

Язык C++

Lambda

Pointer to function

- не члены класса
- статические члены класса

```
int incr(int i) {  
    return i + 1;  
}
```

```
int decr(int i) {  
    return i - 1;  
}
```

```
int main() {  
    int (*funcPtr)(int) = incr;  
  
    std::cout << (void*)incr << std::endl;  
    std::cout << (void*)funcPtr << std::endl;  
  
    std::cout << (*funcPtr)(1) << std::endl;  
    std::cout << funcPtr(1) << std::endl;  
  
    funcPtr = decr;  
    std::cout << funcPtr(1) << std::endl;  
  
    return 0;  
}
```

Pointer to function

```
int* findMax(int* array, size_t size, bool(*compare)(int,int)) {
    int* result = array;
    for(int i = 1; i < size; ++i) {
        if(!compare(*result, *(array + i)))
            result = array + i;
    }

    return result;
}

bool greater(int a, int b) {
    return a > b;
}

int main() {
    int array[] = {1, 4, 5, 3, 10, 9};
    std::cout << *findMax(array, sizeof(array)/sizeof(int), greater);
    return 0;
}
```

Pointer to function

```
using TComparer = bool (*)(int,int)
```

```
int* findMax(int* array, size_t size, TComparer compare) {  
    int* result = array;  
    for(int i = 1; i < size; ++i) {  
        if(!comparer(*result, *(array +i)))  
            result = array + i;  
    }  
  
    return result;  
}
```

Functor

```
template<typename TComparer>
int* findMax(int* array, size_t size, TComparer comparer) {
    int* result = array;
    for(int i = 1; i < size; ++i) {
        if(!comparer(*result, *(array + i)))
            result = array + i;
    }
    return result;
}

int main() {
    int array[] = {1, 4, 5, 3, 10, 9};

    std::cout << *findMax(array, sizeof(array)/sizeof(int), std::greater<int>());
    return 0;
}
```

Pointer to function

```
void print(int value) {  
    std::cout << value << " ";  
}  
  
int main() {  
  
    std::vector<int> v = {1,2,3,4,5,6,7};  
    std::for_each(v.begin(), v.end(), print);  
  
    return 0;  
}
```

Functor

```
struct Printer {  
    void operator()(int value) const {  
        std::cout << value << " ";  
    }  
};  
  
int main() {  
  
    std::vector<int> v = {1,2,3,4,5,6,7};  
    std::for_each(v.begin(), v.end(), Printer{});  
  
    return 0;  
}
```

Functor

```
struct Printer {
    Printer()
        : counter(0)
    {}

    void operator()(int value) const {
        std::cout << value << " ";
        ++counter;
    }

    mutable size_t counter;
};

int main() {
    std::vector<int> v = {1,2,3,4,5,6,7};
    Printer p = std::for_each(v.begin(), v.end(), Printer{});
    std::cout << std::endl << p.counter << std::endl;

    return 0;
}
```


std

- `std::less`
- `std::equal_to`
- `std::plus`
- `std::logical_to`
- [etc \(<functional>\)](#)

Functor

```
#include <functional>
#include <iterator>

int main() {

    std::vector<int> v = {1,2,3,4,5,6,7};

    std::sort(v.begin(), v.end(), std::greater< int>());
    std::copy(v.begin(), v.end(), std::ostream_iterator< int>(std::cout, " "));

    return 0;
}
```

Functor

```
class GreaterThen {
public:
    GreaterThen(int limit)
        : limit_(limit)
    {}

    bool operator()(int value) const {
        return value > limit_;
    }

private:
    int limit_;
};
```

```
int main() {

    std::vector<int> v = {1,2,3,4,5,6,7};

    auto it = std::find_if(
        v.begin(), v.end(),
        GreaterThen{4}
    );

    if(it != v.end())
        std::cout << *it;

    return 0;
}
```

Functor

```
int main() {  
  
    std::vector<int> v = {1,2,3,4,5,6,7};  
  
    auto it = std::find_if(  
        v.begin(), v.end(),  
        std::bind(std::greater<int>{}, std::placeholders::_1, 4)  
    );  
  
    if(it != v.end())  
        std::cout << *it;  
  
    return 0;  
}
```

Functor

- Позволяют параметризовать алгоритмы (и обычные функции)
- Отделены от вызывающего кода
- Использование стандартных функторов в нестандартных ситуациях затруднено

Lambda (C++11)

```
int main() {  
  
    std::vector<int> v = {1,2,3,4,5,6,7};  
  
    auto it = std::find_if(  
        v.begin(), v.end(),  
        [](int value) { return value > 4;}  
    );  
  
    if(it != v.end())  
        std::cout << *it;  
  
    return 0;  
}
```

Lambda

Замыкание, позволяет создавать неименованные функторы, с захватом переменных из текущей области видимости.

[capture] (params) attrs -> return { body }

- (params) - optional
- attrs - optional
- -> return - optional

Lambda

```
int x = 1;
```

```
[]{};
```

```
[](int i) { return i + 1;};
```

```
[](int i) ->float {return i + 1;};
```

```
[x](int i) {return x + i;};
```

```
[](int i) noexcept {return i + 1;};
```

```
[&x](int i) mutable { ++x; return i + x;};
```


Lambda

```
int main() {  
  
    auto f = [](int value) {  
        return value > 4;  
    };  
  
    f(1);  
  
    return 0;  
}
```

```
int main()  
{  
    class __lambda_6_14 {  
    public:  
        bool operator()(int value) const {  
            return value > 4;  
        }  
  
    };  
  
    __lambda_6_14 f = __lambda_6_14{};  
    f.operator()(1);  
    return 0;  
}
```

<https://cppinsights.io/>

Lambda

```
int main()
{
    auto first = [](int x){ return x + 1;};
    auto second = [](int x){ return x + 1;};

    static_assert(
        !std::is_same<
            decltype(first),
            decltype(second)
        >::value,
        "must be different!"
    );
}
```

```
int main()
{
    class __lambda_6_14 {
    public:
        bool operator()(int value) const {
            return x + 1;
        }
    };

    __lambda_6_14 f = __lambda_6_14{};

    class __lambda_6_17 {
    public:
        bool operator()(int value) const {
            return x + 1;
        }
    };

    __lambda_6_17 = __lambda_6_17{};

}
```

Capture

- [x,y] - by value
- [=] - all by value with automatic storage duration
- [&x, &y] - by reference
- [&] - all by reference with automatic storage duration
- [this] - by copy current object
- [*this] - by reference current object

Capture

```
int main() {  
    int x = 1;  
    int y = 2;  
  
    auto f = [x, &y](int v) {  
        return v + x + y;  
    };  
  
}
```

```
int x = 1;  
int y = 2;  
  
class __lambda_7_12  
{  
public:  
    inline int operator()(int v) const {  
        return (v + x) + y;  
    }  
  
private:  
    int x;  
    int & y;  
  
public:  
    __lambda_7_12(int & _x, int & _y)  
    : x{_x}  
    , y{_y}  
    {}  
  
};  
  
__lambda_7_12 f = __lambda_7_12{x, y};
```

Capture

```
struct Foo {  
    int field = 0;  
  
    int func(int i) {  
        auto f = [this](int value) {  
            return field + value;  
        };  
  
        return f(i);  
    }  
};
```

```
struct Foo  
{  
    int field = 0;  
    inline int func(int i) {  
        class __lambda_8_16 {  
        public:  
            inline int operator()(int value) const{  
                return __this->field + value;  
            }  
        private:  
            Foo * __this;  
  
        public:  
            __lambda_8_16(Foo * _this)  
                : __this(_this)  
            {}  
  
        };  
  
        __lambda_8_16 f = __lambda_8_16{this};  
        return this->func(i);  
    }  
};
```

Capture

```
struct Foo {  
    int field = 0;  
  
    int func(int i) {  
        auto f = [*this](int value) {return field  
+ value;};  
  
        return f(i);  
    }  
};
```

```
struct Foo  
{  
    int field = 0;  
    inline int func(int i) {  
        class __lambda_8_16 {  
        public:  
            inline int operator()(int value) const{  
                return (&__this)->field + value;  
            }  
        private:  
            Foo __this;  
  
        public:  
            __lambda_8_16(const Foo & _this)  
                : __this{_this}  
            {}  
  
        };  
  
        __lambda_8_16 f = __lambda_8_16{*this};  
        return this->func(i);  
    }  
};
```

Mutable

```
int main() {  
    int x = 0;  
    auto f = [x]() mutable {  
        ++x;  
        std::cout << x << std::endl;  
    };  
  
    f();  
    std::cout << x << std::endl;  
    f();  
    std::cout << x << std::endl;  
}
```

Immediately Invoked Function

```
int main() {  
    [] () {std::cout << "ITMO\n";} ();  
  
    int x = 2023;  
    [&x] () noexcept {++x;} ();  
    std::cout << x << std::endl;  
  
    return 0;  
}
```


Immediately Invoked Function

```
void SomeHardLogic();
```

```
struct Foo {  
    Foo() {  
        SomeHardLogic();  
    }  
};
```

```
Foo createFooA();
```

```
Foo createFooB();
```

```
int main() {  
    Foo f;    // too expensive  
  
    bool someCondition;  
  
    if(someCondition) {  
        f = createFooA();  
    } else {  
        f = createFooB();  
    }  
  
    return 0;  
}
```

Immediately Invoked Function

```
void SomeHardLogic();
```

```
struct Foo {  
    Foo() {  
        SomeHardLogic();  
    }  
};
```

```
Foo createFooA();
```

```
Foo createFooB();
```

```
int main() {  
    const Foo f; // compile-time error  
  
    bool someCondition;  
  
    if(someCondition) {  
        f = createFooA();  
    } else {  
        f = createFooB();  
    }  
  
    return 0;  
}
```

Immediately Invoked Function

```
void SomeHardLogic(int);
```

```
struct Foo {  
    Foo(int value) {  
        SomeHardLogic(value);  
    }  
};
```

```
Foo createFooA();
```

```
Foo createFooB();
```

```
int main() {  
    Foo f; // compile-time error  
  
    bool someCondition;  
  
    if(someCondition) {  
        f = createFooA();  
    } else {  
        f = createFooB();  
    }  
  
    return 0;  
}
```

Immediately Invoked Function

```
void SomeHardLogic(int);
```

```
struct Foo {
```

```
    Foo(int value) {
```

```
        SomeHardLogic(value);
```

```
    }
```

```
};
```

```
Foo createFooA();
```

```
Foo createFooB();
```

```
int main() {
```

```
    bool someCondition;
```

```
    const Foo f = [someCondition]() {
```

```
        if(someCondition) {
```

```
            return createFooA();
```

```
        } else {
```

```
            return createFooB();
```

```
        }
```

```
    }();
```

```
    return 0;
```

```
}
```

Immediately Invoked Function

```
void SomeHardLogic(int);
```

```
struct Foo {
```

```
    Foo(int value) {
```

```
        SomeHardLogic(value);
```

```
    }
```

```
};
```

```
Foo createFooA();
```

```
Foo createFooB();
```

```
int main() {
```

```
    bool someCondition;
```

```
    const Foo f = std::invoke([someCondition]() {
```

```
        if(someCondition) {
```

```
            return createFooA();
```

```
        } else {
```

```
            return createFooB();
```

```
        }
```

```
    });
```

```
    return 0;
```

```
}
```

Lambda Inheriting

```
template<typename T, typename U>
struct SimpleOverloader : public T, U {
    SimpleOverloader(T t, U u) : T(t), U(u)
    {}

    using T::operator();
    using U::operator();
};
```

```
template<typename T, typename U>
SimpleOverloader<T,U> MakeOverloaded(
    const T& t, const U& u
){
    return SimpleOverloader<T, U>(t, u);
}
```

```
int main() {
    auto o = MakeOverloaded(
        [](int i) { std::cout << "int\n";},
        [](float i) { std::cout << "float\n";}
    );

    o(1);
    o(1.1f);

    return 0;
}
```

Bind

```
int threeArgFunc(int x, int y, int z) {  
    return x + y + z;  
}  
  
int main() {  
  
    auto twoArgFunc = [](int x, int z) { return threeArgFunc(x, 2, z);};  
    auto oneArgFunc = [&](int z) { return twoArgFunc(1, z);};  
  
    std::cout << oneArgFunc(3);  
  
    return 0;  
}
```

Generic Lambda

```
int main() {  
    const auto f = [] (auto x, auto y) {  
        return x + y;  
    };  
  
    std::cout << f(1, 2) << std::endl;  
    std::cout << f(1, 2.3) << std::endl;  
  
    std::cout << f(std::string{ "abc" },  
std::string{ "def" }) << std::endl;  
  
    return 0;  
}
```

```
class __lambda_10_19 {  
public:  
    template<  
        class type_parameter_0_0,  
        class type_parameter_0_1  
    >  
    inline auto operator()(  
        type_parameter_0_0 x,  
        type_parameter_0_1 y  
    ) const {  
        return x + y;  
    }  
};
```


Recursive Lambda

```
int main() {  
    const auto factorial = [] (int n) noexcept {  
        const auto impl = [] (int n, const auto& impl) noexcept -> int {  
            return n > 1 ? n * impl(n - 1, impl) : 1 ;  
        };  
        return impl(n, impl);  
    };  
  
    std::cout << factorial( 4 );  
  
    return 0;  
}
```

Function Pointer & Lambda

```
int main() {  
  
    auto f = [](int value) {  
        return value > 4;  
    };  
  
    return 0;  
}
```

```
class __lambda_4_13  
{  
public:  
    inline /*constexpr */ bool operator()(int value) const  
    {  
        return value > 4;  
    }  
  
    using retType_4_13 = bool (*)(int);  
    inline constexpr operator retType_4_13 () const noexcept  
    {  
        return __invoke;  
    };  
  
private:  
    static inline /*constexpr */ bool __invoke(int value)  
    {  
        return __lambda_4_13{}.operator()(value);  
    }  
  
};
```

Array of lambda

```
int main() {  
    using TFunc = int (*)(int);  
  
    std::vector<TFunc> v;  
  
    v.push_back([](int i){return i + 1;});  
    v.push_back([](int i){return i + 2;});  
    v.push_back([](int i){return i + 3;});  
  
    for(auto& f : v)  
        std::cout << f(1) << std::endl;  
    return 0;  
}
```

std::function

Умеет хранить и вызывать

- Функции
- Лямбды
- Функторы
- Методы класса
- `std::bad_function_call`

std::function

```
int main() {  
    std::function<int(int)> f = Incr{};  
    std::function<int(int)> f2 = incr;  
  
    std::function<int(int)> f3 = [] (int value) {return value + 1;};  
  
    std::function<int(const Foo&, int)> f4 = &Foo::incr;  
  
    std::cout << f(1) << " "  
              << f2(1) << " "  
              << f3(1) << " "  
              << f4(Foo{}, 1) << std::endl;  
  
    return 0;  
}
```

std::function

```
// simple implementation
```