



Uniwersytet
Wrocławski

Wyrażenia lambda (C++)

Zbigniew Koza

Wydział Fizyki i Astronomii

Wprowadzenie

int x;

namespace A {float f;}

namespace B {char c;}

Funkcja

```
void f()  
{  
    float z = ::x + A::f + A::B::c;  
}
```

- Funkcje mają dostęp do obiektów tworzonych lokalnie i do tych istniejących już wcześniej w otaczających je przestrzeniach nazw

W czym problem?

- Sięganie w funkcjach do zmiennych nielokalnych jest wygodne, ale zdecydowanie niezalecane
 - Dlaczego?
 - W C++ pisze się bardzo duże programy
 - Zmienne nielokalne bardzo utrudniają zapanowanie nad poprawnością działania programu
 - Pomyśl o tysiącu funkcji, z których każda może modyfikować ten sam obiekt – sekwencyjnie w nieokreślonej kolejności lub nawet jednocześnie

Rozwiązanie połowiczne (niezalecane)

- Zmienne lokalne w jednostce kompilacji
- Zmienne lokalne w funkcji

static

```
→ static int x = 0;

int f()
{
→ static int y = 0;
  y++;
  return x + y;
}
```

- Zalety:
 - mniej bałaganu

- Wady:
 - Jeden zestaw zmiennych dla wszystkich wywołań takiej funkcji
 - Rekurencja, współbieżność, inicjalizacja etc. => kłopot

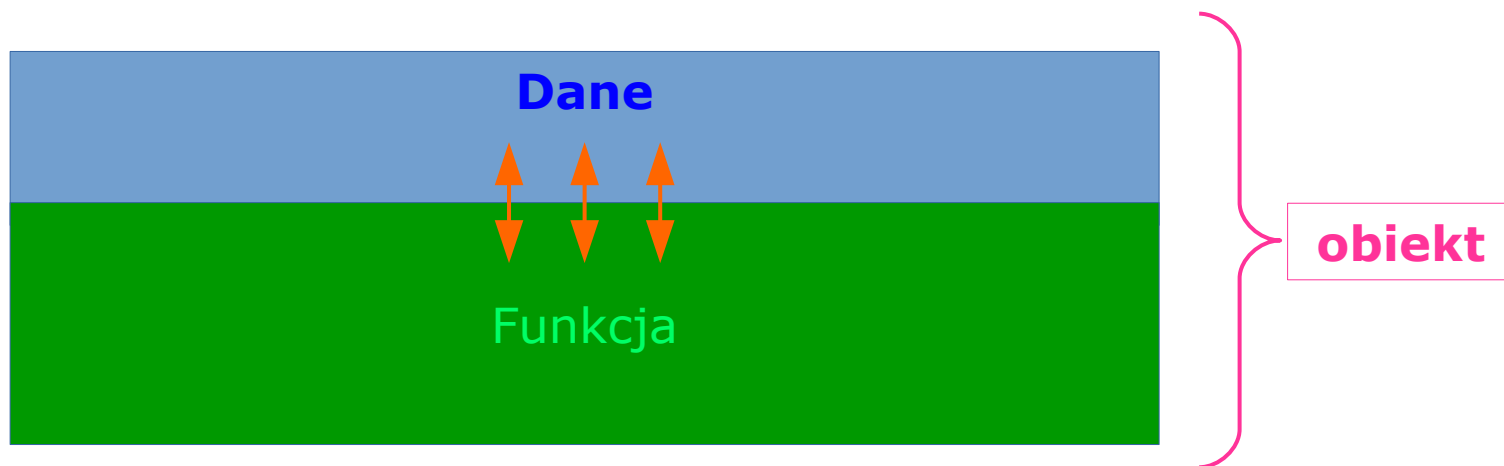
Idealna funkcja

- Czarna skrzynka
z dobrze określonym interfejsem



Rozwiązanie (C++)

- Obiekt: funkcja + dane



- Funkcja zasadniczo jest całym interfejsem obiektu
- Kilka obiektów => kilka funkcji z osobnymi zestawami zewnętrznych danych „prywatnych”
- Rekurencja, współbieżność, etc.: nie ma problemu

Ale po co te komplikacje?

- Np. gdy biblioteka X wymaga w swym interfejsie funkcji o określonej sygnaturze, a dla ciebie to zbyt mocne ograniczenie
 - `std::sort` wymaga funkcji dwuargumentowej, nie ma więc miejsca na dodatkowe parametry, tymczasem ty chcesz sortowanie dostosować do bieżącego kontekstu
 - np. wielokrotne sortowanie punktów wg odległości od punktu A, tyle że za każdym razem A leży gdzie indziej

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 składowych obiektu

```
struct Porownaj
{
    int n;
    Porownaj(int n = 1) : n{n} { }
    bool operator()(int a, int b) const
    {
        return std::pow(a, n) < std::pow(b, n);
    }
};
```

Wada tego rozwiązania

- Obiekty funkcyjne nie zawsze „dziedziczą” swojego środowiska automatycznie
 - Możliwa 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);
    }
};
```

wersja
tradycyjna

```
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?

```
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?

```
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?

Różnica 1

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

Różnica 2


```
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) *domyślnie: ()*

opcjonalny modyfikator *(mutable, constexpr, consteval)*

->zwracany typ *(domyślnie: auto)*

{

ciało funkcji

}

(wartości argumentów z jakimi wyrażenie lambda ma zostać wywołane)

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

Składnia

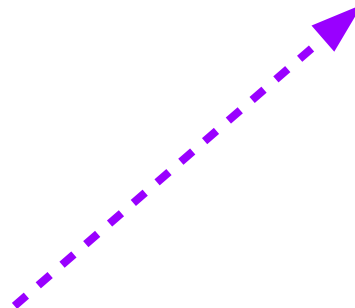
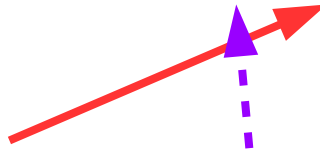
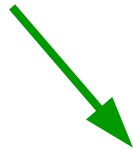
Nieobowiązkowe
prawie zawsze
używane

Miejsce na mutable,
constexpr, consteval

[] (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 przechwyty

- 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)

Lambdy bezargumentowe

- Puste nawiasy () można pominąć

```
#include <iostream>

int main()
{
    auto f = []{std::cout << "Hello!\n";};
    f();
}
```



Hello!

mutable

- Lambdy nie mogą modyfikować swojego środowiska
 - Chyba że zmodyfikujemy je słowem `mutable`

```
#include <iostream>

int main()
{
    auto f = [z = 0] () mutable
    {
        return ++z;
    };
    std::cout << f() << f() << f() << "\n";
}
```



123

referencje

- Lambdy mogą modyfikować środowisko przekazane przez referencję
 - Bo sama referencja jest przypisana do tego samego obiektu (stała)...

```
int main()
{
    int x = 0;
    auto f = [&x] ()
    {
        return ++x;
    };
    std::cout << f() << f() << f() << "\n";
}
```



123

Środowisko z inicjalizatorem

- Środowisko można wprowadzać (a nie przechwytywać)
 - Składnia: `identyfikator = inicjalizator`
 - Typ składowej powielany jest z inicjalizatora

```
#include <iostream>

int main()
{
    auto f = [z = 0] () mutable
    {
        return ++z;
    };
    std::cout << f() << f() << f() << "\n";
}
```

➡ 123

auto-parametry lambda

- Tłumaczone są na szablon operatora()

```
#include <iostream>

struct X
{
    template <typename T>
    auto operator()(T x, T y) const
    {
        return x < y;
    }
};

int main()
{
    X x;
    std::cout << x(2, 3) << " " << x(3.0, 2.0) << "\n";
    auto f = [](auto x, auto y){return x < y;};
    std::cout << f(2, 3) << " " << f(3.0, 2.0) << "\n";
}
```

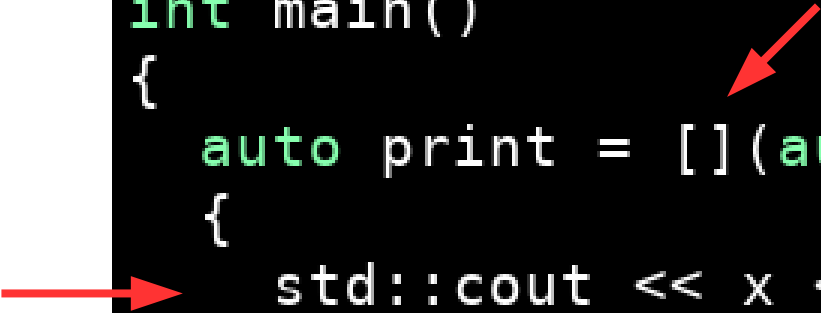


Zmienne globalne

- Nie muszą być przechwytywane

```
#include <iostream>

int main()
{
    auto print = [](auto x)
    {
        std::cout << x << " ";
    };
    print(2);
    print("Ala");
    print(2.71);
    std::cout << "\n";
}
```



Zmienne z przestrzeni nazw

- Nie mogą być przechwytywane

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



g++

```
6.cpp: In function 'int main()':
6.cpp:5:18: error: capture of non-variable 'std'
  5 |     auto print = [&std::cout](auto x)
    |                   ^~~
```



clang++

```
6.cpp:5:21: error: expected ',', or ']' in lambda capture list
  auto print = [&std::cout](auto x)
                ^
```

Przykład: Intel TBB

- Czy rozumiesz ten zapis?

```
void ParallelApplyFoo( float* a, size_t n ) {  
    parallel_for( blocked_range<size_t>(0,n),  
        [=](const blocked_range<size_t>& r) {  
            for(size_t i=r.begin(); i!=r.end(); ++i)  
                Foo(a[i]);  
        }  
    );  
}
```

```
void ParallelApplyFoo(float a[], size_t n) {  
    parallel_for(size_t(0), n, [=](size_t i) {Foo(a[i]);});  
}
```

Kolejne przykłady

```
std::transform(indices.begin(), indices.end(), indices.begin(),  
               [&](auto&& index) { return mapping[index]; });
```

```
boost::write_graphviz(  
    std::cout,  
    g,  
    [&](auto& out, auto v) { out << "[label=\"" << g[v].mesh_id << "\"]"; },  
    [&](auto&, auto e) {}  
);
```

Zapamiętaj

- Funkcje lambda **nie są** funkcjami, anonimowymi funkcjami, wskaźnikami na funkcje; są **funktorami** (**obiektami**) klas automatycznie generowanych przez kompilator
- Ich stosowanie umożliwia kompilatorowi agresywną optymalizację (*inline*, *constexpr*), a nam pisanie krótszego i 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>

Test na zrozumienie wykładu

- (...) 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)))

or in Polish...

- **Domknięcie** – w metodach realizacji języków programowania jest to **obiekt wiążący funkcję (lub referencję do funkcji) ze środowiskiem mającym wpływ na działanie tej funkcji**, w stanie, jaki to środowisko miało **w momencie definiowania domknięcia**. **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.