Семестр 2. Лекция 5. STL (C++98).

Евгений Линский

23 Марта 2018

Введение

- STL = Standard Template Library
- Библиотека описана в стандарте языка (регламентируется в том числе и сложность операций)
- Используются шаблоны и исключения
- Определена в пространстве имен std
- заголовочные файлы без расширения (`include <vector>`)

Основные части:

- контейнеры (структуры данных для хранения объектов в памяти),
- итераторы (унифицированный доступ к элементам контейнера),
- алгоритмы (для работы с последовательностями).

Контейнеры

Структуры данных для хранения объектов в памяти: вектор, список, ассоциативный массив, ...
Требования к хранимым объектам:

- Корректно работает конструктор копий (copy-constructable)
- Корректно работает оператор присваивания (assignable)

Методы, которые есть у всех контейнеров:

1. Конструктор по умолчанию, конструктор копирования, оператор присваивания, деструктор.
2. Операторы сравнения: `==`, `!=`, `>`, `>=`, `<`, `<=`.
3. `size()`, `empty()`.
4. `swap(obj2)`
5. `insert(...)`, `erase(...)`
6. `clear()`.
7. `begin()`, `end()` первый и последний элемент, но это не точно =)

Последовательные контейнеры

1. Сохраняют порядок, в котором были добавлены элементы
2. Добавление в конец: `push_back`
3. std::vector, std::list, std::deque, std::string

std::vector

Динамический массив с автоматическим изменением размера при добавлении элементов.

- Для уменьшения количества вызовов new при добавлении элементов память выделяется с запасом (≈ 2).
- size/capacity
- Сложность добавления элемента в конец вектора и удаления элемента из конца – амортизированное O(1).
- Сложность добавления элемента или удаления его из начала или середины вектора – O(n).
- Сложность доступа к элементу по индексу – O(1).

std::vector

Методы:

1. `size()/resize()` — добавление элементов
2. `capacity()/reserve()` — управление зарезервированной памятью
3. `push_back()/pop_back()` — добавление/удаление последнего
4. `operator[], at()` — at бросает исключения при выходе за границы
5. `data()` — указатель на массив (для “устаревших” функций)

std::vector

```
#include<vector>
std::vector< int > v;
v.push_back(10);
v.pop_back();
v[ 0 ] = 13;
v.at( 0 ) = 14;
size_t size = v.size();
bool isEmpty = v.empty();
v.clear();
v.resize(10); //Функция увеличивает/уменьшает размер до 10 эл-в
//Работает только если есть конструктор по умолчанию
v.resize( 20, 5); //Добавленные элементы (10 штук) будут равны 5
int *dst = new ...
memcpy(dst, v.data(), v.size());
```

std::vector

```
//Резервирование памяти под 100 элементов
//Увеличивается только вместимость, размер вектора при этом не изменяется
v.reserve( 100 );
//Увеличение потенциальной вместимости на 100 элементов
v.reserve( v.size() + 100 );
//Изменит размер вектора до нуля, но capacity при этом не изменится
v.clear();
//что это?  как это работает?
vector< int > (v).swap(v);
```

std::deque

Контейнер с возможностью быстрой вставки и удаления элементов на обоих концах за аморт. $O(1)$. Реализован как массив указателей на массивы фиксированного размера.

- `size()/resize()`
- `operator[], at`,
- `push_back, push_front`
- `pop_back, pop_front`,

std::list

Двусвязный список. В любом месте контейнера вставка и удаление производятся за $O(1)$. Нет обращения по индексу.

- `size()/resize()`
- `push_back, push_front`
- `pop_back, pop_front`
- `merge, splice`

std::string

Контейнер для хранения символьных последовательностей.

1. Метод c_str() для совместимости со старым кодом:

```
std::string res = "Hello";
printf("%s", res.c_str());
```

2. множество алгоритмов вроде substr(), find() (в терминах *индексов*)
3. поддержка преобразований типа с C строками

```
f(const std::string& s);
f("Hello");
```

4. append, operator+, operator+=,
5. string = basic_string<char>
6. wstring = basic_string<wchar_t>

Адаптеры

Адаптеры (используют другие контейнеры для хранения):

- `stack` — реализация интерфейса стека.
- `queue` — реализация интерфейса очереди.
- `priority_queue` — очередь с приоритетом на куче.

`bitset` — служит для хранения битовых масок.

```
std::bitset<256> mask;
```

Итераторы

- Объект, который синтаксически ведет себя как указатель (определены операторы ++, –, *, ->).
- Универсальный способ перебора элементов контейнеров в STL — перебор с помощью итератора.
- Реализованы как вложенные классы для контейнеров.

```
class vector {
  ...
  class iterator {
    operator++() { T* ptr++; }
  }
}
vector::iterator it1;

class list {
  ...
  class iterator {
    operator++() { Node<T> ptr = ptr->next; }
  }
}
vector::iterator it2;
```

Итераторы

Все контейнеры имеют функции, которые возвращают итераторы на первый элемент и на элемент, следующий за последним:

```
vector< int > v;
vector< int >::iterator begin = v.begin();
vector< int >::iterator end = v.end();
```

Для vector и deque реализована арифметика, аналогичная арифметике указателей:

```
int i = *( v.begin() + 5 )
```

Проход по контейнеры (NB: у list нет operator[])

```
list< int > l;
list< int >::iterator it = l.begin();
for( ; it != l.end(); ++it ) cout << *it;
```

Такой итератор не позволяет менять данные, на которые он указывает

```
list< int >::const_iterator cit = l.begin();
```

Итераторы

insert, erase:

```
//Удаление элемента, на который указывает итератор it
it = v.erase( it );
//Вставка элемента 5 на позицию, на которую указывает итератор
it
it = v.insert( it, 5 );
```

“На который указывает” — “начало первого байта элемента”

Инвалидация итераторов

С контейнером произвели операцию и теперь итераторы указавают в неверное место.

```
vector<int> v;
// зачем у erase/insert возвращаемое значение (на следующий/на вставленный)?
it = v.erase( it );
// может ли произойти инвалидация itb после выполнения следующих строк?
vector<int>:: iterator itb = v.begin();
v.push_back(3);
...
v.push_back(5);
```

Чтобы понять происхдит ли инвалидация, нужно знать внутреннее устройство контейнера.

Инвалидация итераторов

Задача: перед каждым четным элементом вектора вставить 0

```
for ( vector< int >::iterator i = v.begin();
      i != v.end(); ++i ) {
  if ( *i % 2 == 0 ) {
    v.insert( i, 0 );
  }
}
```

Инвалидация итераторов

Задача: перед каждым четным элементом вектора вставить 0

```
vector< int > v;
for ( vector< int >::iterator i = v.begin(); i != v.end();
  if ( *i % 2 == 0 ) {
    i = v.insert( i, 0 );
    // При вставке может произойти перераспределение памяти,
    // а insert() возвращает итератор на только что вставленный
элемент
    ++i;
  }
}
```