Интрузивные контейнеры

На примере boost::intrusive

Елена Степанова

Ведущий инженер-программист, ЯЦТМС
Департамент проектирования и разработки пакетного ядра сети

- Пишу на C++ от стандарта 03 до 23
- Научила команду использовать интрузивные контейнеры в продуктовом коде

Что это?

- В чём разница интрузивных и неинтрузивных контейнеров
- Как Boost.Intrusive имплементирует интрузивные контейнеры

Зачем это?

- Какие задачи помогают решать интрузивные контейнеры
- Пример использования в продуктовой задаче

Что такое интрузивные контейнеры

Отличие STL контейнеров от Boost.Intrusive

Как устроена связь объекта с хранилищем

Примеры использования

Definition of 'intrusive'

intrusive

(ɪntruːsɪv)

adjective

Something that is **intrusive** disturbs your mood or your life in a way you do not like.

The cameras were not an intrusive presence.
Staff are courteous but never intrusive.

Synonyms: interfering, disturbing, invasive, unwanted More Synonyms of **intrusive**

More Synonyms of **intrusive**


Напиши
двусвязный список

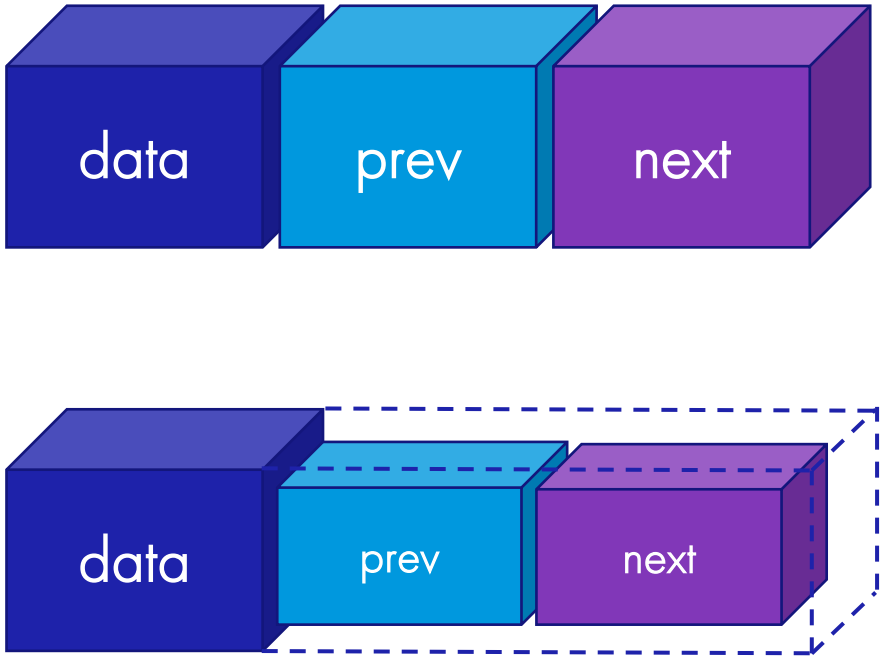
```
template<typename T>
struct LinkedList {
  struct Node {
    T value;
    Node* prev;
    Node* next;
  };

  Node* head;
  Node* tail;
};

LinkedList<int> list;
```

```
template<typename T>
struct LinkedList {

  Node* head;
  Node* tail;
};
```

```
template<typename T>
struct LinkedList {

  T* head;
  T* tail;
};
```

```
template<typename T>
struct LinkedList {
  T* head;
  T* tail;
};

template <typename T>
struct Node {
  T value;
  Node* prev;
  Node* next;
};

LinkedList<Node<int>> list;
```

Главное отличие интрузивного контейнера от неинтрузивного – информация о контейнере «просачивается» в элемент

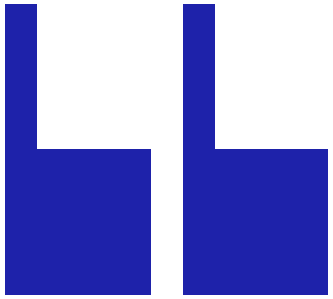


Что такое интрузивные контейнеры

Отличие STL контейнеров от Boost.Intrusive

Как устроена связь объекта с хранилищем

Примеры использования



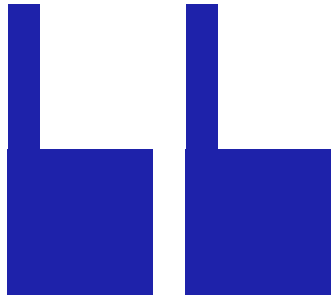
Containers are objects that store other objects. They control allocation and deallocation of these objects through constructors, destructors, insert and erase operations.

ISO IEC 14882, 22.2.1 General container requirements

```
class MyClass {
  // ...
};

std::list<MyClass> data;

MyClass item;

data.push_back(item);
data.emplace_back(item);
data.emplace_back(std::move(item));
```

```
struct _List_node_base
{
  _List_node_base* _M_next;
  _List_node_base* _M_prev;

  static void
  swap(_List_node_base& __x, _List_node_base& __y) _GLIBCXX_USE_NOEXCEPT;

  void
  _M_transfer(_List_node_base* const __first,
              _List_node_base* const __last) _GLIBCXX_USE_NOEXCEPT;

  void
  _M_reverse() _GLIBCXX_USE_NOEXCEPT;
  // ...
};
```

```
/// An actual node in the %list.
template<typename _Tp>
struct _List_node : public __detail::_List_node_base
{
#if __cplusplus >= 201103L
  __gnu_cxx::__aligned_membuf<_Tp> _M_storage;
  _Tp*       _M_valptr()       { return _M_storage._M_ptr(); }
  _Tp const* _M_valptr() const { return _M_storage._M_ptr(); }
#else
  _Tp _M_data;
  _Tp*       _M_valptr()       { return std::__addressof(_M_data); }
  _Tp const* _M_valptr() const { return std::__addressof(_M_data); }
#endif
};
```



Intrusive containers offer predictability when inserting and erasing objects since no memory management is done with intrusive containers.

Properties of Boost.Intrusive containers

```
template<class Config>
class list_impl
   :  private detail::clear_on_destructor_base<list_impl<Config> >
{
   typedef typename Config::value_traits              value_traits;
   typedef typename value_traits::pointer             pointer;
   typedef typename value_traits::node_traits         node_traits;
   // ...
   struct data_t : public value_traits
   {
      // ...
   } data_;
};
```

STL

- Контролирует lifetime
- Хранит элементы
- Хранит все мета-данные о внутренней структуре хранилища внутри контейнера

Boost.Intrusive

- Не контролирует lifetime
- Не хранит элементы
- Требует хранить мета-данные о связи элемента с хранилищем внутри элемента

Что такое интрузивные контейнеры

Отличие STL контейнеров от Boost.Intrusive

Как устроена связь объекта с хранилищем

Примеры использования

Мета-данные внутри элемента

- Элемент должен хранить hook – функтор для связи элемента с контейнером
- hook позволяет изнутри элемента:
  - `hook::is_linked()` – проверить наличие в конкретном контейнере
  - `hook::unlink()` – удалить из контейнера
  - `hook::swap_nodes()` – переложить элемент в другой контейнер

Как хранить hook?

- Boost.Intrusive предлагает несколько механизмов внедрения хуков:
  - `bi::base_hook` — наследование
  - `bi::member_hook` — публичное поле
  - `bi::function_hook` — публичный метод

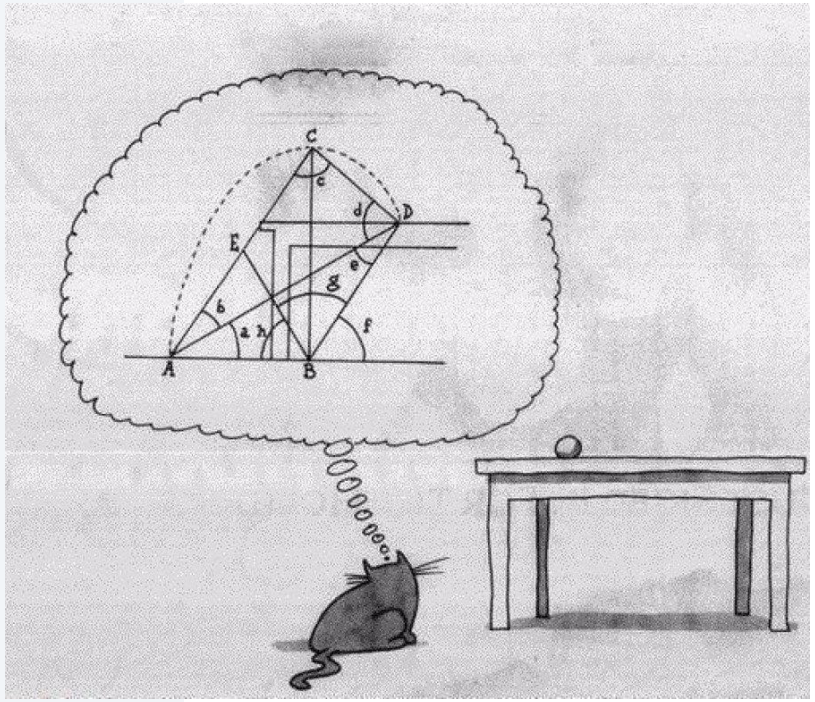
```
namespace bi = boost::intrusive;
class MyClass
  : public bi::list_base_hook<>
{};

bi::list<MyClass,
  bi::base_hook<MyClass,
    bi::list_base_hook<>>
> data;

MyClass item;
data.push_back(item);
```

base_hook

```
namespace bi = boost::intrusive;
class MyClass {
  bi::list_member_hook<> hook;
};

bi::list<MyClass,
  bi::member_hook<MyClass,
    bi::list_member_hook<>,
    &MyClass::hook>
> data;

MyClass item;
data.push_back(item);
```

Настройки контейнера

- `bi::constant_time_size<false>`
  - вычисляемый размер
- `bi::constant_time_size<true>`
  - размер хранится полем



Настройки хуков

- `bi::tag` — идентификатор, чтобы хранить несколько хуков для разных контейнеров

```
namespace bi = boost::intrusive;

struct by_ip;
struct by_fqdn;

class MyClass {
  bi::list_member_hook<bi::tag<by_ip>> hook_by_ip;

  bi::list_member_hook<bi::tag<by_fqdn>> hook_by_fqdn;
  // ...
};
```

Настройки хуков

- `bi::tag` — идентификатор, чтобы хранить несколько хуков для разных контейнеров
- `bi::link_mode` — режим вставки/удаления элементов из контейнера

```
namespace bi = boost::intrusive;


class MyClass {
  bi::list_member_hook<
    bi::link_mode<bi::normal_link>> hook;
  // ...
};
```

normal_link

Ничего не делает

— Пользователь обязан удалить элемент вручную

```
namespace bi = boost::intrusive;


class MyClass {
  bi::list_member_hook<
    bi::link_mode<bi::safe_link>> hook;
  // ...
};
```

safe_link

Следит

— Используется по умолчанию

— Проверяет, что элемент удалился из всех контейнеров (assert)

```
namespace bi = boost::intrusive;


class MyClass {
  bi::list_member_hook<
    bi::link_mode<bi::auto_unlink>> hook;
  // ...
};
```

auto_unlink

Сам удаляет

- Удаляет элемент из всех контейнеров по деструктору хука
- Thread unsafe!
- Несовместим с constant time size

Настройки хуков

- `bi::tag` — идентификатор, чтобы хранить несколько хуков для разных контейнеров
- `bi::link_mode` — режим вставки/удаления элементов из контейнера
- `bi::void_pointer` — пользовательский тип указателя внутри контейнера

```
namespace bi = boost::intrusive;
namespace ip = boost::interprocess;

class MyClass
{
  bi::list_member_hook<
    bi::void_pointer<
      ip::offset_ptr<void>>> hook;
  // ...
};
```

void_pointer

Summary

- Элемент хранит hook для связи с контейнером
- Можно добавлять несколько хуков, чтобы хранить элемент в нескольких контейнерах
- С помощью хука можно проверить, лежит ли элемент в контейнере, и удалить его из контейнера
- Поведение хуков настраивается
- Поведение контейнера настраивается

- slist
- list
- set
- multiset
- unordered_set
- unordered_multiset
- rbtree
- avl_set
- avl_multiset
- avl_tree
- splay_set
- splay_multiset
- splay_tree
- sg_set
- sg_multiset
- sg_tree
- treap_set
- treap_multiset
- treap

Что такое интрузивные контейнеры

Отличие STL контейнеров от Boost.Intrusive

Как устроена связь объекта с хранилищем

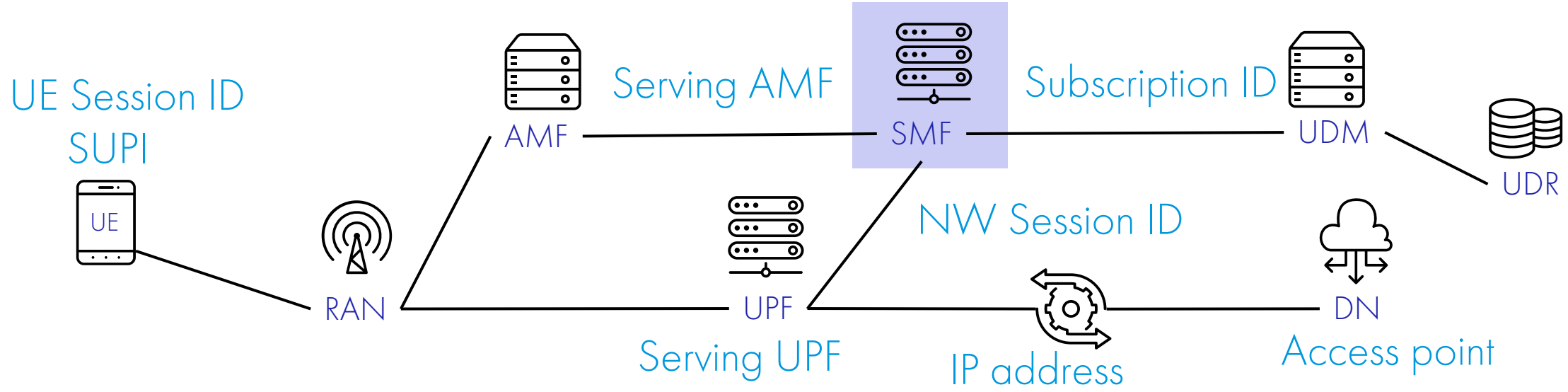
Примеры использования

Для чего использовать

- Построение индексов по множеству ключей, в том числе заранее неизвестным ключам
- Управление пулами коротко живущих объектов
- Хранение объектов со сложной аллокацией
- Получение итератора из указателя/ссылки за константное время
- Строгие требования к времени отклика

```
class Session {
  supi_hook_t hook_by_supi;
  seid_hook_t hook_by_seid;
  ip_hook_t hook_by_ip;
  upf_hook_t hook_by_upf;
  // ...
};
```

```
bi::unordered_multiset<...> by_supi;
bi::unordered_set<...> by_seid;
bi::unordered_set <...> by_ip;
bi::avl_multiset<...> by_upf;
bi::avl_multiset<...> by_amf;
bi::avl_multiset<...> by_apn;
// ...
```



Преимущества

- Ручное управление памятью ⟶ более строгие гарантии производительности
- Константные асимптотики для удаления или подсчёта количества
- Информация о контейнере встроена в элемент

Недостатки

- Ручное управление памятью ⟶ необходимость следить за временем жизни объектов
- Дополнительный расход памяти или замедление операций
- Необходимость изменять объект (добавлять мета-информацию о контейнере)

Q&A

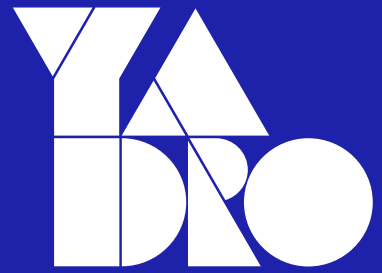
Спасибо!

Boost.Intrusive documentation

Source code glibc std::list

Source code boost::intrusive::list

Benchmark std::list VS bi::list



БУДУЩЕЕ
В НАШИХ
РУКАХ