C++

Лекция XI

Variadic templates

Section 1

Motivation

Motivation I

```
template <class T>
class Wrapper{
public:

    template <class Arg>
    Wrapper(Arg&& arg)
    : t_(arg) {

    }

    template <class Arg1, class Arg2>
    Wrapper(Arg1&& arg1, Arg2&& arg2)
    : t_(arg1, arg2) {}

private:
    T t_;
};
```

Если ли проблемы с такой реализацией?

Motivation I

```
template <class T>
class Wrapper{
public:

    template <class Arg>
    Wrapper(Arg&& arg)
    : t_(arg) {}          // use std::forward for arg

    template <class Arg1, class Arg2>
    Wrapper(Arg1&& arg1, Arg2&& arg2)
    : t_(arg1, arg2) {}  // use std::forward for arg1, arg2

private:
    T t_;
};
```

Motivation I

- Не забываем о perfect-forwarding
- Нужно вручную реализовать все конструкторы: их должно быть больше, так как хотелось бы поддержать все конструкторы у T
- Бойлерплейт
- Дублирование кода: везде инициализируется t_

Motivation I

```cpp
#include<utility>

template <class T>
class Wrapper{
public:
    template <class... Arg>
    Wrapper(Arg&&... arg)
    : t_(std::forward<Arg>(arg)...) {}

private:
    T t_;
};
```

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<A
    return (args + ...);
}
```

Section 2

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)

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);
}
```

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

```cpp
#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);
}
```

`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 3

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 4

CTAD (since c++17)

Class Template Argument Deduction

```cpp
#include <utility>
std::pair<int, double> p{1, 2.0};
std::pair dp{1, 2.0};  // CTAD (since C++17)
```

- применяется при отсутствии аргументов шаблона
- аргументы шаблона выводятся компилятором

User-defined deduction guides

Syntax:

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

- похожи на объявления шаблонных фукнций с trailing return type
- должны быть определены в одном семантическом скоупе вместе с template class (в том же пространстве имен или внешнем классе)
- необязательно шаблон

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>::valu

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

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

deduction guides

- не участвуют в перегрузках *функций*, *методов*, только перегрузки с другими deduction guides — для вывода типа объекта
- алгоритм:
  - генерация фиктивных шаблонных функций
    - для конструкторов
    - для пользовательских deduction guides
  - применяются обычные правила выбора подходящей функции