Zrozumieć lambdy w C++.

MARCIN LISOWSKI

Hej!
Jestem Marcin Lisowski

Pracuję w Siili Szczecin

Prezentacja dostępna na:
meetup.com - https://github.com/szczecin-cpp/meetup-03

Spis treści

NOWE ZNACZY LEPSZE?

PO CO POWSTAŁY LAMBDY?

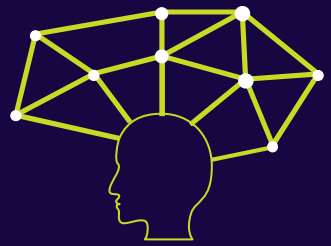
Alternatywy

Funktor

```
struct Functor
{
    void operator()()
    {
        // Do some fancy stuff here
    }
};
```

Funkcja

```
void function()
{
    // Do some fancy stuff here
}
```



[]() -> type { /* ... */ }

Lambda

przykłady

```
[](int a) -> int { return a * 2; }
[](int a, int b) { return a * b; }
[] { std::cout << "szczecin::cpp\n" }
```

C++11

typ lambdy

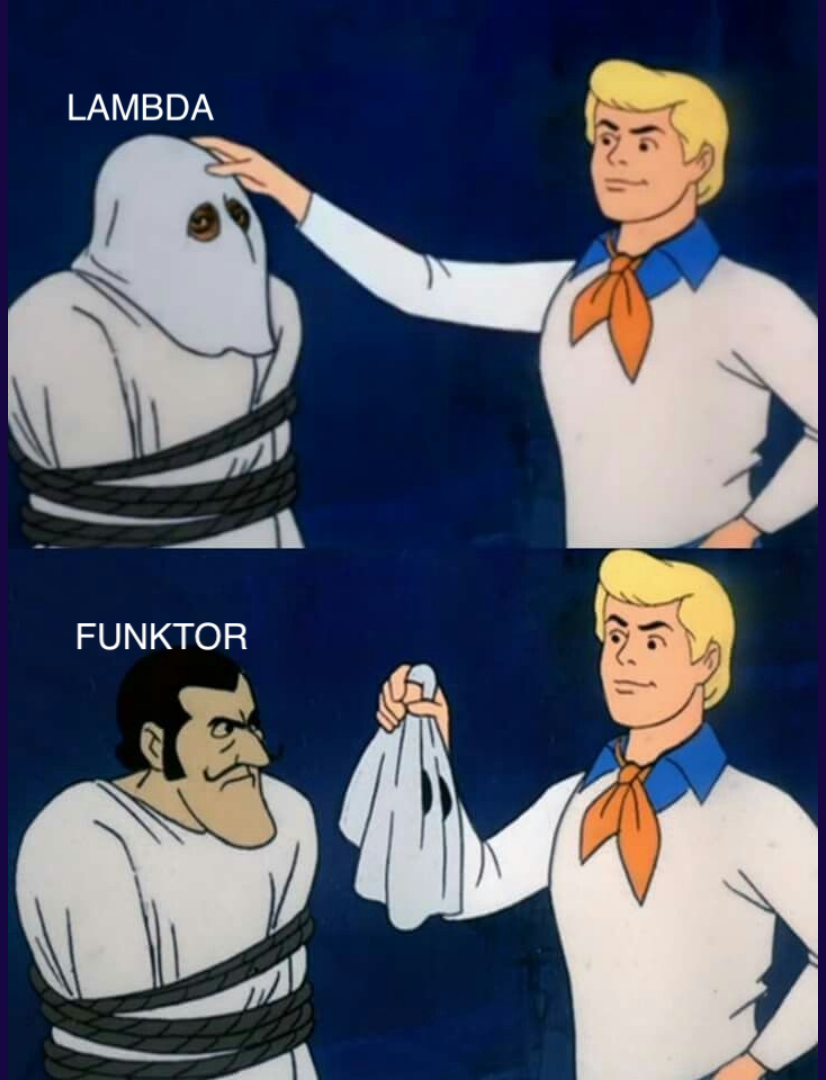
```
auto lambda = [](int a) { return a * 10; };
std::cout << typeid(lambda).name(); // GCC9 output: Z4mainEUIvE_
```

typ lambdy

```
int main()
{
    auto lambda = [](int a)
    {
        return a * 10;
    };
}
```

```
int main()
{
    class __lambda_5_22
    {
    public:
        inline int operator()(int a) const
        {
            return a * 10;
        }
    };

    __lambda_5_22 lambda = __lambda_5_22{};
}
```


LAMBDA
FUNKTOR

typ lambdy

```cpp
int main()
{
    auto lambda = [](int a){};
    lambda = [](int a){};
}
```

```cpp
int main()
{
    class __lambda_7_26
    {
        // Implementation
    };

    class __lambda_8_20
    {
        // Implementation
    };

    __lambda_8_20 lambda = __lambda_8_20();
    lambda = __lambda_7_26();
}
```

lambda a std::function

```
int main()
{
    std::function<bool(int)> fnc = [](int a){ return a > 2; };
    fnc = [](int a){ return a < 100; };
}
```

lambda a std::function

```
void compare(std::function<bool(int, int)> predicate)
{
    // Implementation
}

int main()
{
    compare([](int a, int b)
    {
        return a > b;
    });
}
```

dedukcja zwracanego typu

```
auto lambda = []() { return 0; };
```

```
auto lambda2 = []() -> SomeClass { return 0; };
error: could not convert '0' from 'int' to 'SomeClass'
```

dedukcja zwracanego typu

```
auto lambda = [](int x)
{
    if (x > 0)
        return "OK";
    else
        return "FAIL";
};
```

```
inline const char* operator()(int x) const
{
    if(x > 0) return "OK";
    else return "FAIL";
}
```

capture lista [=]

```
int x = 0;
auto lambda1 = [x](){ return x * 2; };
auto lambda2 = [=](){ return x * 2; };
```

capture lista [=]

```
int main()
{
    int x = 0;
    auto lambda = [x]()
    {
        std::cout << x;
    };
    lambda();
}
```

```
class __lambda_7_22
{
public:
    __lambda_7_22(int x)
        : _x(x)

    inline void operator()() const
    {
        std::cout << _x;
    }

private:
    int _x;
};
...
```

capture lista [=]

```
int main()
{
    int x = 0;
    auto lambda = [=]()
    {
        std::cout << x;
    };
    lambda();
}
```

```
class __lambda_7_22
{
public:
    __lambda_7_22(int x)
        : _x(x)

    inline void operator()() const
    {
        std::cout << _x;
    }

private:
    int _x;
};
...
```

capture lista [&]

```
int x = 0;
auto lambda1 = [&x](){ return ++x; };
auto lambda2 = [&](){ return ++x; };
```

capture lista [=]

```
int main()
{
    int x = 0;
    auto lambda = [&x]()
    {
        std::cout << x;
    };
    lambda();
}
```

```
class __lambda_7_22
{
public:
    __lambda_7_22(int& x)
        : _x(x)

    inline void operator()() const
    {
        std::cout << _x;
    }

private:
    int& _x;
};
...
```

modyfikacja kontekstu

```
int x = 0;
auto lambda = [x]()
{
    x += 1;
};
```

```
error: assignment of read-only variable 'x'
```

modyfikacja kontekstu

```
int x = 0;
auto lambda = [x]()
{
    x += 1;
};
```

```
...
class __lambda_6_22
{
public:
    inline void operator()() const
    {
        x += 1;
    }

private:
    int x;
...
```

mutable

```
const int x = 0;
auto lambda = [x]() mutable
{
    x += 1;
};
```

bez mutable

```
int x = 0;

class __lambda_7_18
{
public:
    inline void operator()() const
    {
        x += 1;
    }

private:
    int x;

...
```

z mutable

```
int x = 0;

class __lambda_7_18
{
public:
    inline void operator()()
    {
        x += 1;
    }

private:
    int x;

...
```

przechwytywanie const

```
const int x = 0;
auto lambda = [x]() mutable
{
    x += 1;
};
```

```
error: assignment of read-only variable 'x'
```

zmienne globalne

```
int global = 0;

int main()
{
    auto lambda = [=]() mutable
    {
        global += 1;
    };

    lambda();
    std::cout << global;
}
```

```
output: 1
```

zmienne globalne

```
int global = 0;

int main()
{
    auto lambda = [global]() mutable
    {
        global += 1;
    };

    lambda();
    std::cout << global;
}
```

```
error: 'global' cannot be captured
because it does not have automatic
storage duration
```

zmienna klasy

```
class SampleClass
{
public:
    int _x;

    void print()
    {
        auto lambda = [=]()
        {
            std::cout << _x;
        };
        lambda();
    }
};
```

zmienna klasy

```
class SampleClass
{
public:
    int _x;

    std::function<void()> get()
    {
        return [=]()
        {
            std::cout << _x;
        };
    }
};
```

```
class __lambda_11_23
{
    public:
    inline void operator()() const
    {
        std::cout << __this->_x;
    }

    private:
        SampleClass* __this;

...
```

lambdy bezstanowe

```
void(*function)(int) = [](int x)
{
    std::cout << x;
};

function(5);
```

lambdy bezstanowe

```
void(*function)(int) = [](int x)
{
    std::cout << x;
};
```

```
...

    using RawFunctionPtr = void(*)(int);
    inline operator RawFunctionPtr() const noexcept
    {
        return __invoke;
    }

private:
    static inline void __invoke(int x)
    {
        std::cout << x;
    }
...
```

lambdy bezstanowe

```
typedef void (*callback)(uint32_t* new_address);
error_code old_important_api(uint32_t* addres, uint32_t length, callback fnc);
```

```
old_important_api(frame_buffer, 1024, [](uint32_t new_address)
{
    // Handle function callback here
});
```

C++14

inicjowanie przechwytywanych obiektów

```
auto lambda = [x = a + b]() { std::cout << x; };
```

```
auto lambda = [ptr = std::move(p)] { /* use ptr normally */ };
```

generyczne lambdy

```
int main()
{
    auto lambda = [](auto x) { std::cout << x; };
    lambda(3);
}
```

generyczne lambdy

```
int main()
{
    class __lambda_6_20
    {
    public:
        template<class T>
        inline auto operator()(T x) const
        {
            std::cout << x;
        }
    };

    __lambda_6_20 lambda = __lambda_6_20();
    lambda(3);
}
```

generyczne lambdy

```
auto lambda = []<typename T>(T a)
{
    std::cout << a;
};
```

generyczne lambdy :(

```
auto lambda = []<typename T>(std::vector<T> a)
{
    std::cout << a;
};
```

C++17

lambda jako constexpr

```
constexpr auto lambda = [] (int x) { return x * x; };
static_assert(lambda(2) == 4);
```

lambda jako constexpr

```
constexpr auto factorial = [](int x)
{
    int result = 1;
    while (x)
    {
        result *= x;
        --x;
    }
    return result;
};
const big_number = factorial(5); // big_number == 120
```

poprawione przechwytywanie this

```
auto lambda = [this]{};         // C++11
auto lambda = [self=*this]{};   // C++14
auto lambda = [*this]{};        // C++17
```

C++20

[=, this]

```
class SampleClass
{
public:
    int _x;

    std::function<void()> get()
    {
        return [=]()
        {
            std::cout << _x;
        };
    }
};
```

```
warning: implicit capture of
'this' via '[=]' is deprecated
in C++20
```

[=, this]

```
class SampleClass
{
public:
    int _x;

    std::function<void()> get()
    {
        return [this]()
        {
            std::cout << _x;
        };
    }
};
```

generyczne lambdy to teraz pełnoprawne szablony

```
auto vector_size = []<typename T>(std::vector<T>& x)
{
    return x.size();
};
std::vector<std::string> vec { "1", "2" };
std::cout << vector_size(vec); // output: 2
```

Tips & Tricks

brak wpływu na wydajność

zwykła funkcja

```
function(int):
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], edi
    mov     eax, DWORD PTR [rbp-4]
    add     eax, 2
    pop     rbp
    ret
```

lambda / funktor

```
lambda::operator()(int):
    push    rbp
    mov     rbp, rsp
    mov     QWORD PTR [rbp-8], rdi
    mov     DWORD PTR [rbp-12], esi
    mov     eax, DWORD PTR [rbp-12]
    add     eax, 2
    pop     rbp
    ret
```

natychmiastowe wywołanie

```
struct SampleObject
{
    const int i = []()
    {
        // Complicated logic
        return 0;
    }();
};
```

Podsumowanie

Lambda == funktor

Lambdy to składnia pomagająca zadeklarować funktor.

Lambdy bezstanowe

Lambdy które nie przechwytują zmiennych, są nazywane bezstanowymi. Można je przekonwertować do wskaźnika na funkcję.

Wydajne

Lambdy nie obciążają dodatkowo wydajności.

Zmienne klasy

Przechwytywanie zmiennych z klasy odbywa się poprzez dereferencję wskaźnika do obiektu.

Zmienne globalne

Zmienne globalne / statyczne dostępne są w lambdach bez konieczności przechwytywania.

Kopia przenosi stan

Kopiowanie lambdy kopiuje także jej aktualny stan.

Linki

- cppinsights.io
- godbolt.org
- medium.com/lambdas-are-not-magic
- web.mst.edu/lambdas-under-the-hood

Dzięki!

Czas na pytania :)