

C++

Fall 2020

compscicenter.ru

Башарин Егор, t.me/egorbasharin

Лекция XIV

Exception safety & Stack.

Variadic templates. CTAD. CRTP.

Section 1

live-coding: exception safety & stack

pop

повторение

- почему pop не возвращает элемент с вершины стека?
- пример с `-fno-elide-constructors`

push

- ищем ошибки в простейшей реализации
- обеспечиваем строгую гарантию безопасности исключений
- `move_if_noexcept`

Section 1.5

Ошибка в лекции 11

Слайд про значения и объекты, на котором упоминается sору-elision.

- Ошибка: связи с sору-elision нет, копировать объект побайтово **нельзя**
- Слайды и видео будут поправлены.

Section 2

Variadic Template Motivation

Motivation I

maybe<T>::emplace

```
template <class Arg>
void emplace(Arg&& arg) {
    reset();
    new (storage_) T{arg};
    has_value_ = true;
}

template <class Arg1, class Arg2>
void emplace(Arg1&& arg1, Arg2&& arg2) {
    reset();
    new (storage_) T{arg1, arg2};
    has_value_ = true;
}
```

Если ли проблемы с такой реализацией функции?

Motivation I

maybe<T>::emplace

```
template <class Arg>
void emplace(Arg&& arg) {
    reset();
    new (storage_) T{arg}; // // use std::forward
    has_value_ = true;
}

template <class Arg1, class Arg2>
void emplace(Arg1&& arg1, Arg2&& arg2) {
    reset();
    new (storage_) T{arg1, arg2}; // use std::forwar
    has_value_ = true;
}
```

- Не забываем о perfect-forwarding
- Дублирование кода

Motivation I

maybe<T>::emplace

```
template <class... Args>
void emplace(Args... args) {
    reset();
    new (storage_) T{std::forward<Args>(args)...};
    has_value_;
}
```

Motivation II

Variadic functions

```
#include <iostream>
#include <cstdarg>
int sumInts(size_t n, ...) {
    va_list args;
    va_start(args, n);
    int sum = 0;
    while (n > 0) {
        --n;
        sum += va_arg(args, int);
    }
    va_end(args);
    return sum;
}

int main() {
    std::cout << sumInts(1,1);
    std::cout << sumInts(2,1,2);
    std::cout << sumInts(3,1,2,3);
}
```

Motivation II

Variadic functions

- Нетипобезопасно
- Нужно передавать информацию о количестве аргументов
- Аргументы передаются только по значению

Motivation II

```
template <class... Args>
int sumInts2(Args&&... args)
{
    static_assert((... && std::is_same_v<std::decay_t<Ar
    return (args + ...);
}
```

Section 3

Parameter pack

Template parameter pack

```
template <int... ints>
struct A { };

template <class... Types>
struct B { };

template <template <class> class... TempTypes>
struct C { };

template <class T>
struct D { };

int main() {
    A<> a1;
    A<1,2,3> a2;
    B<int, double, int> b;
    C<D, D, D> c;
}
```

Пакет параметров может быть у шаблона класса, шаблона функции, шаблона переменной или у псевдонима шаблона (alias)

Template parameter pack

Syntax:

```
type... Args  
class... Args  
template <parameter-list> class... Args
```

- Пакет параметров (parameter pack) может соответствовать 0 и более шаблонным аргументам
- Шаблон с пакетом параметров является вариативным (variadic template)
- Обычно пакет параметров указывается последним в списке параметров (есть исключение для функций)

Example 1

Пишем метафункцию IntSum

```
static_assert(IntSum<>::value == 0);  
static_assert(IntSum<2>::value == 2);  
static_assert(IntSum<1,2,3>::value == 6);
```

Example 2

Пишем Tuple

```
Tuple<> t0;  
Tuple<int> t1{1};  
Tuple<int, double> t2{1, 2.0};
```

Example 3

Примеры из стандартной библиотеки

- `std::variant`
- `std::tuple`

Pack expansion

Syntax:

```
pattern...
```

Pack expansion

```
template <class... Args>
int f(Args&&... args) { }

template <class... Args>
void g(Args... args) { // expands to (A a, B b, C c)

    f(args...); // pattern `args` -> f(a,b,c)
    f(++args...); // pattern `++args` -> f(++a, ++b, ++c)

    f(const_cast<const Args&>(args)...); // pattern `const_cast<const Args&>(args)`
    f(f(args...) + args...);
}

int main() {
    g(1,2,3);
}
```

cppinsight

Pack expansion

```
template <class... Args>
struct Container { };

template <class A, class... Args>
struct MyClass {
    Container<A, Args...> c1;
    Container<Args..., A> c2;

    template <Args... args>
    struct T { };
};

int main() {
    MyClass<int, bool, short> mc;
    MyClass<int, bool, short>::T<true, 1> t;
}
```

Pack expansion

```
#include <iostream>

template <class... Mixins>
class MethodCombiner : public Mixins... {
public:
    MethodCombiner(Mixins... mixins) : Mixins(std::move(mixins))... {}
    using Mixins::method...;
};

struct A {
    void method(int) { std::cout << "A::method" << std::endl; }
};
struct B {
    void method(double) { std::cout << "B::method" << std::endl; }
};

int main() {
    MethodCombiner mc{A{}, B{}};

    mc.method(1);
    mc.method(2.9);
}
```

cppinsight

Pack expansion

Lambdas

```
template <class... Args>
auto f(Args... args) {
    return [args...](auto... another_args){
        g(args..., another_args...);
    };
}
```


sizeof... operator

Оператор для определения количества элементов в пакете параметров

```
template <class... Args>
struct count {
    static const size_t value = sizeof...(Args);
};

int main() {
    static_assert(count<int, int, double>::value == 3);
}
```

Section 4

Fold expression

Motivation

Life without fold expression

```
namespace details {  
int sum_impl() { return 0; }  
  
template <class First, class... Others>  
int sum_impl(First first, Others... others) {  
    return first + sum_impl(others...);  
}  
  
template <class... Args>  
int sum(Args... args) {  
    return details::sum_impl(args...);  
}  
  
int main() {  
    assert(sum(1,2,3,4,5) == 15);  
}
```

Solution with fold expression

```
template <class... Args>  
int sum(Args... args) {  
    return (0 + ... + args);  
}
```

Fold expression

Syntax:

```
(pack op ...)      (1)
(... op pack)      (2)
(pack op ... op init) (3)
(init op ... op pack) (4)
```

Fold expression

with non-associative operators

```
#include <iostream>

template <class... Args>
void example(Args... args){

    auto res1 = (... - args); // ((1 - 2) - 3)
    auto res2 = (args - ...); // (1 - (2 - 3))

    std::cout << res1 << " " << res2;
}

int main() {
    example(1,2,3);
}
```

Fold expression

(pack op ...) (1)

(A1 op (A2 op (A3 op ...)))

(... op pack) (2)

((A1 op A2) op A3) op ...)

Example 1

operator<<

```
template <class... Args>  
void print(const Args&... args) {  
    (std::cout << ... << args);  
}
```


Example 2

comma-operator

```
template <class Arg>
void printThis(const Arg& arg) {
    std::cout << arg << " ";
}

template <class... Args>
void printThese(const Args&... args) {
    (... , printThis(args));
}
```

Section 5

CTAD, deduction guides

Class Template Argument Deduction [C++17]

- в момент определения переменной
- если не хочется писать типы шаблонного класса
- \Rightarrow воспользуемся argument deduction
 - использует deduction guides

std::pair example

```
#include <utility>

std::pair<int, double> p{1, 2.0};
std::pair dp{1, 2.0}; // template argument deduction (s
```

- применяется, только если нет template argument list (даже пустого: \diamond)
- аргументы шаблона выводятся компилятором, хотя и не указаны

Implicitly-generated deduction guides

example

```
template<class T>
struct Foo {
    Foo(T);

    template<class U> Foo(U);
    template<class U> Foo(T, U);
};
// ->
template<class T> Foo<T> FooStub1(T);
template<class T, class U> Foo<T> FooStub2(U);
template<class T, class U> Foo<T> FooStub3(T, U);
// + default & copy ctor

Foo f{42};
Foo f{42.0, 24};
```

User-defined deduction guides

Syntax:

```
[explicit] template-name ( parameter-declaration-clause ) → simple-template-id ;
```

- похожи на объявления шаблонных функций с trailing return type
- должны быть определены в одном семантическом скоупе вместе с template class
 - т.е. в том же пространстве имен или внешнем классе
- необязательно темплейты

User-defined deduction guides example

```
template<class T>
struct container {
    template<class Iter>
    container(Iter beg, Iter end) {};
};

// getting type of value the iter points to
template<typename Iter>
using vtype = typename std::iterator_traits<Iter>::value_type;

template<class Iter>
container(Iter b, Iter e) -> container<vtype<Iter>>;

std::vector<double> v{42.0};
container c(v.begin(), v.end()); // -> container<double>
```


deduction guides

- не функции, не имеют тела
- не участвуют в перегрузках *функций, методов*
 - только перегрузки с другими deduction guides — для вывода типа объекта
- алгоритм:
 - по аргументам выводится тип
 - для полученной инстанции вызывается конструктор

Section 6 (Bonus)

CRTP Idiom

C RTP Idiom

Curiously Recurring Template Pattern

```
template <class Child>
struct Base {
    void foo() const { child().foo(); }
    const Child& child() const { return *static_cast<const Child*>(this); }
};

struct Child1 : public Base<Child1> {
    void foo() const { std::cout << "child_1 foo" << std::endl; }
};

struct Child2 : public Base<Child2> {
    void foo() const { std::cout << "child_2 foo" << std::endl; }
};

template <class T>
void call_foo(const Base<T>& base) {
    base.foo();
}

int main() {
    call_foo(Child1{});
    call_foo(Child2{});
}
```

- Статический полиморфизм
- Примеры использования: [click me](#)

CRTP in STL

- `enable_shared_from_this`