогательные шаблоны.

В C++ поддерживается 2 типа строк:

- Null-terminated массивы символов, которые иначе называются C строки
- Объект обобщенного типа basic_string

Класс basic_string

basic_string – базовый шаблон для всех строковых типов.

```
template <
    class CharType,
    class Traits = char_traits<CharType>,
    class Allocator = allocator<CharType>
>
```

Параметры шаблона:

- CharType – определяет тип данных отдельного символа
 - Traits – описывает основные операции с символами, такие как операции копирования и сравнения символов. Например, можно задать реализацию операции сравнения символов так, чтобы сравнение строк было нечувствительно к регистру.
 - Allocator – определяет модель распределения памяти, используемую строковым классом.
- 2-ой и 3-ий параметры шаблона имеют значения по умолчанию.

В <string> определены следующие специализированные версии класса basic_string:

Typedef	Описание	Пример
typedef basic_string<char> string	Задаёт специализацию шаблона basic_string для типа char.	string s("TEST")
typedef basic_string<char> wstring	Задаёт специализацию шаблона basic_string для типа wchar_t	wstring s(L"TEST")

Преимущества использования: привычный синтаксис операций сравнения и присвоения строк, конкатенации строк, автоматическое управление памятью, в которой хранится строка.

Пример 1: использование типов string и wstring, оператора сравнения ==

```
#include <string>
#include <iostream>
using namespace std;
int main( )
{
    //объявление строки с помощью специализации шаблона
    const basic_string<char> str1("TEST");
    //объявление строки с помощью typedef string
    //эти варианта объявления эквивалентны
    string str2("TEST");
    //вывод строк
    cout<<"String str1: "<<str1<<"\nString str2: "<<str2<<endl;
    //выполнение операции сравнения объектов типа basic_string<char>
    cout<<"Operation: (str1 == str2)"<<endl;
    if(str1 == str2)
        cout<<"Strings str1 & str2 are equal."<<endl;
    else
        cout<<"Strings str1 & str2 are not equal."<<endl;
```



```

//L - используется для объявления UNICODE строк
//объявление строки с помощью специализации шаблона
const basic_string<wchar_t> str3(L"TESTING");
//объявление строки с помощью typedef wstring
//эти варианта объявления эквивалентны
wstring str4(L"JUMPING");
//вывод строк
//поскольку строки UNICODE используется объект потока вывода wcout
wcout<<"\nString str3: " << str3 << " \nString str4: " << str4 <<endl;
//выполнение операции сравнения объектов типа basic_string<wchar_t>
cout<<"Operation: (str3 == str4)"<<endl;
if(str3 == str4)
    cout<<"Strings str3 & str4 are equal."<<endl;
else
    cout<<"Strings str3 & str4 are not equal."<<endl;
return 0;
}

```

```

String str1: TEST
String str2: TEST
Operation: (str1 == str2)
Strings str1 & str2 are equal.

String str3: TESTING
String str4: JUMPING
Operation: (str3 == str4)
Strings str3 & str4 are not equal.

```

Конструкторы класс basic_string

Прототип конструктора	Описание
string ()	Создает пустую строку
string(const char* s)	Создает строку и инициализирует ее значением, строки в стиле C
string(const char* s, size_type len)	Создает строку, инициализированную не более чем len символами строки в стиле C
string(const string& s, size_type pos)	Создает строку, инициализированную символами строки s, начиная с позиции pos
string(const string& s, size_type pos, size_type len)	Создает строку, инициализированную не более чем len символами строки s, начиная с позиции pos
string(size_type n, char c)	Создает строку, состоящую из n элементов, значение каждого элемента равно символу c
string(const_iterator first, const_iterator last)	Создает строку, инициализированную всеми символами интервала [first,end)

Пример 2: использование конструкторов класса basic_string

```

#include <string>
#include <iostream>
using namespace std;
int main()
{
    const char* c_str = "C style string";
    //конструктор, который принимает строку в стиле C
    string s1 = string(c_str);
    cout << s1 << endl;
    //конструктор, который принимает строку в стиле C и количество символов
    string s2 = string(&c_str[2],5);
    cout << s2 << endl;

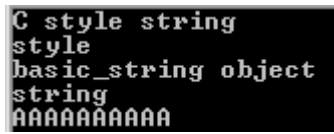
    string s3 = "basic_string object";
    //конструктор, который принимает объект класса basic_string
    string s4 = string(s3);
    cout << s4 << endl;
    //конструктор, который принимает объект класса basic_string,
    //начальный индекс и общее количество символов
    string s5 = string(s3,6,6);
}

```

```

cout << s5 << endl;
//конструктор, который принимает количество символов и значение символа
string s6 = string(10, 'A');
cout << s6 << endl;
return 0;
}

```



```

C style string
style
basic_string object
string
AAAAAAAAAAAA

```

Операторы класса basic_string

Класс basic_string содержит ряд перегруженных операторов, что позволяет использовать объекты класса в выражениях с привычным синтаксисом.

Перегруженные операторы:

- сравнение: !=, ==, <, <=, >, >=
- запись в поток <<
- чтение из потока >>
- конкатенация +
- обращение к символу по индексу []

Пример 3: использование операторов == и != для сравнения объектов типа basic_string<char> между собой и со строками в стиле C.

```

#include <string>
#include <iostream>
using namespace std;
int main( )
{
    //Объявление объектов типа basic_string<char>
    string str1("red");
    string str2("blue");
    cout<<"str1 string is = "<<str1<<endl;
    cout<<"str2 string is = "<<str2<<endl;
    //Объявление строки в стиле C
    char *str3 = "red";
    cout<<"C-style str3 string is = "<<str3<<endl;

    //Оператор !=
    //слева и справа от оператора != объекты basic_string
    cout<<"Operation: (str1 != str2) Result: "<<boolalpha<<(str1 != str2)<<endl;

    //слева - строка в стиле C, справа - объект типа basic_string
    cout<<"Operation: (str3 != str2) Result: "<<boolalpha<<(str3 != str2)<<endl;

    //слева - объект типа basic_string, справа - строка в стиле C
    cout<<"Operation: (str1 != str3) Result: "<< boolalpha<<(str1 != str3)<<endl;

    //Оператор ==
    //слева и справа от оператора == объекты basic_string
    cout<<"Operation: (str1 == str2) Result: "<<boolalpha<<(str1 == str2)<<endl;

    //слева - строка в стиле C, справа - объект типа basic_string
    cout<<"Operation: (str3 == str2) Result: "<<boolalpha<<(str3 == str2)<<endl;

    //слева - объект типа basic_string, справа - строка в стиле C
    cout << "Operation: (str1 == str3) Result: "<<boolalpha<<(str1 == str3)<<endl;
    return 0;
}

```

```

str1 string is = red
str2 string is = blue
C-style str3 string is = red
Operation: (str1 != str2) Result: true
Operation: (str3 != str2) Result: true
Operation: (str1 != str3) Result: false
Operation: (str1 == str2) Result: false
Operation: (str3 == str2) Result: false
Operation: (str1 == str3) Result: true
Для продолжения нажмите любую клавишу . . . _

```

Вопрос: Как перегружены операторы == и !=, если слева от оператора может стоять не объект типа `base_string`, а строка в стиле C?

Пример 4: использование операторов <, >, <=, >=

```

#include <string>
#include <iostream>
using namespace std;
int main()
{
    //Объявление объектов типа basic_string<char>
    string str1("testingthree");
    string str2("testingtwo");
    string str3("testingtwo");
    cout<<"str1 is = "<<str1<<endl;
    cout<<"str2 is = "<<str2<<endl;
    cout<<"str3 is = "<<str3<<endl;

    //Операторы <, >, <=, >=
    cout<<"Operation: (str1 < str2) Result: "<<boolalpha<<(str1 < str2)<<endl;
    cout<<"Operation: (str1 > str2) Result: "<<boolalpha<<(str1 > str2)<<endl;
    cout<<"Operation: (str2 > str3) Result: "<<boolalpha<<(str2 > str3)<<endl;
    cout<<"Operation: (str2 < str3) Result: "<<boolalpha<<(str2 < str3)<<endl;
    cout<<"Operation: (str2 >= str3) Result: "<<boolalpha<<(str2 >= str3)<<endl;
    cout<<"Operation: (str2 <= str3) Result: "<<boolalpha<<(str2 <= str3)<<endl;

    return 0;
}

```

```

str1 is = testingthree
str2 is = testingtwo
str3 is = testingtwo
Operation: (str1 < str2) Result: true
Operation: (str1 > str2) Result: false
Operation: (str2 > str3) Result: false
Operation: (str2 < str3) Result: false
Operation: (str2 >= str3) Result: true
Operation: (str2 <= str3) Result: true

```

Пример 5: использование операторов вывода в поток, чтения из потока

Для чтения строки можно использовать функцию `getline()`, в которую можно передать символ – признак конца строки

```

#include <string>
#include <iostream>
using namespace std;
int main()
{
    string Sample = "Testing the << and >> operators.";
    string Var1, Var2;
    cout<<Sample<<endl;
    cout<<"Enter a string or a word: ";
    getline(cin,Var1,'\n');
    cout<<"Enter another string or a word: ";
    cin>>Var2;
    cout<<"The strings entered are: "<<Var1<<" and "<<Var2<<endl;
    return 0;
}

```

```
Testing the << and >> operators.
Enter a string or a word: test1
Enter another string or a word: test2
The strings entered are: test1 and test2
```

Пример 6: конкатенация строк объектов типа `basic_string<char>` и строк в стиле C.

```
#include <string>
#include <iostream>
using namespace std;
int main()
{
    //Объявление объектов типа basic_string<char>
    string str1("StringOne");
    string str2("StringTwo");
    //Объявление строки в стиле C
    char *str3 = "StringThree";
    //Объявление символа
    char chr = '*';
    cout<<"str1 string is = "<<str1<<endl;
    cout<<"str2 string is = "<<str2<<endl;
    cout<<"str3 C-style string is = "<<str3<<endl;
    cout<<"A character constant chr is = "<<chr<<endl;

    //Конкатенация строк - объектов типа basic_string<char>
    cout<<"\nOperation: str12 = str1 + str2"<<endl;
    string str12 = str1 + str2;
    cout<<"str12 = "<<str12<<endl;
    //Конкатенация строки - объекта типа basic_string<char> со строкой в стиле C
    cout<<"\nOperation: str13 = str1 + str3"<<endl;
    string str13 = str1 + str3;
    cout<<"str13 = "<<str13<<endl;
    //Конкатенация строки - объекта типа basic_string<char> с символьной константой
    cout<<"\nOperation: str13chr = str13 + chr"<<endl;
    string str13chr = str13 + chr;
    cout<<"str13chr = "<<str13chr<<endl;
    return 0;
}
```

```
str1 string is = StringOne
str2 string is = StringTwo
str3 C-style string is = StringThree
A character constant chr is = *

Operation: str12 = str1 + str2
str12 = StringOneStringTwo

Operation: str13 = str1 + str3
str13 = StringOneStringThree

Operation: str13chr = str13 + chr
str13chr = StringOneStringThree*
```

Из приведенных примеров видно, что строки в стиле C могут использоваться практически в любых действиях со строками `basic_string` (например, в операциях сравнения, присвоения, конкатенации). В частности, поддерживается автоматическое преобразование типа `const char*` в строку `basic_string`.

Преобразование строки `basic_string` в строку в стиле C

Автоматическое преобразование строки `basic_string` в строку в стиле C. Возможность этого преобразования исключена, чтобы избежать непреднамеренного преобразования, которое в некоторых случаях может привести к странным последствиям.

Функции преобразования строки `basic_string` в строку в стиле C или в массив символов.

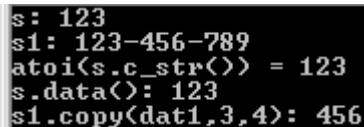
<code>data()</code>	Возвращает содержимое строки в виде массива символов, <code>/0</code> не дописывается.
<code>c_str()</code>	Возвращает содержимое строки в формате C строки, т.е. с завершающим <code>/0</code> .
<code>copy()</code>	Копирует содержимое строки в символьный массив, переданный при вызове, <code>/0</code> не дописывается.

Пример 7: преобразование строки `basic_string` в массив символов.

```
#include <string>
#include <iostream>
using namespace std;
int main()
{
    string s("123");
    string s1("123-456-789");
    cout << "s: " << s << endl;
    cout << "s1: " << s1 << endl;

    //преобразование в строку в стиле C для передачи в функцию
    cout << "atoi(s.c_str()) = " << atoi(s.c_str()) << endl;
    //получение массива символов с помощью метода data
    const char* dat = s.data();
    cout << "s.data(): ";
    for(int i = 0; i < 3; i++) cout << dat[i];
    cout << endl;

    char dat1[3];
    //получение массива символов с помощью метода copy
    //копирует 3 символа, начиная с 4-ого
    s1.copy(dat1, 3, 4);
    cout << "s1.copy(dat1, 3, 4): ";
    for(int i = 0; i < 3; i++) cout << dat1[i];
    cout << endl;
    return 0;
}
```



```
s: 123
s1: 123-456-789
atoi(s.c_str()) = 123
s.data(): 123
s1.copy(dat1, 3, 4): 456
```

Совет: в программе работать с строками `basic_string`, преобразование в C строку выполнять только тогда, когда потребуется содержимое строки в виде массива `char*` (например, для передачи в функцию).

Размер и емкость строки

Функция	Описание
<code>size(), length()</code>	Возвращают количество символов в строке.
<code>max_size()</code>	Возвращает максимальное количество символов, которое может содержаться в строке. Равно максимальное значение типа индекса -1.
<code>capacity()</code>	Возвращает количество символов, которое можно сохранить в строке без выделения дополнительной памяти.
<code>reserve()</code>	Резервирует память для хранения строки.

Пример8: определение размера и емкости строки

```
#include <string>
#include <iostream>
using namespace std;
int main()
{
    string s("How long this string?");
    cout << "s.length() = " << s.length() << endl;
    cout << "s.size() = " << s.size() << endl;
    cout << "s.max_size() = " << hex << s.max_size() << endl;
    cout << "s.capacity() = " << dec << s.capacity() << endl;
    s.reserve(100);
    cout << "After reservation" << endl;
    cout << "s.capacity() = " << s.capacity() << endl;
    return 0;
}
```

```
}
```

```
s.length() = 21  
s.size() = 21  
s.max_size() = ffffffff  
s.capacity() = 31  
After reservation  
s.capacity() = 111
```

Совет: для увеличения быстродействия заранее выполнять резервирование памяти по максимально возможному размеру строки.

Обращение к символам

Символы, содержащиеся в строке можно читать и записывать. Для этого можно использовать оператор [] или функцию at().

Отличия:

- при вызове at() с недопустимым индексом – исключение out_of_range
- при вызове оператора [] с недопустимым индексом – непредсказуемые последствия, как при работе с массивом.

Пример 9: обращение к символам строки по индексу

```
#include <string>  
#include <iostream>  
#include <exception>  
using namespace std;  
int main()  
{  
    string s("ABCDE");  
    //вывод 3-ого символа строки  
    cout << "operator[](2): " << s[2] << endl;  
    cout << "s.at(2): " << s.at(2) << endl;  
    //генерация исключения при выходе за пределы строки  
    try  
    {  
        cout << "try s.at(100): ";  
        cout << s.at(100) << endl;  
    }  
    catch(out_of_range e)  
    {  
        cout << "exception: " << e.what() << endl;  
    }  
    //изменение 1-ого символа строки  
    s[0] = '1';  
    cout << s << endl;  
  
    return 0;  
}
```

```
operator[](2): C  
s.at(2): C  
try s.at(100): exception: invalid string position  
1BCDE
```

Модификация строк

Присвоение строке нового значения

Оператор = может использоваться для присвоения строки в стиле C, строки basic_string, символа

Функция assign() используется, если новое значение описывает несколько аргументов (например, задается диапазон строки).

Пример 10: Присвоение строке начального значения различными способами

```

#include <string>
#include <iostream>
using namespace std;
int main()
{
    string str1;
    const char *str2 = "StRiNg assign()";
    cout<<"str2, C string is: "<<str2<<endl;
    //присвоение C строки
    str1.assign(str2);
    cout<<"Operation: str1.assign(str2) Result: "<<str1<<"\n\n";

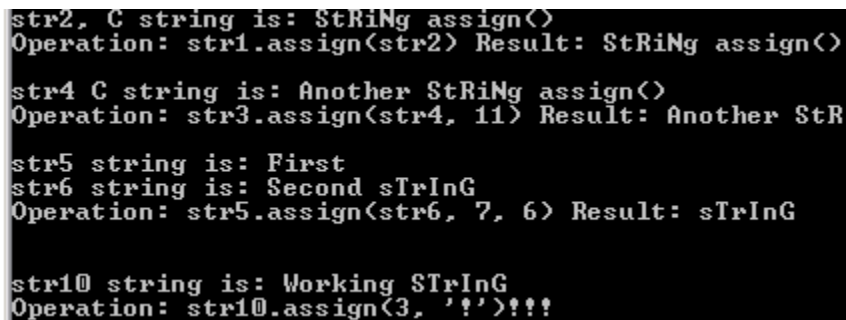
    //присвоение 11 символов из C строки
    string str3;
    const char *str4 = "Another StRiNg assign()";
    cout<<"str4 C string is: "<<str4<<endl;
    str3.assign(str4, 11);
    cout<<"Operation: str3.assign(str4, 11) Result: "<<str3<<"\n\n";

    //присвоени 6 символов строки basic_string, начиная с 7-ой позиции
    string str5("First "), str6("Second sTrInG");
    cout<<"str5 string is: "<<str5<<endl;
    cout<<"str6 string is: "<<str6<<endl;
    str5.assign(str6, 7, 6);
    cout<<"Operation: str5.assign(str6, 7, 6) Result: "<<str5<<"\n\n";

    //Присвоение 3 символов !
    string str10("Working STrInG");
    cout<<"\nstr10 string is: "<<str10<<endl;
    str10.assign(3, '!');
    cout<<"Operation: str10.assign(3, '!')"<<str10<<endl;

    return 0;
}

```



```

str2, C string is: StRiNg assign()
Operation: str1.assign(str2) Result: StRiNg assign()

str4 C string is: Another StRiNg assign()
Operation: str3.assign(str4, 11) Result: Another StR

str5 string is: First
str6 string is: Second sTrInG
Operation: str5.assign(str6, 7, 6) Result: sTrInG

str10 string is: Working STrInG
Operation: str10.assign(3, '!')!!!

```

Обмен значениями

Функция swap() меняет местами содержимое 2 строк

Пример 11: использование функции swap()

```

#include <string>
#include <iostream>

using namespace std;
int main()
{
    string str1("StringOne");
    string str2("StringTwo");
    cout<<"Before swapping string str1 and str2:"<<endl;
    cout<<"str1 string is = "<<str1<<endl;
    cout<<"str2 string is = "<<str2<<endl;
    //обмен значений строк с помощью функции swap
    swap(str1, str2);
    cout<<"\nOperation: swap(str1, str2)"<<endl;
}

```

```

cout<<"After swapping string str1 and str2:"<<endl;
cout<<"str1 string is = "<<str1<<endl;
cout<<"str2 string is = "<<str2<<endl;
return 0;
}

```

```

Before swapping string str1 and str2:
str1 string is = StringOne
str2 string is = StringTwo

Operation: swap(str1, str2)
After swapping string str1 and str2:
str1 string is = StringTwo
str2 string is = StringOne

```

Очистка строки

Метод clear() – очищает содержимое строки

Метод erase() – удаляет заданный диапазон символов

Пример 12: очистка строки, удаление диапазона символов

```

#include <string>
#include <iostream>

using namespace std;
int main()
{
    //Declaring an object of type basic_string<char>
    string str1("StringOne");
    string str2("StringTwo");
    cout<<"str1 string is = "<<str1<<endl;
    cout<<"str2 string is = "<<str2<<endl;
    //очистка строки с помощью clear
    str1.clear();
    cout << "After str1.clear() " << str1 << endl;
    //удаление всех символов, начиная с 6
    str2.erase(6);
    cout << "After str2.erase(6) " << str2 << endl;
    //очистка строки с помощью erase
    str2.erase();
    cout << "After str2.erase() " << str2 << endl;

    return 0;
}

```

```

str1 string is = StringOne
str2 string is = StringTwo
After str1.clear()
After str2.erase(6) String
After str2.erase()

```

Вставка и замена символов

Присоединение символов в конце строки выполняется:

- оператором += - добавляемое значение определяется одним аргументом
- функцией append() – добавляемое значение определяется несколькими аргументом
- функцией push_back() – добавляет 1 символ

Пример 13: вставка символов в конец строки

```

#include <string>
#include <iostream>

using namespace std;
int main()
{
    string str1("Playing ");
    const char *str2 = "with a simple string";
    cout<<"str1: "<<str1<<endl;
}

```



```

cout<<"str2: "<<str2<<endl;
//добавление всей строки str2 в конец строки str1
str1.append(str2);
cout<<"Operation: str1.append(str2)\nResult: "<<str1<<"\n\n";

string str3 ("Travel ");
cout<<"str3: "<<str3<<endl;
//добавление 4 символов строки str2 в конец строки str3
str3.append(str2, 4);
cout<<"Operation: str3.append(str2, 4)\nResult: "<<str3<<"\n\n";

string str4 ("It is ");
cout<<"str4: "<<str4<<endl;
//добавление 6 символов строки str2, начиная с 14-ого, в конец строки str3
str4.append(str2, 14, 6);
cout<<"Operation: str4.append(str2, 14, 6)\nResult: "<<str4<<"\n\n";

//добавление 1 символа с помощью push_back
str4.push_back('!');
cout<<"Operation: str4.push_back('!')\nResult: "<<str4<<"\n\n";

return 0;
}

```

```

str1: Playing
str2: with a simple string
Operation: str1.append(str2)
Result: Playing with a simple string

str3: Travel
Operation: str3.append(str2, 4)
Result: Travel with

str4: It is
Operation: str4.append(str2, 14, 6)
Result: It is string

Operation: str4.push_back('!')
Result: It is string!

```

Для вставки в середину строки используется метод `insert()`, ему первым параметром передается индекс символа, за которым вставляются новые символы.

Для замены символов используется метод `replace()`, ему первым параметром передается индекс символа, начиная с которого выполнять замену, вторым параметром – сколько символов заменять.

Пример 14: вставка символов в заданную позицию строки и замены

```

#include <string>
#include <iostream>

using namespace std;
int main()
{
    string s1 = "one two";
    string s3 = "three";
    string s4 = "four";
    string s5 = "five";
    //вставки строки s3 после 4-ого символа строки s1
    s1.insert(4,s3);
    cout << "s1.insert(4,s3) Result: " << s1 << endl;
    //замена 5-ти символов строки s1, начиная с 4-ого, строкой s4
    s1.replace(4,5,s4);
    cout << "s1.replace(3,5,s4) Result: " << s1 << endl;
    //замена 3-ех символов строки s1, начиная с 0-ого, строкой s5
    s1.replace(0,3,s5);
    cout << "s1.replace(8,3,s5) Result: " << s1 << endl;
    return 0;
}

```

```
s1.insert(4,s3) Result: one three two
s1.replace(3,5,s4) Result: one four two
s1.replace(8,3,s5) Result: five four two
```

Выделение подстроки

Функция `substr()` выделяет подстроку:

- всю исходную строку
- начиная с заданного индекса
- начиная с заданного индекса, включая заданное количество символов

Пример 15: Выделение подстроки

```
#include <string>
#include <iostream>

using namespace std;
int main()
{
    //выделение подстроки
    string s1 = "one string in another string";
    //вся строка
    cout<< "s1.substr(): " << s1.substr() << endl;
    //часть строки, начиная с 14 элемента
    cout<< "s1.substr(14): " << s1.substr(14) << endl;
    //6 символов строки, начиная с 4 элемента
    cout<< "s1.substr(4,6): " << s1.substr(4,6) << endl;
    return 0;
}
```

```
s1.substr(): one string in another string
s1.substr(14): another string
s1.substr(4,6): string
```

Поиск

Возможные варианты поиска:

- поиск отдельных символов
- поиск подстрок
- поиск в прямом и обратном направлении
- поиск, начиная с заданного индекса

Функции:

`find()` – поиск первого вхождения заданного значения

`rfind()` – поиск последнего вхождения заданного значения

`find_first_of()` – поиск первого символа, из входящих в заданное значение

`find_last_of()` – поиск последнего символа, из входящих в заданное значение

Если значение найдено, возвращается индекс первого символа, иначе возвращается значение `pros` (его численное значение = -1). Однако, значение `pros` имеет тип `size_type` (который был указан при специализации шаблона) – это беззнаковый тип, поэтому `pros` является максимальным беззнаковым значением данного типа. Поэтому результат поисковой функции лучше напрямую сравнивать с `pros`.

Пример 16: поиск символа и подстроки в строке

```
#include <string>
#include <iostream>

using namespace std;
int main()
{
    string s = "I am here. Try to find me.";
    //поиск первого вхождения символа e
    cout << "s.find('e'): " << s.find('e') << endl;
    //поиск последнего вхождения символа e
    cout << "s.rfind('e'): " << s.rfind('e') << endl;
    //поиск первого вхождения символа e после 10 индекса
    cout << "s.find('e',10): " << s.find('e',10) << endl;
    //поиск первого вхождения подстроки 'am'
    cout << "s.find('am'): " << s.find("am") << endl;
}
```

```
//поиск первого вхождения любого символа из подстроки 'he'
cout << "s.find_first_of('he')" << s.find_first_of("he") << endl;
//поиск последнего вхождения любого символа из подстроки 'he'
cout << "s.find_last_of('he')" << s.find_last_of("he") << endl;
//поиск символа, который в строке не содержится
cout << "s.find('F')" << s.find('F') << endl;
//проверка возвращаемого значения
if (s.find('F') == string::npos) cout << "Symbol not found" << endl;
return 0;
}
```

```
s.find('e'): 6
s.rfind('e'): 24
s.find('e',10): 24
s.find('am'): 2
s.find_first_of('he')5
s.find_last_of('he')24
s.find('F')4294967295
Symbol not found
```

Анализ и использование классов vector, list, map, multimap

Цель:

- познакомиться с последовательными и ассоциативными контейнерами
- изучить контейнер vector
- изучить контейнер list
- изучить контейнер map
- изучить контейнер multimap

Вопросы для актуализации ранее полученных знаний:

1. Определение контейнера
2. Анализ использования массива в качестве контейнера

Концепции контейнеров

Последовательные контейнеры – каждый элемент имеет определенную позицию (индекс). Контейнер представляет собой линейную последовательность элементов. Поскольку элементы расположены в определенном порядке, возможны такие операции, как вставка в определенную позицию, удаление заданного диапазона.

К последовательным контейнерам относятся: vector, list, queue, deque, priority_queue, stack

ContType – имя класса контейнера

Ассоциативные контейнеры – сортированные коллекции, в которых позиция элемента зависит от его значения (или значения ключа). Обеспечивают быстрый поиск элементов. Вставка элемента в определенную позицию невозможна.

К ассоциативным контейнерам относятся: set, multiset, map, multimap

Ассоциативные контейнеры делятся на:

- контейнеры с уникальным значением ключей (set, map)
- контейнеры с неуникальным значением ключей (multiset, multimap)

Операции, определенные для всех типов контейнеров

Выражение	Комментарий
Конструкторы	
ContType c	Создает пустой контейнер
ContType c(c1)	Создает копию контейнера c1
ContType c(n,t)	Создает контейнер, содержащий n копий значения t
ContType(beg,end)	Создает контейнер и записывает в него элементы диапазона [beg,end)
Определение размеров контейнера	
size()	Текущее количество элементов в контейнере
max_size()	Максимальное количество элементов, которое можно разместить в контейнере
empty()	возвращает true, если в контейнере нет элементов
Получение итератора	
begin()	Возвращает итератор, указывающий на начало контейнера
end()	Возвращает итератор, указывающий на конец контейнера
rbegin()	Возвращает обратный итератор, указывающий первый элемент для обратной итерации
rend()	Возвращает обратный итератор, указывающий последний элемент для обратной итерации
Вставка, удаление элементов	
c.insert(it,t)	Вставляет t перед элементом, на который указывает итератор it
c.insert(it, beg,end)	Вставляет элементы диапазона [beg,end) перед элементом, на который указывает итератор it
c.erase(it)	Удаляет элемент, на который указывает итератор it
c.erase(beg,end)	Удаляет все элементы диапазона [beg,end)
c.clear()	Удаляет все элементы контейнера

Понятие сложности алгоритма

Сложность алгоритма определяется временем его выполнения.

Три уровня сложности:

- время компиляции – действие выполняется на этапе компиляции и во время выполнения программы времени на выполнение не требуется;
- постоянная сложность – время выполнения не зависит от размера контейнера;
- линейная сложность – время выполнения линейно зависит от количества элементов в контейнере;

Например:

вставка в произвольную позицию списка – постоянная сложность (т.к. необходимо только «перенастроить» указатели 2 элементов, между которыми выполняется вставка)

вставка в произвольную позицию вектора – линейная сложность (т.к. необходимо «сдвинуть» все следующие элементы)

Требование к уровню сложности – одна из характеристик STL, т.е. для каждого действия (алгоритма или метода контейнера) указывается уровень сложности.

Элементы контейнера

Для того чтобы объекты класса можно было сохранять в контейнере в классе должны быть предусмотрены (с уровнем доступом public):

- конструктор по умолчанию
- конструктор копирования

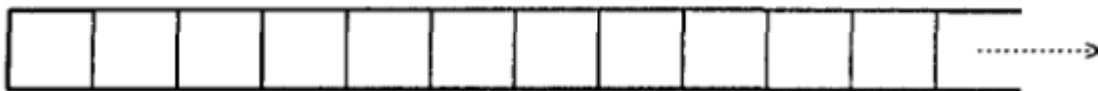
При вставке объекта в контейнер создается его копия, которая и сохраняется в контейнере. Если копирование объекта нежелательно, можно в контейнере вместо самих объектов сохранять указатели.

Если в контейнере хранятся значения элементов, при удалении контейнера вызываются деструкторы для всех элементов контейнера.

Если в контейнере хранятся указатели на объекты, при удалении контейнера деструкторы объектов не вызываются.

Класс vector

Класс vector обеспечивает хранение данных в динамическом массиве. Класс осуществляет управление памятью: память автоматически увеличивается при добавлении новых элементов.



Структура вектора

Вектор обеспечивает произвольный доступ к элементам, т.е. обращение к элементу по индексу выполняется с постоянным временем. Итераторы вектора являются итераторами произвольного доступа, что позволяет использовать этот класс во всех алгоритмах STL.

Операция вставки (удаления) в конец выполняется за постоянное время, операция вставки (удаления) в начало и в середину имеют линейную сложность. После вставки (удаления) в начало или середину вектора ссылки и указатели ссылки, указатели и итераторы, указывающие на другие элементы вектора, становятся недействительными.

Для использования включить заголовочный файл: `#include <vector>`

Объявление класса вектор:

```
template < class Type, class Allocator = allocator<Type> >
class vector
```

Type – тип данных в векторе

Allocator – определяет способ выделения памяти

Пример 1. Различные способы инициализации вектора

```
#include <iostream>
#include <vector>
#include <iterator>
using namespace std;

int main()
{
    //создание вектора, содержащего 3 элемента со значением 0
    vector<int> a(3, 0);
    int ar[4] = {1,2,3,4};
    //создание вектора, содержащего 3 первых элемента массива
    vector<int> b(ar, ar+3);
    //создание копии вектора b
```

```

vector<int> c(b);

//вывод векторов
ostream_iterator<int, char> out(cout, " ");
cout << "Vector a: "; copy(a.begin(), a.end(), out); cout << endl;
cout << "Vector b: "; copy(b.begin(), b.end(), out); cout << endl;
cout << "Vector c: "; copy(c.begin(), c.end(), out); cout << endl;
return 0;
}

```

```

Vector a: 0 0 0
Vector b: 1 2 3
Vector c: 1 2 3

```

Если при вызове конструктора вектора указывается начальное количество элементов, все эти элементы создаются (вызывается конструктор по умолчанию для каждого элемента).

Добавление элементов

void push_back(T val) добавляет элемент со значением val в конец вектора

iterator insert(iterator it, T val) добавляет элемент со значением val перед итератором it

void insert(iterator beg, iterator end) добавляет диапазон элемент [beg, end) перед итератором it

Пример 2. Вставка данных в вектор

```

#include <iostream>
#include <vector>
#include <iterator>
using namespace std;

int main()
{
    //создание пустого вектора
    vector<int> a;
    //заполнение вектора элементами
    for(int i = 0; i < 5; i++)
        a.push_back(i);

    //вывод векторов с использованием итераторов
    vector<int>::iterator it;
    cout << "Initial state: " << endl;
    for(it = a.begin(); it != a.end(); it++)
        cout << *it << " ";
    cout << endl;

    //вставка перед вторым элементом числа 10
    a.insert(a.begin() + 2, 10);
    cout << "After insert 10: " << endl;
    for(it = a.begin(); it != a.end(); it++)
        cout << *it << " ";
    cout << endl;

    //вставка в начало вектора диапазона значений из массива
    int ar[5] = {20, 21, 22, 23, 24};
    a.insert(a.begin(), ar, ar + 4);
    cout << "After insert range: " << endl;
    for(it = a.begin(); it != a.end(); it++)
        cout << *it << " ";
    cout << endl;

    return 0;
}

```

```
Initial state:
0 1 2 3 4
After insert 10:
0 1 10 2 3 4
After insert range:
20 21 22 23 0 1 10 2 3 4
```

Резервирование места и определение емкости

size_type capacity() – количество элементов, которое можно разместить в векторе без перераспределения памяти

void reserve(size_type col) – резервирует память для хранения элементов вектора.

col – количество элементов, на которое резервируется память

void resize(size_type col) – изменяет размер вектора, если размер увеличивается новые элементы заполняются значением по умолчанию, если размер уменьшается, удаляются элементы с конца вектора

col- новое количество элементов в векторе

Пример 3. Демонстрация отличий методов resize() и reserve()

```
#include <iostream>
#include <vector>
#include <iterator>
using namespace std;

int main()
{
    int ar[4] = {1,2,3,4};
    //создание вектора, содержащего 3 первых элемента массива
    vector<int> a(ar, ar+3);

    //вывод векторов
    ostream_iterator<int, char> out(cout, " ");
    cout << "Vector a: "; copy(a.begin(), a.end(), out); cout << endl;
    //вывод размера и емкости вектора
    cout<<"Vector size: "<<a.size()<<" Vector capacity: "<<a.capacity()<< endl;

    //изменение размера вектора
    a.resize(4);
    cout << "Vector a: "; copy(a.begin(), a.end(), out); cout << endl;
    cout<<"Vector size: "<<a.size()<<" Vector capacity: "<<a.capacity()<< endl;

    //выполнение резервирования
    a.reserve(5);
    cout << "Vector a: "; copy(a.begin(), a.end(), out); cout << endl;
    cout<<"Vector size: "<<a.size()<<" Vector capacity: "<<a.capacity()<< endl;

    return 0;
}
```

```
Vector a: 1 2 3
Vector size: 3 Vector capacity: 3
Vector a: 1 2 3 0
Vector size: 4 Vector capacity: 4
Vector a: 1 2 3 0
Vector size: 4 Vector capacity: 5
```

Как видно из полученных результатов, при выполнении resize() добавляется еще один элемент со значением по умолчанию (0), при выполнении reserve() размер вектора не меняется, а емкость увеличивается.

Совет: Для увеличения быстродействия следует выполнять резервирование перед вставкой в вектор значительных объемов данных.

Удаление элементов

void pop_back() удаление элемента в конце вектора

iterator erase(iterator it) удаляет элемент в позиции итератора it

iterator erase(iterator beg, iterator end) удаляет все элементы в диапазоне [beg,end)

Пример 4. Удаление элементов из вектора

```
#include <iostream>
```

```

#include <vector>
#include <iterator>
using namespace std;

int main()
{
    //создание пустого вектора
    vector<int> a;
    //заполнение вектора элементами
    for(int i = 0; i < 10; i++)
        a.push_back(i);

    //вывод векторов
    ostream_iterator<int, char> out(cout, " ");
    cout << "Initial state: "; copy(a.begin(), a.end(), out); cout << endl;

    //удаление последнего элемента вектора
    a.pop_back();
    cout << "After delete last element: "; copy(a.begin(), a.end(), out);
    cout << endl;

    //удаление 2-ого элемента
    a.erase(a.begin() + 1);
    cout << "After delete 2-nd element: "; copy(a.begin(), a.end(), out);
    cout << endl;

    //удаление диапазона из 3-х последних элементов
    a.erase(a.end()-3, a.end());
    cout << "After delete three last element: "; copy(a.begin(), a.end(), out);
    cout << endl;

    return 0;
}

```

```

Initial state: 0 1 2 3 4 5 6 7 8 9
After delete last element: 0 1 2 3 4 5 6 7 8
After delete 2-nd element: 0 2 3 4 5 6 7 8
After delete three last element: 0 2 3 4 5

```

Обращение к элементу вектора

T& operator [](size_type index) возвращает ссылку на элемент в указанной позиции, не контролирует выход позиции за границы массива

T& at(size_type index) возвращает ссылку на элемент в указанной позиции, при выходе позиции за границы массива генерирует исключение.

Пример 5. Обращение к элементам вектора

```

#include <iostream>
#include <vector>
#include <exception>
using namespace std;

int main()
{
    //создание пустого вектора
    vector<int> a;
    //заполнение вектора элементами
    for(int i = 0; i < 10; i++)
        a.push_back(i);

    //вывод векторов с помощью индексов
    for(int i = 0; i < a.size(); i++)
        cout << a[i] << " ";
    cout << endl;

    //обращение по недопустимому индексу
    try
    {
        //cout << a[11] << endl; //эта строка приводит к возникновению ошибки
    }
}

```



```

//времени выполнения
cout << a.at(11) << endl; //эта строка приводит к возникновению
                           // исключения, которое перехватывается обработчиком
}
catch(out_of_range e)
{
    cout << e.what() << endl;
}
return 0;
}

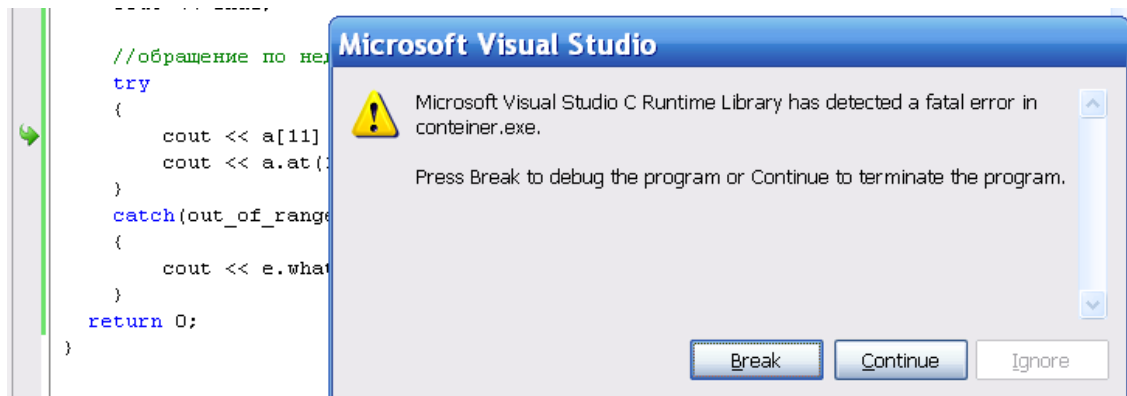
```

```

0 1 2 3 4 5 6 7 8 9
invalid vector<T> subscript

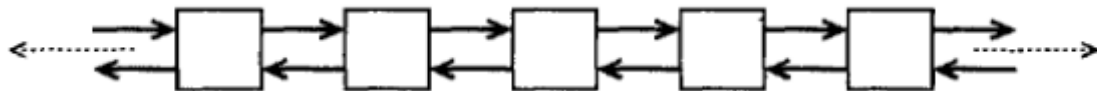
```

Вызов `cout << a.at(11) << endl` генерирует исключение, которое перехватывается обработчиком. Если выполнить команду `cout << a[11] << endl`, будет следующая ошибка времени выполнения:



Класс List

Класс List обеспечивает хранение данных в виде двусвязного списка.



Структура списка

Список не поддерживает произвольный доступ к элементам, т.к. для доступа к элементу надо перебрать все предшествующие элементы, т.е. эта операция имеет линейную сложность. Список обеспечивает вставку (удаление) элементов в любой позиции за постоянное время (поскольку не требует перемещения других элементов). При вставке (удалении) ссылки, указатели и итераторы, связанные с другими элементами списка, остаются действительными.

Итераторы списка являются двунаправленными, т.е. списки нельзя использовать в алгоритмах, требующих итераторы произвольного доступа. Класс List содержит специализированные функции для перемещения элементов, эти функции являются оптимизированными версиями соответствующих алгоритмов.

Для использования включить заголовочный файл: `#include <list>`

Объявление класса списка:

```

template < class Type, class Allocator = allocator<Type> >
class list

```

Type – тип данных в векторе

Allocator – определяет способ выделения памяти

Поскольку список не поддерживает произвольного доступа, для него не определены оператор `[]` (int) и функция `at(int)`.

Добавление элементов

Список поддерживает те же функции добавления элементов, что и вектор, а также функцию вставки в начало списка: `push_front(T val)`.

Пример 6. Добавление элементов в список.

```

#include <iostream>

```

```

#include <list>
using namespace std;

int main()
{
    //создание пустого списка
    list<int> a;
    list<int>::iterator it;

    //вставка элементов в конец списка
    for(int i = 0; i < 10; i++) a.push_back(i);

    //вывод списка с помощью итератора
    for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
    cout << endl;
    a.clear();

    //вставка элементов в начало списка
    for(int i = 0; i < 10; i++) a.push_front(i);
    //вывод списка с помощью итератора
    for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
    cout << endl;

    //вставка элемента после 1-ого элемента списка
    a.insert(++a.begin(), 20);
    for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
    cout << endl;

    //вставка диапазона элементов перед последним элементом списка
    int ar[] = {30, 31, 32};
    a.insert(--a.end(), ar, ar+3);
    for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
    cout << endl;

    return 0;
}

```

```

0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
9 20 8 7 6 5 4 3 2 1 0
9 20 8 7 6 5 4 3 2 1 30 31 32 0

```

Удаление элементов

Список поддерживает те же функции добавления элементов, что и вектор, а также некоторые функции, специфичные для списка:

pop_front() – удаление элемента из начала списка

remove(T val) – удаление всех элементов со значением val

remove_if(predicate) – удаление всех элементов, для которых предикат возвращает true

Пример 7. Удаление элементов списка.

```

#include <iostream>
#include <list>
using namespace std;

bool isEven(int a)
{
    return (a%2 == 0);
}

int main()
{
    //создание пустого списка
    list<int> a;
    list<int>::iterator it;

    //заполнение списка элементами
    for(int i = 0; i < 20; i++) a.push_back(i);

```

```

//вывод списка с помощью итератора
for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
cout << endl;

//удаление 1-ого элемента списка
a.pop_front();
//удаление последнего элемента списка
a.pop_back();
for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
cout << endl;
a.push_back(1);

//удаление всех элементов со значением 1
a.remove(1);
for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
cout << endl;

//удаление всех элементов с четными значениями
a.remove_if(isEven);
for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
cout << endl;

//удаление элементов списка со 2-ого до предпоследнего
a.erase(++a.begin(),--a.end());
for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
cout << endl;

return 0;
}

```

```

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
3 5 7 9 11 13 15 17
3 17

```

Специальные операции модификации списков

Название метода	Описание
unique()	Удаляет дубликаты (элементы с одинаковыми значениями)
unique(predicate_pred)	Удаляет дубликаты, для определения равенства элементов используется предикат
splice(iterator it, list c)	Перемещает все элементы из списка с в целевой список перед позицией итератора it
splice(iterator it, list c, iterator pos)	Перемещает элемент в позиции pos списка в целевой список перед позицией итератора it
splice(iterator it, list c, iterator beg, iterator end)	Перемещает диапазон элементов [beg,end) из списка с в целевой список перед позицией итератора it
sort()	Сортирует список
sort(Traits cmp)	Сортирует список, используя заданную функцию для сравнения элементов
merge(list c)	Перемещает все элементы из списка с в целевой список с сохранением сортировки (предполагается, что оба списка являются сортированными)
merge(list c, Traits cmp)	Перемещает все элементы из списка с в целевой список с сохранением сортировки (для сортировки используется функция сравнения cmp)
reverse()	Переставляет элементы списка в обратном порядке

Пример 8. Использование метода unique().

```

#include <iostream>
#include <list>
using namespace std;

int main()
{
    //начальная инициализация списка
    int ar[] = {1,1,2,3,3,2,1};
    list<int> a(ar,ar+6);
    list<int>::iterator it;
}

```

```

//вывод списка с помощью итератора
for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
cout << endl;

//удаление повторов (повторы удаляются, только если встречаются подряд)
a.unique();

for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
cout << endl;

return 0;
}

```

```

1 1 2 3 3 2
1 2 3 2

```

Пример 9. Использование метода splice().

```

#include <iostream>
#include <list>
using namespace std;

int main()
{
    //начальная инициализация списков
    int ar[] = {1,2,3};
    list<int> a(ar,ar+3);

    int br[] = {11,12,13};
    list<int> b(br,br+3);

    list<int> c(3,20);

    int dr[] = {31,32,33};
    list<int> d(dr,dr+3);

    //вывод начального состояния
    list<int>::iterator it;
    cout << "Initial state: " << endl;
    cout << "list a: ";
    for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
    cout << endl;
    cout << "list b: ";
    for(it = b.begin(); it != b.end(); it++) cout << *it << " ";
    cout << endl;
    cout << "list c: ";
    for(it = c.begin(); it != c.end(); it++) cout << *it << " ";
    cout << endl;
    cout << "list d: ";
    for(it = d.begin(); it != d.end(); it++) cout << *it << " ";
    cout << endl;

    //перемещение элементов списка b в список a
    a.splice(a.begin(),b);
    cout << endl << "After splice b into a: " << endl;
    cout << "list a: ";
    for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
    cout << endl;
    cout << "list b: ";
    for(it = b.begin(); it != b.end(); it++) cout << *it << " ";
    cout << endl;

    //перемещение 1-ого элемента списка c в список a
    a.splice(a.begin(),c,c.begin());
    cout << endl << "After splice first element from list c: " << endl;
    cout << "list a: ";
    for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
    cout << endl;
}

```

```

//перемещение диапазона элементов списка d в список a
a.splice(a.begin(),d,++d.begin(),d.end());
cout << endl << "After splice a range from list d: " << endl;
cout << "list a: ";
for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
cout << endl;
return 0;
}

```

```

Initial state:
list a: 1 2 3
list b: 11 12 13
list c: 20 20 20
list d: 31 32 33

After splice b into a:
list a: 11 12 13 1 2 3
list b:

After splice first element from list c:
list a: 20 11 12 13 1 2 3

After splice a range from list d:
list a: 32 33 20 11 12 13 1 2 3

```

Пример 10. Использование метода sort().

```

#include <iostream>
#include <string>
#include <list>
using namespace std;

bool desc(int a1, int a2)
{
    return (a1 > a2);
}

int main()
{
    //начальная инициализация списка
    list<int> a;
    for(int i = 0; i < 10; i++) a.push_back(rand()%10);

    //Вывод начального состояния
    list<int>::iterator it;
    cout << "Initial state: " << endl;
    for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
    cout << endl;

    //сортировка по возрастанию
    a.sort();
    cout << "After ascending sort: " << endl;
    for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
    cout << endl;

    //сортировка по убыванию
    a.sort(desc);
    cout << "After descending sort: " << endl;
    for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
    cout << endl;

    //сортировка массива строк
    string br[] = {"STL", "C#", "Java", "C++"};
    list<string> b(br, br+4);

    //Вывод начального состояния
    list<string>::iterator it1;
    cout << "Initial state: " << endl;
    for(it1 = b.begin(); it1 != b.end(); it1++) cout << *it1 << " ";
}

```

```

    cout << endl;

    //сортировка по возрастанию
    b.sort();
    cout << "After ascending sort: " << endl;
    for(it1 = b.begin(); it1 != b.end(); it1++) cout << *it1 << " ";
    cout << endl;
    return 0;
}

```

```

Initial state:
1 7 4 0 9 4 8 8 2 4
After ascending sort:
0 1 2 4 4 4 7 8 8 9
After descending sort:
9 8 8 7 4 4 4 2 1 0
Initial state:
STL C# Java C++
After ascending sort:
C# C++ Java STL

```

Пример 11. Использование метода merge().

```

#include <iostream>
#include <string>
#include <list>
using namespace std;

int main()
{
    //начальная инициализация списков
    int ar[] = {1,12,23};
    list<int> a(ar,ar+3);

    int br[] = {11,12,13};
    list<int> b(br,br+3);

    int cr[] = {42,12,31};
    list<int> c(cr,cr+3);

    //Вывод начального состояния
    list<int>::iterator it;
    cout << "Initial state: " << endl;
    cout << "List a: ";
    for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
    cout << endl;
    cout << "List b: ";
    for(it = b.begin(); it != b.end(); it++) cout << *it << " ";
    cout << endl;

    //вставка списка b в список a
    //!!! оба списка должны быть отсортированы
    a.merge(b);

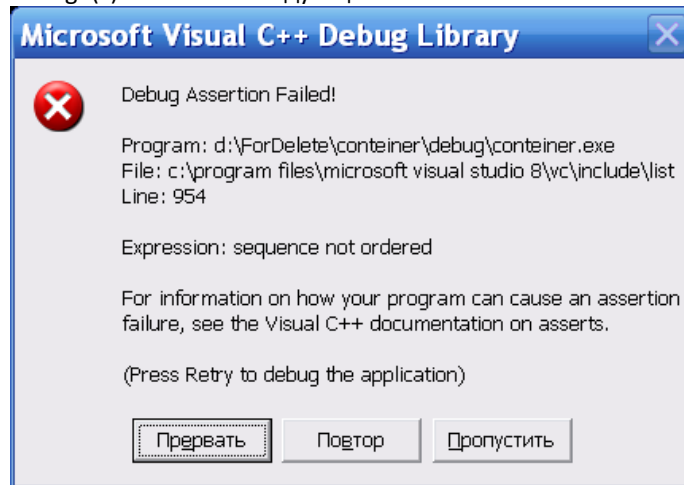
    cout << "After merge: " << endl;
    cout << "List a: ";
    for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
    cout << endl;
    cout << "List b: ";
    for(it = b.begin(); it != b.end(); it++) cout << *it << " ";
    cout << endl;

    //пример ошибки при вставке несортированного массива
    a.merge(c);
    return 0;
}

```

```
Initial state:
List a: 1 12 23
List b: 11 12 13
After merge:
List a: 1 11 12 12 13 23
List b:
```

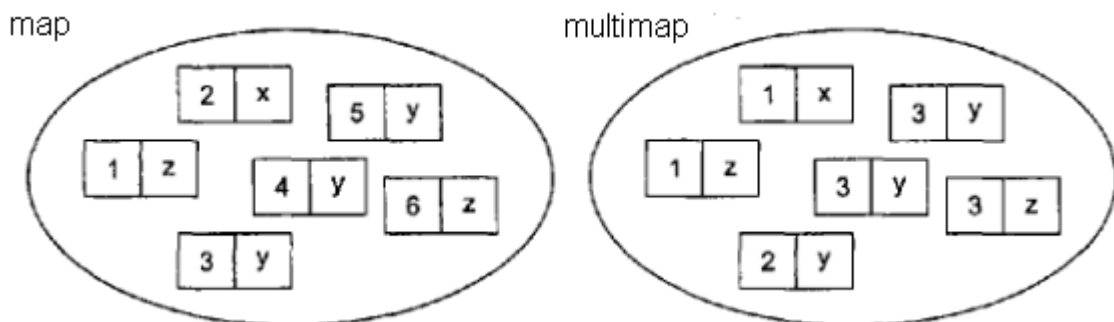
При выполнении строки `a.merge(c)` возникает следующая ошибка:



Как видно из сообщения об ошибке, она появляется при запуске Debug варианта программы. При запуске версии Release сообщение об ошибке не возникает, но результирующий список получается неупорядоченным.

Классы `map` и `multimap`

Элементами контейнеров `map` и `multimap` являются пары «ключ - значение». При размещении элементов в контейнере выполняется их сортировка на основании значений ключа. Отличие классов `map` и `multimap` состоит в том, что `map` не может содержать дублирующие элементы, а `multimap` может.



Структура контейнеров `map` и `multimap`

Контейнеры `map` и `multimap` реализуются в виде бинарного дерева, в котором элементы упорядочены по значению ключа. Такая организация контейнера обеспечивает быстрый поиск элемента по значению ключа. В других типах контейнеров такой поиск выполняется с линейной сложностью (т.к. выполняется последовательный перебор элементов).

Для использования включить заголовочный файл: `#include <map>`

Классы `map` и `multimap` определены как шаблонные классы:

```
template < class Key, class Type, class Traits = less<Key>, class Allocator=allocator<pair <const Key, Type> > >
class map
```

Key – тип ключа

Type – Тип данных

Traits – функтор, который используется для упорядочения ключей. По умолчанию используется сортировка по возрастанию.

Функция вставки элементов

`insert(const value_type& val)` вставляет значение в контейнер

value_type – тип пары «ключ - значение», определяемый контейнером

Определен как: `typedef pair<const Key, Type> value_type`

insert(iterator beg, iterator end) – вставляет диапазон элементов [beg, end)

Для формирования пары при вставке можно использовать:

- обозначение value_type
- тип pair
- функцию make_pair, которая создает объект из 2 компонентов, переданных в аргументах

Для map можно также использовать способ вставки с использованием индексирования, т.е. выполнять присвоение значения, используя ключ в качестве индекса.

Для обращения к ключу и значению итератор имеет два поля: first и second.

Пример 12. Заполнение map и вывод данных.

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

//для сокращения записи вводим обозначение:
typedef map<int, string> MyMap;
typedef MyMap::iterator Iter;
int main()
{
    //начальная инициализация map
    MyMap m;
    //использование value_type для формирования пары "ключ - значение"
    m.insert(MyMap::value_type(3, "Ivanov"));
    //можно непосредственно использовать тип pair
    m.insert(pair<int, string>(2, "Petrov"));
    //использование функции make_pair
    m.insert(make_pair(3, "Sidorov"));

    for(Iter it = m.begin(); it != m.end(); it++)
    {
        //использование свойств итератора:
        //first для обращения к ключу,
        //second для обращения к значению
        cout << it->first << " " << it->second << endl;
    }

    return 0;
}
```

```
2 Petrov
3 Ivanov
```

Как видно из полученного результата, при вставке в map дублирующие значения ключа игнорируются и выполняется сортировка по значению ключа.

Если заменить

```
typedef map<int, string> MyMap;
```

на

```
typedef multimap<int, string> MyMap;
```

то дублирующие значения ключа будут включаться в контейнер

```
2 Petrov
3 Ivanov
3 Sidorov
```

Функции удаления элементов

erase(const key_type k) удаляет все элементы со значением ключа k

erase(iterator it) удаляет элемент, на который указывает итератор

erase(iterator beg, iterator end) удаляет все элементы диапазона [beg, end)

clear() удаляет все элементы контейнера

Специальные операции поиска

Название метода	Описание
count(const key_type& k)	Подсчитывает количество элементов со значением ключа k
find(const key_type& k)	Возвращает позицию первого элемента с ключом k
lower_bound(const key_type& k)	Возвращает первый элемент с ключом >= k
upper_bound(const key_type& k)	Возвращает первый элемент с ключом > k
equal_range(const key_type& k)	Возвращает первую и последнюю позицию интервала, в котором значение ключей = k

Пример 13. Выполнение операций поиска в multimap

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

//для сокращения записи вводим обозначение:
typedef multimap<char, string> MyMap;
typedef MyMap::iterator Iter;
int main()
{
    //начальная инициализация map
    MyMap m;
    //использование value_type для формирования пары "ключ - значение"
    //в карте хранится список городов и их начальных букв
    m.insert(MyMap::value_type('L', "London"));
    m.insert(MyMap::value_type('P', "Paris"));
    m.insert(MyMap::value_type('M', "Madrid"));
    m.insert(MyMap::value_type('M', "Moskow"));
    m.insert(MyMap::value_type('M', "Milan"));

    //вывод содержимого контейнера
    for(Iter it = m.begin(); it != m.end(); it++)
        cout << it->first << " " << it->second << endl;

    //подсчет количества элементов с заданным значением ключа
    cout << endl << "There are " << m.count('M') << " cities starting on M" << endl;

    //получение диапазона элементов с заданным значением ключа
    pair<Iter, Iter> p = m.equal_range('M');
    cout << endl << "Cities starting on M: " << endl;
    for(Iter it = p.first; it != p.second; it++)
        cout << it->second << " ";
    cout << endl << endl;

    //поиск элемента по значению ключа
    Iter res = m.find('P');
    if (res != m.end())
        cout << "The first city starting on P is: " << res->second << endl;
    else
        cout << "City starting on P is not found" << endl;

    return 0;
}
```

```
L London
M Madrid
M Moskow
M Milan
P Paris

There are 3 cities starting on M

Cities starting on M:
Madrid Moskow Milan

The first city starting on P is: Paris
```

Использование контейнера map как ассоциативного массива

Контейнер map поддерживает использование оператора [] для обращения к своим элементам. Однако вместо индекса используется значение ключа.

Type& operator[](const key_type& k) – возвращает ссылку на значение элемента с ключом k.

Если элемента с заданным значением ключа не существует в map вставляется элемент, который создается путем вызова конструктора по умолчанию для типа Type.

Пример 14. Ассоциативный массив, в котором ключами являются названия нот, а значениями их длительности.

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

//для сокращения записи вводим обозначение:
typedef multimap<char, string> MyMap;
typedef MyMap::iterator Iter;
int main()
{
    //карта в качестве ключей используются названия нот, в качестве значений их
    частоты
    map<string, int> Tones;
    Tones["Do"] = 523;
    Tones["Re"] = 587;
    Tones["Mi"] = 659;
    Tones["F"] = 698;
    Tones["Sol"] = 783;
    Tones["La"] = 879;
    Tones["Si"] = 987;
    Tones["La#"]; //вместо значения вставляет значения по умолчанию ( 0 - для int)
    map<string, int>::iterator it;
    for(it = Tones.begin(); it!=Tones.end(); it++)
    {
        cout << (*it).first << " " << (*it).second << endl;
    }
    cout << Tones["Si#"] << endl; //ошибки не возникает - выводится значения по
    //умолчанию

    return 0;
}
```

```
Do 523
F 698
La 879
La# 0
Mi 659
Re 587
Si 987
Sol 783
0
```

Практические примеры использования классов контейнеров

Цель:

- Рассмотреть примеры решения типовых задач с использованием контейнеров

Вопросы для актуализации ранее полученных знаний:

3. Какие типы контейнеров Вы знаете?
4. В каких типах задачах оптимально использовать контейнер vector?
5. В каких типах задачах оптимально использовать контейнер list?
6. В чем состоит отличие контейнеров map и multimap?

Пример 1.

Необходимо разработать программу для управления счетами в банке. О каждом счете должна храниться следующая информация:

- номер счета
- имя владельца
- сумма на счету

Программа должна позволять:

- добавлять новый счет
- выполнять сортировку по владельцам
- выполнять сортировку по сумме вклада по возрастанию и по убыванию

Для хранения информации о счете создадим отдельный класс. В этом классе определим: конструктор по умолчанию, конструктор с параметрами, оператор сравнения, который будет использоваться при сортировке по имени. Как дружественные функции определим: функцию вывода в поток, функции для выполнения сравнения по сумме вклада.

```
#include <iostream>
#include <string>
#include <list>

using namespace std;

class Account //класс для представления банковского счета
{
    int ID; //номер счета
    string Name; //имя клиента
    float Sum; //сумма на счету
public:
    Account(int iID, string iName, float iSum) : ID(iID), Name(iName), Sum(iSum)
    {}
    Account()
    {
        ID = 0; Sum = 0; Name = "";
    }
    //перегруженный оператор сравнения (используется алгоритмом сортировки)
    bool operator < (const Account& a) {return Name < a.Name;}

    friend bool CmpBySumAsc(const Account& a1, const Account& a2);
    friend bool CmpBySumDesc(const Account& a1, const Account& a2);
    friend ostream& operator <<(ostream &out, const Account& a);
};

//перегруженный оператор вывода в поток
ostream& operator <<(ostream &out, const Account& a)
{
    out << "ID: " << a.ID << endl;
    out << "Name: " << a.Name << endl;
    out << "Sum: " << a.Sum << endl;
    return out;
}
```

```

//функция для сортировки по сумме на счету по возрастанию
bool CmpBySumAsc(const Account& a1, const Account& a2)
{
    return (a1.Sum < a2.Sum);
}

//функция для сортировки по сумме на счету по возрастанию
bool CmpBySumDesc(const Account& a1, const Account& a2)
{
    return (a1.Sum > a2.Sum);
}

//список для хранения банковских счетов
list<Account> bank;

//вывод содержимого списка банковских счетов
void print()
{
    list<Account>::iterator it;
    cout << "Accounts: " << endl;
    for(it = bank.begin(); it != bank.end(); it++)
        cout << *it;
    cout << endl;
}

int main()
{
    //добавление новых счетов
    bank.push_back(Account(1, "Jim", 300));
    bank.push_back(Account(2, "Tom", 200));
    bank.push_back(Account(3, "Tim", 100));
    print();

    //сортировка (по имени владельца)
    cout << "Sort by Name" << endl;
    bank.sort();
    print();

    //сортировка (по сумме на счету по возрастанию)
    cout << "Sort by sum ascending" << endl;
    bank.sort(CmpBySumAsc);
    print();

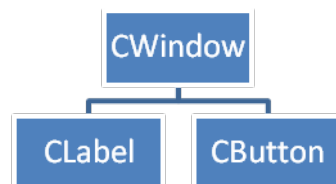
    //сортировка (по сумме на счету по убыванию)
    cout << "Sort by sum descending" << endl;
    bank.sort(CmpBySumDesc);
    print();

    return 0;
}

```

Пример 2.

Предположим, Вы разработали собственные визуальные компоненты. У Вас имеется следующая иерархия:



Вы динамически размещаете элемент на форме и для последующего взаимодействия с этими элементами сохраняете их в контейнере. Для того чтобы в контейнере можно было сохранять элементы производных классов необходимо в качестве типа элементов контейнера объявить тип указателя на базовый класс (CWindows*).

При хранении в контейнере указателей возникает интересная проблема с сортировкой. Сортировка выполняется на основании значений указателей, т.е. никак не связана с содержимым объектов, на которые эти указатели указывают. Для устранения этой проблемы следует в функцию sort() передать функцию сравнения, которая будет сравнивать объекты, хранящиеся в контейнере на основании заданного критерия.

Следует подчеркнуть, что при удалении из контейнера указателей на объекты вызов оператор delete не происходит, поэтому освобождение памяти надо выполнять самостоятельно.

```
#include <vector>
#include <string>
#include <iostream>
#include <algorithm>
using namespace std;

class CWindow
{
protected:
    string caption;
public:
    CWindow(string cap):caption(cap){}
    virtual void print() = 0;

    friend bool cmpName(const CWindow* c1, const CWindow* c2);
};

class CLabel: public CWindow
{
public:
    CLabel(string cap):CWindow(cap){}
    void print() {cout << "Label: " << caption << endl;}
    void clickLabel() {cout << "The label " << caption << " is clicked: " << endl;}
};

class CButton: public CWindow
{
public:
    CButton(string cap):CWindow(cap){}
    void print() {cout << "Button: " << caption << endl;}
    void pressButton() {cout << "The button " << caption << " is pressed: " << endl;}
};

bool cmpName(const CWindow* c1, const CWindow* c2)
{
    return c1->caption < c2->caption;
}

int main()
{
    //заполнение контейнера
    vector<CWindow*> controls;
    controls.push_back(new CButton("Button1"));
    controls.push_back(new CLabel("Label1"));
    controls.push_back(new CButton("Button2"));
    controls.push_back(new CButton("Button3"));
    controls.push_back(new CLabel("Lable2"));

    //вывод начального состояния
    vector<CWindow*>::iterator it;
    cout << "Initial state: " << endl;
    for(it = controls.begin(); it != controls.end(); it++) (*it)->print();

    //неудачная попытка сортировки
    //сортировка выполняется по значениям указателей
    cout << endl << "After sort by pointers: " << endl;
    sort(controls.begin(), controls.end());
    for(it = controls.begin(); it != controls.end(); it++) (*it)->print();

    //правильный вариант сортировки
    cout << endl << "Sort by Name: " << endl;
    sort(controls.begin(), controls.end(), cmpName);
    for(it = controls.begin(); it != controls.end(); it++) (*it)->print();

    //выполнение действий, специфичных для каждого элемента управления
    cout << endl << "Use control: " << endl;
}
```

```

cout << endl << "Sort by Name: " << endl;
for(it = controls.begin(); it != controls.end(); it++)
{
    CButton* b;
    CLabel* l;
    if (l = dynamic_cast<CLabel*>(*it)) l->clickLabel();
    if (b = dynamic_cast<CButton*>(*it)) b->pressButton();
}

//освобождение памяти перед удалением элементов из контейнера
for(it = controls.begin(); it != controls.end(); it++) delete *it;
controls.clear();
return 0;
}

```

Пример 3.

Реализация 2-мерного массива с использованием вектора.

В векторе элементами могут быть не только простые типы данных, но и другие контейнеры, например, вектор. Вектор, в котором содержится вектора, позволяет реализовать динамический 2-мерный массив.

```

#include <vector>
#include <iostream>
using namespace std;

int main()
{
    int n = 3; //количество строк
    int m = 4; //количество столбцов

    vector<int> v;
    vector<vector<int> > a;

    for(int i = 0; i < n; i++)
    {
        //заполнение строки элементами
        for(int j = 0; j < m; j++) v.push_back(i*10 + j);
        //добавление строки в матрицу
        a.push_back(v);
        v.clear();
    }

    //вывод матрицы
    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < m; j++)
            cout << a[i][j] << " ";
        cout << endl;
    }

    return 0;
}

```

Пример 4.

Обновление ключа элемента в map. Значения ключей хранятся как константные ссылки, поэтому напрямую изменить значения ключа нельзя. Для изменения значения следует удалить элемент со старым значением ключа и вставить с новым значением.

```

#include <map>
#include <string>
#include <iostream>
using namespace std;

int main()
{
    map <string, string> m;
    //заполнение катры названиями городов и достопримечательностями
    m.insert(make_pair("Leningrad", "Winter Palace"));
    m.insert(make_pair("London", "Big Ben"));
    m.insert(make_pair("Parice", "Louvre"));

    map <string, string>::iterator it;
    cout << "Initial State: " << endl;
    for(it = m.begin(); it != m.end(); it++)
        cout << it->first << " " << it->second << endl;

    //Лененград переименовали
    //непосредсвенно ключ заменить нельзя:
    //m.begin()->first = "Peregburg"; //ошибка присвоения значения операнду типа
const string
    //Для изменения надо удалить, а потом добавить:
    m.erase("Leningrad");
    m.insert(make_pair("Peregburg", "Winter Palace"));
    cout << endl << "After change key: " << endl;
    for(it = m.begin(); it != m.end(); it++)
        cout << it->first << " " << it->second << endl;
}

```

Использование функторов

Цель:

- познакомиться с понятием функторов
- рассмотреть преимущества использования функторов
- научиться применять функторы в алгоритмах
- научиться применять функторы для специализации шаблонного класса
- рассмотреть стандартные функторы STL
- рассмотреть применение функциональных адаптеров

Вопросы для актуализации ранее полученных знаний:

1. Для чего используется указатель на функцию?
2. Как и для каких целей выполняется перегрузка оператора () ?

Понятие функтора

Объектом функции или функтором называется объект класса, в котором перегружен оператор ().

Рассмотрим пример простого класса, в котором перегруженный оператор () выполняет возведение числа в квадрат.

Пример 1. Функтор для возведения числа в квадрат.

```
#include <iostream>
using namespace std;

//класс, в котором перегружен оператор ()
class square
{
public:
    int operator() (int a) {return a*a;}
};

int main()
{
    //объявление функтора
    square sq;
    //использование функтора
    cout << sq(5);
    return 0;
}
```

Основные преимущества функторов:

- функтор может обладать определенным состоянием, для обычных функций это невозможно
- объект функции имеет тип, поэтому его можно использовать при специализации шаблона

Наглядным примером использования внутреннего состояния является применение объекта функции в методах типа `remove_if()`. В метод `remove_if()` надо передавать предикат, которые возвращает `true` для элементов, которые надо удалить. Если этот предикат задавать обычной функцией, то в ней можно будет задать только жесткий критерий отбора (например, число < 100). Для другого критерия (например, число < 200) надо создавать отдельную функцию. Поскольку объект функции обладает определенным состоянием, это состояние можно использовать для гибкой настройки критерия отбора.

Пример 2. Использование функтора для передачи значения для сравнения в метод `remove_if()`.

```
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>
using namespace std;

class cmp
{
    int Val;
```



```

public:
    cmp(int iVal) : Val(iVal) {}
    //оператор () используется для сравнения элементов
    //контейнера со значением, сохраненным в классе
    bool operator() (int a) {return a < Val;}
};

int main()
{
    //объявление и заполнение списка
    int ar[] = {1,3,2,5,8,6,7,9,4,0};
    list<int> a = list<int>(ar, ar+10);
    ostream_iterator<int> out(cout, " ");
    //вывод начального состояния списка
    cout << "Initial state: " << endl;
    copy(a.begin(), a.end(), out); cout << endl;

    //удаление элементов со значением < 3 с использованием функтора
    a.remove_if(cmp(3));
    cout << endl << "After remove less then 3: " << endl;
    copy(a.begin(), a.end(), out); cout << endl;

    //удаление элементов со значением < 6 с использованием функтора
    a.remove_if(cmp(6));
    cout << endl << "After remove less then 6: " << endl;
    copy(a.begin(), a.end(), out); cout << endl;

    return 0;
}

```

```

Initial state:
1 3 2 5 8 6 7 9 4 0

After remove less then 3:
3 5 8 6 7 9 4

After remove less then 6:
8 6 7 9

```

В шаблонных классах set, multiset, map, multimap одним из параметров шаблона является функтор, который задает правило сравнения ключей контейнера. Это правило определяет способ упорядочения ключей в контейнере.

Например, шаблон для set объявлен следующим образом:

```

template < class Key, class Traits = less<Key>, class Allocator=allocator<pair <const Key, Type> >>
class set

```

Key – тип ключа

Traits – функтор, который должен содержать бинарный предикат для сравнения ключей.

Предположим, имеется класс, в котором хранится следующая информация:

- предметная область
- тематика
- название

Необходимо сохранить объекты этого класса в set, предусмотрев упорядочение по предметной области, я для каждой предметной области по теме. Этот критерий упорядочения можно реализовать с помощью функции, однако эту функцию нельзя будет использовать при специализации шаблона в качестве значения параметра типа Traits. Значение этого параметра можно задать только с помощью функтора.

Пример 3. Использование функтора при специализации шаблона

```

#include <iostream>
#include <set>
#include <iterator>
#include <algorithm>
#include <string>
using namespace std;

```

```

//класс, описывающий книгу
class Book
{
public:
    string subject; //предметная область
    string theme;   //тема
    string name;    //название
    Book(string iSubject, string iTheme, string iName): subject(iSubject),
    theme(iTheme), name(iName){}
};

//функтор, используется для задания критерия упорядочения контейнера
class cmpBooks
{
public:
    //оператор () используется для сравнения книг по области знаний
    //a в пределах обной оласти знаний по тематике
    bool operator() (const Book& a, const Book& b)
    {
        if (a.subject == b.subject) return a.theme < b.theme;
        return (a.subject < b.subject);
    }
};

//функция сравнения (не может бять использована в качестве значения шаблонного
параметра)
bool cmpBooksFunc(const Book& a, const Book& b)
{
    if (a.subject == b.subject) return a.theme < b.theme;
    return (a.subject < b.subject);
}

//ОШИБКА!!! - функция не может использоваться в качестве значения параметра шаблона
//typedef set<Book, cmpBooksFunc> setBooks;

//функтор - может использоваться в качестве значения параметра шаблона
typedef set<Book, cmpBooks> setBooks;

int main()
{
    setBooks b; //множество книг
    //добавление книг
    b.insert(Book("Nature", "Ocean", "The Pacific"));
    b.insert(Book("Techncis", "Harware", "PC"));
    b.insert(Book("Nature", "Animals", "Sheathbill"));

    //Вывод книг
    setBooks::iterator it;
    for(it = b.begin(); it != b.end(); it++)
        cout << (*it).subject << " " << (*it).theme << " " << (*it).name << endl;

    return 0;
}

```

```

Nature Animals Sheathbill
Nature Ocean The Pacific
Techncis Harware PC

```

Стандартные объекты функций

Библиотека STL содержит ряд стандартных функторов.

Выражение	Описание
<code>negate<T>()</code>	Изменение знака параметра: <code>-param</code>
<code>plus<T>()</code>	<code>param1 + param2</code>
<code>minus<T>()</code>	<code>param1 - param2</code>
<code>multiplies<T>()</code>	<code>param1 * param2</code>
<code>divides<T>()</code>	<code>param1 / param2</code>
<code>modulus<T>()</code>	<code>param1 % param2</code>
<code>equal_to<T>()</code>	<code>param1 == param2</code>
<code>not_equal_to<T>()</code>	<code>param1 != param2</code>
<code>less<T>()</code>	<code>param1 < param2</code>
<code>greater<T>()</code>	<code>param1 > param2</code>
<code>less_equal<T>()</code>	<code>param1 <= param2</code>
<code>greater_equal<T>()</code>	<code>param1 >= param2</code>
<code>logical_not<T>()</code>	<code>!param</code>
<code>logical_and<T>()</code>	<code>param1 && param2</code>
<code>logical_or<T>()</code>	<code>param1 param2</code>

Для использования необходимо включить: `#include <functional>`

Функтор `less<T>()` является параметром по умолчанию при сортировке объектов.

Пример 4. Использование стандартных функторов для сортировки списка, для изменения знака элементов контейнера

```
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

int main()
{
    int ar[] = {10,12,8,};
    list<int> a = list<int>(ar, ar+3);
    list<int>::iterator it;

    //Вывод списка
    cout << "Initial state: " << endl;
    for(it = a.begin(); it != a.end(); it++) cout << *it << " ";

    //по умолчанию используется less<int>(), выполняется сортировка по возрастанию
    a.sort();
    cout << endl << "Sort with default criteria: " << endl;
    for(it = a.begin(); it != a.end(); it++) cout << *it << " ";

    //явное задание функтора для сортировки: функтор greater<int>()
    //обеспечивает сортировку по убыванию
    a.sort(greater<int>());
    cout << endl << "Sort with greater<int> criteria: " << endl;
    for(it = a.begin(); it != a.end(); it++) cout << *it << " ";

    //изменение знака всех элементов списка
    transform(a.begin(), a.end(), a.begin(), negate<int>());
    cout << endl << "After negate: " << endl;
    for(it = a.begin(); it != a.end(); it++) cout << *it << " ";
    cout << endl;
    return 0;
}
```

```
Initial state:
10 12 8
Sort with default criteria:
8 10 12
Sort with greater<int> criteria:
12 10 8
After negate:
-12 -10 -8
```

Пример 5. Использование стандартных функторов для попарного сложения, вычитания, умножения, деления элементов 2-х векторов.

```
#include <iostream>
#include <functional>
#include <iterator>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    //задание векторов
    int ar[] = {20,30,40};
    int br[] = {1,2,3};
    vector<int> a = vector<int>(ar, ar+3);
    vector<int> b = vector<int>(br, br+3);
    ostream_iterator<int> out(cout, " ");
    cout << "a: ";
    copy(a.begin(),a.end(),out);cout << endl;
    cout << "b: ";
    copy(b.begin(),b.end(),out); cout << endl;
    //использование функторов для выполнения попарного сложения элементов векторов
    cout << "use plus<int>(): ";
    transform(a.begin(),a.end(),b.begin(),out,plus<int>()); cout << endl;
    //использование функторов для выполнения попарного вычитания элементов векторов
    cout << "use minus<int>(): ";
    transform(a.begin(),a.end(),b.begin(),out,minus<int>()); cout << endl;
    //использование функторов для выполнения попарного деления элементов векторов
    cout << "use divides<int>(): ";
    transform(a.begin(),a.end(),b.begin(),out,divides<int>()); cout << endl;
    //использование функторов для выполнения попарного умножения элементов векторов
    cout << "use multiplies<int>(): ";
    transform(a.begin(),a.end(),b.begin(),out,multiplies<int>()); cout << endl;

    return 0;
}
```

```
a: 20 30 40
b: 1 2 3
use plus<int>(): 21 32 43
use minus<int>(): 19 28 37
use divides<int>(): 20 15 13
use multiplies<int>(): 20 60 120
```

Функциональные адаптеры

Функциональным адаптером называется объект, который позволяет комбинировать объекты функций друг с другом, с определенными значениями или со специальными функциями.

Стандартные функциональные адаптеры

Адаптер	Описание	Результат применения
bind1st(op,value)	Связывает 1-ый элемент заданного функтора с указанным значением	op(value,param)
bind2nd(op,value)	Связывает 2-ой элемент заданного функтора с указанным значением	op(param, value)
not1(op)	Инверсия унарного функтора	!op(param)
not2(op)	Инверсия бинарного функтора	!op(param1, param2)

Пример 6. Использование стандартного функтора и функционального адаптера для:

- удаления всех элементов списка со значением больше 10
- умножения всех элементов списка на заданное число
- вычитание элементов списка из заданного числа

```
#include <iostream>
#include <functional>
#include <iterator>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    int ar[] = {2,4,6,8,10,12,14};

    list<int> a = list<int>(ar, ar+7);

    ostream_iterator<int> out(cout, " ");
    cout << "Initial state: ";
    copy(a.begin(), a.end(), out); cout << endl;

    //удаление элементов списка со значением большим или равным заданному (10)
    //2-ой паратер функтора связывается с числом 10
    a.remove_if(bind2nd(greater_equal<int>(), 10));
    cout << "After delete elements >= 10: ";
    copy(a.begin(), a.end(), out); cout << endl;

    //умножение элементов списка на заданное число (2)
    //2-ой паратер функтора связывается с числом 2
    cout << "list element * 2: ";
    transform(a.begin(), a.end(), out, bind2nd(multiplies<int>(), 2));
    cout << endl;

    //вычитание элементов списка из заданного число (100)
    //1-ый паратер функтора связывается с числом 100
    cout << "100 - list element: ";
    transform(a.begin(), a.end(), out, bind1st(minus<int>(), 100));
    cout << endl;
    return 0;
}
```

```
Initial state: 2 4 6 8 10 12 14
After delete elements >= 10: 2 4 6 8
list element * 2: 4 8 12 16
100 - list element: 98 96 94 92
```

Написание пользовательских объектов функций

Для того чтобы объект функцию можно было использовать с функциональными адаптерами в нем необходимо определить типы аргументов и результата. Для этого в библиотеку STL включены 2 структуры:

```
template<class _A, class _R>
struct unary_function
{
    typedef _A Argument_Type;
    typedef _R Result_Type;
};
```

```
template<class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```

Унарные функторы следует наследовать от unary_function, а бинарные от binary_function.

Пример 7. Создание и использование объекта функции для возведения первого числа в степень, заданную вторым числом.

```
#include <iostream>
#include <cmath>
#include <functional>
#include <iterator>
#include <vector>
#include <algorithm>
using namespace std;

//создание собственного функционального объекта для возведения числа в степень
//наследование от binary_function для определения типов аргументов и типа результата
template <class Arg1, class Arg2>
struct MyPower: public binary_function<Arg1, Arg2, Arg1>
{
    Arg1 operator() (Arg1 a, Arg2 b) const {return pow(a,b);}
} ;

int main()
{
    int ar[] = {2,4,6};

    vector<int> a = vector<int>(ar, ar+3);

    ostream_iterator<int> out(cout, " ");
    cout << "Initial state: ";
    copy(a.begin(),a.end(),out); cout << endl;

    //использование функциоального адаптера с собственным функтором:
    //для вывода квадратов элементов
    cout << "vecot elements in power 2: ";
    transform(a.begin(),a.end(),out,bind1st(MyPower<float, int>(),2));cout << endl;

    //для вчисла 2 в степени, равной элементам массива
    cout << "2 in power of vecot elements: ";
    transform(a.begin(),a.end(),out,bind2nd(MyPower<float,int>(),2));cout << endl;

    return 0;
}
```

```
Initial state: 2 4 6
vecot elements in power 2: 4 16 64
2 in power of vecot elements: 4 16 36
```

Использование предикатов

Цель:

- познакомиться с понятием предиката

Предикат – функция, возвращающая логическое значение. По количеству принимаемых аргументов предикаты делятся на:

- унарные – принимают 1 аргумент
- бинарные - принимают 2 аргумента

Предикаты с большим количеством аргументов практически не используются.

Предикаты применяются в следующих ситуациях:

- унарный предикат в алгоритмах поиска может использоваться для задания критерия, которому должен удовлетворять искомый элемент
- бинарный предикат может передаваться алгоритму сортировки для задания критерия, который используется для сравнения элементов
- для модификации операции в численных алгоритмах

Предикат может задаваться с помощью функции или объекта функции (функтора).

Пример: использование предиката для удаления элементов, значения которых попадают в заданный диапазон.

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <queue>
#include <map>
using namespace std;

//функтор реализует предикат для проверки попадания элемента в диапазон
class DelRange
{
private:
    int lo, hi;
public:
    DelRange(int lo, int hi)
    {
        this->lo = lo;
        this->hi = hi;
    }
    bool operator() (int val)
    {
        return ((lo <= val) && (val <= hi));
    }
};

int main()
{
    int ar[] = {1,2,3,4,5,6,7,8,9};
    vector<int> a(ar,ar+9);
    vector<int>::iterator it;
    ostream_iterator<int> out(cout, " ");

    cout << "Initial state: " << endl;
    copy(a.begin(),a.end(),out); cout << endl;

    //использование предиката для удаления элементов
    it = remove_if(a.begin(),a.end(),DelRange(2,5));
    a.erase(it, a.end());

    cout << "After delete range from 2 to 5: " << endl;
    copy(a.begin(),a.end(),out); cout << endl;

    return 0;
}
```

```
Initial state:
1 2 3 4 5 6 7 8 9
After delete range from 2 to 5:
1 6 7 8 9
```

Пример: использование предиката для сортировки массива символов без учета регистра символа

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <queue>
#include <map>
using namespace std;

//перевод символа в верхний регистр
bool cmp(char a, char b)
{
    if ((a >= 'a') && (a <= 'z')) a = a - 'a' + 'A';
    if ((b >= 'a') && (b <= 'z')) b = b - 'a' + 'A';
    return a < b;
}

int main()
{
    char ar[] = {'a','h','A','K','b','z','X','I'};
    vector<char> a(ar,ar+8);
    ostream_iterator<char> out(cout," ");

    cout << "Initial state: " << endl;
    copy(a.begin(),a.end(),out); cout << endl;

    sort(a.begin(),a.end(),cmp);

    cout << "After sort: " << endl;
    copy(a.begin(),a.end(),out); cout << endl;

    return 0;
}
```

```
Initial state:
a h A K b z X I
After sort:
a A b h I K X z
```


Использование алгоритмов

Цель:

- познакомиться с понятием алгоритма
- рассмотреть классификацию алгоритмов

Вопросы для актуализации ранее полученных знаний:

1. Какие подходы использовались в C++ для применения одного и того же алгоритма обработки к различным типам данных?
2. Какие виды обработки наиболее часто приходится использовать при работе с массивами?

Особенности алгоритмов STL

Алгоритмы работают с итераторами, с помощью которых задается один или несколько интервалов значений, предназначенных для обработки алгоритмом. Алгоритмы работают в режиме замены, а не в режиме вставки, поэтому перед применением алгоритма надо убедиться, что принимающий контейнер имеет достаточное количество элементов. Если необходимо использовать режим вставки, в алгоритм следует передавать специальные итераторы вставки.

В некоторые алгоритмы можно передавать пользовательские функции или функторы, которые используются при внутренней работе алгоритма.

Для использования алгоритмов надо подключить заголовочный файл `<algorithm>`.

Классификация алгоритмов

Немодифицирующие алгоритмы сохраняют порядок следования элементов и их значения. Работают с итераторами ввода и прямыми итераторами. Могут использоваться со всеми типами контейнеров.

Модифицирующие алгоритмы изменяют значения элементов последовательности, с которой работают.

Алгоритмы удаления удаляют отдельные элементы или все элементы диапазона.

Перестановочные алгоритмы изменяют порядок следования элементов в диапазоне.

Алгоритмы сортировки выполняют сортировку заданного интервала.

Если в названии алгоритма используется суффикс `_if`, в этот алгоритм должен передаваться предикат, который используется для проверки выполнения условия.

Например:

```
replace(ForwardIterator first, ForwardIterator last, const Type& old, const Type& new)
```

замещает заданное значение (old) новым (new)

```
replace_if(ForwardIterator first, ForwardIterator last, UnaryPredicate pred, const Type& new)
```

замещает значение, для которого предикат `pred` возвращает true новым (new)

Если в названии алгоритма используется суффикс `_copy`, алгоритм не изменяет значения в исходном интервале, результат его работы записывается в другой интервал (в этом же или другом контейнере).

Например:

```
replace_copy(ForwardIterator first, ForwardIterator last, OutputIterator out, const Type& old, const Type& new)
```

копирует значения заданного интервала в выходной контейнер, начиная с позиции `out`, замещая заданное значение (old) новым (new)

Немодифицирующие алгоритмы

Подсчет элементов

Сложность линейная

```
difference_type count(InputIterator _First, InputIterator _Last, const Type& _Val);  
подсчитывает количество элементов со значением равным заданному (_Val)  
difference_type count_if(InputIterator _First, InputIterator _Last, Predicate _Pred);  
подсчитывает количество элементов, для которых предикат _Pred возвращает true
```

Пример: подсчет количества отрицательных элементов вектора

```
#include <iostream>  
#include <functional>  
#include <vector>  
#include <algorithm>
```

```
using namespace std;

int main()
{
    int ar[] = {-1,2,-2,4,6};

    vector<int> a = vector<int>(ar, ar+5);
    //вывод количества элементов < 0
    cout << "The number of element < 0: " <<
        count_if(a.begin(), a.end(), bind2nd(less<int>(),0)) << endl;

    return 0;
}
```

```
The number of element < 0: 2
```

Минимум и максимум

Сложность линейная

InputIterator min_element(InputIterator_First, InputIterator_Last);

находит минимальный элемент заданного диапазона

InputIterator min_element(InputIterator_First, InputIterator_Last, CompFunc op);

находит минимальный элемент заданного диапазона, используя для сравнения элементов функцию op

InputIterator max_element(InputIterator_First, InputIterator_Last);

находит максимальный элемент заданного диапазона

InputIterator max_element(InputIterator_First, InputIterator_Last, CompFunc op);

находит максимальный элемент заданного диапазона, используя для сравнения элементов функцию op

Пример: поиск минимального, максимального элемента вектора, а также элементов с максимальным и минимальным абсолютным значением

```
#include <iostream>
#include <functional>
#include <vector>
#include <algorithm>
#include <cmath>

using namespace std;

bool CmpAbs(int a, int b)
{
    return abs(a) < abs(b);
}

int main()
{
    int ar[] = {-1,2,-8,4,6};

    vector<int> a = vector<int>(ar, ar+5);
    //функции возвращают итератор, поэтому надо производить его разименование
    cout << "min element: " << *min_element(a.begin(), a.end()) << endl;
    cout << "min element by absolute value: " <<
        *min_element(a.begin(), a.end(), CmpAbs) << endl;
    cout << "max element: " << *max_element(a.begin(), a.end()) << endl;
    cout << "max element by absolute value: " <<
        *max_element(a.begin(), a.end(), CmpAbs) << endl;

    return 0;
}
```

```
min element: -8
min element by absolute value: -1
max element: 6
max element by absolute value: -8
```

Поиск элементов

Сложность линейная

InputIterator find(InputIterator_First, InputIterator_Last, const Type& _Val);

находит 1-ый элемент с заданным значением (_Val)

InputIterator find_if(InputIterator_First, InputIterator_Last, Predicate_Pred);

находит 1-ый элемент, который удовлетворяет заданному условию (*_Pred*)

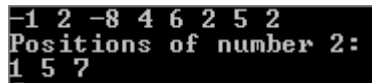
Пример: поиск всех элементов со значение 2

```
#include <iostream>
#include <functional>
#include <iterator>
#include <vector>
#include <algorithm>
#include <cmath>

using namespace std;

int main()
{
    int ar[] = {-1, 2, -8, 4, 6, 2, 5, 2};
    int* a = ar;
    ostream_iterator<int> out(cout, " ");
    copy(ar, ar + 8, out);
    cout << endl << "Positions of number 2: " << endl;
    do
    {
        a = find(a, ar + 8, 2);
        if (a != ar + 8)
        {
            cout << a - ar << " ";
            a++;
        }
        else break;
    }
    while(true);
    cout << endl;

    return 0;
}
```



```
-1 2 -8 4 6 2 5 2
Positions of number 2:
1 5 7
```

Модифицирующие алгоритмы

Копирование

Линейная сложность

`OutputIterator copy(InputIterator _First, InputIterator _Last, OutputIterator dest);`

копирует элементы заданного интервала в выходной интервал, начиная с позиции `dest`

`OutputIterator backword_copy (InputIterator _First, InputIterator _Last, OutputIterator dest);`

копирует элементы заданного интервала в обратном порядке в выходной интервал, начиная с позиции `dest`

Алгоритмы возвращают позицию за последним скопированным элементов в выходном интервале. Чтобы при копировании выполнялась вставка в выходной интервал, надо использовать итератор вставки.

Пример: копирование массива в вектор и использованием итератора вывода и итератора вставки.

```
#include <iostream>
#include <functional>
#include <iterator>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    int ar[] = {1,2,3,4,5,6,7,8};
    cout << "Array to copy: ";
    ostream_iterator<int> out(cout, " ");
    copy(ar, ar + 8, out); cout << endl;

    vector<int> a(10,0);
```

```

//копирование массива в вектор a, который уже соержит 10 элементов
//при копировании новые значения переписываются поверх старых
copy(ar, ar+8, a.begin());
cout << "Vector a: ";
copy(a.begin(), a.end(), out); cout << endl;

vector<int> b;
insert_iterator<vector<int> > it(b, b.begin());
//копирование массива в вектор b нулевого размера
//с использованием итератора вставки
copy(ar, ar+8, it);
cout << "Vector b: ";
copy(b.begin(), b.end(), out); cout << endl;

vector<int> c;
//ошибка: попытка копирования в вектор с нульвого размера
//без использования итератора вставки
copy(ar, ar+8, c.end());

return 0;
}

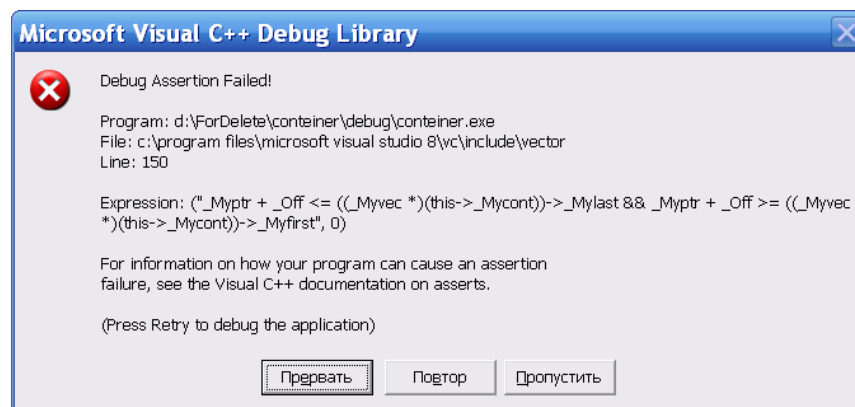
```

```

Array to copy: 1 2 3 4 5 6 7 8
Vector a: 1 2 3 4 5 6 7 8 0 0
Vector b: 1 2 3 4 5 6 7 8

```

При копировании в вектор a, который имеет 10 элементов, новые значения записываются поверх старых. При копировании в вектор b, который не имеет элементов, используется итератор вставки, который добавляет элементы в процессе копирования. При копировании в вектор c возникает ошибка, т.к. он не содержит элементов.



Преобразование

Линейная сложность

`OutputIterator transform(InputIterator _First, InputIterator _Last, OutputIterator _Dest, UnaryFunc op);`

Для каждого элемента интервала вызывается функция `op`, результат, который возвращает эта функция, записывается в выходной интервал, начиная с позиции `Dest`. Значения `_First` и `_Dest` могут быть идентичны, т.е. алгоритм может использоваться для модификации данных внутри интервала.

`OutputIterator transform(InputIterator _First, InputIterator _Last, InputIterator _Second, OutputIterator _Dest, BinaryFunc op);`

Для каждой пары соответствующих элементов интервалов `[_First, _Last)` и `[_Second, ...)` вызывается функция `op`, результат, который возвращает эта функция, записывается в выходной интервал, начиная с позиции `Dest`.

Алгоритмы возвращают позицию за последним скопированным элементом в выходном интервале.

Пример: изменения знака чисел в векторе на противоположный. Парное сложение элементов 2 векторов и вывод результата на экран.

```

#include <iostream>
#include <functional>
#include <vector>
#include <algorithm>

using namespace std;

```

```

int main()
{
    int ar[] = {1,-2,3,-4,5,-6,7,-8};
    vector<int> a(ar, ar+8);
    cout << "Initial state: ";
    ostream_iterator<int> out(cout, " ");
    copy(a.begin(),a.end(), out); cout << endl;

    //изменение знака каждого элемента на противоположный
    transform(a.begin(), a.end(), a.begin(), negate<int>());
    cout << "Reverse sign: ";
    copy(a.begin(),a.end(), out); cout << endl;

    //попарное перемножение элементов вектора и массива
    cout << "Sum of vector and array: ";
    transform(a.begin(), a.end(), ar, out, multiplies<int>());
    cout << endl;
    return 0;
}

```

```

Initial state: 1 -2 3 -4 5 -6 7 -8
Reverse sign: -1 2 -3 4 -5 6 -7 8
Sum of vector and array: -1 -4 -9 -16 -25 -36 -49 -64

```

Присвоение

void fill(ForwardIterator _First, ForwardIterator _Last, const T& new);

присвоение всем элементам интервала заданного значения (new)

void generate(ForwardIterator _First, ForwardIterator _Last, Func op);

присвоение всем элементам интервала значения, возвращаемого функцией op.

Пример: заполнение вектора случайными значениями

```

#include <iostream>
#include <functional>
#include <iterator>
#include <vector>
#include <algorithm>

using namespace std;

int rangeRand()
{
    return rand()%10;
}

int main()
{
    vector<int> a(10);
    generate(a.begin(), a.end(), rangeRand);
    ostream_iterator<int> out(cout, " ");
    copy(a.begin(),a.end(), out); cout << endl;
    return 0;
}

```

```

1 7 4 0 9 4 8 8 2 4

```

Замена элементов

void replace(ForwardIterator _First, ForwardIterator _Last, const Type& old, const Type& new)

замещает заданное значение (old) новым (new)

void replace_if(ForwardIterator _First, ForwardIterator _Last, UnaryPredicate pred, const Type& new)

замещает значение, для которого предикат pred возвращает true новым (new)

replace_copy(ForwardIterator _First, ForwardIterator _Last, OutputIterator _Dest, const Type& old, const Type& new)

копирует элементы заданного диапазона в выходной диапазон, начиная с _Dest, при копировании значения old заменяются на new

replace_copy_if(ForwardIterator _First, ForwardIterator _Last, OutputIterator _Dest, UnaryPredicate pred, const Type& new)

копирует элементы заданного диапазона выходной диапазон, начиная с `_Dest`, при копировании значения элементов, для которых предикат `pred` возвращает `true`, заменяются на `new`

Пример: замена всех отрицательных элементов нулями, замена всех нулей на -1 с копированием в другой контейнер.

```
#include <iostream>
#include <functional>
#include <iterator>
#include <vector>
#include <algorithm>

using namespace std;

int rangeRand()
{
    return rand()%30 - 10;
}

int main()
{
    vector<int> a(10);
    generate(a.begin(), a.end(), rangeRand);
    ostream_iterator<int> out(cout, " ");
    cout << "Initial state: ";
    copy(a.begin(), a.end(), out); cout << endl;

    //замена всех отрицательных элементов на 0 в том же векторе
    replace_if(a.begin(), a.end(), bind2nd(less<int>(), 0), 0);
    cout << "After set negative to 0: ";
    copy(a.begin(), a.end(), out); cout << endl;

    //замена всех элементов со значением 0 на -1 с копированием в другой вектор
    //используем итератор вставки в конец, т.к. вектор b изначально пустой
    vector<int> b;
    replace_copy(a.begin(), a.end(), back_inserter(b), 0, -1);
    cout << "Copy in b and replace 0 with -1: ";
    copy(b.begin(), b.end(), out); cout << endl;
    return 0;
}
```

```
Initial state: 1 7 -6 0 19 -6 8 8 12 4
After set negative to 0: 1 7 0 0 19 0 8 8 12 4
Copy in b and replace 0 with -1: 1 7 -1 -1 19 -1 8 8 12 4
```

Алгоритмы удаления

`ForwardIterator remove(ForwardIterator _First, ForwardIterator _Last, const Type& val)`

удаляет все элементы со значением `val`

`ForwardIterator remove_if(ForwardIterator _First, ForwardIterator _Last, UnaryPredicate pred)`

удаляет все элементы, для которых предикат возвращает значение `true`

`ForwardIterator remove_copy(ForwardIterator _First, ForwardIterator _Last, OutputIterator _Dest, const Type& val)`

копирует все элементы заданного диапазона в выходной диапазон, начиная с позиции `_Dest`, кроме элементов со значением `val`

`ForwardIterator remove_copy_if(ForwardIterator _First, ForwardIterator _Last, OutputIterator _Dest, UnaryPredicate pred)`

копирует все элементы заданного диапазона в выходной диапазон, начиная с позиции `_Dest`, кроме элементов, для которых предикат `pred` возвращает `true`

Возвращают новый логический конец интервала, т.е. итератор, указывающий на позицию за последним элементом, который не был удален.

Алгоритмы не изменяют указатель на конец контейнер `end()`. Для изменения этого указателя надо использовать метод контейнера `erase()`.

Пример: удаление всех чисел со значением 0.

```
#include <iostream>
```

```

#include <functional>
#include <iterator>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    int ar[] = {1,0,2,0,3,0,4 };
    vector<int> a(ar,ar+7);
    ostream_iterator<int> out(cout, " ");
    cout << "Initial state: ";
    copy(a.begin(),a.end(), out); cout << endl;

    //удаление элементов со значением 0
    //итератор end() не перемещается, поэтому при выводе
    //отображаются лишние элементы
    vector<int>::iterator it;
    it = remove(a.begin(), a.end(),0);
    cout << "After remove 0: ";
    copy(a.begin(),a.end(), out); cout << endl;

    //перемещение итератора end() в новый конец контейнера
    cout << "After correct end: ";
    a.erase(it,a.end());
    copy(a.begin(),a.end(), out); cout << endl;

    return 0;
}

```

```

Initial state: 1 0 2 0 3 0 4
After remove 0: 1 2 3 4 3 0 4
After correct end: 1 2 3 4

```

ForwardIterator unique(ForwardIterator_First, ForwardIterator_Last)

удаляет из интервала идущие подряд повторяющиеся значения

ForwardIterator unique_copy(ForwardIterator_First, ForwardIterator_Last, OutputIterator_Dest)

копирует заданный интервал в выходной интервал, удаляя из интервала идущие подряд повторяющиеся значения

Возвращает новый логический конец интервала, т.е. итератор, указывающий на позицию за последним элементом, который не был удален.

Например: Удаление повторов из вектора

```

#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    int ar[] = {1,1,2,2,3,3,3 };
    vector<int> a(ar, ar + 6);
    cout << "Initial state: ";
    ostream_iterator<int> out(cout, " ");
    copy(a.begin(),a.end(), out); cout << endl;

    //удаление стоящих рядом повторов,
    //перемещение указател end() на новый конец вектора
    a.erase(unique(a.begin(),a.end()),a.end());
    cout << "After remove duplicates: ";
    copy(a.begin(),a.end(), out); cout << endl;
}

```

```

    return 0;
}

```

```

Initial state: 1 1 2 2 3 3
After remove duplicates: 1 2 3

```

Перестановочные алгоритмы

`void reverse(BidirectionalIterator _First, BidirectionalIterator _Last)`

переставляет элементы интервала в обратном порядке

`void reverse_copy(BidirectionalIterator _First, BidirectionalIterator _Last, OutputIterator _Dest)`

копирует элементы заданного интервала в выходной интервал, начиная с позиции `_Dest`, в обратном порядке

`void rotate(ForwardIterator _First, ForwardIterator _newBeg, ForwardIterator _Dest)`

выполняет циклический сдвиг элементов, после которого началом становится `_newBeg`

Пример: вывод вектора в обратном порядке, циклический сдвиг вектора на 1 элемент

```

#include <iostream>
#include <functional>
#include <iterator>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    int ar[] = {1,2,3,4,5,6 };
    vector<int> a(ar, ar + 6);
    cout << "Initial state: ";
    ostream_iterator<int> out(cout, " ");
    copy(a.begin(), a.end(), out); cout << endl;

    cout << "Reverse order: ";
    reverse_copy(a.begin(), a.end(), out); cout << endl;

    //циклический сдвиг на одну позицию
    rotate(a.begin(), a.begin() + 1, a.end());
    cout << "After rotate: ";
    copy(a.begin(), a.end(), out); cout << endl;

    return 0;
}

```

```

Initial state: 1 2 3 4 5 6
Reverse order: 6 5 4 3 2 1
After rotate: 2 3 4 5 6 1

```

`void random_shuffle(RandomAccessIterator _First, RandomAccessIterator _Last)`

переставляет элементы интервала в случайном порядке

Алгоритмы сортировки

Быстродействие $n \times \log(n)$, где n – количество элементов в контейнере

`void sort(RandomAccessIterator _First, RandomAccessIterator _Last)`

сортирует заданный диапазон элементов

`void sort(RandomAccessIterator _First, RandomAccessIterator _Last, BinaryPredicate op)`

сортирует заданный диапазон элементов, используя заданную функцию `op`, для попарного сравнения элементов

Алгоритм не может использоваться со списками, т.к. списки не поддерживают итератор произвольного доступа. Для выполнения сортировки списка в этом классе `list` имеется метод `sort()`.

Пример: сортировка массива по возрастанию, затем по убыванию

```

#include <iostream>
#include <iterator>
#include <algorithm>

```



```
using namespace std;

int main()
{
    int ar[] = {3,2,5,6,1,3,2,5,5,6};
    vector<int> v(ar,ar+10);
    ostream_iterator<int> out(cout," ");

    //сортировка по возрастанию
    sort(v.begin(),v.end());
    copy(v.begin(), v.end(), out); cout << endl;

    //сортировка по убыванию
    sort(v.begin(),v.end(), greater<int>());
    copy(v.begin(), v.end(), out); cout << endl;
    return 0;
}
```

```
1 2 2 3 3 5 5 5 6 6
6 6 5 5 5 3 3 2 2 1
```

Алгоритмы для упорядоченных интервалов

Работают с контейнерами, элементы в которых упорядочены. Вызов этих алгоритмов для неупорядоченных интервалов приводит к непредсказуемым последствиям. Как правило, эти алгоритмы эффективнее аналогичных алгоритмов, работающих с неупорядоченной последовательностью, т.к. имеют логарифмическую, а не линейную скорость.

Поиск элемента в упорядоченном контейнере

`bool binary_search(ForwardIterator _First, ForwardIterator _Last, const T& val)`
выполняет поиск значения `val` в заданном интервале методом бинарного поиска

Пример: поиск заданного числа в векторе, содержащем упорядоченные значения

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
    int ar[] = {3,2,5,6,1,3,2,5,5,6};
    vector<int> v(ar,ar+10);
    ostream_iterator<int> out(cout," ");

    copy(v.begin(), v.end(), out); cout << endl;
    int n;
    cout << "Number to search ";
    cin >> n;
    sort(v.begin(), v.end());
    if (binary_search(v.begin(), v.end(), n))
        cout << "The number " << n << " is present" << endl;
    else
        cout << "The number " << n << " is not present" << endl;

    return 0;
}
```

```
3 2 5 6 1 3 2 5 5 6
Number to search 4
The number 4 is not present
```

Суммирование 2 упорядоченных множеств

`OutputIterator merge(InputIterator _St1, InputIterator _Fn1, InputIterator _St2, InputIterator _Fn2 OutputIterator _Dest)`

записывает в приемный интервал все элементы, содержащиеся в интервалах $[_St1, _Fn1)$ и $[_St2, _Fn2)$

OutputIterator set_intersection(InputIterator_ $_St1$, InputIterator_ $_Fn1$, InputIterator_ $_St2$, InputIterator_ $_Fn2$
OutputIterator_ $_Dest$)

записывает в приемный интервал все элементы, которые входят одновременно в оба интервала $[_St1, _Fn1)$ и $[_St2, _Fn2)$

OutputIterator set_difference(InputIterator_ $_St1$, InputIterator_ $_Fn1$, InputIterator_ $_St2$, InputIterator_ $_Fn2$
OutputIterator_ $_Dest$)

записывает в приемный интервал все элементы, которые входят в интервал $[_St1, _Fn1)$, но не входят в интервал $[_St2, _Fn2)$

OutputIterator set_symmetric_difference(InputIterator_ $_St1$, InputIterator_ $_Fn1$, InputIterator_ $_St2$, InputIterator_ $_Fn2$ OutputIterator_ $_Dest$)

записывает в приемный интервал все элементы, которые входят либо в интервал $[_St1, _Fn1)$, либо в интервал $[_St2, _Fn2)$ (но не в оба сразу)

Во всех случаях в приемном интервале сохраняется порядок сортировки. Алгоритмы возвращают итератор, указывающий на элемент, следующий за последним вставленным.

Практические примеры использования функторов, предикатов, алгоритмов

Цель:

- рассмотреть примеры использования библиотеки STL для решения типовых задач

Пример 1. Имеется массив, заполненный случайными элементами. Необходимо записать в вектор 3 элемента этого массива с наибольшими значениями.

Для сортировки до тех пор, пока не будет получено требуемое количество отсортированных элементов, используется алгоритм `partial_sort_copy`. Применение копирующего алгоритма приводит к тому, что исходный массив не изменяется, а результат записывается в вектор. Для сортировки по убыванию в качестве функции сравнения передается функтор `greater<int>()`.

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <queue>

using namespace std;

int main()
{
    int ar[] = { 3,2,5,6,1,45,2,7,8,9 };
    vector<int> v(3);
    partial_sort_copy(ar,ar+10,v.begin(),v.end(),greater<int>());
    ostream_iterator<int> out(cout, " ");
    copy(v.begin(),v.end(),out);
    return 0;
}
```

45 9 8

Пример 2. Имеется вектор, заполненный случайными элементами. Необходимо найти количество повторов каждого значения и записать результат в карту, используя значение элемента в качестве ключа, а количество повторов в качестве значения.

Для решения задачи создается вспомогательный вектор (`v1`), в который копируются все элементы из исходного вектора без повторов. Для копирования без повторов элементы исходного вектора сортируются алгоритмом `sort()` и затем копируются алгоритмом `unique_copy()`. Затем выполняется цикл по элементам вспомогательного вектора и подсчитывается количество вхождения каждого элемента в исходный вектор. Для подсчета количества вхождений используется алгоритм `count()`.

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <queue>
#include <map>
using namespace std;

int main()
{
    int ar[] = {3,2,5,6,1,3,2,5,5,6};
    vector<int> v(ar,ar+10);

    vector<int> v1;
    vector<int>::iterator it;

    map<int,int> m;
    map<int,int>::iterator itm;

    for(it = v.begin(); it != v.end(); it++) cout << *it << " ";
    cout << endl;
}
```

```

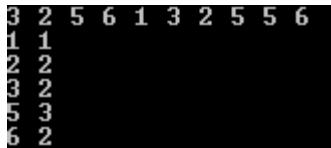
sort(v.begin(),v.end());

//копирование с удалением повторов
unique_copy(v.begin(),v.end(),back_inserter(v1));

//подсчет количества повторов и запись в map
for(it = v1.begin(); it != v1.end(); it++)
    m[*it] = count(v.begin(), v.end(), *it);

//вывод результатов
for(itm = m.begin(); itm != m.end(); itm++)
    cout << itm->first << " " << itm->second << endl;
return 0;
}

```



```

3 2 5 6 1 3 2 5 5 6
1 1 2 2 3 2 3 3 5 2

```

Пример 3. В игре необходимо реализовать возможность выполнения выстрелов по следующему алгоритму:

- источник выстрелов может перемещаться вдоль оси X, но не перемещается вдоль оси Y
- последовательно можно выполнять произвольное количество выстрелов
- в результате выстрела появляется пуля, которая должна перемещаться вдоль оси Y, позиция по оси X не изменяется
- необходимо хранить координаты всех пуль (для последующего отображения)
- когда пуля уходит за пределы экрана, ее необходимо удалять

Для реализации поставленной задачи разработаем класс fire.

Поставленная задача не предполагает добавление или извлечение элементов в произвольную позицию контейнера. Выстрелы добавляются в конец контейнера и извлекаются из его начала, поэтому наиболее подходящим контейнером для хранения выстрелов является контейнер queue.

Поскольку источник выстрелов не перемещается относительно оси Y, его положение можно хранить в переменной класса и передавать в конструктор. Также в классе хранится размер окна по вертикали, который используется для проверки выхода выстрела за пределы окна (в этом случае выстрел удаляется).

Метод AddShot() используется для добавления выстрела, в него передается координата выстрела по оси X. В методе создается объект типа POINT и помещается в конец очереди выстрелов.

Метод OneStep() выполняет сдвиг выстрелов в очереди и удаление выстрелов, вышедших за пределы окна. Для сдвига используется алгоритм transform, который вызывает функцию move для каждого элемента очереди. Функция move выполняет увеличение координаты выстрела по оси Y на 2 ед. Поскольку новые выстрелы добавляются в конец очереди и имеют значение координаты Y равное stY, выстрелы в очереди оказываются упорядоченными по убыванию по значению Y. Поэтому для удаления элементов, вышедших за пределы окна можно проверять элементы с начала очереди. Для получения элемента, находящегося в начале очереди используется метод front(), для удаления 1-ого элемента метод pop(). На первый взгляд кажется, что более простой вариант решения этой задачи – использование алгоритма remove_if(). Однако этот алгоритм будет просматривать все элементы в очереди, тогда как предложенный метод просматривает только первые элементы до тех пор, пока не найден элемент со значением Y < fnY.

Метод Print() выводит на экран очередь выстрелов. Алгоритм for_each() используется для вызова функции printShot() для каждого элемента очереди.

Для тестирования этого класса создается объект класса. В цикле с задержкой в 1 сек. выполняется сдвиг очереди и вывод ее на экран. Добавление выстрела выполняется по нажатию пробела.

```

#include <iostream>
#include <iterator>
#include <algorithm>
#include <queue>
#include <conio.h>
#include <windows.h>

using namespace std;

//класс для реализации движения выстрелов
class fire
{
private:

```

```

queue<POINT> q; //очередь для хранения координат выстрелов
int stY; //положение источника выстрелов по оси Y
int fnY; //размер окна по вертикали

//перемещение одного выстрела вдоль оси Y на 2 ед.
static POINT move(const POINT& p)
{
    POINT p1 = p;
    p1.y += 2;
    return p1;
}

//вывод координат выстрела на экран
static void printShot(const POINT& p)
{
    cout << "x = " << p.x << "\ty = " << p.y << endl;
}

public:

//кнструктор
//stY - положение источника выстрелов по оси Y
//fnY - горизонтальный размер экрана
fire(int stY, int fnY)
{
    this->stY = stY;
    this->fnY = fnY;
}

//добавление нового выстрела
//x - координата выстрела,
//y не передается, т.к. источник выстрелов не смещается по оси Y
void AddShot(int x)
{
    //создается точка с координатами выстрела и помещается в очередь
    POINT p = {x, stY};
    q.push(p);
}

//смещение всех выстрелов на 1 шаг
void OneStep()
{
    if (q.empty()) return;
    //алгоритм transform выполняет функцию move для всех элементов очереди
    //для получения итераторов begin(), end() выполняется обращение к
    //контейнеру (vector), который используется для реализации очереди
    transform(q.c.begin(), q.c.end(), q.c.begin(), move);
    //проверка выполняется для элементов начиная с начала очереди, т.к.
    //элементы в очереди упорядочены по убыванию Y
    //обязательно надо выполнять проверку !q.empty(), т.к. если
    //в результате удаления в очереди не останется элементов,
    //q.front() вернет неверную ссылку
    while(!q.empty() && (q.front().y > fnY)) q.pop();
}

//вывод всех выстрелов на экран
void Print()
{
    if (q.empty())
    {
        cout << "no shots" << endl;
        return;
    }
    //использование алгоритма for_each
    //для выполнения функции printShot для всех элементов контейнера
    for_each(q.c.begin(), q.c.end(), printShot);
}

};

```

```

//тестирование разработанного класса
int main()
{
    fire f(0,10);
    cout << "Space - add new shot" << endl;
    cout << "q - Exit" << endl;
    do
    {
        if (_kbhit())
        {
            char c = getch();
            if (c == 'q') break;
            if (c == 32) f.AddShot(rand()%20);
        }
        Sleep(1000);
        f.OneStep();
        cout << "fire: " << endl;
        f.Print();
    }
    while(true);
    return 0;
}

```

```

fire:
x = 1    y = 4
x = 7    y = 2
fire:
x = 1    y = 6
x = 7    y = 4
x = 14   y = 2
fire:
x = 1    y = 8
x = 7    y = 6
x = 14   y = 4
x = 0    y = 2
fire:
x = 1    y = 10
x = 7    y = 8
x = 14   y = 6
x = 0    y = 4
fire:
x = 7    y = 10
x = 14   y = 8
x = 0    y = 6
fire:
x = 14   y = 10
x = 0    y = 8

```