



Uniwersytet
Wrocławski

Zaawansowane programowanie w C++

Zbigniew Koza

Wydział Fizyki i Astronomii

O czym będą te zajęcia?

Program wykładu

- Zaawansowane C++ (98)
- C++11/14

Program wykładu nieco rozwinięty

- Zaawansowane C++ (98)
 - Szablony
 - Wyjątki
 - Biblioteki (tworzenie i używanie)
- C++11/14
 - Nowości składni
 - Wyrażenia lambda
 - `&&` i `std::move`
 - `#include <thread, future, etc.>`
 - OpenMP (jeśli zdążymy)?

Program wykładu nieco rozwinięty

- Zaawansowane C++ (98)
 - Szablony
 - Wyjątki
 - Biblioteki (tworzenie i używanie)
- C++11/14
 - Nowości w składni
 - Wyrażenia lambda
 - && i std::move
 - #include <thread, future, etc.>
 - OpenMP (jeśli zdążymy)?

Czy coś mam dopisać?

- ***
- ***

Wyrażenia lambda

Zaczniemy od formalnej definicji

- In programming languages, **closures** (also lexical closures or function closures) are techniques for implementing lexically scoped name binding in languages with first-class functions. Operationally, **a closure is a record storing a function together with an environment**: a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created. A closure—unlike a plain function—allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.
([en.wikipedia.org/wiki/Closure_\(computer_programming\)](https://en.wikipedia.org/wiki/Closure_(computer_programming)))

I po polsku...

- **Domknięcie** – w metodach realizacji języków programowania jest to **obiekt wiążący funkcję lub referencję do funkcji oraz środowisko mające wpływ na tę funkcję** w momencie jej definiowania. **Środowisko przechowuje wszystkie nielokalne obiekty wykorzystywane przez funkcję.** Realizacja domknięcia jest zdeterminowana przez język, jak również przez kompilator. Domknięcia występują głównie **w językach funkcyjnych**, w których funkcje mogą zwracać inne funkcje (tzw. funkcje wyższego rzędu), wykorzystujące zmienne utworzone lokalnie.

Uff, przykład...

- JavaScript

```
function mnozenie_przez(x) {  
    return function(y) {  
        return x * y;  
    };  
}
```

```
var iloczyn_5_przez = mnozenie_przez(5);  
console.log(iloczyn_5_przez(12)); // 60
```

I python...

```
def f(x):  
    def g(y):  
        return x + y  
    return g
```

Środowisko funkcji g

Funkcja nazwana

```
def h(x):  
    return lambda y: x + y
```

Funkcja anonimowa

domknięcia

→ a = f(1)

→ b = h(1)

f(1)(5)

h(1)(5)


Jak to się robi w C++?

- Wersja „ułamna” to obiekty funkcyjne

```
struct Porownaj
{
    bool operator()(int a, int b) const
    {
        return std::abs(a) < std::abs(b);
    }
};
```

Anonimowy obiekt funkcyjny

```
int main()
{
    std::vector<int> v {1, -2, 3, 8, 0};
    std::sort(v.begin(), v.end(), Porownaj());
}
```



Jak to się robi w C++?

- obiekty funkcyjne:
 - Mogą być używane jak funkcje (składnia)
 - Mogą przechowywać swój stan (środowisko) w zmiennych obiektu

```
struct Porownaj
{
    int n = 0;
    bool operator()(int a, int b) const
    {
        n++;
        return std::abs(a) < std::abs(b) + n; // jak to działa?
    }
};
```

Wada tego rozwiązania

- Obiekty funkcyjne nie „dziedziczą” swojego środowiska automatycznie
 - Wymagana jest ręczna obsługa „środowiska”

```
struct Porownaj
{
    int n = 0;
    bool operator()(int a, int b) const
    {
        return std::pow(a, n) < std::pow(b, n);
    }
    void set_n(int m) { n = m; }
};
```

Na pomoc wzywamy funkcje lambda!

```
struct Porownaj
{
    bool operator()(int a, int b) const
    {
        return std::abs(a) < std::abs(b);
    }
};

int main()
{
    std::vector<int> v {1, -2, 3, 8, 0};
    std::sort(v.begin(), v.end(), Porownaj());
}
```

Na pomoc wzywamy funkcje lambda!

```
struct Porownaj  
{  
    bool operator()(int a, int b) const  
    {  
        return std::abs(a) < std::abs(b);  
    }  
};
```

```
int main()  
{  
    std::vector<int> v {1, -2, 3, 8, 0};  
    std::sort(v.begin(), v.end(), [ ](int a, int b){  
        return std::abs(a) < std::abs(b);  
    });  
}
```

Przykład nr 2


```
auto f = [ ](int a, int b) { return std::abs(a) < std::abs(b); }  
...  
std::sort(v.begin(), v.end(), f);
```

- Funkcje lambda zachowują się jak funkcje anonimowe
- Można je przypisywać zmiennym, przekazywać jako argumenty do i z funkcji...
- Wewnątrz funkcji lambda można definiować inne funkcje lambda

Jak to działa?

```
struct nazwa_0001
{
    bool operator()(int a, int b) const
    {
        return std::abs(a) < std::abs(b);
    }
};
```

```
int main()
{
    std::vector<int> v {1, -2, 3, 8, 0};
    std::sort(v.begin(), v.end(), [ ](int a, int b){
        return std::abs(a) < std::abs(b);
    });
}
```




Jak to działa?

- Funkcje lambda są **obiektami funkcyjnymi**

```
struct nazwa_0001
{
    bool operator()(int a, int b) const
    {
        return std::abs(a) < std::abs(b);
    }
};
```

```
int main()
{
    std::vector<int> v {1, -2, 3, 8, 0};
    std::sort(v.begin(), v.end(), nazwa_0001());
}
```



Jak to działa?

- Funkcje lambda są **obiektami funkcyjnymi**
- Ale w praktyce traktuje się je jak **funkcje anonimowe**

```
int main()
{
    std::vector<int> v {1, -2, 3, 8, 0};
    std::sort(v.begin(), v.end(), [ ](int a, int b){
        return std::abs(a) < std::abs(b);
    });
}
```

Składnia

[miejsce na przechwycenie środowiska]

(argumenty funkcji)

->zwracany typ (domyślnie: auto)

{

ciało funkcji

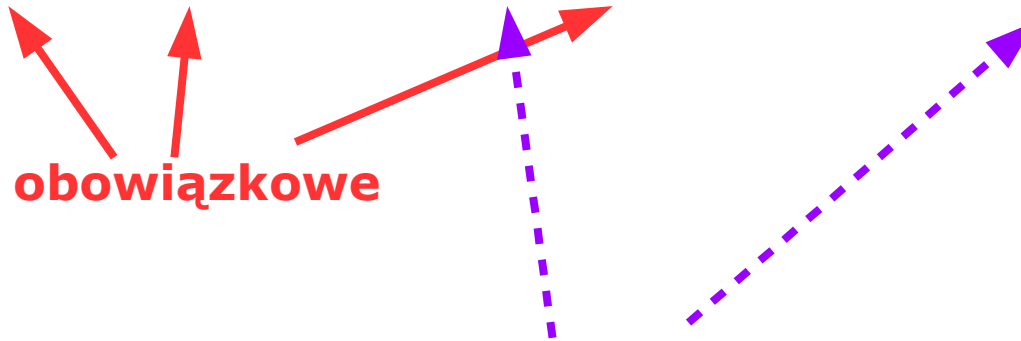
}

(wartości argumentów z jakimi wyrażenie lambda ma zostać wywołane)

[](int a, int b) -> int { return a < b;} (5, 7);

Składnia

[] (int a, int b) -> int { return a < b;} (5, 7);



obowiązkowe

opcjonalne, rzadko widywane

Przechwyt środowiska - przykład

```
struct X
{
    int x, y;
    int operator()(int);
    void f()
    {
        // Kontekstem funkcji lambda jest funkcja X::f
        [=]()->int
        {
            return operator()(this->x + y); // X::operator()(this->x + (*this).y)
                                           // typem this jest X*
        };
    }
};
```

Przechwyt środowiska - przykład

```
struct X
{
    int x, y;
    int operator()(int);
    void f()
    {
        // Kontekstem funkcji lambda jest funkcja X::f
        [=]()->int
        {
            return operator()(this->x + y); // X::operator()(this->x + (*this).y)
                                           // typem this jest X*
        };
    }
};
```

Przechwyt (ang. *captures*)

- [] - nic
- [=] - wszystkie zmienne automatyczne środowiska przez wartość, ale *this przez referencję
- [&] - wszystkie zmienne automatyczne środowiska przez referencję
- [a, &x] – tylko a (wartość) i x (przez ref.)
- [=, &y] – wszystko przez wartość, ale y przez ref.
- [&, a, b, c] – wiadomo...

Moment przechwytu

- Podczas definiowania funkcji lambda (!)

```
14 int main()
15 {
16     int x = 10;
17     auto f = [&x](int a) { return a + x; };
18     cout << f(50) << "\n";
19     {
20         int x = 5;
21         cout << f(1);
22     }
23
24     return 0;
25 }
26
```

60

11

Przechwyt

- Na początku najczęściej będziesz używać pustego przechwyty

[]

- Te nawiasy oznaczają albo początek funkcji lambda, albo tablicę w stylu języka C

Typ wyniku

```
[ ] (int a, int b) -> int { return a < b; }
```

- Najczęściej jest pomijany

```
[ ] (int a, int b) { return a < b; }
```

- Kompilator dedukuje typ wyniku na podstawie postaci funkcji return (lub jej braku) (C++14)

Zapamiętaj

- Funkcje lambda nie są funkcjami ani anonimowymi funkcjami, ani wskaźnikami na funkcje, tylko funktorami (obiektami) klas automatycznie generowanych przez kompilator
- Ich stosowanie umożliwia kompilatorowi stosowanie *inliningu*, a nam pisanie krótszego i bardziej czytelnego kodu bez utraty jego wydajności
- Można bez nich żyć, ale...

Dalsza lektura

- <https://stackoverflow.com/questions/7627098/what-is-a-lambda-expression-in-c11>
- <http://www.stroustrup.com/C++11FAQ.htm#lambda>