Язык C++

STL. Итераторы и основные алгоритмы - II

Функциональные объекты

- minus;
- multiplies
- ….
- equal_to
- not_equal_to
- greater
- ...
- logical_and
- …
- bit_and

Контейнеры

- Контейнеры последовательностей:
  - vector<T>
  - deque<T>
  - list<T>
  - array<T>
  - forward_list<T>
- Ассоциативные контейнеры:
  - set<Key> (multiset)
  - map<Key,T> (multimap)
- Неупорядоченные ассоциативные контейнеры
  - unordered_set<Key> (multiset)
  - unordered_map<Ket, T> (multimap)

Named Requirements

- ***Container***
- ***ReversibleContainer***
- ***AllocatorAwareContainer***
- ***SequenceContainer***
- ***ContiguousContainer***
- ***AssociativeContainer***
- ***UnorderedAssociativeContainer***
- ***etc***

Sequence containers

- array
- vector
- deque
- forward_list
- list

```
template<class T,  class Allocator = std::allocator<T>>
class vector;
```

std::array

- *Container*
- *ReversibleContainer*
- *SequenceContainer*
- *ContiguousContainer*

std::array Container requirements

```
template <class _Tp, size_t _Size>
struct array
{
  typedef _Tp                   value_type;
  typedef value_type&           reference;
  typedef const value_type&     const_reference;
  typedef value_type*           iterator;
  typedef const value_type*     const_iterator;
  typedef size_t                size_type;
  typedef ptrdiff_t             difference_type;

  _Tp __elems_[_Size];
……..
};
```

std::array Container requirements

```
iterator begin() {return iterator(data());}
const_iterator begin() const {return const_iterator(data());}
iterator end() {return iterator(data() + _Size);}
const_iterator end() const {return const_iterator(data() + _Size);}


size_type size() const {return _Size;}
size_type max_size() const {return _Size;}
bool empty() const {return _Size == 0;}
```

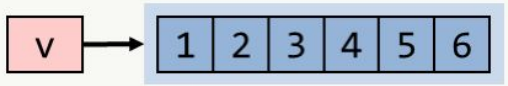
std::array *ReversibleContainer* requirements

```
typedef _VSTD::reverse_iterator<iterator>       reverse_iterator;
typedef _VSTD::reverse_iterator<const_iterator> const_reverse_iterator;



reverse_iterator rbegin() {return reverse_iterator(end());}
const_reverse_iterator rbegin() const {return const_reverse_iterator(end());}
reverse_iterator rend() {return reverse_iterator(begin());}
const_reverse_iterator rend() const {return const_reverse_iterator(begin());}
```

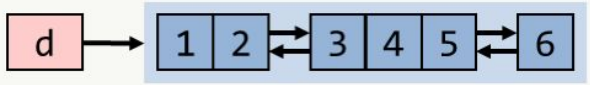
Sequence containers

array<T,n>

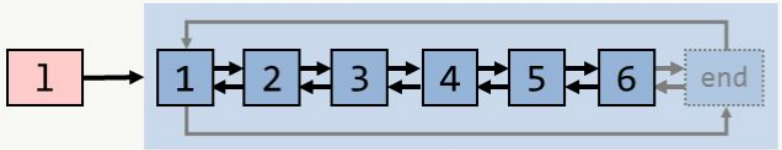
vector<T>



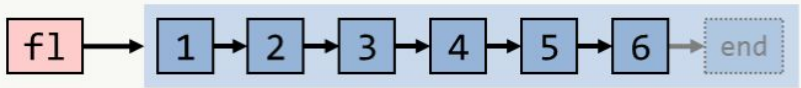
deque<T>



list<T>



forward_list<T>



Associative containers

- set
- map
- multiset
- multimap

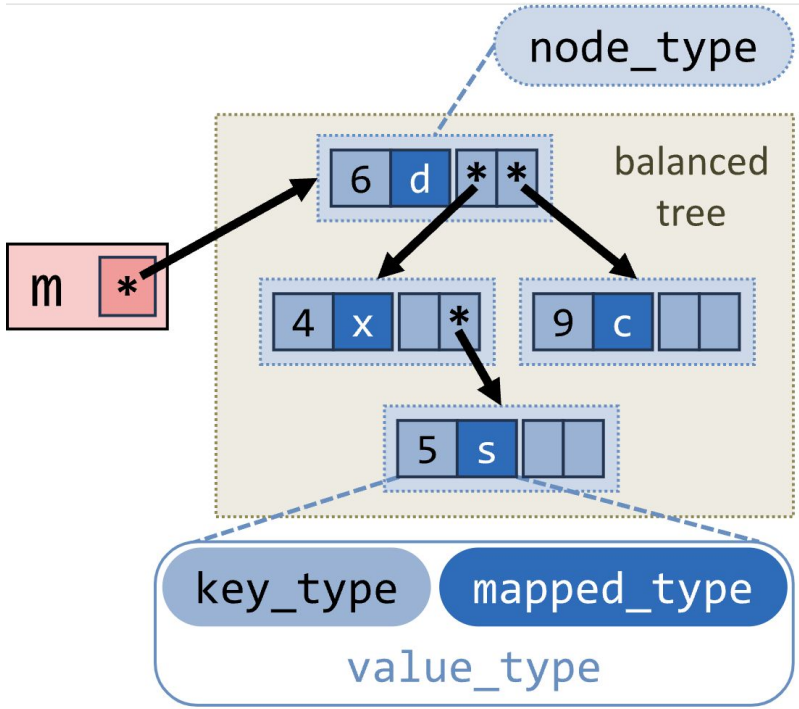
```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
```

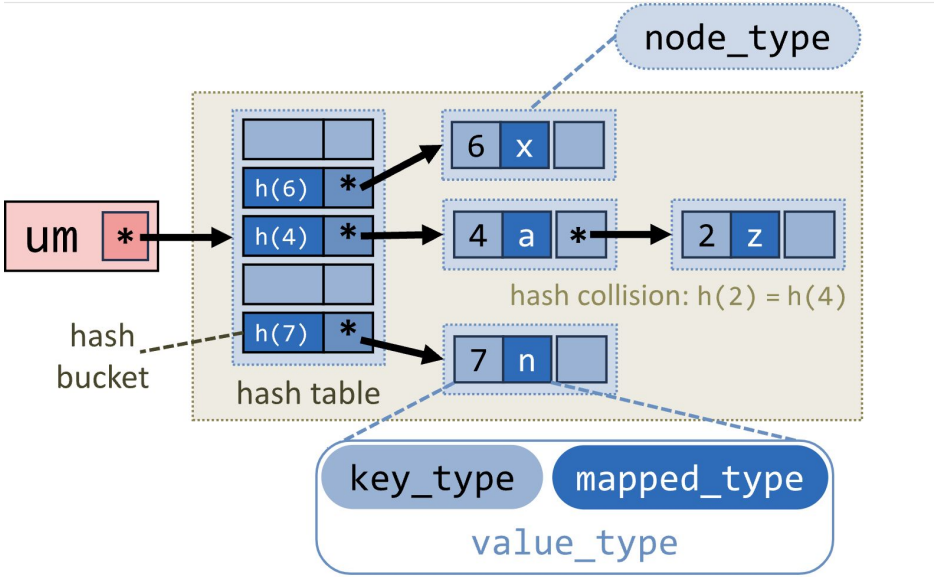
Unordered associative containers

- unordered_set
- unordered_map
- unordered_multiset
- unordered_multimap

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_map;
```

Associative containers





Iterator Invalidation

- Insert/erase
- Capacity change
- After
- Before
- Rehash

Allocator

Класс, отвечающий требованиям, основная задача - инкапсулировать стратегию выделения/очистки памяти и созданий/удаления объектов.

- allocation
- deallocation
- construction
- destruction

Allocator

```
struct SPoint {
   int x;
   int y;
};

int main () {
   std::allocator_traits<CSimpleAllocator<int>> at;
   std::vector<SPoint, CSimpleAllocator<SPoint>> data;

   data.push_back({10,20});

   data.pop_back();

   return 0;
}
```

Allocator

```
template <typename T>
class CSimpleAllocator {
public:
   typedef size_t size_type ;
   typedef ptrdiff_t difference_type ;
   typedef T* pointer ;
   typedef const T* const_pointer ;
   typedef T& reference ;
   typedef const T& const_reference ;
   typedef T value_type ;
};
```

Allocator

```
template <typename T>
class CSimpleAllocator {
public:
   pointer allocate( size_type size) {
       pointer result = static_cast <pointer >(malloc(size * sizeof(T)));
       if(result == nullptr ) {
           // error
       }
       std::cout << "Allocate count" << size << " elements. Pointer:" << result << std::endl;
       return result;
   }

   void deallocate(pointer p, size_type n) {
       std::cout << "Deallocate pointer: " << p << std::endl;
       free(p);
   }
};
```

StackAllocator