Лекция 1. Стандартная библиотека

Контейнеры

Стандартная библиотека

- Минимальный полный набор возможностей для обеспечения базовых потребностей.
- Если что-то содержит, должна реализовывать лучшим образом (стать стандартом де-факто) – не всегда так (e.g. unordered_map ☹)
- Алгоритмически эффективна.
- Расширяема (может использовать как стандартные, так и пользовательские типы).

Состав стандартной библиотеки

- Включает в себя CRT (C Run Time Library) и STL (Standard Template Library)
- Содержит
  - контейнеры: `vector`, `list`, `map`, `unordered_set`, …
  - утилиты: пары, аллокаторы, …
  - итераторы
  - алгоритмы: `for_each`, `sort`, `lower_bound`, …
  - диагностика: `assert`, `exception`, …
  - строки: `string`, `wstring`
  - ввод/вывод: потоки `cout`, `cin`; `printf`, …

Состав стандартной библиотеки (2)

- Содержит (продолжение):
  - локализацию
  - поддержку языка: `numeric_limits`, `typeinfo`, `new/delete`
  - числовые операции: `complex`, <cmath>
  - свойства типов: `is_class, is_numeric,…`
  - …
- Для хидера <*name.h*> из CRT есть хидер <*cname*> с теми же объявлениями в пространстве имен `std`
- Используйте только стандартные хидера для объявления стандартных типов и функций
- Не забывайте про библиотеку Boost

Контейнеры STL

- Реализуют стандартные операции с соответствующими именами и смыслом.
- Однородны – все хранимые объекты одного типа. Но могут хранить указатели (желательно умные) на полиморфные.
- Требуют минимум от хранимого типа.
- Универсально итерируемы.
- Можно настроить (аллокаторы, компараторы).

Типы контейнеров

- Списочные: `array`, `vector`, `list`, `deque`, `forward_list`
- Ассоциативные: `map`, `set`, `multiset`, `multimap`
- Unordered: `unordered_map`, `unordered_multiset`, …
- Адапторы: `stack`, `queue`, `priority_queue`
- Прочие контейнеры: `bitset`, `string`, …

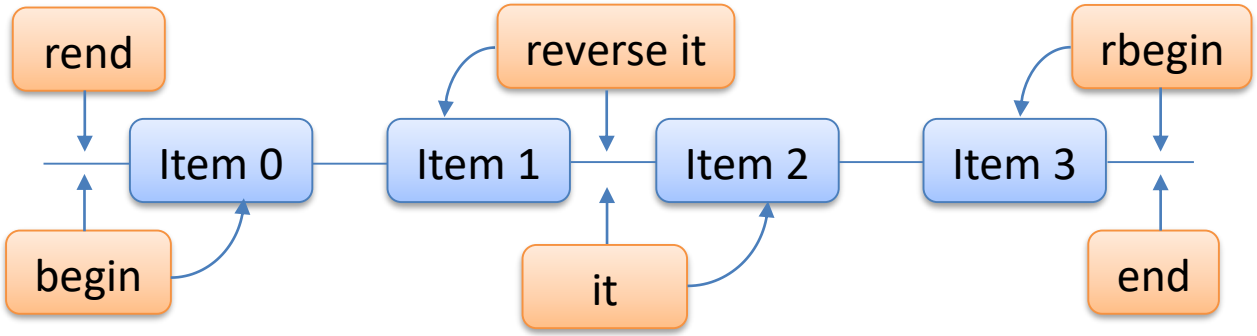
Контейнер vector

```
1   // <vector> header
2   template<class T, class Allocator = std::allocator<T>>
3   class vector
4   {
5       typedef typename T                              value_type;
6       typedef Allocator                               allocator_type;
7       typedef typename Allocator::size_type           size_type;
8       typedef typename Allocator::difference_type     difference_type;
9       typedef typename Allocator::pointer             pointer;
10      typedef typename Allocator::const_pointer       const_pointer;
11      typedef typename Allocator::reference           reference;
12      typedef typename Allocator::const_reference     const_reference;
13
14      typedef impl-defined iterator;
15      typedef impl-defined const_iterator;
16
17      typedef reverse_iterator<iterator>       reverse_iterator;
18      typedef reverse_iterator<const_iterator> const_reverse_iterator;
19      // ...
20
21  };
```

Итераторы

```
1.  template<class T, class Allocator = std::allocator<T>>
2.  class vector
3.  {
4.      // ...
5.      iterator       begin();
6.      const_iterator begin() const;
7.
8.      iterator       end();
9.      const_iterator end() const;
10.
11.     reverse_iterator       rbegin();
12.     const_reverse_iterator rbegin() const;
13.
14.     reverse_iterator       rend();
15.     const_reverse_iterator rend() const;
16.
17.     const_iterator cbegin() const;
18.     const_iterator cend  () const;
19.
20.     const_reverse_iterator crbegin() const;
21.     const_reverse_iterator crend  () const;
22.     // ...
23. };
```

Итераторы




```
1.  &(*reverse_it) != &(*(reverse_it.base()));
2.  &(*reverse_it) == &(*(std::prev(reverse_it.base())));
```

- Типы итераторов: Output; Input, Forward, Bidirectional, RandomAccess

По вектору в обратном направлении

```
1.  // forward
2.  for (size_t i = 0; i < v.size(); ++i)
3.      cout << v[i] << endl;
4.
5.  // naive backward (hmmm, with a surprise)
6.  for (size_t i = v.size() - 1; i >= 0; --i)
7.      cout << v[i] << endl;
8.
9.  // correct
10. for (vector<int>::const_reverse_iterator it  = v.rbegin();
11.                                          it != v.rend(); ++it)
12.     cout << *it << endl;
13.
14. // correct and tiny
15. for (auto it = v.rbegin(); it != v.rend(); ++it)
16.     cout << *it << endl;
17.
18. // range-based for. Does it look good?
19. for (auto item : reverse(v))
20.     cout << item << endl;
```

Как устроен вектор?

```
1.   bool empty () const;
2.
3.   size_type size  () const;
4.   void      resize(size_type size, T val = T());
5.
6.   size_type capacity() const;
7.   void      reserve (size_type size);
8.
9.   void     clear         ();
10.  void     shrink_to_fit();
11.  iterator erase         (const_iterator where);
```

- Вектор – динамический массив.
- При нехватке места увеличивается ~ в 1,5 раза
- Трудоемкость вставки в конец (!) – амортизированная O(1)
- (*) Чтобы увеличение вектора вызывало move конструкторы элементов, а не копирования, они должна быть объявлена как noexcept

Вставка и удаление из вектора

```
1.  // could be const_iterator since C++11
2.  vector<T>::iterator from, to, where;
3.
4.  // random place
5.  v.insert(where, val);
6.
7.  // usual erase
8.  v.erase(from, to);
9.  v.erase(where);
10.
11. // erase items with value by 'remove' function
12. // or use 'remove_if' for predicate usage
13. v.erase(remove(v.begin(), v.end(), 42), v.end());
```

- Функция `remove` и `remove_if` ничего не удаляют, а лишь переносят в конец.

Вектор. Концевые элементы и индекс

- Вектор позволяет эффективно вставлять и удалять элементы с конца. Но (!) не из начала.

```
1.  void push_back(T const& value);
2.  void push_back(T && value);
3.
4.  template< class... Args >
5.  void emplace_back( Args&&... args );
6.
7.  void pop_back();
8.
9.  reference       back();
10. const_reference back() const;
11.
12. reference       front();
13. const_reference front() const
14.
15. pointer data(); // and const
16.
17. reference operator[](size_type index);
18. reference         at(size_type index);
```

Конструирование вектора

- Конструктор по умолчанию, копирования, move-конструктор
- А также:

```
1.  vector<T> v0(from, to);
2.  vector<T> v1(num, value);
3.
4.  // reassign already constructed vector
5.  v0.assign(from, to);
6.
7.  // initializer list (since C++11), any mistake?
8.  vector v2 = {1, 42, 10, 20};
```

Контейнер sdt::list

```
1.  // <list> header
2.  template<class T, class Alloc = std::allocator<T>>
3.  class list;
```

- Двунаправленный список
- Вставка, удаление из любого места O(1)
- Нет RandomAccess итератора, только Bidirectional
- Полезны `splice`, `sort`
- За O(1), в отличие от вектора: `pop_front`, `push_front`

Контейнер `std::deque`

- Есть RandomAccess итератор (возможно, несколько менее эффективно, чем в векторе)
- Позволяет вставлять/удалять в начало и конец за O(1) (но не в произвольное место)
- Часто реализуется через вектор векторов

Адаптеры

- std::stack: push, pop, top
- std::queue: push, pop, front, back
- std::priority_queue: push, pop, top, *constructor*

```
1.  template<class T,
2.            class Container = vector<T>,
3.            class Pr = less<typename Container::value_type>>
4.  class priority_queue;
```

Ассоциативные контейнеры

- Наиболее популярный – `std::map`

```
1.  template<
2.      class Key,
3.      class Value,
4.      class Predicate = less<Key>,
5.      class Alloc = allocator<pair<const Key, Value>>>
6.  class map
7.  {// usually, some balanced tree (e.g. red-black)
8.  public:
9.      typedef Key                         key_type;
10.     typedef Value                       mapped_type;
11.     typedef pair<const Key, Value>      value_type;
12.     typedef Predicate                   key_compare;
13.     // ...
14. };
```

Итерирование map

```
1.  map<string, size_t> marks;
2.
3.  for (auto it = marks.begin(); it != marks.end(); ++it)
4.      cout << it->first << " " << it->second << endl;
5.
6.  //...
7.  it->second = 5;          // ok
8.  it->first  = "Vasya";    // nope, cannot change key
```

- Итератор разыменовывается в пару. Ключ – first, значение – second.

Поиск и вставка в map

```
1.  typedef std::map<string, size_t> map_t;
2.  typedef map_t::iterator          iterator;
3.
4.  map_t m;
5.
6.  // simple insert
7.  auto it = m.find(str);
8.  if (it != n.end())
9.      return ...;
10.
11. pair<iterator, bool> p =
12.     m.insert(make_pair(str, 5));
13.
14. // with 'hint'
15. auto it = m.lower_bound(str);
16. if (it != m.end() && it->second == str)
17.     return ...;
18.
19. m.insert(it, make_pair(str, 5));
```

Индексирование

```
1.  typedef std::map<string, size_t> map_t;
2.  typedef map_t::iterator          iterator;
3.
4.  map_t m;
5.
6.  m[str] = 5;
```

- Если элемента не было в map – добавит
- Если был – оператора вернет ссылку и произойдет замена текущего значения (в отличие от insert)
- Оператор **[]** всегда неконстантный. Есть только у `map`, нет у `set`, `multiset`, `multimap`.

Удаление из map (set)

```
1.  m.erase(key);
2.  m.erase(from, to);
3.
4.  // with predicate
5.  for (auto it = m.begin(); it != m.end(); ~~++it~~)
6.  {
7.      if (predicate(*it))
8.      {
9.          m.erase(it++);
10.         // or
11.         it = m.erase(it);
12.     }
13.     else
14.         ++it;
15. }
```

Другие ассоциативные контейнеры

- `std::set` – содержит только ключи
- `std::multimap`, `std::multiset` – ключи могут повторяться
- `std::unordered_map/set`
  - используют вычисление хэшей и предикат равенства вместо предиката порядка
  - вставка, удаление, поиск – амортизированная O(1)
  - Требуют больше памяти, чем обычные map/set
  - Есть более быстрые реализации: Google sparse/dense hash map, MCT hash map

Спасибо за внимание!
Вопросы?