Writing good std::future<C++>

Anton Bikineev, 2016

STE||AR GROUP

C++17 Standardization process. What we wanted and expected…

- concepts
- ranges
- modules
- parallel algorithms
- *.then* on futures and other concurrency features
- executors
- coroutines with *await*
- transactional memory
- contracts
- fold expressions
- uniform function call syntax
- constexpr lambdas
- filesystem library
- networking library
- library features, e.g. *any*, *optional*, etc…
- etc…

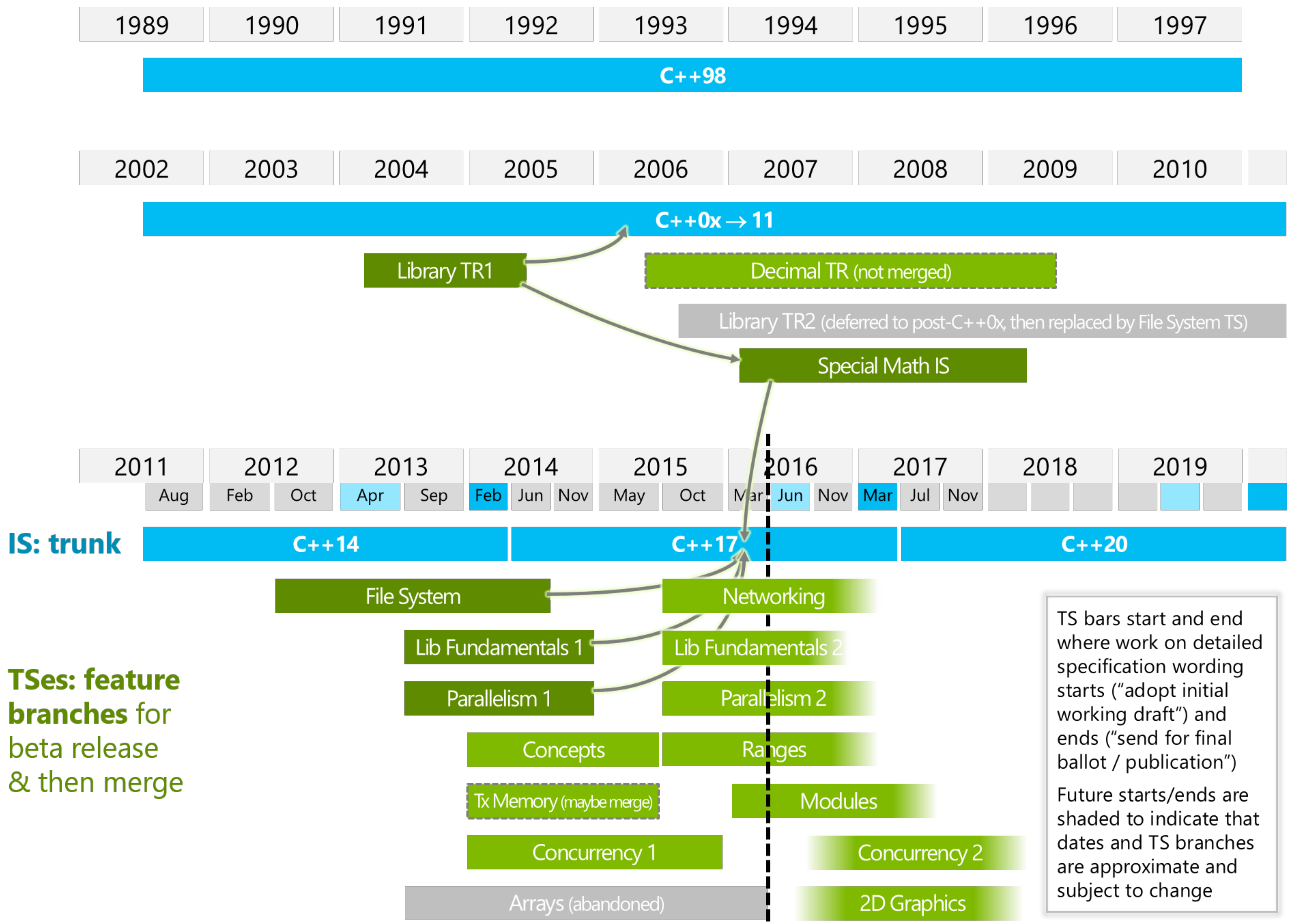
C++17 Standardization process. What we've got.

- concepts
- ranges
- modules
- parallel algorithms
- *.then* on futures and other concurrency features
- executors
- coroutines with *await*
- transactional memory
- contracts
- fold expressions
- uniform function call syntax
- constexpr lambdas
- filesystem library
- networking library
- library features, e.g. *any*, *optional*, etc…
- etc…

C++17 Standardization process. What we’ve got.

- concepts
- ranges
- modules
- parallel algorithms
- *.then* on futures and other concurrency features
- executors
- coroutines with *await*
- transactional memory
- contracts
- fold expressions
- uniform function call syntax
- constexpr lambdas
- filesystem library
- networking library
- library features, e.g. *any*, *optional*, etc…
- etc…

Kenny Kerr @kennykerr 7 march
C++11 attempted too much. C++14 fixed that.
C++17 attempts too little. We need to fix that.


1989
1990
1991
1992
1993
1994
1995
1996
1997
C++98
2002
2003
2004
2005
2006
2007
2008
2009
2010
C++0x → 11
Library TR1
Decimal TR (not merged)
Library TR2 (deferred to post-C++0x, then replaced by File System TS)
Special Math IS
2011
2012
2013
2014
2015
2016
2017
2018
2019
Aug
Feb
Oct
Apr
Sep
Feb
Jun
Nov
May
Oct
Mar
Jun
Nov
Mar
Jul
Nov
IS: trunk
C++14
C++17
C++20
File System
Networking
Lib Fundamentals 1
Lib Fundamentals 2
Parallelism 1
Parallelism 2
Concepts
Ranges
Tx Memory (maybe merge)
Modules
Concurrency 1
Concurrency 2
Arrays (abandoned)
2D Graphics
TSes: feature branches for beta release & then merge
TS bars start and end where work on detailed specification wording starts ("adopt initial working draft") and ends ("send for final ballot / publication")
Future starts/ends are shaded to indicate that dates and TS branches are approximate and subject to change

STE||AR GROUP

C++’s view on polymorphism

C++'s view on polymorphism

```
class Factory
{
public:
    virtual AbstractButton* createButton() = 0;
};
```

C++’s view on polymorphism

```
class Factory
{
public:
    virtual std::unique_ptr<AbstractButton> createButton() = 0;
};
```

C++'s view on polymorphism

```
class Factory
{
public:
    virtual std::unique_ptr<AbstractButton> createButton() = 0;
};


// use of polymorphic code
void createForm(Factory* factory)
{
    auto button = factory->createButton();
}
```

C++'s view on polymorphism

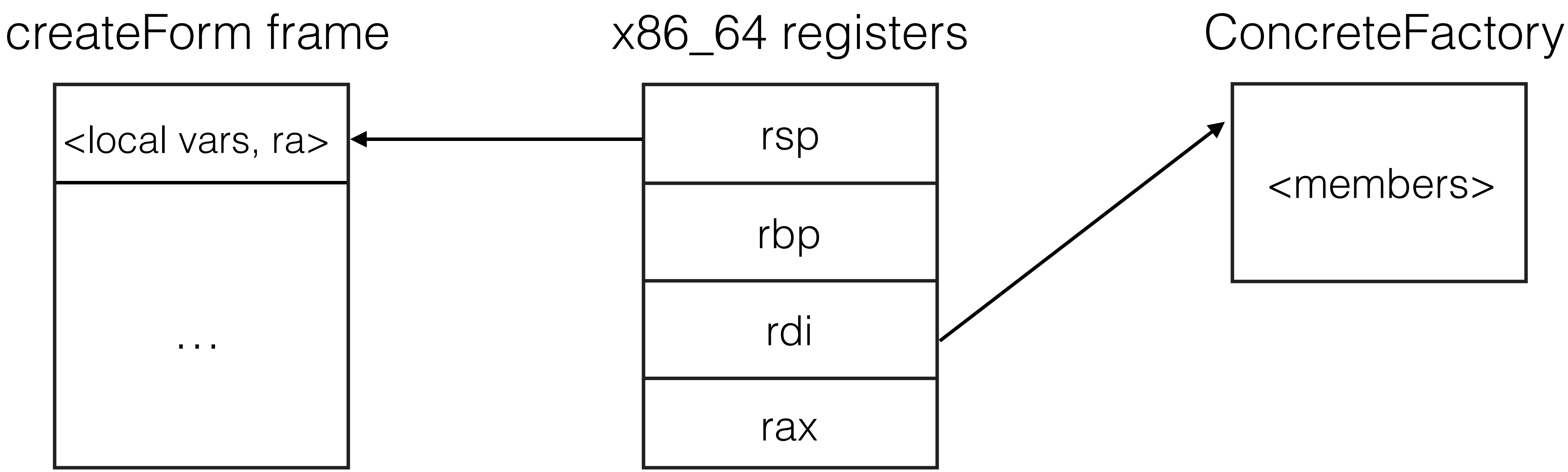
```
// use of polymorphic code
void createForm(Factory* factory)
{
    auto button = factory->createButton();
}
```

C++’s view on polymorphism

```
// use of polymorphic code
void createForm(Factory* factory)
{
    auto button = factory->createButton();
}
```

```
__Z10createFormP7Factory:
LFB2560:
    pushq   %rbp
LCFI0:
    movq    %rsp, %rbp
LCFI1:
    subq    $32, %rsp
    movq    %rdi, -24(%rbp)
    movq    -24(%rbp), %rax
    movq    (%rax), %rax
    addq    $8, %rax
    movq    (%rax), %rax
    leaq    -16(%rbp), %rdx
    movq    -24(%rbp), %rcx
    movq    %rcx, %rsi
    movq    %rdx, %rdi
    call    *%rax
    leaq    -16(%rbp), %rax
    movq    %rax, %rdi
    call    __ZNSt10unique_ptrI14AbstractButtonSt14default_deleteIS0_EED1Ev
    nop
    leave
LCFI2:
    ret
```

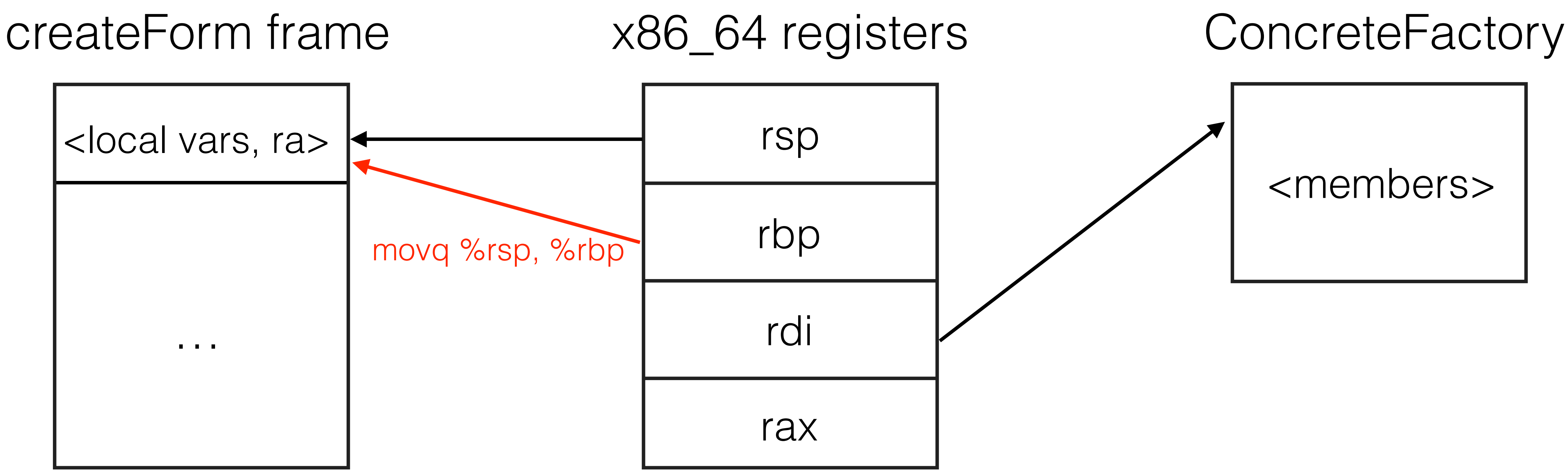
Dynamic polymorphism



the function itself

```
__ZN15ConcreteFactory12createButtonEv:
    pushq   %rbp
    ... ; some asm instrs
```

Dynamic polymorphism

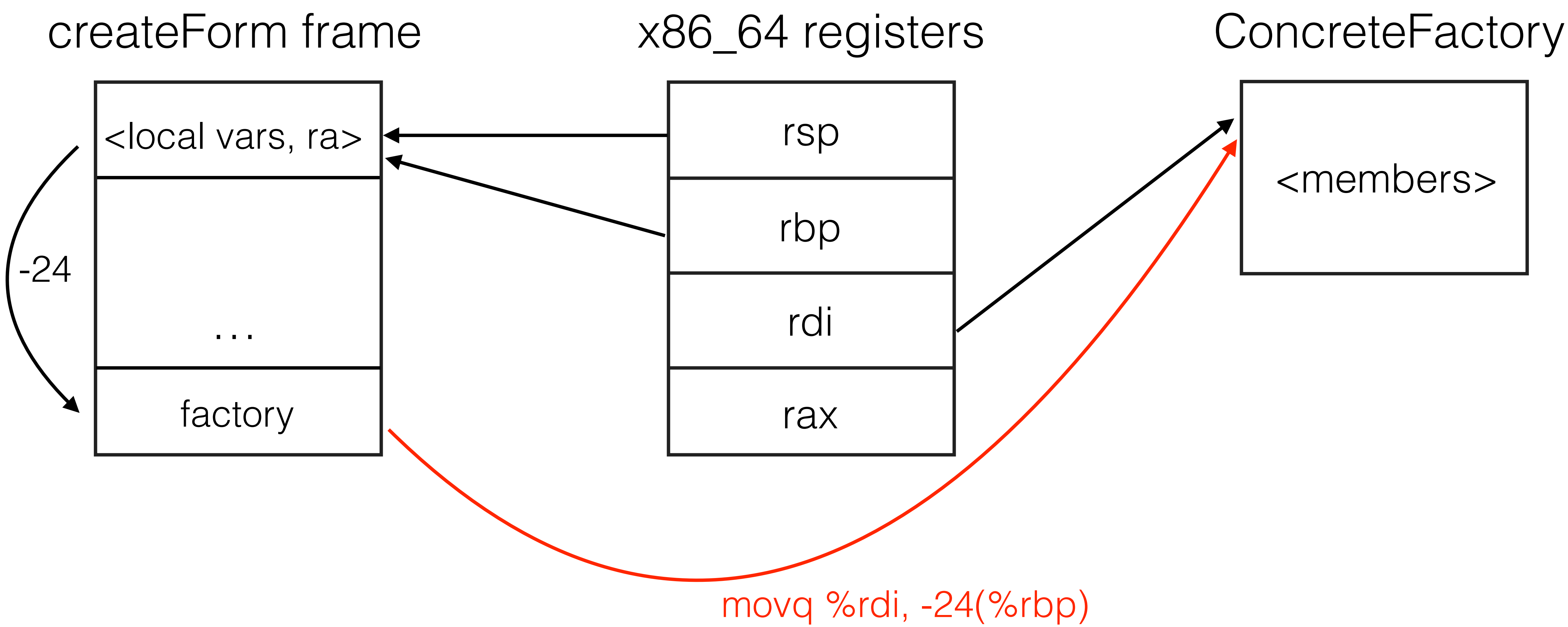


the function itself

```
__ZN15ConcreteFactory12createButtonEv:
    pushq   %rbp
    ... ; some asm instrs
```


STE||AR GROUP

Dynamic polymorphism

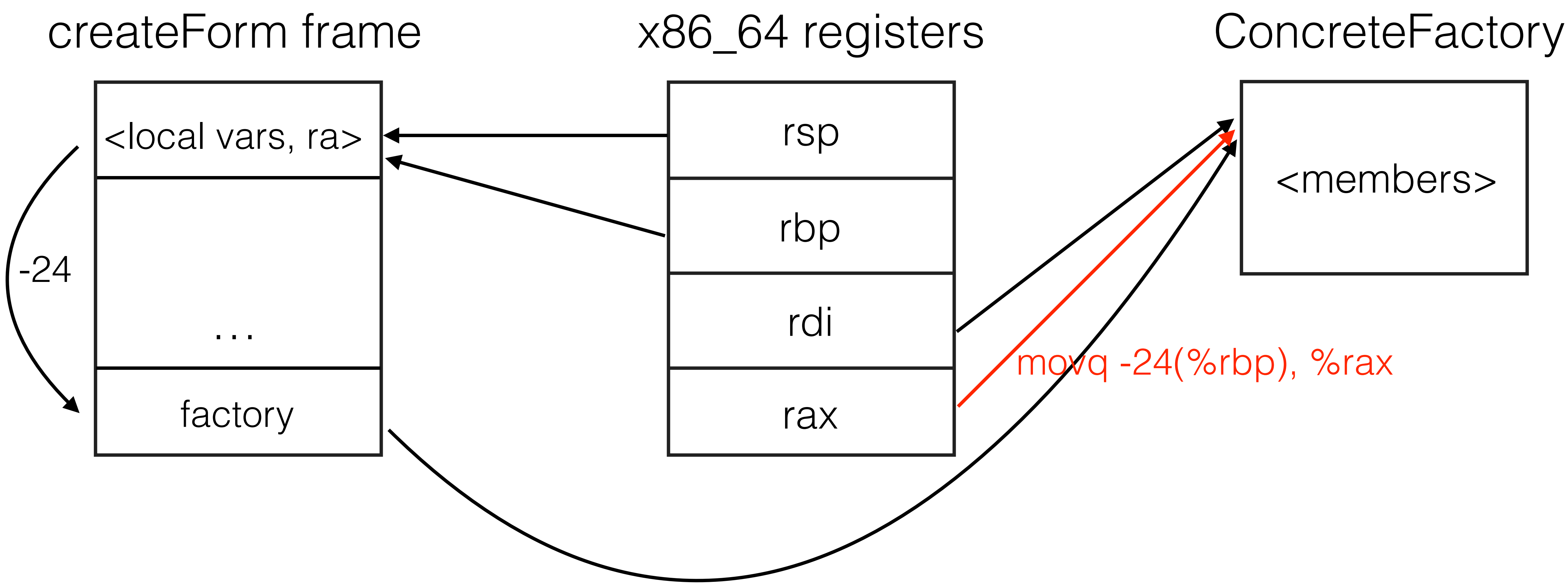


the function itself

```
__ZN15ConcreteFactory12createButtonEv:
    pushq   %rbp
    ... ; some asm instrs
```


STE||AR GROUP

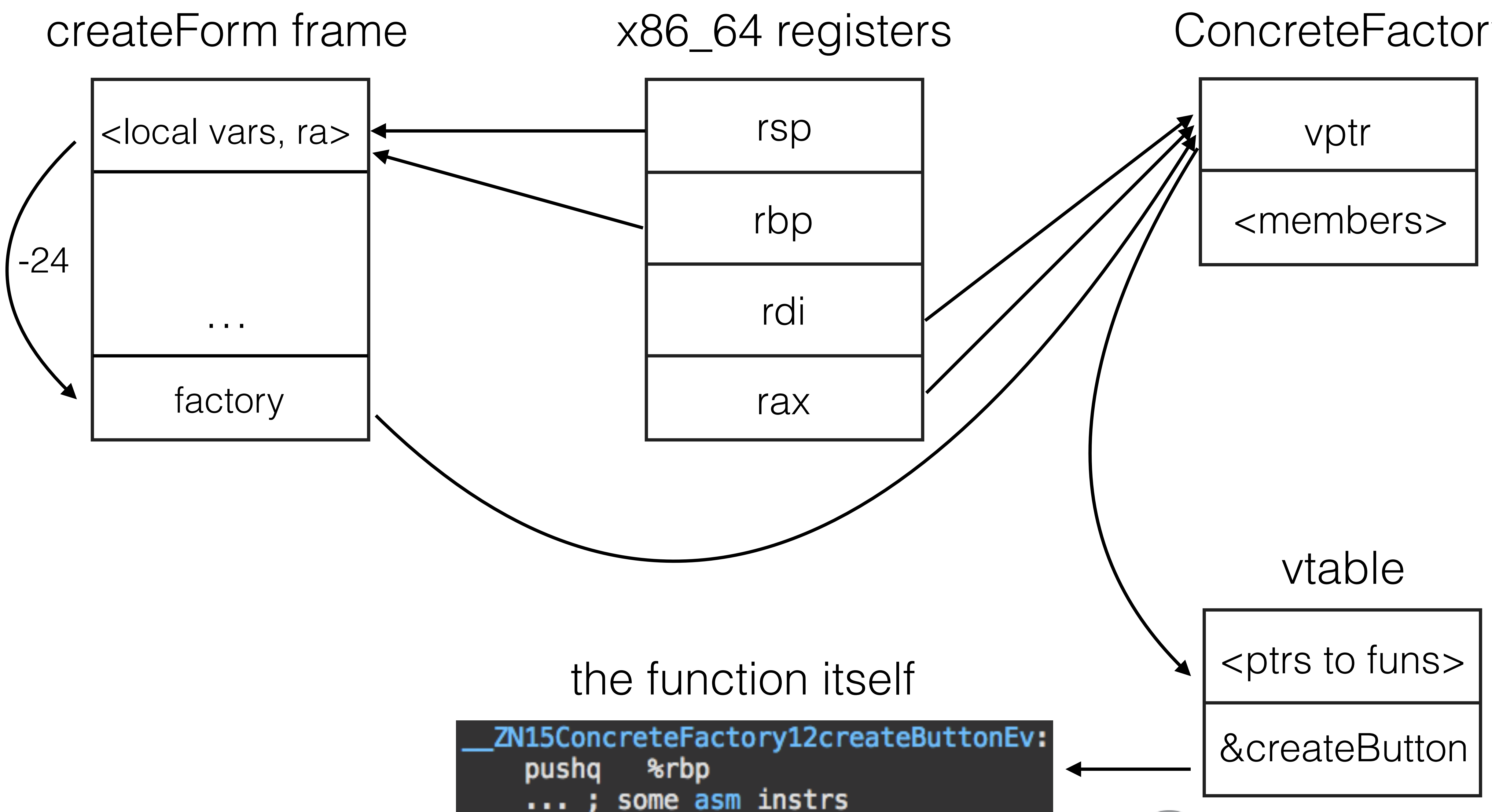
Dynamic polymorphism



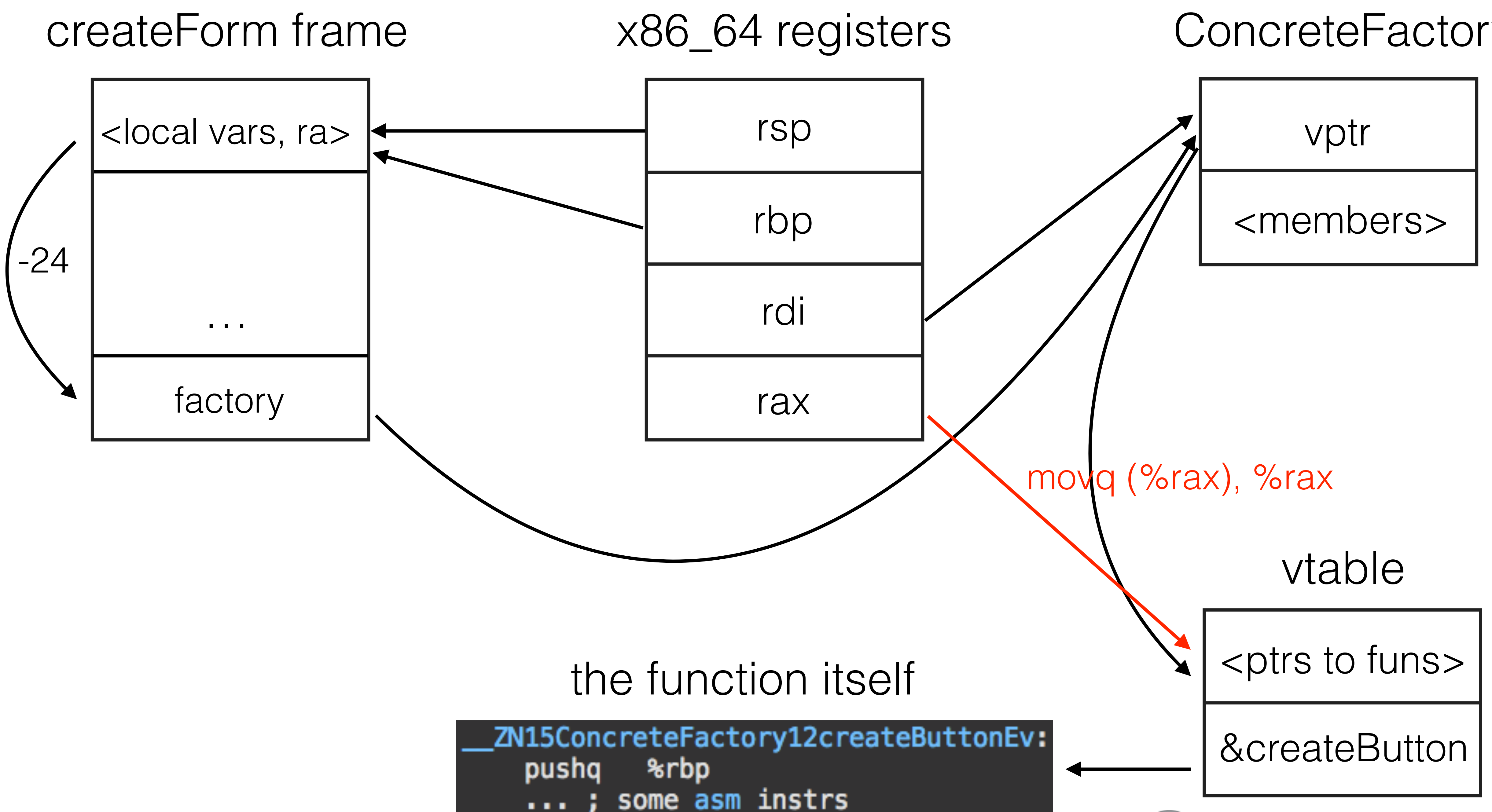
the function itself

```
__ZN15ConcreteFactory12createButtonEv:
    pushq   %rbp
    ... ; some asm instrs
```

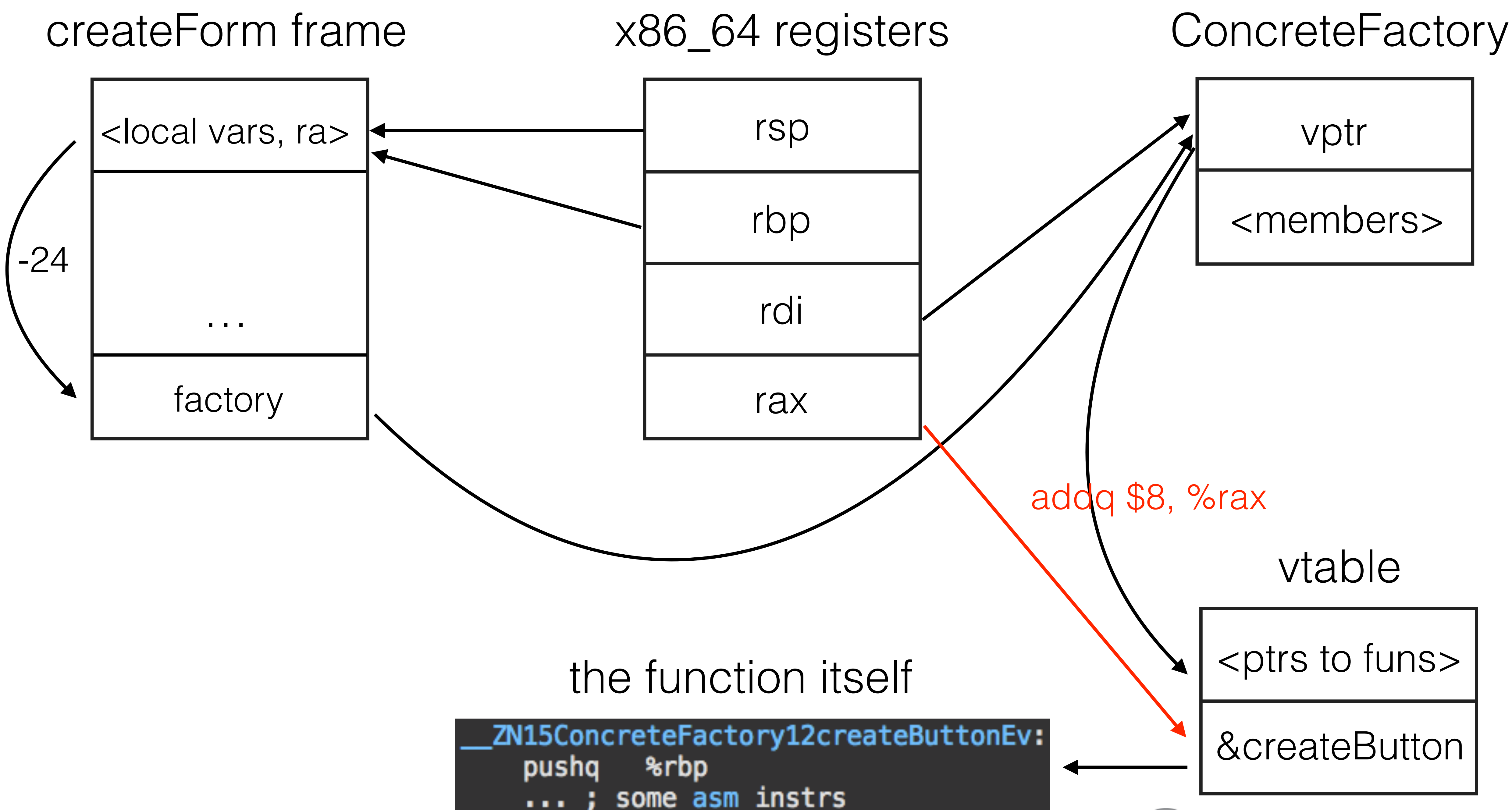
Dynamic polymorphism



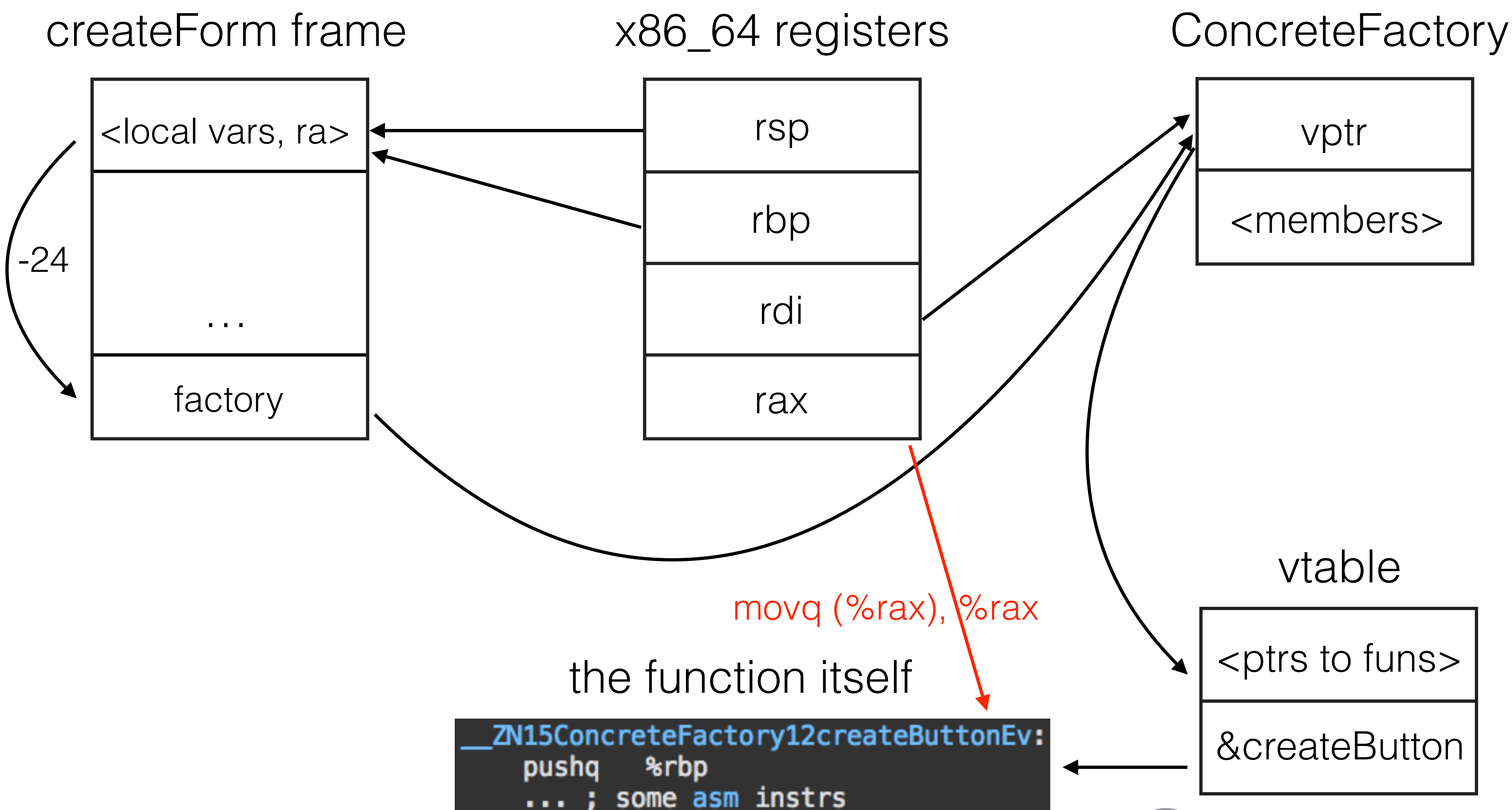
Dynamic polymorphism



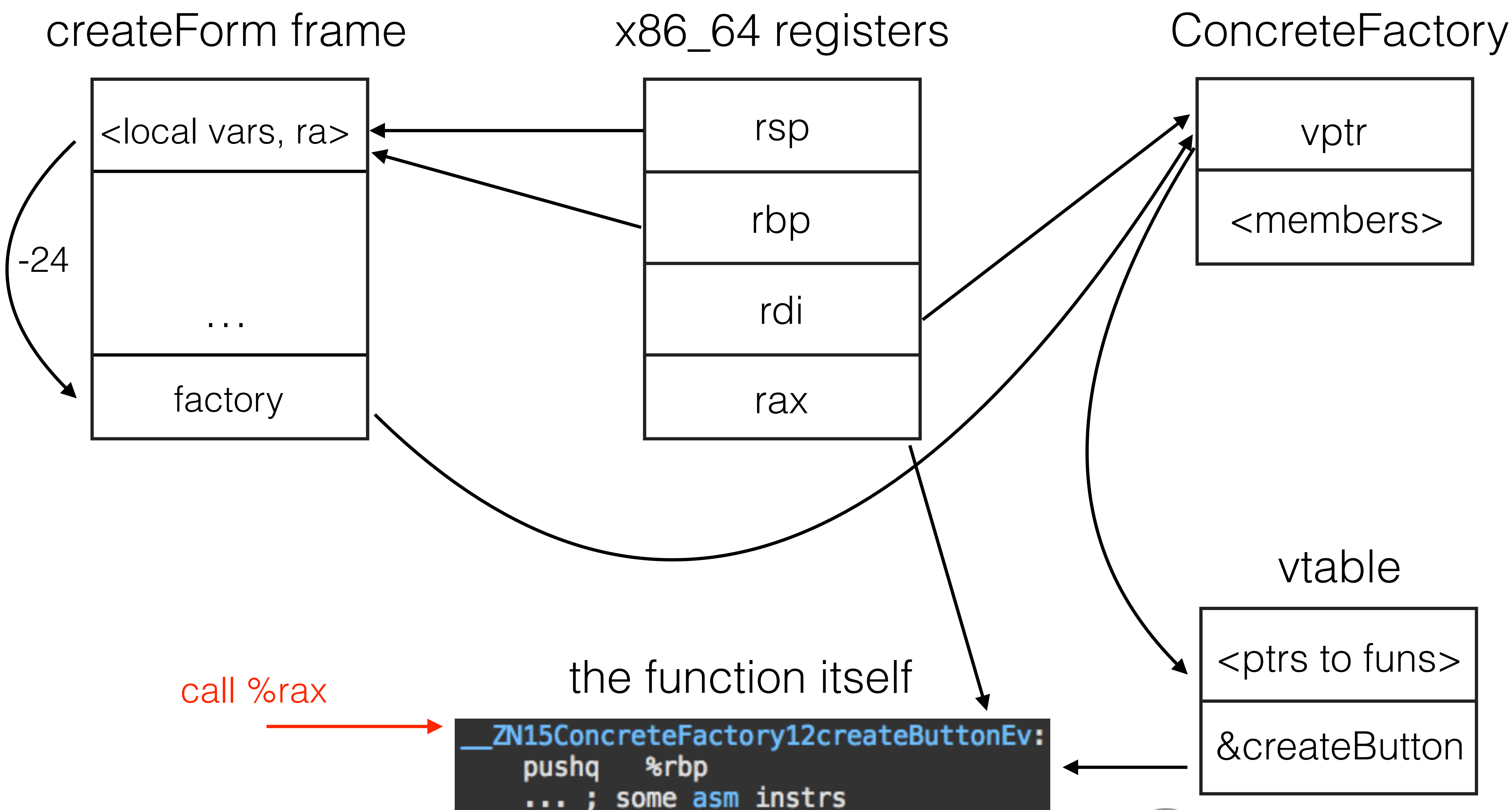
Dynamic polymorphism



Dynamic polymorphism

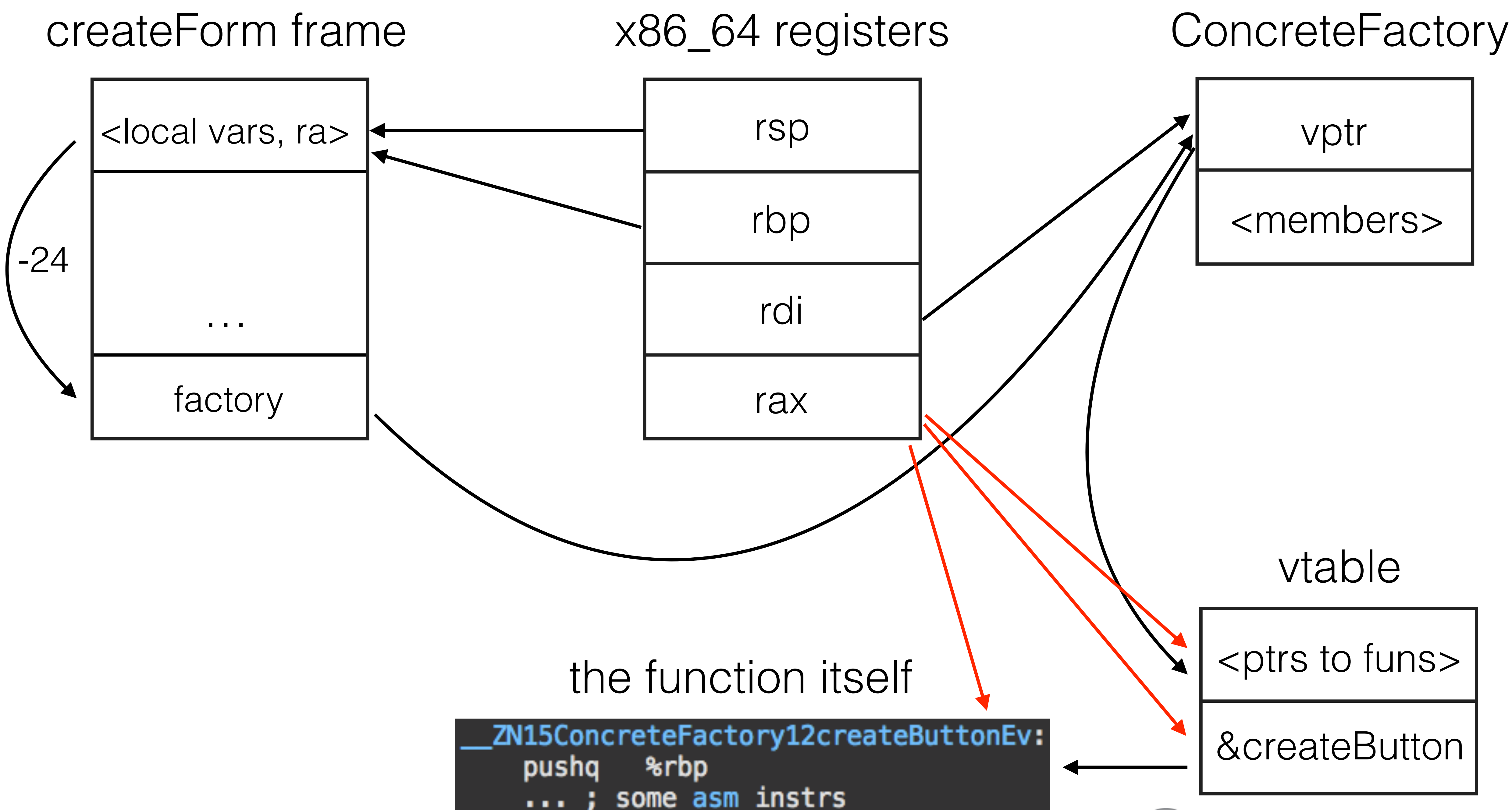


Dynamic polymorphism



STE||AR GROUP

Dynamic polymorphism overhead



STE||AR GROUP

C++ templates — solution?

```
template <class Factory>
void createForm(Factory& factory)
{
    auto button = factory.createButton();
}
```

C++ templates — solution?

```cpp
template <class Factory>
void createForm(Factory& factory)
{
    auto button = factory.createButton();
}

template void createForm(ConcreteFactory&);
```

```asm
__Z10createFormI15ConcreteFactoryEvRT_:
LFB2819:
    pushq   %rbp
LCFI27:
    movq    %rsp, %rbp
LCFI28:
    subq    $48, %rsp
    movq    %rdi, -40(%rbp)
    leaq    -16(%rbp), %rax
    movq    -40(%rbp), %rdx
    movq    %rdx, %rsi
    movq    %rax, %rdi
    call    __ZN15ConcreteFactory12createButtonEv
```

C++ static interface. CRTP

```
template <class Impl>
struct Factory
{
    std::unique_ptr<AbstractButton> createButton()
    {
        return static_cast<Impl*>(this)->createButton();
    }
};
```

C++ static interface. CRTP

```cpp
template <class Impl>
struct Factory
{
    std::unique_ptr<AbstractButton> createButton()
    {
        return static_cast<Impl*>(this)->createButton();
    }
};

struct ConcreteFactory: Factory<ConcreteFactory>
{
    std::unique_ptr<AbstractButton> createButton()
    {
        return std::make_unique<ConcreteButton>();
    }
};
```

C++ static interface. Usage

```
template <class Impl>
void createForm(Factory<Impl>& fact)
{
    auto button = fact.createButton();
}

template void createForm(Factory<ConcreteFactory>&);
```

```
__Z10createFormI15ConcreteFactoryEvR7FactoryIT_E:
LFB2820:
    pushq   %rbp
LCFI27:
    movq    %rsp, %rbp
LCFI28:
    subq    $48, %rsp
    movq    %rdi, -40(%rbp)
    leaq    -16(%rbp), %rax
    movq    -40(%rbp), %rdx
    movq    %rdx, %rsi
    movq    %rax, %rdi
    call    __ZN15ConcreteFactory12createButtonEv
```

CRTP problems

- looks like a hack;
- no possibility to get nested types from Impl;
- Impl is incomplete at point of instantiation.

Concepts — solution?

```
template <class T>
concept bool factory = requires (T t)
    {
        {t.createButton()} -> std::unique_ptr<AbstractButton>;
    };
```

Concepts — solution?

```
template <class T>
concept bool factory = requires (T t)
   {
      {t.createButton()} -> std::unique_ptr<AbstractButton>;
   };

template <class T>
   requires factory<T>
void createForm(T& fact)
{
   auto button = fact.createButton();
}
```

Concepts — solution?

```
template <class T>
concept bool factory = requires (T t)
    {
        {t.createButton()} -> std::unique_ptr<AbstractButton>;
    };

template <factory T>
void createForm(T& fact)
{
    auto button = fact.createButton();
}
```

Concepts — solution?

```
template <class T>
concept bool factory = requires (T t)
    {
        {t.createButton()} -> std::unique_ptr<AbstractButton>;
    };

factory{T}
void createForm(T& fact)
{
    auto button = fact.createButton();
}
```

Concepts — solution?

```
template <class T>
concept bool factory = requires (T t)
    {
        {t.createButton()} -> std::unique_ptr<AbstractButton>;
    };

void createForm(factory& fact)
{
    auto button = fact.createButton();
}
```

Defining concepts

```
template <class T>
concept bool variable_concept = constraint-expression;


template <class T>
concept bool function_concept()
{
    return constraint-expression;
}
```

Defining concepts. Type traits

```
template <class T>
concept bool integral = std::is_integral<T>::value;
```

Defining concepts. Type traits

```
template <class T>
concept bool integral = std::is_integral<T>()();
```

Defining concepts. Type traits

```
// from library fundamentals TS
namespace std::experimental
{
    template <class T>
    constexpr bool is_integral_v = is_integral<T>::value;
}


namespace stde = std::experimental;

template <class T>
concept bool integral = stde::is_integral_v<V>;
```


STE||AR GROUP

Concepts. Iterator example

Table 112 — Relations among iterator categories

Random Access	$\rightarrow$ **Bidirectional**	$\rightarrow$ **Forward**	$\rightarrow$ **Input**
			$\rightarrow$ **Output**

Concepts. Iterator example

A type `X` satisfies the `Iterator` requirements if:

- `X` satisfies the `CopyConstructible`, `CopyAssignable`, and `Destructible` requirements (17.6.3.1) and lvalues of type `X` are swappable (17.6.3.2), and
- the expressions in Table 113 are valid and have the indicated semantics.

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`*r`	unspecified		pre: `r` is dereferenceable.
`++r`	`X&`		

```
template <class T>
concept bool iterator = ...;
```

Concepts. Iterator example

```
template <class T>
concept bool swappable = requires (T t)
    {
        {std::swap(t, t)} -> void;
    }
    or requires (T t)
    {
        // for ADL
        {swap(t, t)} -> void;
    };
```

Concepts. Iterator example

```
template <class T>
concept bool iterator = stde::is_copy_constructible_v<T>
    and stde::is_copy_assignable_v<T>
    and stde::is_destructible_v<T>
    and swappable<T>
    and requires (T r)
    {
        *r;
        {++r} -> T&;
    };
```

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`*r`	unspecified		pre: `r` is dereferenceable.
`++r`	`X&`		

Concepts. Input iterator

```
template <class T>
concept bool equality_comparable = requires (T t)
    {
        {t == t} -> bool;
    };

template <class T>
concept bool input_iterator = iterator<T>
    and equality_comparable<T>;
```

Concepts. Forward iterator

A class or pointer type `X` satisfies the requirements of a forward iterator if

- `X` satisfies the requirements of an input iterator (24.2.3),
- X satisfies the `DefaultConstructible` requirements (17.6.3.1),
- if `X` is a mutable iterator, `reference` is a reference to `T`; if `X` is a const iterator, `reference` is a reference to `const T`,
- the expressions in Table 116 are valid and have the indicated semantics, and
- objects of type `X` offer the multi-pass guarantee, described below.

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`r++`	convertible to `const X&`	`{ X tmp = r; ++r; return tmp; }`	

STE||AR GROUP

Concepts. Forward iterator

```
template <class T>
concept bool forward_iterator = input_iterator<T>
    and stde::is_default_constructible_v<T>
    and requires (T r)
    {
        {r++} -> const T&;
    };
```

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition

`r++`	convertible to `const X&`	`{ X tmp = r;`
		`++r;`
		`return tmp; }`	

STE||AR GROUP

Concepts. Bidirectional iterator

```
template <class T>
concept bool bidirectional_iterator = forward_iterator<T>
    and requires (T r)
    {
        {--r} -> T&
        {r--} -> const T&;
    };
```

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`--r`	`X&`		pre: there exists `s` such that `r == ++s`. post: `r` is dereferenceable. `--(++r) == r`. `--r == --s` implies `r == s`. `&r == &--r`.

`r--`	convertible to `const X&`	`{ X tmp = r;`
		`--r;`
		`return tmp; }`	

Concepts. Random access iterator

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition

`r += n`	`X&`	`{ difference_type m = n;`
		`if (m >= 0)`
		`while (m--)`
		`++r;`
		`else`
		`while (m++)`
		`--r;`
		`return r; }`	

`a + n` `n + a`	`X`	`{ X tmp = a;` `return tmp += n; }`	`a + n == n + a`.
`r -= n`	`X&`	`return r += -n;`	
`a - n`	`X`	`{ X tmp = a;` `return tmp -= n; }`	
`b - a`	`difference_-` `type`	`return n`	pre: there exists a value `n` of type `difference_type` such that `a + n == b`. `b == a + (b - a)`.
`a[n]`	convertible to `reference`	`*(a + n)`	
`a < b`	contextually convertible to `bool`	`b - a > 0`	`<` is a total ordering relation
`a > b`	contextually convertible to `bool`	`b < a`	`>` is a total ordering relation opposite to `<`.
`a >= b`	contextually convertible to `bool`	`!(a < b)`	
`a <= b`	contextually convertible to `bool`.	`!(a > b)`	

Concepts. Random access iterator

```
template <class T>
concept bool random_access_iterator = bidirectional_iterator<T>
    and requires (T it,
            typename std::iterator_traits<T>::difference_type n)
    {
        {it += n} -> T&;
        {it + n} -> T;
        {n + it} -> T;
        {it -= n} -> T&;
        {it - it} ->
            typename std::iterator_traits<T>::difference_type;
        {it[n]} ->
            typename std::iterator_traits<T>::reference;
        {it < it} -> bool;
        {it <= it} -> bool;
        {it > it} -> bool;
        {it >= it} -> bool;
    };
```

STE||AR GROUP

Concepts. Iterators. Let's test!

```
static_assert(forward_iterator<
    std::forward_list<int>::iterator>);
```

Concepts. Iterators. Let's test!

```
static_assert(forward_iterator<
    std::forward_list<int>::iterator>);

static_assert(bidirectional_iterator<
    std::list<int>::iterator>);
static_assert(bidirectional_iterator<
    std::map<char, int>::iterator>);
```

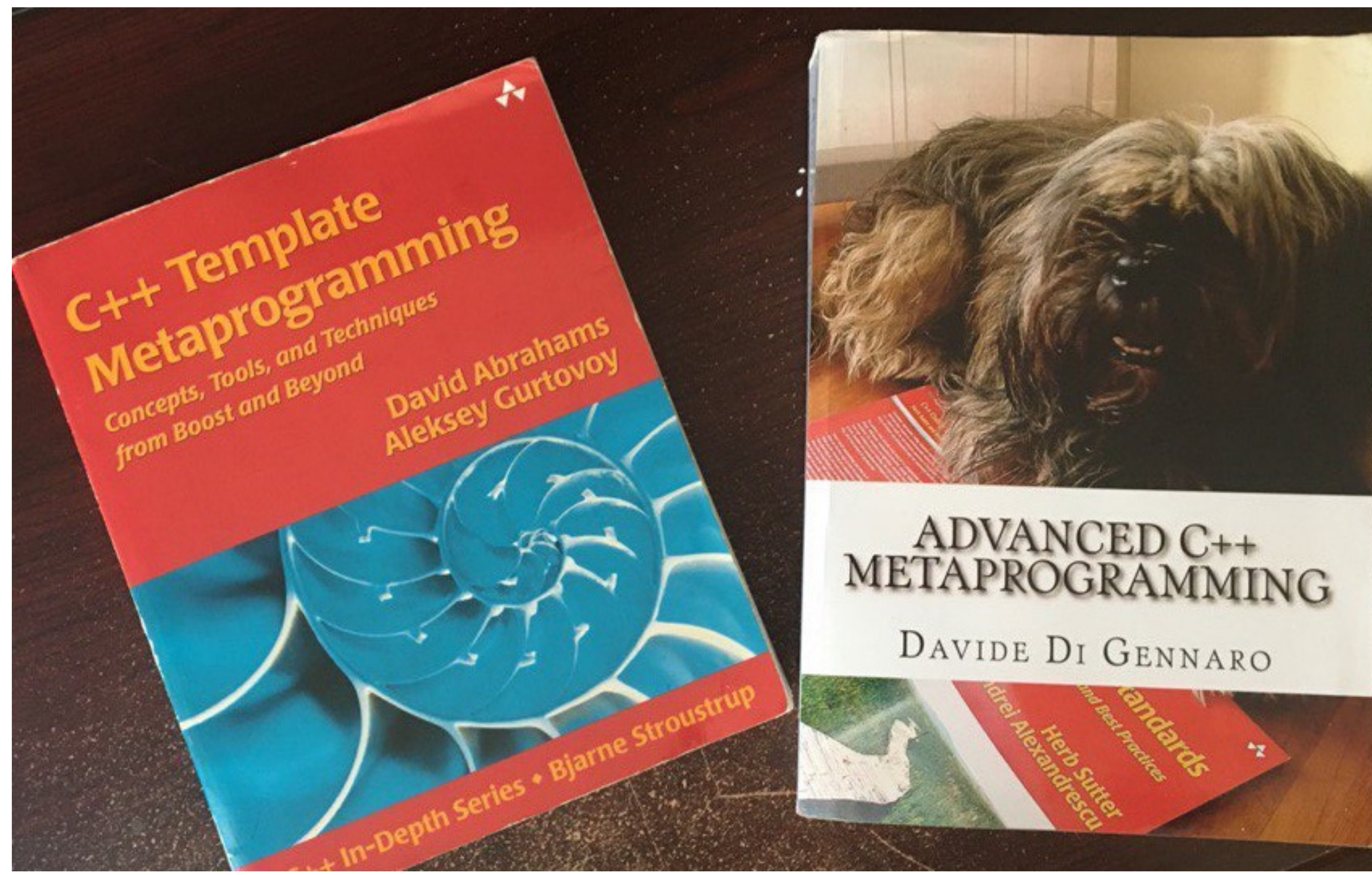
Concepts. Iterators. Let's test!

```
static_assert(forward_iterator<
    std::forward_list<int>::iterator>);

static_assert(bidirectional_iterator<
    std::list<int>::iterator>);
static_assert(bidirectional_iterator<
    std::map<char, int>::iterator>);

static_assert(random_access_iterator<int*>);
static_assert(random_access_iterator<
    std::vector<int>::iterator>);
static_assert(random_access_iterator<
    std::deque<int>::iterator>);
```

C++ generic programming and TMP



Without concepts: libc++

```
template <class _InputIterator>
    vector(_InputIterator __first,
        typename enable_if<
            __is_input_iterator<_InputIterator>::value &&
            !__is_forward_iterator<_InputIterator>::value &&
                is_constructible<
                    value_type,
                    typename iterator_traits<
                        _InputIterator>::reference>::value,
                _InputIterator>::type __last);
template <class _ForwardIterator>
    vector(_ForwardIterator __first,
        typename enable_if<
            __is_forward_iterator<_ForwardIterator>::value &&
            is_constructible<
                value_type,
                typename iterator_traits<
                    _ForwardIterator>::reference>::value,
            _ForwardIterator>::type __last);
```

With concepts: libc++

```
vector(input_iterator __first, input_iterator __last);
vector(forward_iterator __first, forward_iterator __last);
```

That's it!

The greediness of T&&

The greediness of T&&

```
// from Meyers's EMC++ Item 26
std::multiset<std::string> names;

void logAndAdd(const std::string& name)
{
    auto now = std::chrono::system_clock::now();

    log(now, "logAndAdd");
    names.emplace(name);
}
```

The greediness of T&&

```
// from Meyers's EMC++ Item 26
std::multiset<std::string> names;

void logAndAdd(const std::string& name)
{
    auto now = std::chrono::system_clock::now();

    log(now, "logAndAdd");
    names.emplace(name);
}

std::string petName("Darla");
logAndAdd(petName); // 1)
logAndAdd(std::string("Persephone")); // 2)
logAndAdd("Patty Dog"); // 3)
```


STE||AR GROUP

The greediness of T&&

```
// from Meyers’s EMC++ Item 26
std::multiset<std::string> names;

void logAndAdd(const std::string& name)
{
    auto now = std::chrono::system_clock::now();

    log(now, "logAndAdd");
    names.emplace(name);
}

std::string petName("Darla");
logAndAdd(petName); // 1) okay, just copying
logAndAdd(std::string("Persephone")); // 2) extra copy
logAndAdd("Patty Dog”); // 3) no need to copy or move
```

The greediness of T&&

```
// from Meyers's EMC++ Item 26
template <typename T>
void logAndAdd(T&& name)
{
    auto now = std::chrono::system_clock::now();

    log(now, "logAndAdd");
    names.emplace(std::forward<T>(name));
}


std::string petName("Darla");
logAndAdd(petName); // 1) okay, just copying
logAndAdd(std::string("Persephone")); // 2) moving!
logAndAdd("Patty Dog"); // 3) neither moving nor copying,
                        //    just creating inplace
```

The greediness of T&&

```
// from Meyers's EMC++ Item 26
std::string nameFromIdx(int idx);

void logAndAdd(int idx)
{
    auto now = std::chrono::system_clock::now();

    log(now, "logAndAdd");
    names.emplace(nameFromIdx(idx));
}

short nameIdx;
logAndAdd(nameIdx); // error!
```

The greediness of T&&

Meyers, “Effective Modern C++”,
Item 26:
Avoid overloading on universal references.

The greediness of T&&

Meyers, “Effective Modern C++”,
Item 26:
Avoid overloading on universal references.

...

The greediness of T&&

Meyers, “Effective Modern C++”,
Item 26:
Avoid overloading on universal references.

...

but not on constrained universal references!

Constrained T&&
as the best parameter type for efficiency and safety

```
void logAndAdd(int idx);

template <typename T>
void logAndAdd(T&& name)
{
    ...
    names.emplace(std::forward<T>(name));
}
```

Constrained T&&
as the best parameter type for efficiency and safety

```
void logAndAdd(int idx);

template <typename T, typename =
    std::enable_if_t<
        stde::is_convertible_v<
            T, std::string>>>
void logAndAdd(T&& name)
{
    ...
    names.emplace(std::forward<T>(name));
}
```

Constrained T&&
as the best parameter type for efficiency and safety

```
void logAndAdd(int idx);

template <typename T, typename =
    std::enable_if_t<
        stde::is_convertible_v<
            std::remove_reference_t<T>, std::string>>>
void logAndAdd(T&& name)
{
    ...
    names.emplace(std::forward<T>(name));
}
```

Constrained T&&
as the best parameter type for efficiency and safety

```
void logAndAdd(int idx);

template <typename T, typename =
    std::enable_if_t<
        stde::is_convertible_v<
            std::remove_reference_t<T>, std::string>>>
void logAndAdd(T&& name)
{
    ...
    names.emplace(std::forward<T>(name));
}


short nameIdx;
logAndAdd(nameIdx); // calls logAndAdd(int) just fine!
```

Constrained T&&
as the best parameter type for efficiency and safety

```
template <class From, class To>
concept bool convertible =
    stde::is_convertible_v<
        std::remove_reference_t<From>, To>;
```

Constrained T&&
as the best parameter type for efficiency and safety

```
template <class From, class To>
concept bool convertible =
    stde::is_convertible_v<
        std::remove_reference_t<From>, To>;


void logAndAdd(convertible<std::string>&& name)
{
    ...
    names.emplace(std::forward<decltype(name)>(name));
}


short nameIdx;
logAndAdd(nameIdx); // calls logAndAdd(int) just fine!
```

Constrained T&&
as the best parameter type for efficiency and safety

```
// from Meyers's EMC++ Item 26
class Person {
public:
    template <typename T>
    explicit Person(T&& n)  // perfect forwarding ctor
    : name(std::forward<T>(n)) {}

    explicit Person(int idx)
    : name(nameFromIdx(idx)) {}

private:
    std::string name;
};
```

Constrained T&&
as the best parameter type for efficiency and safety

```
// from Meyers's EMC++ Item 26
class Person {
public:
    template <typename T>
    explicit Person(T&& n)  // perfect forwarding ctor
    : name(std::forward<T>(n)) {}

    explicit Person(int idx)
    : name(nameFromIdx(idx)) {}

private:
    std::string name;
};

Person p("Nancy");
auto cloneOfP(p);  // WOW!
```

Constrained T&&
as the best parameter type for efficiency and safety

```
// from Meyers's EMC++ Item 26
class Person {
public:
    explicit Person(convertible<std::string>&& n)
    : name(std::forward<decltype(n)>(n)) {}

    explicit Person(int idx)
    : name(nameFromIdx(idx)) {}

private:
    std::string name;
};

Person p("Nancy");
auto cloneOfP(p);  // now works fine!
```

std::future<C++> example:

flat_map

C++17 example: boost::flat_map

```
namespace boost::container
{
    template <class Key, class Value,
              class Compare = std::less<Key>,
              class Allocator =
                  std::allocator<std::pair<Key, Value>>>
    class flat_map;
}
```

C++17 example: my flat_map

```
namespace mine::container
{
    template <class Key, class Value,
              class Compare = std::less<Key>,
              class UnderlyingContainer =
                  std::vector<std::pair<Key, Value>>>
    class flat_map;
}
```

C++17 example: my flat_map

```
namespace mine::container
{
    template <class Key, class Value,
              class Compare = std::less<Key>,
              class UnderlyingContainer =
                  std::vector<std::pair<Key, Value>>>
        requires default_constructible<Value>
    class flat_map;
}
```

C++17 example: my flat_map

```
namespace mine::container
{
    template <class Key, class Value,
              class Compare = std::less<Key>,
              class UnderlyingContainer =
                  std::vector<std::pair<Key, Value>>>
        requires default_constructible<Value>
              and random_access_iterator<
                  typename UnderlyingContainer::iterator>
    class flat_map;
}
```

C++17 example: my flat_map

```
namespace mine::container
{
    template <class Key, class Value,
              class Compare = std::less<Key>,
              class UnderlyingContainer =
                  std::vector<std::pair<Key, Value>>>
        requires default_constructible<Value>
            and random_access_iterator<
                typename UnderlyingContainer::iterator>
            and requires (Compare cmp, Key key)
            {
                {cmp(key, key)} -> bool;
            }
    class flat_map;
}
```


STE||AR GROUP

flat_map: typedefs

```
template <...>
class flat_map final: private UnderlyingContainer
{
    using key_type = Key;
    using mapped_type = Value;
    using underlying_type = UnderlyingContainer;

    using typename underlying_type::value_type;
    static_assert(stde::is_same_v<value_type,
            std::pair<key_type, mapped_type>>);

    using key_compare = Compare;
    using value_compare = struct {
        ...
    };

};
```

flat_map: typedefs

```
template <...>
class flat_map final: private UnderlyingContainer
{
    using value_compare = struct {
        bool operator()(const value_type& left,
                        const value_type& right) const {
            return key_compare()(left.first, right.first);
        }
        bool operator()(const key_type& left,
                        const value_type& right) const {
            return key_compare()(left, right.first);
        }
        bool operator()(const value_type& left,
                        const key_type& right) const {
            return key_compare()(left.first, right);
        }};
};
```

flat_map: typedefs

```
template <...>
class flat_map final: private UnderlyingContainer
{
    using typename underlying_type::size_type;
    using typename underlying_type::difference_type;
    using typename underlying_type::allocator_type;
    using typename underlying_type::reference;
    using typename underlying_type::const_reference;
    using typename underlying_type::pointer;
    using typename underlying_type::const_pointer;
    using typename underlying_type::iterator;
    using typename underlying_type::const_iterator;
    using typename underlying_type::reverse_iterator;
    using typename underlying_type::const_reverse_iterator;


};
```

flat_map: laconic wrapping

```
template <...>
class flat_map final: private UnderlyingContainer
{
    using underlying_type::begin; // laconic wrapping
    using underlying_type::cbegin;
    using underlying_type::end;
    using underlying_type::cend;
    using underlying_type::rbegin;
    using underlying_type::crbegin;
    using underlying_type::rend;
    using underlying_type::crend;




};
```

flat_map: laconic wrapping

```
template <...>
class flat_map final: private UnderlyingContainer
{
    using underlying_type::swap;
    using underlying_type::clear;
    using underlying_type::empty;
    using underlying_type::size;
    using underlying_type::max_size;
    using underlying_type::capacity;
    using underlying_type::reserve;
    using underlying_type::shrink_to_fit;



};
```

flat_map: ctors

```
template <...>
class flat_map final: private UnderlyingContainer
{
    flat_map() = default;

    template <class Iterator>
    flat_map(Iterator first, Iterator last):
        underlying_type{first, last}
    {
        std::sort(begin(), end());
    }

    flat_map(std::initializer_list<value_type> list):
        flat_map{list.begin(), list.end()}
    {
    }
};
```

flat_map: ctors

```
template <...>
class flat_map final: private UnderlyingContainer
{
    flat_map() = default;


    flat_map(input_iterator first, input_iterator last):
        underlying_type{first, last}
    {
        std::sort(begin(), end());
    }

    flat_map(std::initializer_list<value_type> list):
        flat_map{list.begin(), list.end()}
    {
    }
};
```

flat_map: ctors

```
template <...>
class flat_map final: private UnderlyingContainer
{
    flat_map() = default;


    flat_map(input_iterator first, input_iterator last):
        underlying_type{first, last}
    {
        stde::sort(stde::par, begin(), end());
    }

    flat_map(std::initializer_list<value_type> list):
        flat_map{list.begin(), list.end()}
    {
    }
};
```

flat_map: binary_search

```
namespace detail
{
    template <class Iterator, class Key, class Compare>
    Iterator binary_search(Iterator begin,
            Iterator end,
            const Key& key,
            const Compare& cmp)
    {
        auto it = std::lower_bound(begin, end, key, cmp);
        if (it == end || cmp(key, *it))
            return end;

        return it;
    }


}
```

flat_map: binary_search

```
namespace detail
{

    auto binary_search(forward_iterator begin,
            forward_iterator end,
            const auto& key,
            const auto& cmp)
    {
        auto it = std::lower_bound(begin, end, key, cmp);
        if (it == end || cmp(key, *it))
            return end;

        return it;
    }


}
```

flat_map: find

```
template <...>
class flat_map final: private UnderlyingContainer
{
    iterator find(const key_type& key)
    {
        return detail::binary_search(
            begin(), end(), key, value_compare{});
    }

    const_iterator find(const key_type& key) const
    {
        return detail::binary_search(
            cbegin(), cend(), key, value_compare{});
    }


};
```

flat_map: at

```
template <...>
class flat_map final: private UnderlyingContainer
{
    mapped_type& at(const key_type& key)
    {
        const auto it = find(key);
        if (it == end()) throw std::range_error("no key");

        return it->second;
    }

    const mapped_type& at(const key_type& key) const
    {
        // same as above
    }

};
```

flat_map: emplace

```
template <...>
class flat_map final: private UnderlyingContainer
{
    std::pair<iterator, bool> emplace(
            auto&& first, auto&&... args)
    {
        value_compare comp;
        const auto it = std::lower_bound(
            begin(), end(), first, comp);
        if (it == end() || comp(first, *it))
            return {underlying_type::emplace(it,
                       std::forward<decltype(first)>(first),
                       std::forward<decltype(args)>(args)...),
                    true};
        return {it, false};
    }
};
```

flat_map: insert

```
template <...>
class flat_map final: private UnderlyingContainer
{
    std::pair<iterator, bool> insert(const value_type& value)
    {
        return emplace(value);
    }









};
```


STE||AR GROUP

flat_map: insert

```
template <...>
class flat_map final: private UnderlyingContainer
{
    std::pair<iterator, bool> insert(const value_type& value)
    {
        return emplace(value);
    }

    std::pair<iterator, bool> insert(value_type&& value)
    {
        return emplace(std::move(value));
    }



};
```


STE||AR GROUP

flat_map: insert

```
template <...>
class flat_map final: private UnderlyingContainer
{
    std::pair<iterator, bool> insert(
        convertible<value_type>&& value)
    {
        return emplace(std::forward<decltype(value)>(value));
    }

    // no need to provide anything else!

};
```

flat_map: insert

```
template <...>
class flat_map final: private UnderlyingContainer
{
    void insert(input_iterator first,
                input_iterator last)
    {
        const auto count = std::distance(first, last);
        const auto room = capacity() - size();

        if (room < count)
            reserve(size() + count);

        for (; first != last; ++first)
            emplace(*first);
    }

};
```

flat_map: insert

```
template <...>
class flat_map final: private UnderlyingContainer
{
    void insert(std::initializer_list<value_type> list)
    {
        insert(list.begin(), list.end());
    }









};
```

flat_map: index operator

```
template <...>
class flat_map final: private UnderlyingContainer
{
    mapped_type& operator[](convertible<key_type>&& key)
    {
        return emplace(
                std::forward<decltype(key)>(key),
                mapped_type{}
            ).first->second;
    }




};
```

flat_map: erase

```
template <...>
class flat_map final: private UnderlyingContainer
{
    using underlying_type::erase;

    size_type erase(const key_type& key)
    {
        const auto it = find(key);
        if (it == end())
            return 0u;

        erase(it);
        return 1u;
    }


};
```

Modularizing flat_map. Interface

```
// flat_map.hpp
#include <...>



namespace container
{
    template <class Key, class Value,
              class Compare = std::less<Key>,
              class UnderlyingContainer =
                  std::vector<std::pair<Key, T>>>
    class flat_map
    {
        ...
    };
}
```

Modularizing flat_map. Interface

```
// flat_map.ixx
#include <...>

module container.flat_map;

export namespace container
{
    template <class Key, class Value,
              class Compare = std::less<Key>,
              class UnderlyingContainer =
                  std::vector<std::pair<Key, T>>>
    class flat_map
    {
        ...
    };
}
```


STE||AR GROUP

Modularizing flat_map. Consuming

```
// main.cpp
import std.io;
import container.flat_map;

int main()
{
    container::flat_map<std::string, int> map =
        {{"second", 2}, {"first", 1}, {"third", 3}};

    for (const auto& p: map)
        std::cout << p.first << ": " << p.second << std::endl;

    return 0;
}
```

Some more slides…

Static polymorphism with concepts. Improving

```
template <class T>
concept bool factory = requires (T t)
    {
        {t.createButton()} -> std::unique_ptr<AbstractButton>;
    };

void createForm(factory& fact)
{
    auto button = fact.createButton();
}
```

Static polymorphism with concepts. Improving

```
struct AbstractButton
{
    virtual void onClick(std::function<void ()>) = 0;
    virtual void onMove (std::function<void ()>) = 0;
};
```

Static polymorphism with concepts. Improving

```
namespace detail // or std? or whatever
{
    constexpr auto empty = [](auto...){};
}

template <class T>
concept bool button = requires (T t)
    {
        t.onClick(detail::empty);
        t.onMove(detail::empty);
    };
```

Static polymorphism with concepts. Improving

```
template <class T>
concept bool button = ...;

template <class T>
concept bool factory = requires (T t)
    {
        {t.createButton()} -> button;
    };
```

Static polymorphism with concepts. Improving

```
class TButton
{
   void onClick(...);
   void onMove(...);
};
```

```
class QPushButton
{
   void onClick(...);
   void onMove(...);
};
```

Static polymorphism with concepts. Improving

```
class TButton
{
    void onClick(...);
    void onMove(...);
};
```

```
class VCLFactory
{
    TButton createButton()
    {
        return TButton{};
    }
};
```

```
class QPushButton
{
    void onClick(...);
    void onMove(...);
};
```

```
class QtFactory
{
    QPushButton createButton()
    {
        return QPushButton{};
    }
};
```

Static polymorphism with concepts. Using

```
void createForm(factory& fact)
{
    auto button = fact.createButton();
    button.onClick([]{ /* pum pum pum */ });
}

int main()
{
    QtFactory factory;
    createForm(factory);

    QApplication::run();
}
```

Thank you for your attention!