Не такие уж стандартные контейнеры

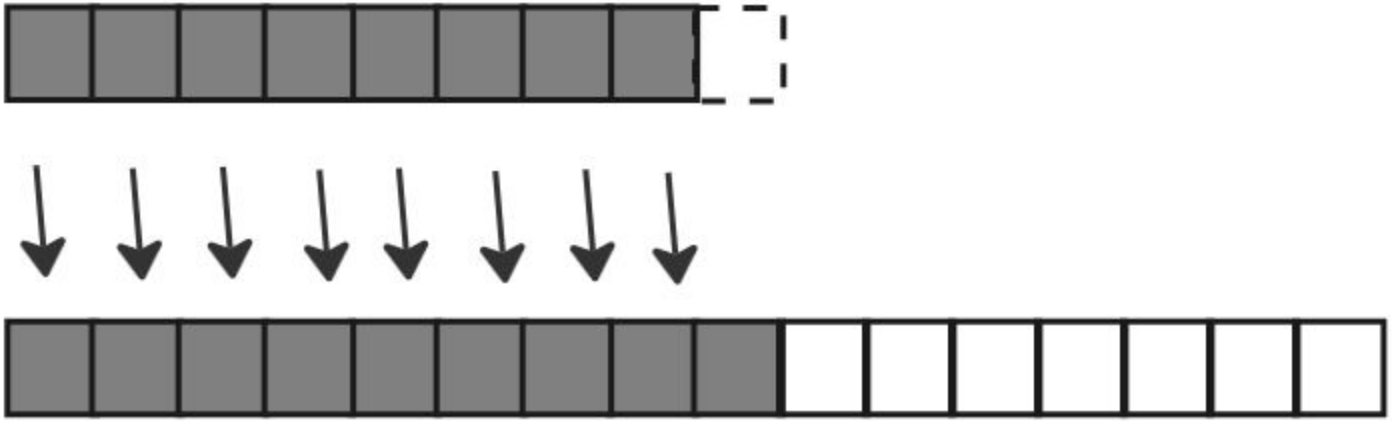
или улучшенные версии стандартных контейнеров из библиотеки boost

Мещерин Илья
t.me/mesyarik, youtube.com/@mesyarik

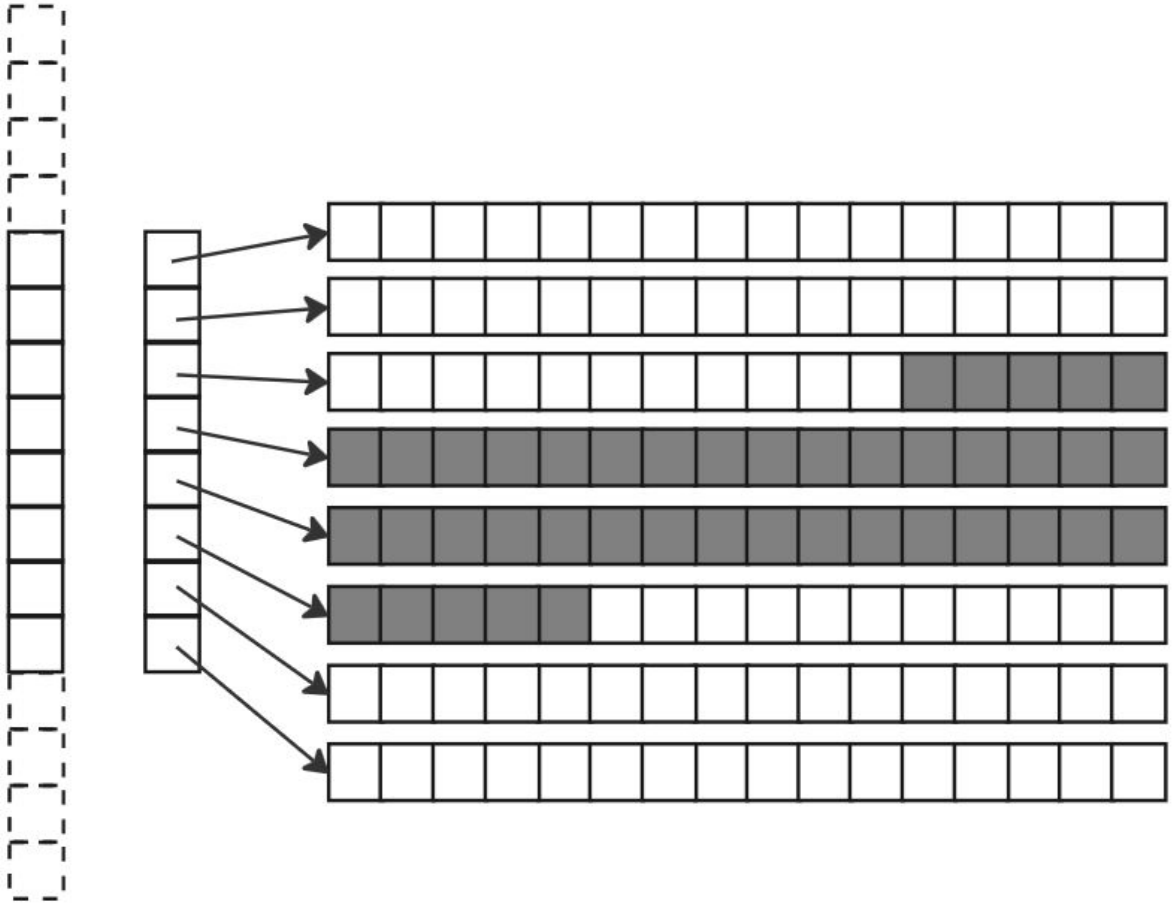
Revision of standard containers

- vector
- deque
- list
- map
- unordered_map

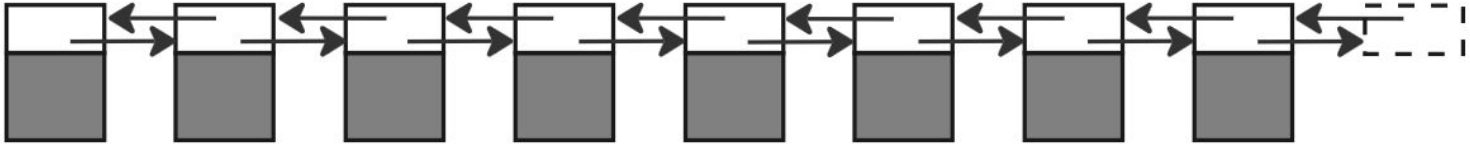
std::vector



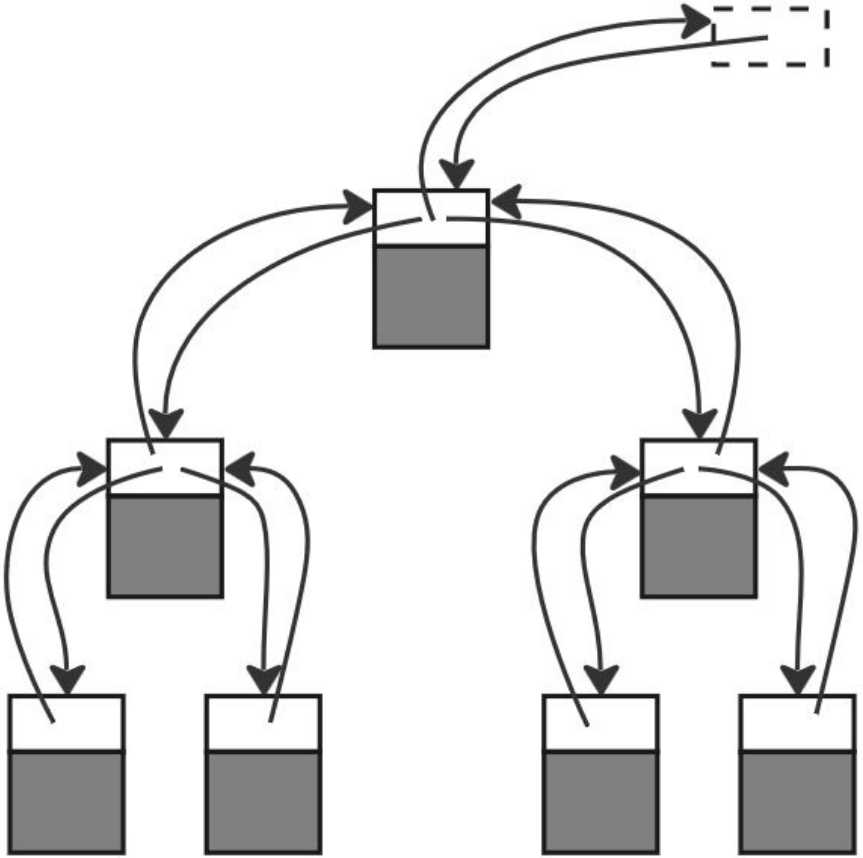
std::deque



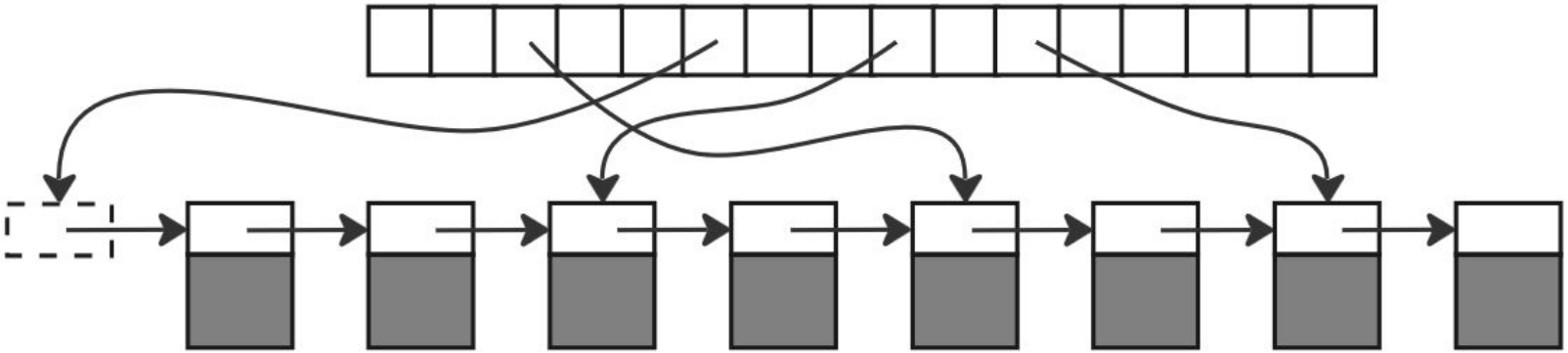
std::list



std::map



std::unordered_map



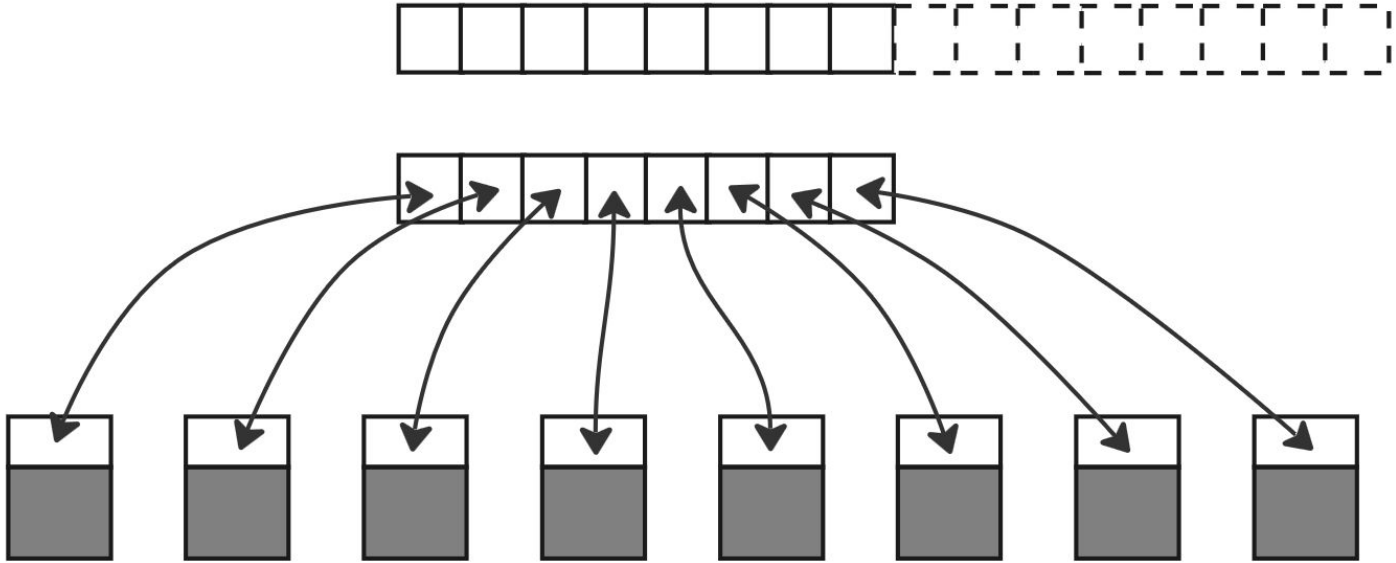
Introducing not so standard containers

- stable_vector
- devector
- circular_buffer
- flat_map
- bimap

Introducing not so standard containers

- **stable_vector**
- devector
- circular_buffer
- flat_map
- bimap

boost::stable_vector



stable_vector example

```cpp
#include <boost/container/stable_vector.hpp>
#include <iostream>

using boost::container::stable_vector;

int main() {
    stable_vector<int> v;
    for (int i = 0; i < 5; ++i) {
        v.push_back(i);
    }
    auto it = v.begin();
    v.push_back(5); // old iterators remain valid

    for (; it != v.end(); ++it) {
        std::cout << *it << ' '; // 0 1 2 3 4 5
    }
}
```

stable_vector: сложность операций

Operation	**std::vector**	**std::deque**	**std::list**	**stable_vector**
push_back	O(1) amortized	O(1) with reallocs	O(1)	O(1) with reallocs
push_front	-	O(1) with reallocs	O(1)	-
operator[]	1x dereference	2x dereference	-	2x dereference
iterator increment	no dereference	0-1x dereference	1x dereference	1x dereference
iterator += n	no dereference	0-1x dereference	-	1x dereference

stable_vector: итераторы

Property	**std::vector**	**std::deque**	**std::list**	**stable_vector**
iterator category	contiguous	random access	bidirectional	random access
Iterators invalidation on push_back	Yes	Yes	No	No
References invalidation on push_back	Yes	No	No	No
Iterators invalidation on insert	Yes	Yes	No	No
References invalidation on insert	Yes	Only moved elements	No	No

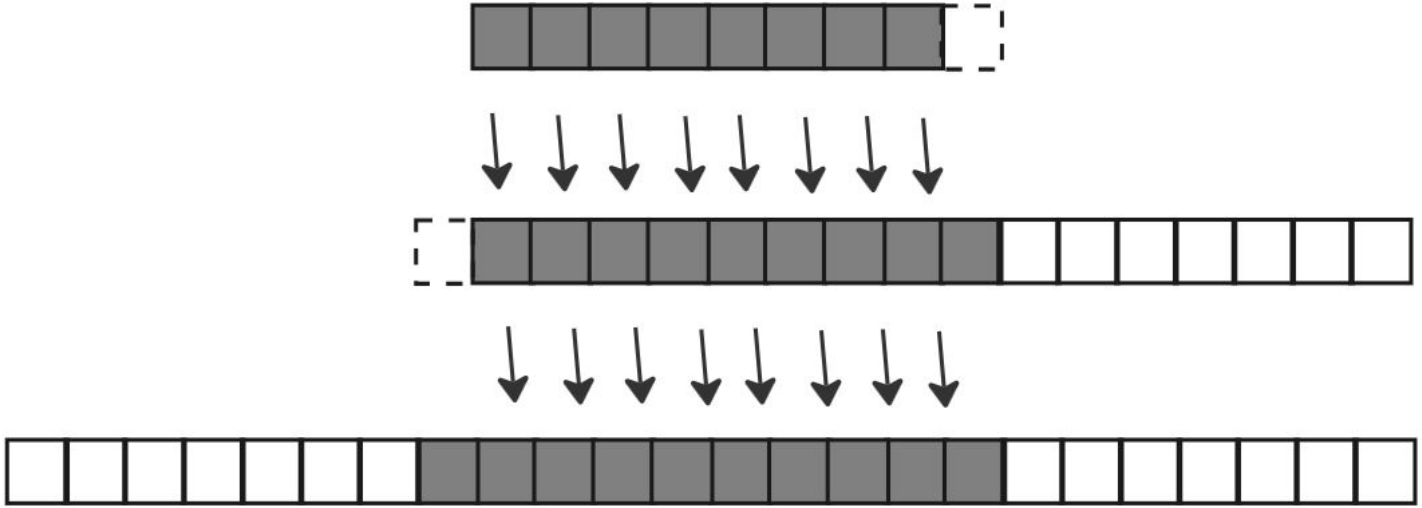
stable_vector: exception safety

Property	**std::vector**	**std::deque**	**std::list**	**stable_vector**
push_back exception safe?	Sometimes	Yes	Yes	Yes
insert in the middle exception safe?	No	No	Yes	Yes

Introducing not so standard containers

- stable_vector
- **devector**
- circular_buffer
- flat_map
- bimap

boost::devector



devector: сложность операций

Operation	**std::vector**	**std::deque**	**std::list**	**devector**
push_back	O(1) amortized	O(1) with reallocs	O(1)	O(1) amortized
push_front	-	O(1) with reallocs	O(1)	O(1) amortized
operator[]	1x dereference	2x dereference	-	1x dereference
iterator increment	no dereference	0-1x dereference	1x dereference	no dereference
iterator += n	no dereference	0-1x dereference	-	no dereference

devector: итераторы

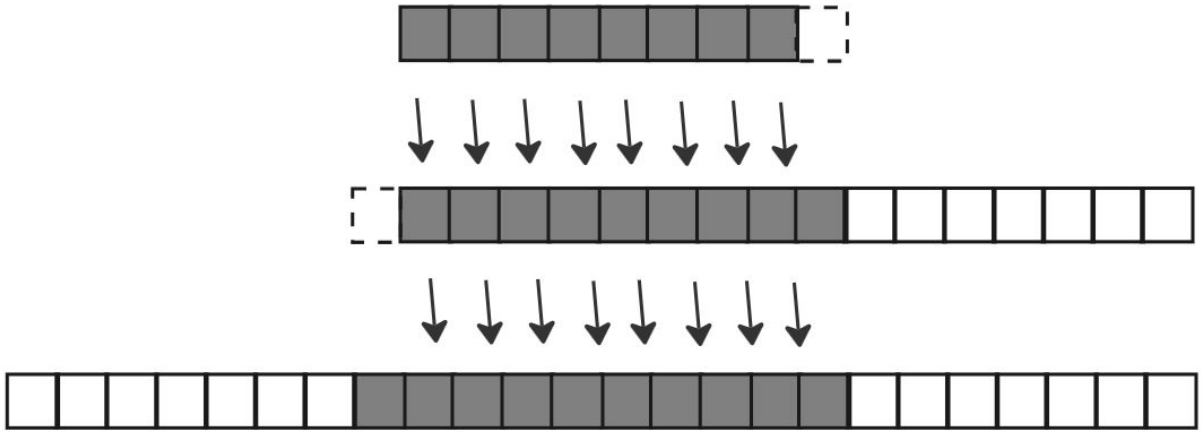
Property	**std::vector**	**std::deque**	**std::list**	**devector**
iterator category	contiguous	random access	bidirectional	contiguous
Iterators invalidation on push_back	Yes	Yes	No	Yes
References invalidation on push_back	Yes	No	No	Yes
Iterators invalidation on insert	Yes	Yes	No	Yes
References invalidation on insert	Yes	Only moved elements	No	Yes

devector: exception safety

Property	std::vector	std::deque	std::list	devector
push_back exception safe?	Sometimes	Yes	Yes	Sometimes
insert in the middle exception safe?	No	No	Yes	No

devector: сложность операций

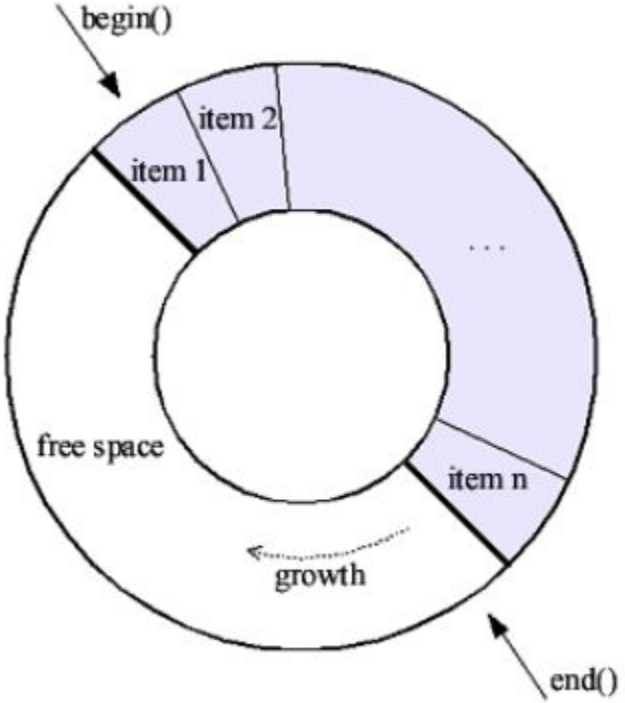
Каким образом достигается амортизированное O(1)?



Introducing not so standard containers

- stable_vector
- devector
- **circular_buffer**
- flat_map
- bimap

boost::circular_buffer



Introducing not so standard containers

- stable_vector
- devector
- circular_buffer
- flat_map
- bimap

Intrusive containers

- intrusive::list
- set, rbtree
- avl_set, avltree
- splay_set, splaytree
- treap_set, treap

Intrusive containers

- **intrusive::list**
- set, rbtree
- avl_set, avltree
- splay_set, splaytree
- treap_set, treap

intrusive_list: first example

```cpp
#include <boost/intrusive/list.hpp>
#include <iostream>

class MyClass : public boost::intrusive::list_base_hook<> {
public:
    int value;
    MyClass(int value): value(value) {}
};

int main() {
    MyClass obj1{1}, obj2{2};

    boost::intrusive::list<MyClass> myList;
    myList.push_back(obj1);
    myList.push_back(obj2);
    obj2.value = 999;
    for (const auto& obj : myList) {
        std::cout << obj.value << " ";   // 1 999
    }
}
```

intrusive_list example 2: crash

```
#include <boost/intrusive/list.hpp>
#include <iostream>

class MyClass : public boost::intrusive::list_base_hook<> {
public:
    int value;
    MyClass(int value): value(value) {}
};

int main() {
    boost::intrusive::list<MyClass> myList;

    MyClass obj1{1}, obj2{2}, obj3{3}; // WRONG, objects must be destroyed after the list
    myList.push_back(obj1);
    myList.push_back(obj2);
    myList.push_back(obj3);
    for (const auto& obj : myList) {
        std::cout << obj.value << " ";
    }
}
```

intrusive_list example 2: crash

```
a.out: /usr/include/boost/intrusive/detail/generic_hook.hpp:48: void
boost::intrusive::detail::destructor_impl(Hook&, link_dispatch<boost::intrusive::safe_link>) [with
Hook = boost::intrusive::generic_hook<boost::intrusive::CircularListAlgorithms,
boost::intrusive::list_node_traits<void*>, boost::intrusive::dft_tag, boost::intrusive::safe_link,
boost::intrusive::ListBaseHookId>]: Assertion `!hook.is_linked()' failed.
Aborted (core dumped)
```

intrusive_list example 3: member hooks

```
#include <boost/intrusive/list.hpp>
#include <iostream>

using boost::intrusive::list_member_hook;

class MyClass {
public:
    int value;
    list_member_hook<> hook;
};

using IntrusiveList = boost::intrusive::list<
        MyClass,
        boost::intrusive::member_hook<MyClass, list_member_hook<>, &MyClass::hook>
    >;

int main() {
    MyClass obj1{1}, obj2{2}, obj3{3};
    IntrusiveList myList;
    // ...
```

intrusive_list: 3 types of hooks

- `base_hook`
- `member_hook`
- `function_hook`

intrusive_list example 4: single element, multiple lists

```
#include <boost/intrusive/list.hpp>
#include <iostream>

struct MyTag1;
struct MyTag2;

using namespace boost::intrusive;

using BaseHook1 = list_base_hook<tag<MyTag1>>;
using BaseHook2 = list_base_hook<tag<MyTag2>>;

class MyClass: public BaseHook1, public BaseHook2 {
public:
    int value;
    MyClass(int value): value(value) {}
};
```

intrusive_list example 4: single element, multiple lists

```
int main() {
    MyClass obj1{1}, obj2{2}, obj3{3}, obj4{4};

    list<MyClass, base_hook<BaseHook1>> first;
    list<MyClass, base_hook<BaseHook2>> second;

    first.push_back(obj1); first.push_back(obj2); first.push_back(obj3);
    second.push_back(obj2); second.push_back(obj3); second.push_back(obj4);

    obj3.value = 999;
    for (const auto& obj : first) {
        std::cout << obj.value << " "; // 1 2 999
    }
    for (const auto& obj : second) {
        std::cout << obj.value << " "; // 2 999 4
    }
}
```

intrusive_list crash revision

```cpp
#include <boost/intrusive/list.hpp>
#include <iostream>

class MyClass : public boost::intrusive::list_base_hook<> {
public:
    int value;
    MyClass(int value): value(value) {}
};

int main() {
    boost::intrusive::list<MyClass> myList;

    MyClass obj1{1}, obj2{2}, obj3{3}; // WRONG, objects must be destroyed after the list
    myList.push_back(obj1);
    myList.push_back(obj2);
    myList.push_back(obj3);
    for (const auto& obj : myList) {
        std::cout << obj.value << " ";
    }
}
```

intrusive_list example 5: safe_link vs normal_link

```
class MyClass : public list_base_hook< link_mode<normal_link> > {
public:
    int value;
    MyClass(int value): value(value) {}
};

int main() {
    boost::intrusive::list<MyClass> myList;

    {
        MyClass obj1{1}, obj2{2};
        myList.push_back(obj1);
        myList.push_back(obj2);
    } // Runtime error in case of link_mode<safe_link>

    for (const auto& obj : myList) {
        std::cout << obj.value << " "; // UB, because objects are already destroyed
    }
}
```

intrusive_list example 5: properties of safe_link

- Hook's constructor puts the hook in a well-known default state.
- Hook's destructor checks if the hook is in the well-known default state. If not, an assertion is raised. (That’s our case)
- Every time an object is inserted in the intrusive container, the container checks if the hook is in the well-known default state. If not, an assertion is raised.
- Every time an object is being erased from the intrusive container, the container puts the erased object in the well-known default state.

intrusive_list: constant time size() option

`list` receives the type to be inserted in the container (`T`) as the first parameter and optionally, the user can specify options.

- **`base_hook<class Hook>` / `member_hook<class T, class Hook, Hook T::* PtrToMember>`**
- **`constant_time_size<bool Enabled>`**: Specifies if a constant time `size()` function is demanded for the container. This will instruct the intrusive container to store an additional member to keep track of the current size of the container. By default, constant-time size is activated.

intrusive_list: constant time size() option

`list` receives the type to be inserted in the container (T) as the first parameter and optionally, the user can specify options.

- **`base_hook<class Hook>`** / **`member_hook<class T, class Hook, Hook T::* PtrToMember>`**
- **`constant_time_size<bool Enabled>`**: Specifies if a constant time `size()` function is demanded for the container. This will instruct the intrusive container to store an additional member to keep track of the current size of the container. By default, constant-time size is activated.

```
void splice(const_iterator p, list& x, const_iterator f, const_iterator e);
```

Effects: Transfers the range pointed by f and e from list x to this list, before the element pointed by p. No destructors or copy constructors are called.

Complexity: Linear to the number of elements transferred if constant-time size option is enabled. Constant-time otherwise.

intrusive_list example 6: auto_unlink

```
using auto_unlink_hook = list_base_hook< link_mode<auto_unlink> >;
```

- Similar properties to the safe link, and
- When the destructor of the hook is called, the hook checks if the node is inserted in a container. If so, the hook removes the node from the container.
- The hook has a member function called `unlink()` that can be used to unlink the node from the container at any time, without having any reference to the container, if the user wants to do so.
- Can only be used with non-constant time size containers

intrusive::list vs. std::list

```
void push_back(reference value);
void push_front(reference value);
void pop_back();
template<typename Disposer> void pop_back_and_dispose(Disposer disposer);
void pop_front();
template<typename Disposer> void pop_front_and_dispose(Disposer disposer);
iterator erase(const_iterator i);
iterator erase(const_iterator b, const_iterator e);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator b, const_iterator e,
                             Disposer disposer);
void clear();
template<typename Disposer> void clear_and_dispose(Disposer disposer);
iterator insert(const_iterator p, reference value);
template<typename Iterator>
  void insert(const_iterator p, Iterator b, Iterator e);
```

intrusive::list operations complexity

```
iterator erase(const_iterator b, const_iterator e);
```

Requires: b and e must be valid iterators to elements in *this. n must be std::distance(b, e).

Effects: Erases the element range pointed by b and e No destructors are called.

Returns: the first element remaining beyond the removed elements, or end() if no such element exists.

Complexity: Linear to the number of erased elements if it's a safe-mode or auto-unlink value is enabled. Constant-time otherwise.

```
void clear();
```

Effects: Erases all the elements of the container. No destructors are called.

Complexity: Linear to the number of elements of the list, if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Note: Invalidates the iterators (but not the references) to the erased elements.

Intrusive containers

- intrusive::list
- **set, rbtree**
- avl_set, avltree
- splay_set, splaytree
- treap_set, treap

intrusive_set: first example

```
#include <boost/intrusive/set.hpp>
#include <iostream>

class MyClass : public boost::intrusive::set_base_hook<> {
public:
    int key;
    MyClass(int key): key(key) {}
};

struct Compare {
    bool operator()(const MyClass& lhs, const MyClass& rhs) const {
        return lhs.key < rhs.key;
    }
};

using MySet = boost::intrusive::set<MyClass, boost::intrusive::compare<Compare>>;
```

intrusive_set: first example

```
int main() {
    MyClass obj1{1};
    MyClass obj2{2};
    MyClass obj3{3};

    MySet mySet;  // It's important that set will be destroyed before the objects
    mySet.insert(obj2);
    mySet.insert(obj1);
    mySet.insert(obj3);

    for (const auto& obj : mySet) {
        std::cout << obj.key << " "; // 1 2 3
    }
    std::cout << std::endl;
}
```

intrusive_set: member hook

```
#include <boost/intrusive/set.hpp>

using boost::intrusive::set_member_hook;

class MyClass {
public:
    int key;
    set_member_hook<> hook;
    MyClass(int key): key(key) {}
};

struct Compare {
    bool operator()(const MyClass& lhs, const MyClass& rhs) const {
        return lhs.key < rhs.key;
    }
};

using MySet = boost::intrusive::set<
                MyClass,
                boost::intrusive::compare<Compare>,
                boost::intrusive::member_hook<MyClass, set_member_hook<>, &MyClass::hook>
            >;
```

intrusive_set: hook options

`set_base_hook` and `set_member_hook` receive the same options explained in the section How to use Boost.Intrusive plus a size optimization option:

- **`tag<class Tag>`** (for base hooks only): This argument serves as a tag, so you can derive from more than one base hook. Default: `tag<default_tag>`.
- **`link_mode<link_mode_type LinkMode>`**: The linking policy. Default: `link_mode<safe_link>`.
- **`void_pointer<class VoidPointer>`**: The pointer type to be used internally in the hook and propagated to the container. Default: `void_pointer<void*>`.
- **`optimize_size<bool Enable>`**: The hook will be optimized for size instead of speed. The hook will embed the color bit of the red-black tree node in the parent pointer if pointer alignment is even. In some platforms, optimizing the size might reduce speed performance a bit since masking operations will be needed to access parent pointer and color attributes, in other platforms this option improves performance due to improved memory locality. Default: `optimize_size<false>`.

intrusive_set: hook options

`set_base_hook` and `set_member_hook` receive the same options explained in the section How to use Boost.Intrusive plus a size optimization option:

- **`tag<class Tag>`** (for base hooks only): This argument serves as a tag, so you can derive from more than one base hook. Default: `tag<default_tag>`.
- **`link_mode<link_mode_type LinkMode>`**: The linking policy. Default: `link_mode<safe_link>`.
- **`void_pointer<class VoidPointer>`**: The pointer type to be used internally in the hook and propagated to the container. Default: `void_pointer<void*>`.
- **`optimize_size<bool Enable>`**: The hook will be optimized for size instead of speed. The hook will embed the color bit of the red-black tree node in the parent pointer if pointer alignment is even. In some platforms, optimizing the size might reduce speed performance a bit since masking operations will be needed to access parent pointer and color attributes, in other platforms this option improves performance due to improved memory locality. Default: `optimize_size<false>`.

Why there is no intrusive::map?

This is because a `map<Key, Value>` is actually `set<std::pair<Key const, Value>>` and Boost.Intrusive and Boost.MultiIndex sets allow you to use one of the value members as a key. In other words, there is no need for `map` if `find` can accept a comparable key, rather than the entire value to search for which has been a long-standing unresolved issue with `std::map` and `std::set`:

> The associative container lookup functions (find, lower_bound, upper_bound, equal_range) only take an argument of key_type, requiring users to construct (either implicitly or explicitly) an object of the key_type to do the lookup. This may be expensive, e.g. constructing a large object to search in a set when the comparator function only looks at one field of the object. There is strong desire among users to be able to search using other types which are comparable with the key_type.

Share Improve this answer Follow

edited May 15, 2014 at 22:15

answered May 13, 2014 at 14:11

Maxim Egorushkin

134k ● 17 ● 190 ● 281

Summary about intrusive containers

Issue	Intrusive	Non-intrusive
Memory management	External	Internal through allocator
Insertion/Erasure time	Faster	Slower
Memory locality	Better	Worse
Can insert the same object in more than one container	Yes	No
Exception guarantees	Better	Worse
Computation of iterator from value	Constant	Non-constant
Insertion/erasure predictability	High	Low
Memory use	Minimal	More than minimal
Insert objects by value retaining polymorphic behavior	Yes	No (slicing)
User must modify the definition of the values to insert	Yes	No
Containers are copyable	No	Yes
Inserted object's lifetime managed by	User (more complex)	Container (less complex)
Container invariants can be broken without using the container	Easier	Harder (only with containers of pointers)
Thread-safety analysis	Harder	Easier

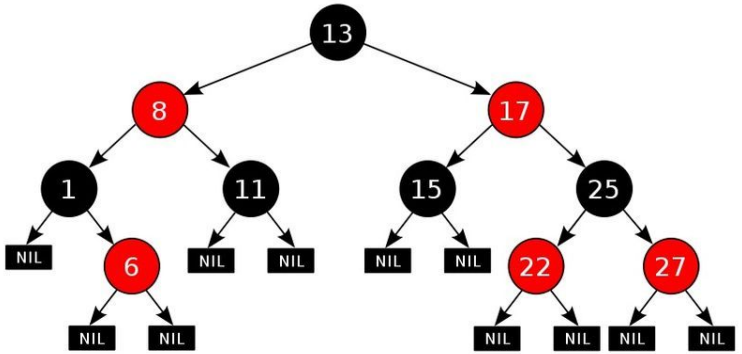
Intrusive containers

- intrusive::list
- set, rbtree
- **avl_set, avltree**
- splay_set, splaytree
- treap_set, treap

Red-black tree

Красно-черное дерево:

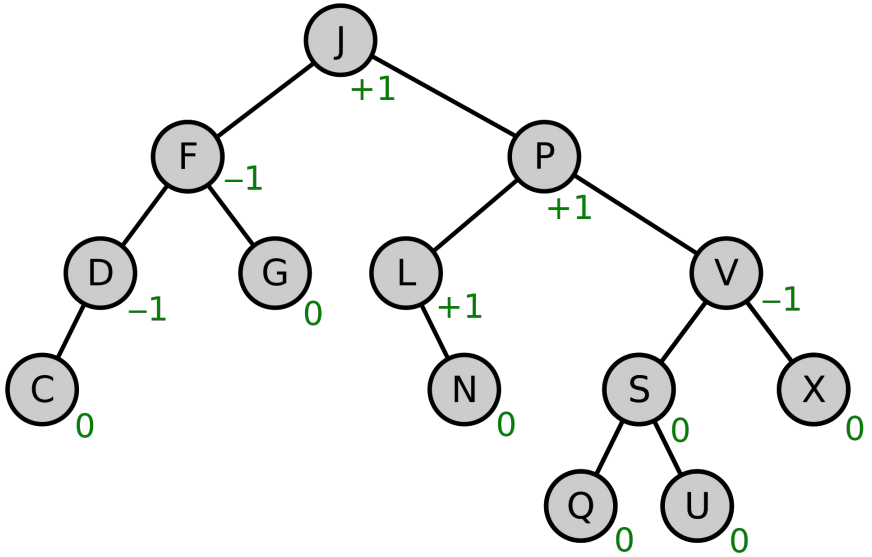
- Каждая вершина - красная или черная
- Родители красных вершин - всегда черные
- На любом пути от корня до листа одинаковое количество черных вершин
- Следствие: длины любых двух ветвей отличаются не более чем в 2 раза
- Один бит дополнительной информации в вершинах



Red-black tree vs. AVL tree

AVL дерево:

- У каждой вершины высота левого поддерева не более чем на 1 отличается от высоты правого поддерева
- Высота не превосходит log_φ (n), где φ примерно равно 1.6 - золотое сечение
- Два бита дополнительной информации в вершинах



intrusive::avl_set

```
#include <boost/intrusive/avl_set.hpp>
#include <iostream>

class MyClass : public boost::intrusive::avl_set_base_hook<
                            boost::intrusive::optimize_size<true>
                        > {
public:
    int value;
    MyClass(int value): value(value) {}
    bool operator <(const MyClass& other) const {
        return value < other.value;
    }
};

int main() {
    MyClass obj1{1};
    boost::intrusive::avl_set<MyClass> myAVLSet;
    myAVLSet.insert_unique(obj1);
}
```

Intrusive containers

- intrusive::list
- set, rbtree
- avl_set, avltree
- splay_set, splaytree
- treap_set, treap

Сравнение различных деревьев

Operation	**std::set**	**avl_set**	**splay_set**	**treap_set**
insert	O(log n)	O(log n), bigger constant	O(log n) amortized	O(log n) expected
erase	O(log n)	O(log n), bigger constant	O(log n) amortized	O(log n) expected
find	O(log n)	O(log n), smaller constant	O(log n) amortized	O(log n) expected

Semi-intrusive containers

- intrusive::unordered_set
- Intrusive::unordered_multiset

Why **semi**-intrusive?

intrusive::unordered_set

```
#include <boost/intrusive/unordered_set.hpp>
#include <iostream>

class MyType : public boost::intrusive::unordered_set_base_hook<> {
public:
  int key;
  MyType(int k) : key(k) {}
};

using boost::intrusive::unordered_set;

int main() {
  unordered_set<MyType>::bucket_type buckets[100];
  unordered_set<MyType>::bucket_traits traits(buckets, 100);
  unordered_set<MyType> mySet(traits);

  MyType obj1(1);
  mySet.insert(obj1);
```

Спасибо за внимание

Мещерин Илья

t.me/mesyarik

youtube.com/@mesyarik