Лекция 14.

Стандартная библиотека шаблонов STL. Контейнеры последовательностей.

Литература по STL:

1) *Д. Мюссер, Ж. Дердж, А. Сейни,* C++ и STL: справочное руководство, 2-е изд. (серия C++ in Depth).: Пер. с англ. — М.: Вильямс, 2010. — 432 с., илл.

2) *С. Мейерс,* Эффективное использование STL. Библиотека программиста. — М.: Питер, 2002. — 224 с., илл.

3) *Л. Аммерааль,* STL для программистов на C++: Пер. с англ. — М.: ДМК, 1999. — 240 с., илл.

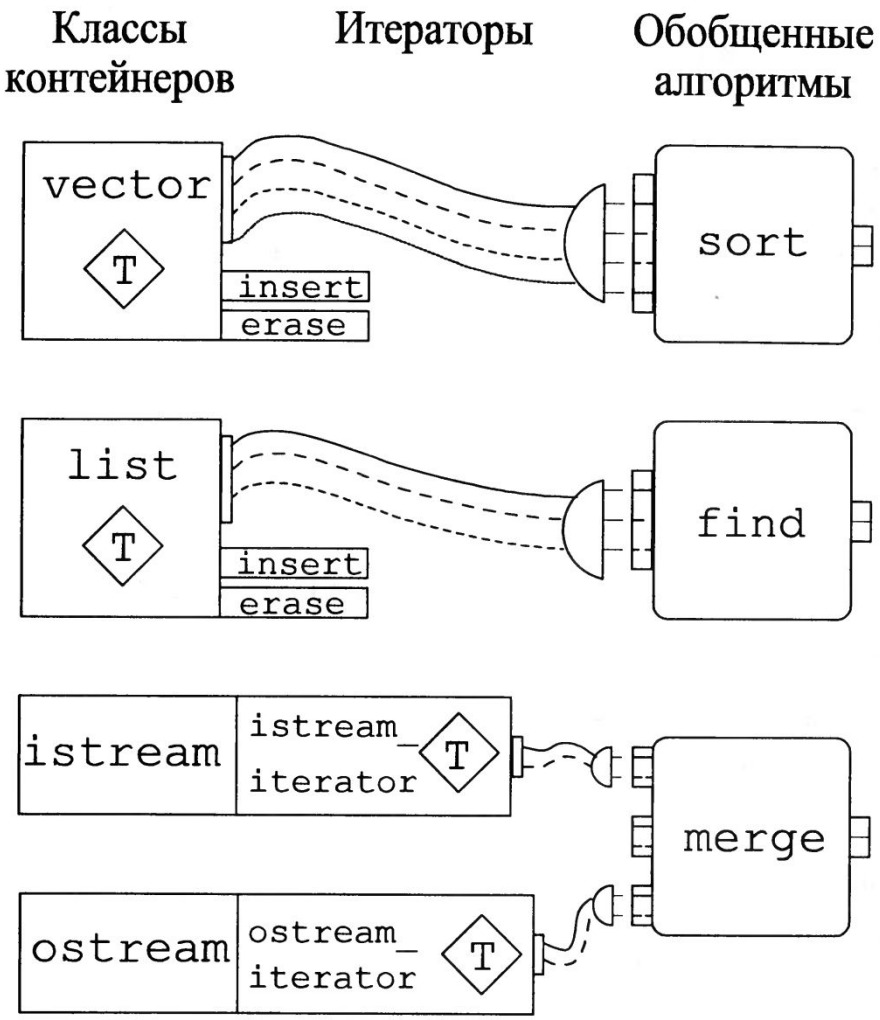
Стандартная библиотека шаблонов - Standard Template Library (STL)

- является частью стандартной библиотеки языка C++
- предоставляет набор классов-контейнеров
- предоставляет набор фундаментальных алгоритмов
- использует методы обобщенного программирования
- большинство компонентов реализованы через шаблоны C++

Главные особенности библиотеки STL:

1) высокая степень адаптивности компонентов друг к другу
2) высокая эффективность (скорость) работы алгоритмов

Взаимодействие между основными компонентами STL



Не всякий алгоритм может работать с любым итератором!
Совместимость алгоритмов и контейнеров обеспечивается за счет контроля типов C++.

Компоненты STL – контейнеры последовательностей.

- **vector<T>.** Вектор, динамический массив с произвольным доступом к элементам (время доступа O(1)). Обеспечивает вставку и удаление элементов с конца последовательности с амортизированным O(1). Элементы вектора хранятся с памяти последовательно.
- **deque<T>.** Дек, аналогичен вектору, но с возможностями вставки и удаления элементов с обоих концов последовательности. Доступ к элементу — O(1), вставка/удаление — амортизированное O(1). Реализован в виде двусвязного списка линейных массивов.
- **list<T>.** Двусвязный список, элементы которого хранятся в различных участках памяти. Время доступа к элементам — линейное (0[N), где N — текущее число элементов), время вставки/удаления — O[1] в любом месте последовательности.

Обычный массив a[N] также считается контейнером последовательности, и для него работают все алгоритмы STL. Кроме того, строковый тип string (из заголовочного файла <string>) совместим с алгоритмами STL.

Пример: работа с векторами STL

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

int main()
{
   vector<int> v; // создаем вектор нулевой длины
   int i;

   // Отображаем исходный размер объекта v
   cout << "размер = " << v.size() << endl;

   // Помещаем значения в конец вектора:
   // в результате вектор будет расти
   for(i=0; i<10; i++) v.push_back(i);

   // Отображаем текущий размер объекта v.
   cout << "размер сейчас = " << v.size() << endl;
```

```
   // Доступ к содержимому вектора
   // можно получить, используя индексы
   for(i=0; i<10; i++)
      cout << v[i] << " ";
   cout << endl;

   // Доступ к первому и последнему
   // элементам вектора.
   cout << "первый = " << v.front() << endl;
   cout << "последний = " << v.back() << endl;

   // Доступ через итератор.
   vector<int>:: iterator p = v.begin();
   while(p != v.end())
   {
      cout << *p << " " ;
      p++;
   }
   return 0;
}
```

Обзор функций-членов класса *vector*

```
iterator begin();
iterator end();
void push_back(const T& x);
reverse_iterator rbegin();
reverse_iterator rend();
```

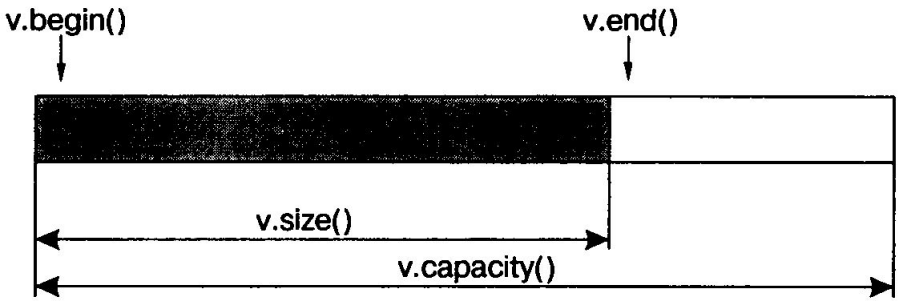
базовые функции для вставки элемента и последовательного обхода элементов в прямом и обратном порядке

```
const_iterator begin() const;
const_iterator end() const;
const_reverse_iterator rbegin() const;
const_reverse_iterator rend() const;
```

обход элементов в режиме "только чтение"

```
size_type size() const;
size_type capacity() const;
void reserve(size_type n);
size_type max_size() const;
bool empty() const;
```

размер и емкость вектора (см. рис.)



```
vector();
vector(size_type n, const T& value = T());
vector(const vector<T>& x);
vector(const_iterator first, const_iterator last);
~vector();
```

конструкторы и деструктор

Примеры использования:

```
vector<int> v;              // (1) Конструктор по умолчанию
vector<int> w(5, -3);       // (2) Создает 5 элементов,
                            //     все они равны -3
vector<int> w(5);
vector<int> w1(w);          // (2) Создает 5 элементов

vector<int> w2(w.begin()+1, w.begin() + 5);
                            // (4) Копирует четыре элемента
                            //     из w в w2
```

```
reference operator[](size_type n);
const_reference operator[](size_type n) const;
```

доступ к элементам по индексу

Пример:

```
w[3] = 3 * w[2] + 1;
```

```
vector<T>& operator=(const vector<T>& x);
```

оператор присваивания

Пример:

```
vector<int> w(5, -3), v;
v = w;
```

```
reference front();
reference back();
const_reference front() const;
const_reference back() const;
```

доступ к первому и последнему элементам

Пример:

```
v.front() = 1000;
cout << v.front() << " " << v.back() << " ";
```

```
void swap(vector<T>& x);
```

меняет местами содержимое двух векторов одного типа

Примеры:

```
v.swap(w);
w.swap(v);
```

Функции вставки и удаления элементов. Занимают время O(n), где n - номер позиции для вставки/удаления.

```
iterator insert(iterator position, const T& x);
```

Пример:

```
vector<int> v;
for (int k=10; k<15; k++) v.push_back(k);
vector<int>::iterator i = v.begin() + 1, j;
j = v.insert(i, 123);
```

вставка одного элемента в позицию position

```
void insert(iterator position, const_iterator first,
   const_iterator last);

void insert (iterator position,
             size_type n, const T& x);
```

вставка нескольких элементов

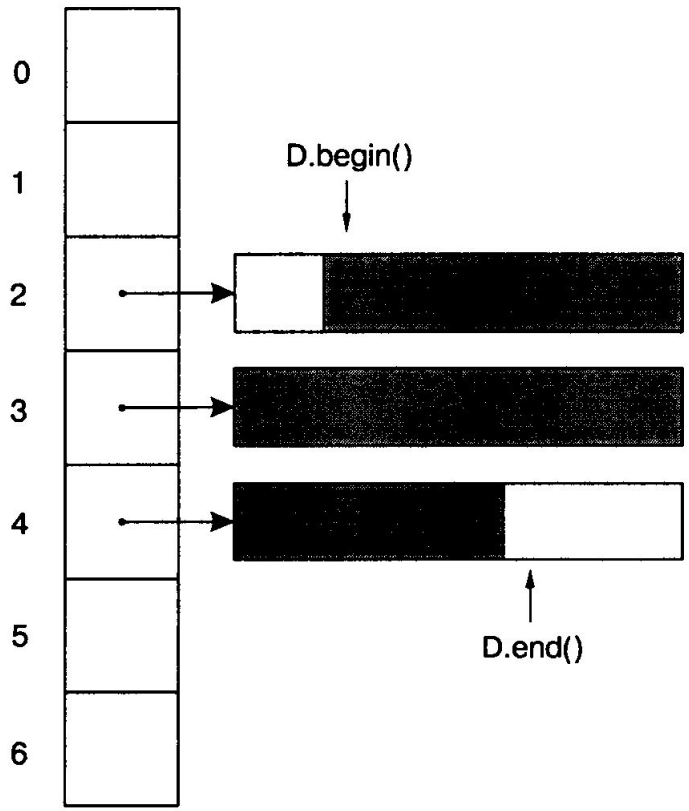
```
void pop_back();
void erase(iterator position);
void erase(iterator first, iterator last);
```

удаление элементов по одному и группой

Дополнительные функции-члены класса *deque*

Большинство функций дека повторяет соответствующие функции вектора (см. выше). Далее мы рассмотрим только *специфические* для дека функции.

К примеру, дек поддерживает вставку элементов не только в конец, но и в начало очереди (за константное время). С другой стороны, функции capacity и reserve для дека не определены.



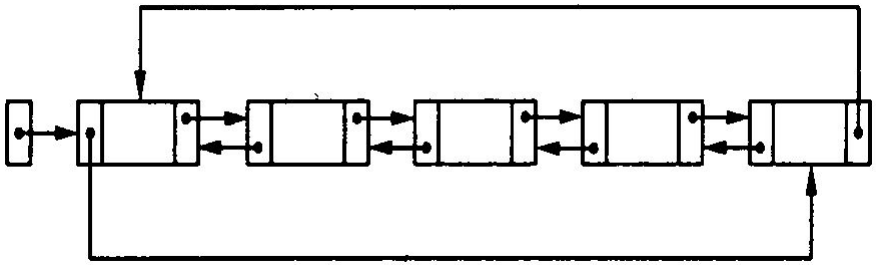

```
void push_front(const T& x);
void pop_front();
reference front();
```

Пример:

```
v.back() = D.front();
D.pop_front();
```

Дополнительные функции-члены класса *list*

Связанный список (list) обеспечивает вставку/удаление элементов в любой позиции за константное время. Однако доступ к элементу требует O(n) операций. Как следствие, для списка не реализован оператор доступа по индексу [].



```
void sort();
void unique();
```

Пример:

```
Initial contents: 123 123 123 123 123 100 123 123
After L.unique(): 123 100 123
After L.sort():   100 123 123
```

```
int main()
{  list<int> L(5, 123);
   L.push_back(100);
   L.push_back(123);
   L.push_back(123);
   out("Initial contents: ", L);
   L.unique();
   out("After L.unique(): ", L);
   L.sort();
   out("After L.sort():   ", L);
   return 0;
}
```

Функции сцепки списков (splicing) - перемещение одного или более элементов из одного списка в другой.

```
void splice(iterator position, list<T>& x);
```

Вставляет содержимое списка x в текущий список перед позицией position, и оставляет список x пустым.

```
void splice(iterator position, list<T>& x, iterator j);
```

Вставляет элемент из списка x (в позиции j) в текущий список (перед позицией position).

```
void splice(iterator position, list<T>& x,
      iterator first, iterator last);
```

Вставляет элементы диапазона [first, last] из списка x в текущий список (перед позицией position).

Пример:

```
L.splice(i, M, j1, j2);
```

Адаптер стека. Функции-члены класса *stack*

Стек представляет собой структуру данных, допускающую только две операции, изменяющие ее размер: 1) добавление элемента в конец последовательности, 2) извлечение элемента из конца последовательности.

В STL стек реализуется на основе вектора, дека или связанного списка. Таким образом, стек – не новый тип контейнера. Он реализован в виде адаптера к существующим контейнерам vector, deque или list. Если при создании стека тип используемого контейнера не указан, то по умолчанию используется дек.

Для стека определены следующие функции:

`type top();`	Значение верхнего элемента в стеке
`void push(type);`	Поместить элемент в стека
`void pop();`	Извлечь элемент из стека
`int size();`	Количество элементов в стеке
`bool empty();`	Стек пуст?

Пример:

```
// stackcmp.cpp: Сравнение и присваивание для стек
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

int main()
{  stack <int, vector<int> > S, T, U;
   S.push(10); S.push(20); S.push(30);
   cout << "Pushed onto S: 10 20 30\n";
   T = S;
   cout << "After T = S; we have ";
   cout << (S == T ? "S == T" : "S != T") << endl;
   U.push(10); U.push(21);
   cout << "Pushed onto U: 10 21\n";
   cout << "We now have ";
   cout << (S < U ? "S < U" : "S >= U") << endl;
   return 0;
}
```

```
Pushed onto S: 10 20 30
After T = S; we have S == T
Pushed onto U: 10 21
We now have S < U
```

Адаптер очереди. Функции-члены класса *queue*

Очередь представляет собой структуру данных, в которой новый элемент добавляется в конец последовательности, а извлечение элемента происходит из начальной позиции.

В STL очередь реализуется на основе дека или связанного списка, то есть как адаптер к контейнерам deque или list. Если при создании очереди тип контейнера не указан, то по умолчанию используется дек.

Для очереди определены следующие функции:

`type front();`	Значение первого элемента в очереди
`type back();`	Значение последнего элемента в очереди
`void push(type);`	Поместить элемент в очередь
`void pop();`	Удалить элемент из очереди
`int size();`	Количество элементов
`bool empty();`	Очередь пуста?

Пример:

```
// queue.cpp:  Использование очереди; демонстрация
//             функций-членов push, pop, back и front.

#include <iostream>
#include <list>
#include <queue>

using namespace std;

int main()
{  queue <int, list<int> > Q;
   Q.push(10); Q.push(20); Q.push(30);
   cout << "After pushing 10, 20 and 30:\n";
   cout << "Q.front() = " << Q.front() << endl;
   cout << "Q.back()  = " << Q.back() << endl;
   Q.pop();
   cout << "After Q.pop():\n";
   cout << "Q.front() = " << Q.front() << endl;
   return 0;
}
```

```
After pushing 10, 20 and 30:
Q.front() = 10
Q.back()  = 30
After Q.pop():
Q.front() = 20
```

Адаптер очереди с приоритетом. Функции-члены класса *priority_queue*

Очередь с приоритетом является структурой данных, из которой можно удалить только наибольший элемент (т.е. элемент с наибольшим приоритетом).

В STL очередь с приоритетом реализуется на основе вектора или дека, как адаптер к контейнерам vector или deque. По умолчанию используется дек.

Определены следующие функции:

`type top();`	Значение верхнего элемента
`void push(type);`	Поместить элемент
`void pop();`	Удалить элемент
`int size();`	Количество элементов
`bool empty();`	Очередь пуста?

Пример:

```
// prqueue.cpp: Очередь с приоритетами; программа
//              демонстрирует функции-члены push, pop, empty и top.

#include <iostream>
#include <vector>
#include <functional>
#include <queue>

using namespace std;

int main()
{  priority_queue <int, vector<int>, less<int> > P;
   int x;
   P.push(123); P.push(51); P.push(1000); P.push(17);
   while (!P.empty())
   {  x = P.top();
      cout << "Retrieved element: " << x << endl;
      P.pop();
   }
   return 0;
}
```

```
Retrieved element: 1000
Retrieved element: 123
Retrieved element: 51
Retrieved element: 17
```