Лекция 11

КОНТЕЙНЕРЫ
ПОСЛЕДОВАТЕЛЬНЫЕ КОНТЕЙНЕРЫ. ИТЕРАТОРЫ.
АССОЦИАТИВНЫЕ КОНТЕЙНЕРЫ.
АЛГОРИТМЫ STL.

Контейнеры

Контейнер – это объект, содержащий другие объекты. В него можно добавлять объекты и удалять из него объекты.

Примерами контейнеров могут служить массивы, линейные списки или стеки.

Для каждого типа контейнера определены методы для работы с его элементами, не зависящие от конкретного типа данных, которые хранятся в контейнере, поэтому один и тот же вид контейнера можно использовать для хранения данных различных типов. Эта возможность реализована с помощью шаблонов классов, поэтому часть библиотеки C++, в которую входят контейнеры, а также алгоритмы и итераторы, называют **стандартной библиотекой шаблонов (STL – Standard Template Library).**

Контейнеры

Использование контейнеров позволяет значительно повысить надежность программ, их переносимость и универсальность, а также уменьшить сроки их разработки.

Стандартные контейнерные классы STD:

последовательные — обеспечивают хранение конечного количества однотипных величин в виде непрерывной последовательности; (векторы (**`vector`**), двусторонние очереди (**`deque`**) и списки (**`list`**), а также так называемые адаптеры (то есть варианты) контейнеров – стеки (**`stack`**), очереди (**`queue`**) и очереди с приоритетами (**`priority_queue`**). Каждый вид контейнера обеспечивает свой набор действий над данными.

ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу, построены на основе сбалансированных деревьев (словари (**`map`**), словари с дубликатами (**`multimap`**), множества (**`set`**), множества с дубликатами (**`multiset`**) и битовые множества (**`bitset`**)).

Последовательные контейнеры

- **`vector`** – динамический массив — структура, эффективно реализующая произвольный доступ к элементам, добавление в конец и удаление из конца.
- **`deque`** – двусторонняя очередь (дек) – эффективно реализует произвольный доступ к элементам, добавление в оба конца и удаление из обоих концов
- **`list`** – линейный список – эффективно реализует вставку и удаление элементов в произвольное место, но не имеет произвольного доступа к своим элементам
- **`stack`** – стек,
- **`queue`** – очередь,
- **`priority_queue`** – очередь с приоритетами;

vector

Объект, хранимый в **vector**, должен иметь:

- конструктор по умолчанию;
- операторы < и ==.

Возможности **vector**:

В классе вектор определены операторы сравнения **==, <, <=, !=, >, >=.**

Новые элементы могут включаться с помощью функций **insert**(), **push_back**(), **resize**(), **assign**().

Существующие элементы могут удаляться с помощью функций **erase**(), **pop_back**(), **resize**(), **clear**().

Доступ к отдельным элементам осуществляется с помощью итераторов **begin**(), **end**(), **rbegin**(), **rend**().

Манипулирование контейнером, сортировка, поиск в нем и тому подобное возможно с помощью глобальных функций, подключаемых заголовочным файлом <**algorithm**>

Примеры вызова конструкторов вектора

а) `vector <one> m1 (10);`

/* вектор из 10 объектов класса one (работает конструктор one без параметров) */

б) `vector <int> v1 (10, 1);`// вектор из 10 единичных элементов

в) `vector <int> v2 (v1.begin(), v1.begin() + 1);` /* вектор из двух элементов, равных первым двум элементам v1 */

г) `vector <int> v3 (v1);` // вектор, равный вектору v1

д) `vector <Wword> m2 (5, Wword("test"));` /* вектор из 5 объектов класса Wword заданным именем (работает конструктор с параметром char*)*/

Основные типы, используемые в шаблонах

value_type - тип элемента контейнера;
size_type - тип индексов элементов и т. д.;

iterator - тип "итератор" – указатель на элемент;
const_iterator - тип "константный итератор" – используется для неизменных данных;
reverse_iterator - обратный итератор – для просмотра от конца к началу;
const_reverse_iterator - константный обратный итератор;

key_type - тип ключа (для ассоциативных контейнеров);
key_compare - тип результата сравнения ключей.

reference - ссылка на элемент;
const_reference - константная ссылка на элемент;

Итераторы

Итераторы **(`iterators`)** – это объекты, которые по отношению к контейнеру играют роль указателей. Они позволяют получить доступ к содержимому контейнера и сканировать его элементы, примерно так же, как указатели используются для доступа к элементам массива. С итераторами можно работать так же, как с указателями. К ним можно применять операции *, инкремент, декремент. Тип итератора **`iterator`** определён в различных контейнерах.

Существует пять типов итераторов:

1. **Итераторы ввода (`input iterator`)** поддерживают операции равенства, разыменования и инкремента: **`==, !=, *i, ++i, i++, *i++`**. Специальным случаем итератора ввода является **`istream_iterator`**.

2. **Итераторы вывода (`output iterator`)** поддерживают операции разыменования, допустимые только с левой стороны присваивания, и инкремента: **`++i, i++, *i = t, *i++ = t`**. Специальным случаем итератора вывода является **`ostream_iterator`**.

Итераторы

3. **Однонаправленные итераторы (`forward iterator`)** поддерживают все операции итераторов ввода/вывода и позволяют без ограничения применять присваивание: **`==, !=, =, *i, ++i, i++, *i`**.

4. **Двунаправленные итераторы (`bidirectional iterator`)** обладают всеми свойствами **`forward-итераторов`**, а также имеют дополнительную операцию декремента (**`--i, i--, *i--`**), что позволяет им проходить контейнер в обоих направлениях.

5. **Итераторы произвольного доступа (`random access iterator`)** обладают всеми свойствами **`bidirectional-итераторов`**, а также поддерживают операции сравнения и адресной арифметики, то есть непосредственный доступ к элементу по индексу: **`i += n, i + n, i -= n, i - n, i1 - i2, i[n], i1 < i2, i1 <= i2, i1 > i2, i1 >= i2.`**

В STL также поддерживаются обратные итераторы (**`reverse iterators`**). Обратными итераторами могут быть либо двунаправленные итераторы, либо итераторы произвольного доступа, проходящие последовательность в обратном направлении.

Операции над итераторами

Категория итератора	Допустимые операции	Контей-неры
Входной (**input**) – перемещается вперед и допускает только чтение элементов	`x=*i,` `++i,i++`	все
Выходной (**output**) – перемещается вперед и допускает запись элементов	`*i = x,` `++i, i++`	все

Прямой (**forward**) – допускает запись и чтение элементов при движении вперед.
Реверсивный (**reverse**) – допускает запись и чтение элементов при движении назад.
	`x =*i, *i = x,`
	`++i, i++`
	`x =*i, *i = x,`
	`--i, i--`
		все

Двунаправленный (**bidirectional**) – работает вперед и назад	`x=*i,` `++i,i++,--i,i--`	все

Произвольного доступа (**random access**)	`x = *i, *i = x,`
	`--i, i--`
	`i + n, i - n,`
	`i += n, i -= n`
	`i < j, i > j,`
	`i <= j, i >= j`
		все, кроме **list**

Методы получения адресов для инициализации итераторов

Указатель на первый:

```
iterator begin()
const_iterator begin() const
```

Указатель на следующий за последним:

```
iterator end()
const_iterator end() const
```

Указатель на первый при обратном просмотре:

```
reverse_iterator rbegin()
const_reverse_iterator rbegin() const
```

Указатель на элемент, следующий за последним, при обратном просмотре:

```
reverse_iterator rend()
const_reverse_iterator rend() const
```

Методы доступа к элементам контейнеров STL

Метод	Описание
`front()`	ссылка на первый элемент
`back()`	ссылка на последний элемент
`operator [](i)`	доступ по индексу без проверки
`at(i)`	доступ по индексу с проверкой

Методы для включения и исключения элементов

Операция	Метод	vector	deque	list
Вставка в начало	**push_front()**	-	+	+
Удаление из начала	**pop_front()**	-	+	+
Вставка в конец	**push_back()**	+	+	+
Удаление из конца	**pop_back()**	+	+	+
Вставка в произвольное место	**insert()**	+	+	+
Удаление из произвольного места	**erase()**	+	+	+
Произвольный доступ	**[ ], at**	+	+	-

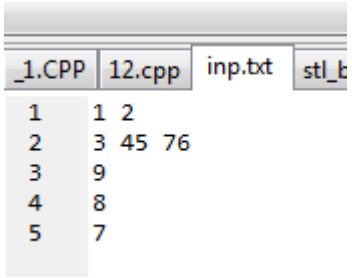
Другие операции с контейнерами STL

size() - количество элементов
empty() - определяет, пуст ли контейнер
capacity() - память, выделенная под вектор (только для векторов)
reserve(n) - выделяет память для контейнера под n элементов
resize(n) - изменяет размер контейнера (только для векторов, списков и очередей с двумя концами)
swap(x) - обмен местами двух контейнеров
==, !=, < - операции сравнения
operator =(x) - контейнеру присваиваются элементы контейнера **x**
assign(n, x) - присваивание контейнеру **n** копий элементов **x** (не для ассоциативных контейнеров)
assign(first, last) - присваивание элементов из диапазона **[first:last]**
operator [](k) - доступ к элементу с ключом **k**
find(k) - находит элемент с ключом **k**
lower_bound(k) - находит первый элемент с ключом, меньшим **k**
upper_bound(k) - находит первый элемент с ключом, большим **k**
equal_range(k) - находит **lower_bound** (нижнюю границу) и **upper_bound** (верхнюю границу) элементов с ключом **k**

Пример использования

```
#include <fstream>
#include <vector>
using namespace std;
void main()
{
   ifstream in ("inp.txt");
   vector<int> v; // инстанцирование шаблона

  int x;
  while ( in >> x )
      v.push_back(x); // добавление элемента в конец
   for (vector<int>::iterator i = v.begin();  i !=
   v.end(); ++i)
                  cout << *i << " ";
   system("pause");
 }
```




```
1 2 3 45 76 9 8 7
```

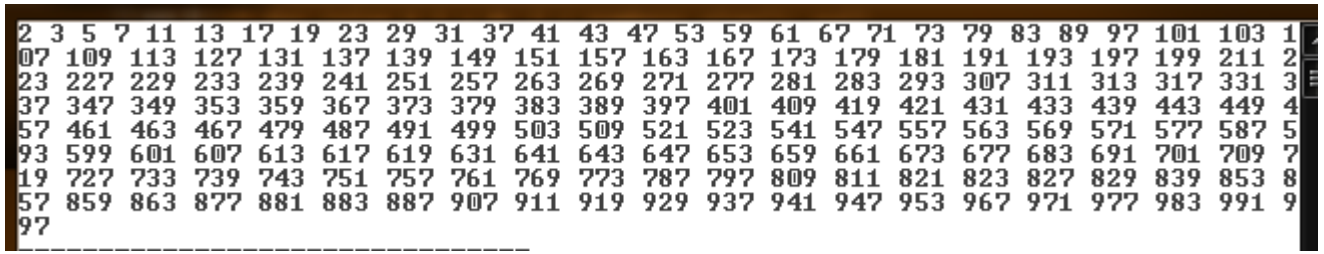
Пример использования

```
#include <iostream>
#include <vector>
using namespace std;
int main () {
unsigned limit = 1000;
 vector <unsigned> numbers;
 numbers.reserve(limit - 1);
 for (unsigned i = 2; i < limit; ++i)
        numbers.push_back(i);
for (size_t i = 0; i < numbers.size(); ++i)  {
      size_t j = i + 1;
      while (j < numbers.size()) {
       if (numbers[j] % numbers[i] == 0)

   numbers.erase((vector<unsigned>::iterator)&numbers[j]);
        else
            ++j;
      }
   }
```

Пример использования

```
    for (vector<unsigned>::iterator i = numbers.begin();
                                        i != numbers.end();
   ++i)
        cout << *i << " ";
numbers.clear();
}
// обработка 2 вариант
for (vector<unsigned>::iterator i = numbers.begin(); i !=
   numbers.end(); ++i)
    {
      vector<unsigned>::iterator j = i + 1;
      while (j != numbers.end()) {
       if (*j % *i == 0)
          numbers.erase(j);
        else
            ++j;
      }
   }
```



Ассоциативные контейнеры

- **map** – словарь уникальных ключей,
- **multimap** – словарь ключей с дубликатами,
- **set** - множество,
- **multiset** – мультиножество,
- **bitset** – битовое множество (набор битов).

Словари часто называют также *ассоциативными массивами* или *отображениями*.

Словарь построен на основе пар значений, первое из которых представляет собой ключ для идентификации элемента, а второе – собственно элемент.

map – это последовательность пар {ключ, значение}, которая обеспечивает быстрое получение значения по ключу. Контейнер map предоставляет двунаправленные итераторы.

Контейнер **map** требует, чтобы для типов ключа существовала операция <. Он хранит свои элементы отсортированными по ключу так, что перебор происходит по порядку.

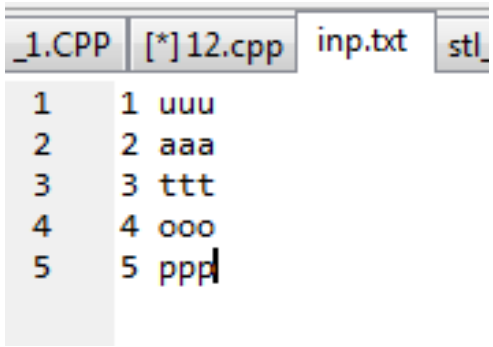
Компонентная функция **find()** ищет элемент контейнера по заданному ключу.

Пример работы с ассоциативным контейнером

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
typedef map <string, long, less <string> > map_sl;
int main()
{  map_sl m;     string str;   long num;
   ifstream in("inp.txt");
   while (in >> num)
   {
      in.get();
      getline(in, str);
      m[str] = num;
      cout << str << "   " << num << endl;
   }
```

Пример работы с ассоциативным контейнером

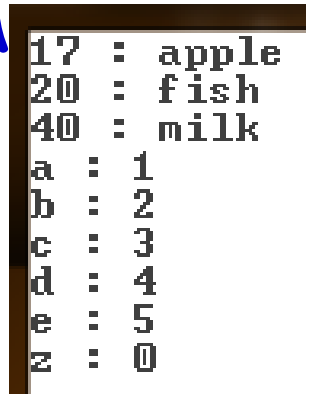
```
    // Вывод элемента по ключу
    cout << "Name: " << endl;
    getline(cin, str); // ввод ключа
    if (m.find(str) != m.end()) // поиск значения по
    ключу
       cout << m[str]<<endl;
    else
       cout << "Error" ;
    system("pause");
}
```



Работа с ассоциативным массивом map

```
#include <iostream>
#include <string>
#include <map> //подключили библиотеку для работы с map
using namespace std;
int main() {
//Сделаем и инициализируем map
map <int,string> myFirstMap;
myFirstMap.insert(pair<int, string>(17, "apple"));
myFirstMap.insert(pair<int, string>(40, "milk"));
myFirstMap.insert(pair<int, string>(17, "bread"));//не вставится!
myFirstMap.insert( pair<int, string>(20, "fish"));
for (map <int,string>::iterator it = myFirstMap.begin(); it
   myFirstMap.end(); it++)
cout << it->first << " : " << it->second << endl;
```

Работа с ассоциативным массивом




```
//Ещё один map
char c='a';
map <char, int> mySecondMap;
for (int i = 0; i < 5; ++i,++c)
mySecondMap.insert ( pair<char, int>(c,i+1));
mySecondMap.insert ( pair<char, int>('d',5) ); //не вставится!
mySecondMap.insert(mySecondMap.begin(), make_pair('e',1));
//так тоже не вставится
mySecondMap.insert(mySecondMap.begin(), make_pair('z',0)
   );
//а так можно
for (map <char, int>::iterator it = mySecondMap.begin();
   it != mySecondMap.end(); it++)
cout << (*it).first << " : " << (*it).second << endl;
cin.get(); return 0;
}
```

Работа с множеством set

```
#include <iostream>
#include <string>
#include <set>   //заголовочный файл множеств и мультимножеств
#include <iterator>
using namespace std;
int main() {
set <char> mySet;
string str("Abracadabra");
for (int i=0; i<str.length(); i++) mySet.insert(str[i]);
copy(mySet.begin(), mySet.end(),
   ostream_iterator<char>(cout," "));
cin.get(); return 0;
}
```

Алгоритмы STL

Алгоритмы **(algorithms)** выполняют операции над содержимым контейнера. Существуют алгоритмы для инициализации, просмотра, сортировки, поиска, замены содержимого контейнеров.

Каждый алгоритм выражается шаблоном функции или набором шаблонов функций. Таким образом, один и тот же алгоритм может работать с разными контейнерами, содержащими значения разнообразных типов. Алгоритмы, которые возвращают итератор, как правило, для сообщения о неудаче используют конец входной последовательности. Алгоритмы не выполняют проверки диапазона на их входе и выходе. Когда алгоритм возвращает итератор, это будет итератор того же типа, что и был на входе.

Алгоритмы определены в заголовочном файле <**algorithm**>, приведём имена некоторых наиболее распространённых функций-алгоритмов.

Алгоритмы STL

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

Пример

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
void myfunction (int i) { cout << ' ' << i; }
int main () {
vector<int> myvector;
myvector.push_back(10);
myvector.push_back(20);
myvector.push_back(30);
for_each (myvector.begin(), myvector.end(), myfunction);
cin.get(); return 0;
}
```

```
10 20 30
```