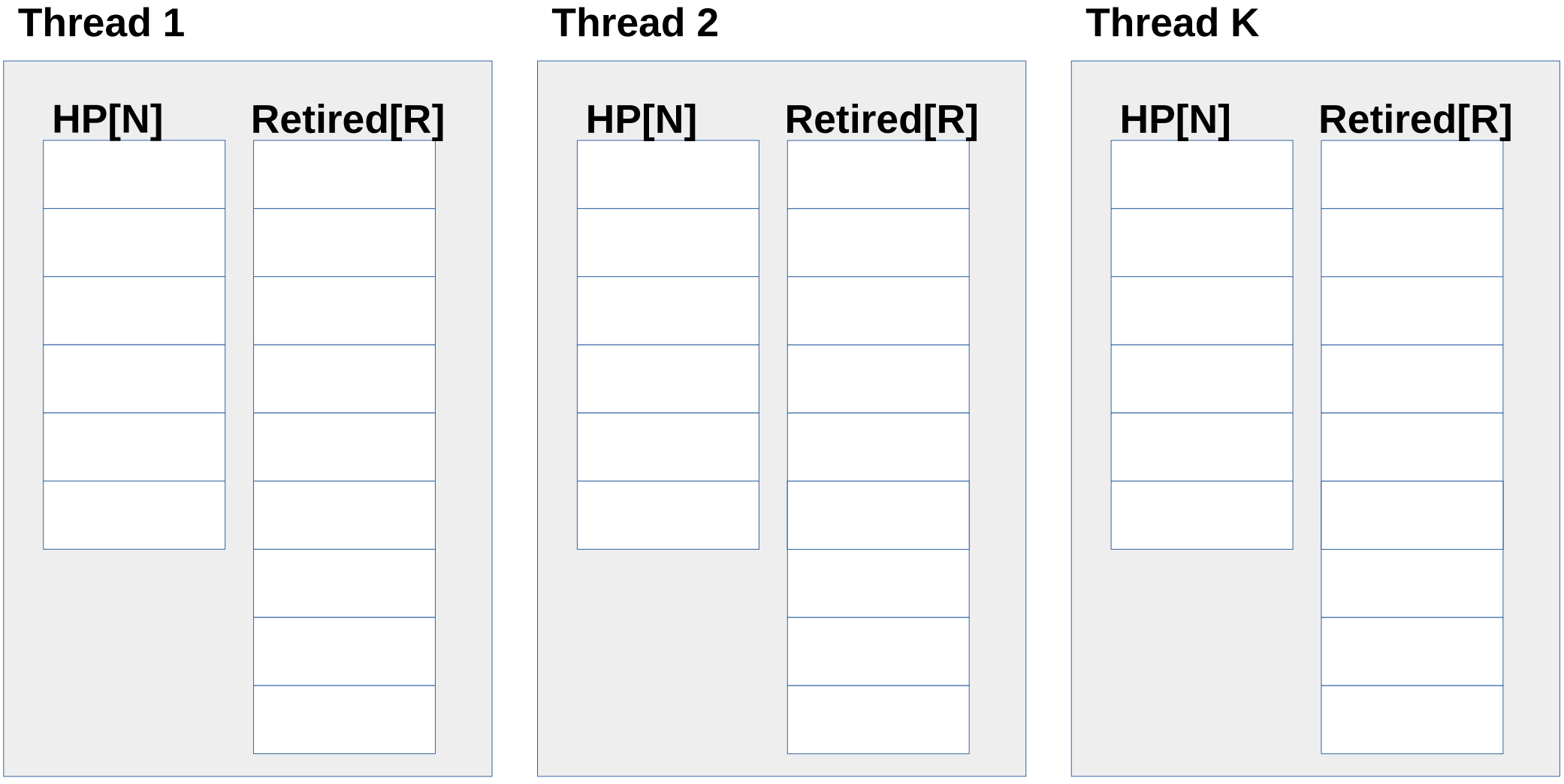
Hazard Pointer

P1121R0

```
struct Foo {
    int id;
    std::string name;
    // ...
};
// Глобальные данные
std::atomic<Foo*> g_foo;

// Вызывается часто, из разных threads
void print_foo() {
    Foo* foo = g_foo.load();
    printf( "id=%d name=%s\n", foo->id, foo->name.c_str());
}

// Вызывается редко, из разных threads
void update_foo(foo* new_foo)
{
    Foo* prev = g_foo.exchange( new_foo );
    delete prev;  // упс...
}
```

```cpp
struct Foo: hazard_pointer_obj_base<Foo> {
    int id;
    std::string name;
    // ...
};
// Глобальные данные
std::atomic<Foo*> g_foo;

// Вызывается часто, из разных threads
void print_foo() {
    hazard_pointer hp = make_hazard_pointer();
    Foo* foo = hp.protect( g_foo ); // g_foo.load();
    printf( "id=%d name=%s\n", foo->id, foo->name.c_str());
}

// Вызывается редко, из разных threads
void update_foo(foo* new_foo)
{
    Foo* prev = g_foo.exchange( new_foo );
    prev->retire(); // aka delete prev;
}
```

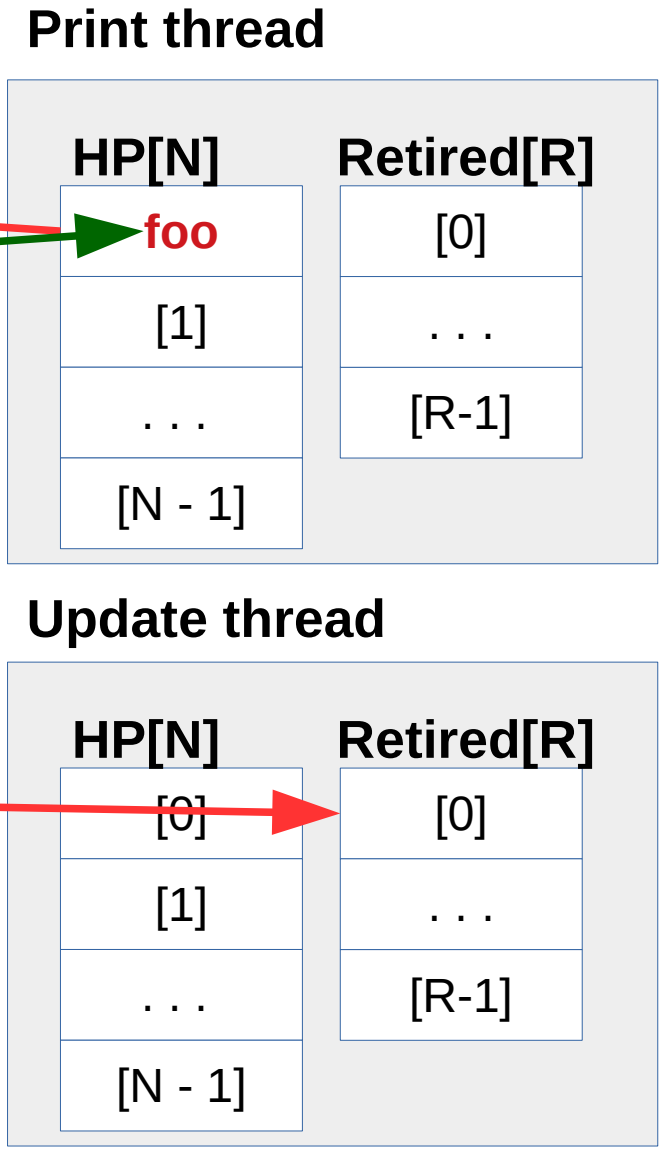

Thread 1
HP[N]
Retired[R]
Thread 2
HP[N]
Retired[R]
Thread K
HP[N]
Retired[R]
Hazard Pointer Domain, R > N * K

[P1121]

```
class hazard_pointer_domain;

template <
  typename T,
  typename D = default_delete<T>>
class hazard_pointer_obj_base;

class hazard_pointer;
```

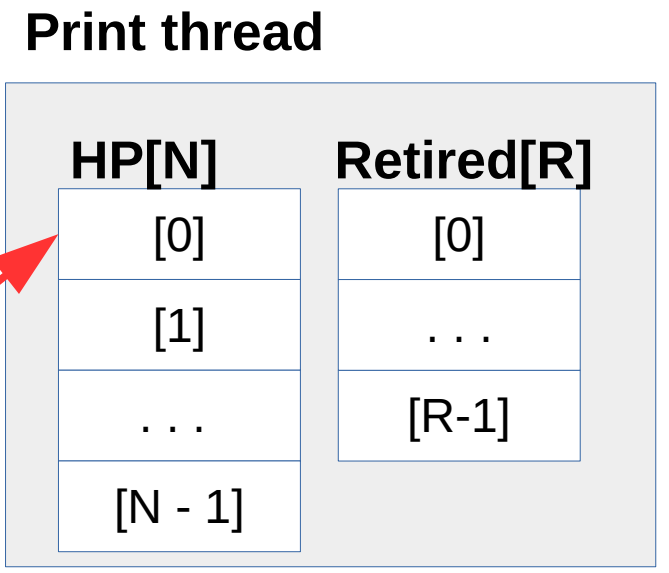
+ несколько функций

```
// Вызывается часто, из разных threads
void print_foo() {
    hazard_pointer hp =
        make_hazard_pointer();
    Foo* foo = hp.protect( g_foo );
    printf( "id=%d name=%s\n",
      foo->id, foo->name.c_str());
}


// Вызывается редко, из разных threads
void update_foo(foo* new_foo)
{
    Foo* prev = g_foo.exchange( new_foo );
    prev->retire();
}
```



```
// Вызывается часто, из разных threads
void print_foo() {
    hazard_pointer hp =
        make_hazard_pointer();
    Foo* foo = hp.protect( g_foo );
    printf( "id=%d name=%s\n",
      foo->id, foo->name.c_str());
}
```

```
class hazard_pointer {
  implementation_defined  hp_;
public:
  template <typename T>
  T* protect( std::atomic<T*> const& p )
  {
    T* ptr;
    do {
      hp_ = ptr = p.load();
    } while( ptr != p.load());
    return ptr;
  }
};
```

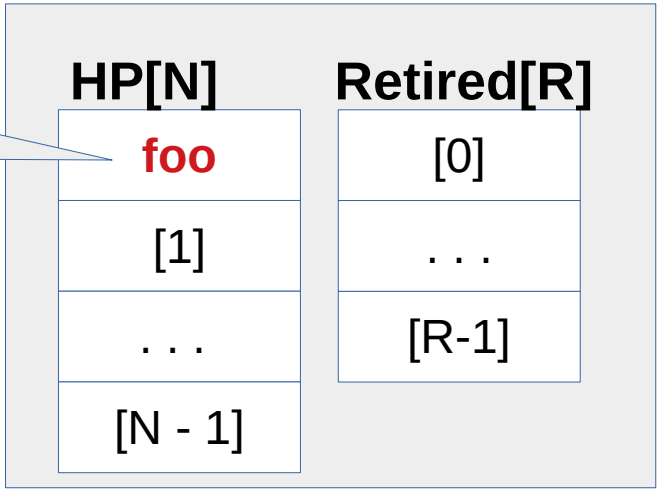


!

HP[i] — пишет только поток-владелец!

Что такое hazard_pointer?

P1121: `hazard_pointer_obj_base<T, D>*`
vs
libcds: `void*`

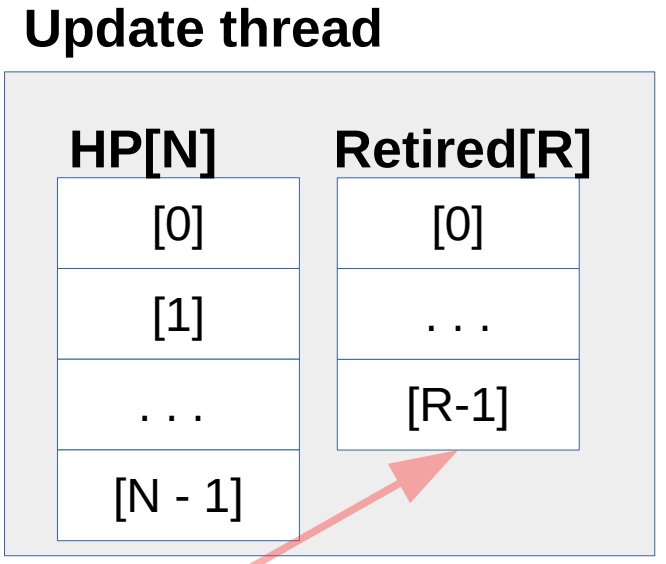



```
class hazard_pointer {
public:
   hazard_pointer()
   { //alloc free slot from HP[N] }
   hazard_pointer( hazard_pointer&& hp)
     : hp_( hp.hp_ )
   {
       hp.hp_ = nullptr;
   }
   bool empty() const { return hp_ == nullptr; }
   // ...
private:
     HP*   hp_; // ptr на HP[i]
};
```

! Обычно N — малое (4 .. 8)

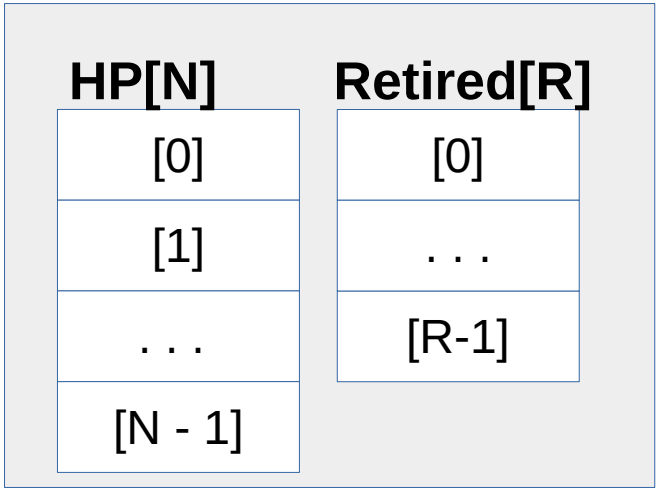
```
template <typename T>
struct guarded_ptr {
   hazard_pointer hp_;    // защищает элемент
   T *            ptr_;   // элемент списка
   guarded_ptr(std::atomic<T *>& p)
         { ptr_ = hp_.protect( p ); }
   guarded_ptr( guarded_ptr&& gp): hp( std::move(gp.hp_)),
       ptr_( gp.ptr_ ) { gp.ptr_ = nullptr; }
   ~guarded_ptr() { hp_.clear(); }
   T * operator ->() const { return ptr_; }
   explicit operator bool() const { return ptr_ != nullptr; }
};
guarded_ptr gp = list.find( key ); // lock-free list
if ( gp ) {
   // можно безопасно обращаться к полям T через gp
}
```

```
struct Foo: hazard_pointer_obj_base<Foo>
{ … }

// Вызывается редко, из разных threads
void update_foo(foo* new_foo)
{
    Foo* prev = g_foo.exchange( new_foo );
    prev->retire();
}
```




```
template <typename T, typename D = std::default_delete<T>>
class hazard_pointer_obj_base {
public:
    void retire( D reclaim = {},
      hazard_pointer_domain& domain = default_domain())
    {
        domain.cur_thread.Retired.push( this, reclaim );
    }
};
```

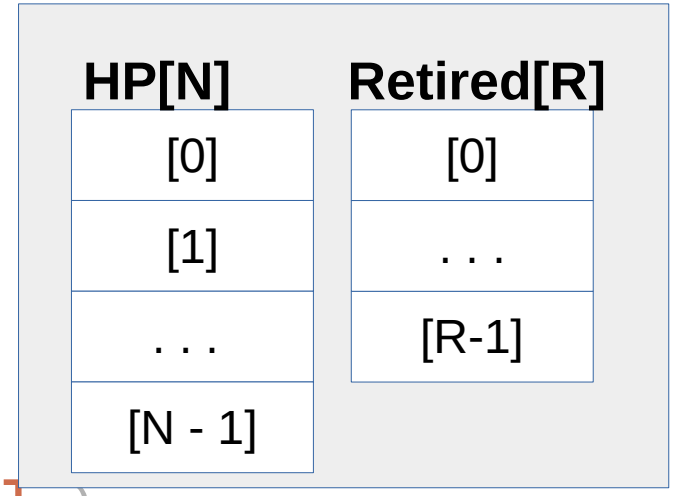
```
domain.Retired.push( this, reclaim ):

Push to current_thread.Retired

if ( current_thread.Retired is full ) {

   impl_defined guarded[N*K] =
            union HP[N] for all K thread;

   foreach ( p in current_thread.Retired[R] )
      if ( p not in guarded[] )
         reclaim( p ); // ~= delete p
}
```




```
template <typename T, typename D = std::default_delete<T>>
class hazard_pointer_obj_base { … }
```

R > N * K !!!

```
domain.Retired.push( this, reclaim ):

Push to current_thread.Retired

if ( current_thread.Retired is full ) {
   // чтение HP[i] всех потоков
   impl_defined guarded[N*K] =
            union HP[N] for all K thread;

   foreach ( p in current_thread.Retired[R] )
      if ( p not in guarded[] )  // сравнение указателей!!!
         reclaim( p ); // ~= delete p
}
```




```
[P1121R0]
struct Foo: Bar, Baz, hazard_pointer_obj_base<Foo> {
   // ...
};
```

[P1121R0]

```
struct Foo: Bar, Baz, hazard_pointer_obj_base<Foo> {
    // ...
};
```

HP[i] stores hazard_pointer_obj_base<Foo>*

[libcds]

```
struct Foo: Bar, Baz {
    // ...
};
```

HP[i] stores Foo* casted to void*

Для контейнера Container<T>
всегда в HP[i] хранит reinterpret_cast<void*>(T*)
Соблюдение этого правила — на совести разработчика

[P1121]

```
template <typename T, typename D = std::default_delete<T>>
class hazard_pointer_obj_base {
public:
    void retire( D reclaim = {},
      hazard_pointer_domain& domain = default_domain())
    {
       domain.cur_thread.Retired.push( this, reclaim );
    }
};
```

Класс с единственным методом — переходником.
Должен быть базой, если хотим применять Hazard Pointer.

! Объект знает, как себя удалять.

```
template <typename T, typename D = std::default_delete<T>>
class hazard_pointer_obj_base {
public:
   void retire( D reclaim = {},
     hazard_pointer_domain& domain = default_domain())
   {
      domain.cur_thread.Retired.push( this );
   }
};
```

```
// можно было бы так
class hazard_pointer_domain {
public:
   template <typename T, typename D = std::default_delete<T>>
   void retire( T* ptr, D disposer = {} )
   {
      cur_thread.Retired.push( ptr, disposer );
      // что есть Retired?..
   }
};
```

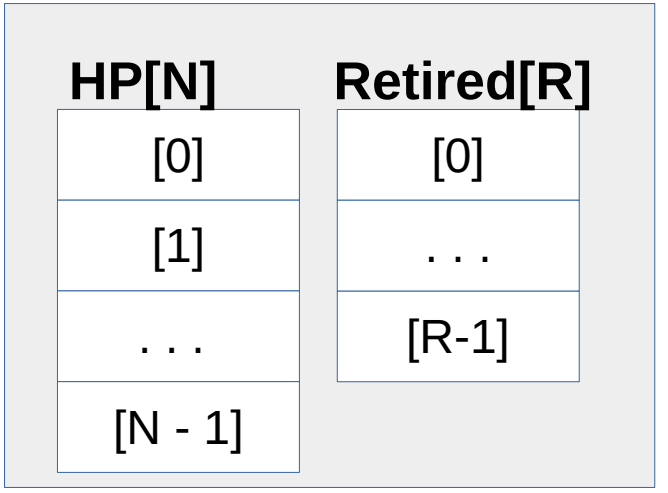
Что такое Retired[i]?

[P1121R0]

```
typedef hazard_pointer_obj_base<T,D>
        retired_ptr; // bad
```

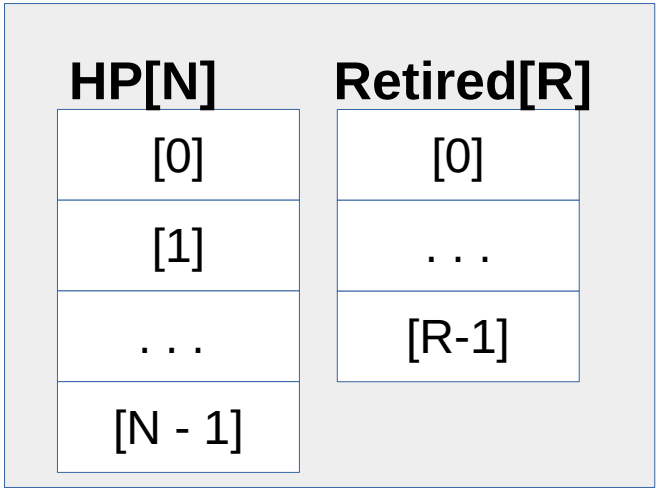
[libcds]

```
struct retired_ptr {
    void* ptr_;
    void (* disposer_)(void*);
}; // потерян тип?..
```



[libcds]

```
struct retired_ptr {
    void* ptr_;
    void (* disposer_)(void*);
};

template <typename T, typename D>
void retire( T* ptr, D ) {
    struct type_recovery {
        static void func(void* ptr) {
          D()( static_cast<T*>( ptr ));
        }
    };
    current_thread().Retired.push(
      retired_ptr( ptr, type_recovery::func );
    );
}

// Вызов удаления: Retired[i].disposer_( Retired[i].ptr_ );
```



Интрузивные контейнеры

boost::intrusive (bi): container<T>

- no copy for T
- no move for T
- no allocation — best для lock-free контейнеров!

Boost — сильная интрузивность — no allocation

Libcds — слабая интрузивность:
No allocation for user data T
Можно аллоцировать внутренние структуры (nodes)

Гвоздь в P1121?..

Std:

```
template <typename T>
class list {
   struct node {
      node* next_;
      node* prev_;
      T     data;
   };
   // ...
};
```

Каждый контейнер имеет тип ноды:

```
struct list_node {
    list_node* next_;
    list_node* prev_;
};
```

Мы встраиваем тип ноды в свой тип (hook)

Base hook:

```
typedef bi::list_base_hook<> my_list_hook;
struct Foo: public my_list_hook {
    std::string data_;
    //  ...
};

bi::list<Foo, bi::base_hook<my_list_hook>> list_;
```

```
struct list_node {
    list_node* next_;
    list_node* prev_;
};
```

Member hook:

```
typedef bi::list_member_hook<> my_list_hook;
struct Foo {
    std::string  data_;
    my_list_hook hook_;
    //   ...
};

bi::list<Foo,
    bi::member_hook<Foo, my_list_hook, &Foo::hook_>
> list_;
```

Function hook:

```
typedef bi::list_member_hook<> my_list_hook;
typedef bi::set_member_hook<>  my_set_hook;
struct Foo {
    std::string data_;
    union {
       my_list_hook  list_hook_;
       my_set_hook   set_hook_;
    };
    //  ...
};
```

В контейнере задаются две функции (traits):

- `T* node_to_value( hook* node );`
- `hook* value_to_node( T* val );`

Одни и те же данные — во многих контейнерах:

```
struct primary_tag {};
struct key_tag {};
typedef bi::list_base_hook<>                list_hook;
typedef bi::set_base_hook<primary_tag> primary_hook;
typedef bi::set_base_hook<key_tag>       key_hook;

struct Foo:
    public list_hook,     // для списка
    public primary_hook, // индекс
    public key_hook       // индекс
{
   int  primary_key_; // первичный ключ
   std::string key_;  // вторичный ключ
   // ...
};

bi::list<Foo, bi::base_hook<list_hook>>    list_;
bi::set< Foo, bi::base_hook<primary_hook>> primary_set_;
bi::set< Foo, bi::base_hook<key_hook>>     index_set_;
```

hazard_pointer_obj_base:

```
struct primary_tag {};
struct key_tag {};
typedef bi::list_member_hook<>                list_hook;
typedef bi::set_member_hook<primary_tag> primary_hook;
typedef bi::set_member_hook<key_tag>       key_hook;

struct Foo // наследование запрещено
{
    int  primary_key_; // первичный ключ
    std::string key_;  // вторичный ключ
    list_hook       list_hook_;
    primary_hook   primary_key_hook_;
    key_hook        key_hook_;
    // . . . прочие поля
};
```

Куда воткнуть hazard_pointer_obj_base<Foo>?..

Интрузивные контейнеры и hazard pointer

[P1121R0]

Требуют наследования типа T от `hazard_pointer_obj_base<T,D>`

Следствие: применим только base_hook

[libcds]

Hazard pointer == void*

Ничего не требует от типа T

Можно применять любой тип hook

Сохранение правильного ptr — на совести разработчика

[P1121]

https://github.com/facebook/folly

[libcds]

https://github.com/khizmax/libcds