Тема 2.2.

Основные контейнеры vector и deque

Подготовила Пономарёва Анастасия
Группа 271 ПИ

Структура презентации:

- Стандартный контейнер **Vector**
- Стандартный контейнер **Deque**
- Различия между вектором и очередью
- Выбор между вектором и очередью
- Использованная литература

Vector 1

- Вектор – один из типов последовательных контейнеров.

- Векторы реализуются, как динамические массивы. Такие контейнеры хранят элементы в соседних ячейках памяти, что позволяет осуществлять доступ к его элементам не только с помощью итераторов, но с помощью смещения указателей элементов.
- Векторы занимаются автоматическим размещением себя в памяти, расширением и сужением своего размера по мере вставки и удаления элементов.

- Векторы удобно использовать, когда необходимо:

- осуществить доступ к отдельным элементам с помощью индекса;
- использовать итерацию для любых элементов;
- добавлять и удалять элементы в конце вектора.

Размещение вектора в памяти

Vector 2

- Векторы по сравнению с очередями, например, более эффективны по времени с точки зрения доступа к элементам и их добавления/удаления в конце контейнера.

- Изнутри, векторы, как и остальные контейнеры имеют размер –количество элементов в векторе. Также они обладают определенным объемом, который является количеством занимаемой памяти и может быть равным или большим фактического размера.

Vector 3 Варианты объявления

```
#include <vector>

  using namespace std;
  int main(int argc, char* argv[])
{
  vector<int> v1; // вектор с членами типа int
  vector<double> v2; // вектор с членами типа double
  vector<bool> v3; // вектор с членами типа bool
  vector<string> v4; // вектор с членами типа string
    return EXIT_SUCCESS;
}

vector<vector<int> > matrix; // вектор векторов
```

Vector 4 Варианты инициализации

```
#include <vector>
using namespace std;
int main(int argc, char* argv[])
{
vector<int> v1; // initially empty
vector<int> v2(5); // 5 elements, initialized to 0
vector<int> v3(10, 1); // 10 elements, initialized to 1
vector<int> v4(v3); // v4 is a copy of v3
return EXIT_SUCCESS;
}
```

Vector 5

- **(constructor)** Конструктор вектора (public member function) **(destructor)** Деструктор вектора(public member function) **operator=** Копирование содержимого вектора(public member function)
- **Итераторы**:

begin возвращает итератор начала(public member type)
end возвращает итератор конца (public member function)
rbegin возвращает противоположный итератор для противоположного конца (public member function)
rend возвращает противоположный итератор для противоположного конца (public member function);

- **Емкость**:

size возвращает размер вектора (public member function)
max_size максимальный размер (public member function)
resize изменение размера(public member function)
capacity возвращает объем занимаемой памяти (public member function)
empty проверяет, не пустой ли вектор (public member function)
reserve запрос об изменении объема(public member function)

Vector 6

- **Доступ к элементам:**

operator[] доступ к элементам(public member function)
at доступ к элементу (public member function)
front доступ к первому элементу (public member function)
back доступ к последнему элементу (public member function)

- **Модификаторы**:

assign присвоить содержание вектора(public member function)
push_back добавить элемент в конец(public member function)
pop_back удалить последний элемент (public member function)
insert добавить элемент(public member function)
erase стереть элементы (public member function)
swap обменять содержимое контейнера (public member function)
clear очистить содержимое контейнера (public member function)

- **Распределение**:

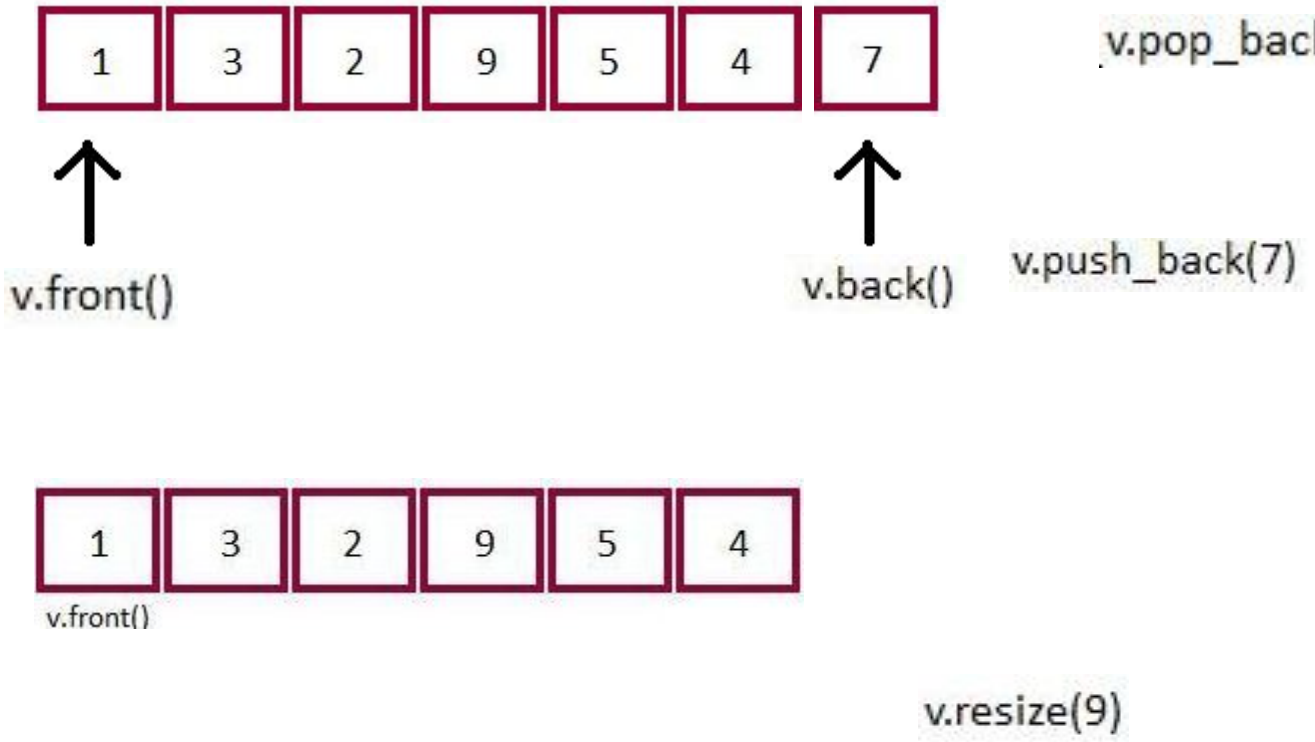
- **get_allocator** вернуть модель распределения памяти (public member function)

Vector 7 Примеры

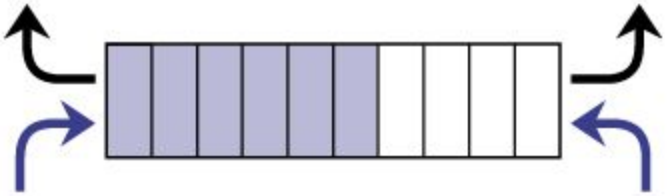
```
// Пустой вектор
vector<int> v;
// Добавление элементов в конец
for (int i = 0; i < 5; i++) {
v.push_back(i);
cout << "Size: " << v.size() << endl;
}
// Удаление элементов из вектора
for (int j = 0; j < 5; j++) {
v.pop_back();
cout << "Size: " << v.size() << endl;
}
cout << endl << v.empty() << endl;


//Варианты доступа к элементам:
        vector<int> v(12);
        v[1] = 3;
        v.at(2) = 90;
          v.front() = v.back();
```

```
// Доступ к элементам с помощью
   итераторов
vector<int>::iterator it;
for (it = v.begin(); it != v.end(); it++) {
cout << *it << endl;

}


// Вставить элементы в начало
v.insert(v.begin(), 70);
cout << v[0] << endl;
cout << v.size() << endl;

// Удалить 6 последних элементов
vector<int>::iterator it = v.end() - 6;
v.erase(it, v.end());
cout << v.size() << endl;
```

Vector 8 Механизм работы некоторых функций для вектора



Deque 1



- Deque – акроним для double-ended queue.
- Очереди с двухсторонним доступом представляют собой разновидность последовательных контейнеров и чем-то схожи с векторами и связными списками.
- Как и вектор, это тип контейнера поддерживает произвольный доступ к элементам.
- Если говорить о размещении в памяти, то двухсторонние очереди могут размещаться в нескольких сегментах памяти, не обязательно в смежных. Это приводит к снижению скорости доступа к элементам.

Deque 2

- Объявление такого типа очередей может быть осуществлено с помощью различных библиотек, но в любом случае они обеспечивают доступ к элементам через произвольную выборку итератора.

- Очереди с двухсторонним доступом имеют следующие свойства:
- Отдельные элементы могут быть доступны через их индексы;
- Итераторы элементов могут быть реализованы в любом порядке;
- Элементы могут быть успешно добавлены или удалены с обоих концов очереди.

В сущности этот тип контейнеров содержит тот же набор функций для членов последовательности, что и вектор, поэтому список методов не привожу второй раз.

Deque 3 Объявление

- deque(); // пустая deque
- deque(size_type size); // deque размера size
- deque(size_type num, const TYPE &val); //с параметрами num иval - num копий величины val в deque
- deque(input_iteratordeque(input_iterator start, input_iterator end); //С параметрами start иend – заполнение deque элементами между значением начала (start) до конца(end).

Deque 4

assign

- *Syntax:* void assign(input_iterator void assign(input_iterator start, input_iterator end);
- void assign(Size num, const TYPE &val);
- Объявление очереди с элементами между итераторами начала и конца.

at

- *Syntax:* reference at(size_type pos);
- Возвращает ссылку на элемент по указанной позиции pos.

back

- *Syntax:* reference back();
- Возвращает ссылку на конец очереди.

begin

- *Syntax:* iterator begin();
- Возвращает итератор первого элемента очереди.

clear

- *Syntax:* void clear();
- Удаление всех элементов из очереди.

empty

- *Syntax:* bool empty();
- Возвращает true, если очередь не содержит элементов. В противном случае возвращает false.

end

- *Syntax:* iterator end();
- Возвращает итератор последнего элемента очереди.

erase

- *Syntax:* iterator iterator erase(iterator pos);
- Функция стирает элемент на указанной позиции.

front

- *Syntax:* reference front();
- Возвращает ссылку на начало очереди.

Deque 5

insert

- *Syntax:* iterator iterator insert(iterator iterator insert(iterator pos, size_type num, const TYPE &val);
- Вставка *num* копий величины*val* после элемента на позиции *pos.*

max_size

- *Syntax:* size_type max_size();
- Возвращает максимальное число элементов, которые может содержать очередь.

pop_back

- *Syntax:* void pop_back();
- Удаление последнего элемента очереди.

pop_front

- *Syntax:* void pop_front();
- Удаление первого элемента очереди.

push_back

- *Syntax:* void push_back(const TYPE &val);
- Добавление элемента в конец очереди.

push_front

- *Syntax:* void push_front(const TYPE &val);
- Добавление элемента в начало очереди.

rbegin

- *Syntax:* reverse_iterator rbegin();
- Возвращает обратный итератор на конец очереди.

rend

- *Syntax:* reverse_iterator rend();
- Возвращает обратный итератор на начало очереди.

resize

- *Syntax:* void resize(size_type num, TYPE val);
- Изменени количества элементов очереди на *num*, добавляя в очередь элементы, равные v*al*.

size

- *Syntax:* size_type size();
- Возвращает размер очереди.

swap

- *Syntax:* void swap(deque &target);
- Замена элементов на double.

Deque 6 Пример

```
// очередь с десятью '1' в ней
deque<int> dq( 10, 1 );

//создание итератора очереди
 <int>::iterator iter;

// добавляет в конец
 dq.push_back (8);

// добавляет в начало
 dq.push_front (9);

// удаляет первый элемент
 dq.pop_front ();

// вывод элементов очереди
  for( iter = dq.begin(); iter != dq.end(); iter++ )
 { cout << *iter << endl; }

 //удаление всех элементов из очереди
 dq.clear();
```

Vector vs Deque: Различия

- Vector и Deque очень схожи, но существует несколько различий:

  1) в deque можно добавлять элементы в начало.

  2) Главное отличие: размещение в памяти(!)

- vector всегда будет размещаться в памяти последовательно. При добавлении элементов объем памяти увеличивается в 2 раза (принято в Visual C++ 6.0). Этот запас необходим. Когда есть запасная память, элемент пишется туда. Когда ее нет, выделяется непрерывный кусок памяти достаточный для n*K элементов, где K – коэффицент, который должен быть меньше, чем (1+sqrt(5))/2. В него копируются предыдущие n, добавляется наш новый элемент, старый кусок размером n освобождается. Если бы запаса не было, то память бы выделялась каждый раз при добавлении нового элемента, что неэффективно. Из-за последовательного размещения в памяти произвольный доступ к элементам очень быстрый.
- deque размещается в памяти сегментированно и обычно реализуется как массив массивов. Поэтому доступ к элементам будет помедленнее. Но зато при добавлении элементов не будет возникать тех проблем, что имеют место у vector с выделением последовательных кусков памяти.

Vector vs Deque

- 1. **deque** выполняет функции insert() and erase() за постоянное 0 (1) время в начале контейнера,а вектор – нет. Вектор выполняет функции capacity(), back(), clear(), destroy(), value(), pop_back().

- 2. **deque** проще в использовании и более эффективен для добавления новых элементов, чем **vector**. Есть только две функции, которые определены для **vector** и не определены для **deque** -capacity() and reserve().

- 3. Из-за страничной организации памяти deque реализует быстрее vector'a оператор push_back()

- 4. Функция выделения дополнительной памяти reserve() обоими контейнерами при фиксированном количестве элементов выполняется приблизительно за одно время. Был проведен эксперимент по вставке в deque и vector 9874 элементов средней длины1755.85. Получены следующие результаты:

Vector

Mean	0.603724814 sec
Maximum	0.738313000 sec
Minimum	0.559959000 sec
Std. Dev	0.037795736 sec
6-Sigma	0.226774416 sec

Deque

Mean	0.588021114 sec
Maximum	0.615617000 sec
Minimum	0.567503000 sec
Std. Dev	0.009907800 sec
6-Sigma	0.059446800 sec

Что использовать: deque или vector?

- **Если планируете добавлять много новых элементов или хотите иметь доступ к началу контейнера- используй deque.**

  В связи со страничным выделением памяти для deque добавление элементов в нем осуществляется быстрее.

- **Если планируете работать с произвольными элементами контейнера, и нужен быстрый доступ к ним - используйте vector**

  В частности можно применить **vector::at() .** Из-за последовательного размещения в памяти произвольный доступ к элементам очень быстрый.

Использованная литература:

- 1. Р. Лафоре «Объектно-ориентированное программирование в C++», 4-е издание.
- 2. Бьерн Страуструп «Язык программирования C++», специальное издание.
- 3. Материалы сайта www.cplusplus.com
- 4. Материалы практических занятий по предметы «Алгоритмы и структуры данных».
- 5. Mark Allen Weiss “Data Structures and Problem Solving Using C++”.
- 6. Материалы сайта www.cppreference.com