

Kurs języka C++

14. Obiekty funkcyjne i lambdy

Spis treści

- ▶ Funktory i predykaty
- ▶ Predefiniowane obiekty funkcyjne
- ▶ Funkcje lambda

Obiekty funkcyjne

- ▶ Obiekt funkcyjny to obiekt, w którym jest zdefiniowany operator wywołania funkcji `operator()`.
- ▶ Obiekty funkcyjne są obiektami działającymi jak funkcje.
- ▶ Zalety obiektów funkcyjnych:
 - ▶ posiadają stan (pamięć),
 - ▶ mają własny typ (mogą być parametrami szablonów),
 - ▶ działają co najmniej tak szybko jak wskaźniki do funkcji.

Obiekt funkcyjny jako kryterium sortowania

```
class Person {
public:
    string firstname() const;
    string lastname() const;
    ...
};

struct PersonSortCriterion {
    bool operator() (const Person &p1, const Person &p2) const {
        return p1.lastname()<p2.lastname() or
            p1.lastname()==p2.lastname() and p1.firstname()<p2.firstname();
    }
};

...

set<Person, PersonSortCriterion> coll;
```

Funktory i predykaty

- ▶ **Funktor** to obiekt klasy z operatorem wywołania funkcji.
- ▶ **Predykat** to funktor, który wyniku zwraca wartość boolowską.
- ▶ Obiekt funkcji łączący dwa obiekty funkcyjne nosi nazwę **adaptatora funktorów**.

Obiekt funkcyjny ze stanem wewnętrznym

```
class IntSequence {  
private:  
    int value;  
public:  
    // konstruktor  
    IntSequence (int init = 0) : value(init) {}  
    // operator wywołania funkcji  
    int operator() () {  
        return ++value;  
    }  
};
```

Algorytm `for_each`

- ▶ Algorytm `for_each` aplikuje funkcje zdefiniowaną w obiekcie funkcyjnym do wszystkich elementów kolekcji.
- ▶ Algorytm `for_each` zwraca swój obiekt funkcyjny.
- ▶ Stan danego obiektu funkcyjnego możemy więc sprawdzić, analizując wartość zwróconą przez algorytm `for_each`.

Algoritm for_each

```
class MeanValue {
    int num = 0; // number of elements
    long sum = 0; // sum of all element values
public:
    // MeanValue() : num(0), sum(0) {}
    void operator() (int elem) {
        ++num; // increment count
        sum += elem; // add value
    }
    double value () {
        return static_cast<double>(sum) / num;
    }
};

...
vector<int> coll = /* ... */;
MeanValue mv =
    for_each(coll.begin(), coll.end(), MeanValue());
cout << mv.value() << endl;
```

Algorytm `for_each` dla funktora 1-argumentowego

```
template<typename elementType>
struct DisplayElement {
    void operator () (const elementType &element) const {
        cout << element << ' ';
    }
};

...
vector<int> vec;
...
for_each (vec.begin(), vec.end(), DisplayElement<int>());
cout << endl;
```

Predykaty

- ▶ Predykaty są to funkcje lub obiekty funkcyjne zwracające wartość boolowską albo wartość, którą można niejawnie przekonwertować na typ `bool`.
- ▶ Predykaty są często wewnętrznie kopiowane przez algorytmy STL - dlatego predykat powinien być bezstanowy (predykat nie powinien zmieniać swojego stanu w wyniku wywołania, a kopia predykatu powinna posiadać ten sam stan co oryginał).
- ▶ W przypadku lambda problem ten nie występuje, a to dzięki możliwości współdzielenia stanu pomiędzy wszystkimi kopiami obiektu funkcyjnego.

Predefiniowane obiekty funkcyjne

- ▶ Stosowanie predefiniowanych obiektów funkcyjnych wymaga włączenia pliku nagłówkowego `<functional>`.
- ▶ Arytmetyczne obiekty funkcyjne: `negate<>`, `plus<>`, `minus<>`, `multiplies<>`, `divides<>`, `modulus<>`.
- ▶ Obiekty funkcyjne porównujące: `less<>` (domyślne kryterium przy sortowaniu czy wyszukiwaniu binarnym), `greater<>`, `less_equal<>`, `greater_equal<>`, `equal_to<>`, `not_equal_to<>`.
- ▶ Obiekty funkcyjne do tworzenia wyrażeń logicznych: `logical_not<>`, `logical_and<>`, `logical_or<>`.
- ▶ Obiekty funkcyjne używające operatorów bitowych: `bit_not<>`, `bit_and<>`, `bit_xor<>`, `bit_or<>`.

Adaptator bind()

- ▶ Adaptator funkcji jest to obiekt funkcyjny, który umożliwia składanie obiektów funkcyjnych ze sobą nawzajem, z określonymi wartościami lub ze specjalnymi funkcjami.
- ▶ Adaptator wiązania argumentów `bind()` pozwala na:
 - ▶ adaptację i kompozycję nowych obiektów funkcyjnych z istniejących i predefiniowanych obiektów funkcyjnych;
 - ▶ wywoływanie funkcji globalnych;
 - ▶ wywoływanie funkcji składowych na rzecz obiektów, wskaźników do obiektów i inteligentnych wskaźników do obiektów.
- ▶ Argumenty przekazane do wywołania obiektu wiążącego są w wyrażeniu wiążącym widoczne jako symbole zastępcze `std::placeholders::_1`, `std::placeholders::_2` itd.

Adaptator bind()

```
auto plus10 = bind(  
    plus<int>(),  
    std::placeholders::_1,  
    10);  
cout << "+10: " << plus10(7) << endl;
```

```
auto inversDiv = bind(  
    divides<double>(),  
    std::placeholders::_2,  
    std::placeholders::_1);  
cout << "invdiv: " << inversDiv(49,7) << endl;
```

Rachunek lambda

- ▶ Nazwa wywodzi się od *rachunku lambda* stworzonego przez Alonzo Churcha w 1932 roku, gdzie symbol greckiej litery λ oznaczał wszystko co można wywołać przez funkcje.
- ▶ Rachunek lambda okazał się być modelem obliczeń równoważnym maszynie Turinga.
- ▶ Rachunek lambda bez typów stanowił inspirację dla powstania programowania funkcyjnego. Rachunek lambda z typami jest podstawą dzisiejszych systemów typów w językach programowania.

Wyrażenia lambda w C++

- ▶ Wyrażenia lambda zostały po raz pierwszy wprowadzone w standardzie C++11.
- ▶ Wyrażenia lambda w języku C++ są takimi anonimowymi obiektami funkcyjnymi - są podobne do zwykłych funkcji i można je pamiętać w zmiennych albo przekazywać do funkcji jako argumenty.
- ▶ Najczęściej wyrażenie lambda pozwala zdefiniować anonimową funkcję, a dokładniej obiekt funkcyjny, w miejscu użycia.
- ▶ Wyrażenia lambda posiadają swoją treść, w którym mogą realizować jakieś obliczenia; mogą przyjmować argumenty oraz zwracać wartości.

Kopiowanie wyrażeń lambda

- ▶ Wyrażenie lambda jest takim elementem języka C++ bez którego można tworzyć oprogramowanie, jednak wymaga to dużo więcej kodu - lambdy upraszczają zapis i powodują, że kod staje się przejrzysty i czytelny.
- ▶ Funkcje lambda to anonimowe obiekty funkcyjne.
- ▶ Przykład:

```
auto f = [] (int x, int y) { return x + y; }
```
- ▶ Lambdy nie posiadają ani konstruktora domyślnego ani operatora przypisania ale posiadają konstruktor kopiujący (bez zastosowania).

Wyrażenia lambda w C++

- ▶ Lambdę można traktować jak anonimową funkcję.
- ▶ Główne zastosowanie funkcji lambda to ich użycie jako argumentu sterującego obliczeniami w zaimplementowanych algorytmach.
- ▶ Wyrażenie lambda jest używane wszędzie tam gdzie jest potrzebne jakieś kryterium (najczęściej predykatowe) w formie funkcji - główne zastosowanie to algorytmy z biblioteki STL, jak na przykład `sort` albo `find` (plik nagłówkowy `<algorithm>`).
- ▶ Wyrażenie lambda jest wygodnym sposobem definiowania anonimowego obiektu funkcyjnego w miejscu użycia.

Budowa wyrażeń lambda

- ▶ Wyrażenie lambda w C++ składa się z 5 elementów, z czego część jest opcjonalna:
 - ▶ `[]` - kwadratowe nawiasy oznaczają początek wyrażenia lambda; między te nawiasy można wpisać listę przechwytywanych nazw zewnętrznych;
 - ▶ `()` - nawiasy okrągłe, analogicznie jak przy zwykłej funkcji, zawierają argumenty, jakie ma przyjmować wyrażenie lambda (opcjonalne);
 - ▶ atrybuty wyrażenia lambda - z możliwych atrybutów najistotniejszy jest `mutable`, który sprawia że zmienne przechwycone przez wartość mogą być modyfikowane wewnątrz ciała wyrażenia (opcjonalne);
 - ▶ `-> T` - wartość typu T zwracana przez wyrażenie lambda (opcjonalne)
 - ▶ `{ }` - treść wyrażenia lambda, czyli kod do wykonania gdy wyrażenie zostanie wywołane.
- ▶ Najprostsza lambda: `[] { }`

Wyrażenia lambda

- ▶ Można utworzyć obiekt funkcyjny anonimowego typu reprezentujący lambdę :

```
auto lambda = [] (...)->...{ ... };
```

Do takiej lambda można się potem odwołać jak do funkcji:

```
lambda (...);
```

- ▶ Przykład:

```
vector<int> v {9, 4, 1, 6, 8};
```

```
bool sensitive = true;
```

```
// ...
```

```
auto lambda =
```

```
    [sensitive] (int x, int y)
```

```
    { return sensitive ? x<y : abs(x) < abs(y);
```

```
    }
```

```
// ...
```

```
sort(v.begin(), v.end(), lambda);
```

Proste przykłady wyrażeń lambda

- ▶ Przykład 1: sześćcian liczby typu int (zmienna funkcyjna)

```
auto kw = [](int x){ return x * x * x; };
```

...

```
cout << kw(5) << endl;
```

- ▶ Przykład 2: kwadrat liczby typu int (wywołanie funkcji)

```
cout << [](int x){ return x * x; }(7)
```

```
<< endl;
```

Przechwytywanie nazw w wyrażeniach lambda

- ▶ Dostęp do lokalnych zmiennych lub pól w obiekcie określa się w funkcji lambda za pomocą domknięcia, czyli wewnątrz początkowych nawiasów kwadratowych `[]` na początku definicji.
- ▶ Domknięcie puste `[]` oznacza, że funkcja lambda nie potrzebuje dostępu do zmiennych z lokalnego środowiska (zdefiniowanych poza funkcją lambda).
- ▶ Domknięcie `[&]` oznacza, że wszystkie zmienne z lokalnego środowiska są dostępne przez referencję.
- ▶ Domknięcie `[=]` oznacza, że wszystkie zmienne z lokalnego środowiska są dostępne przez wartość (kopiowanie wartości następuje w miejscach, w których funkcja lambda odwołuje się do zewnętrznych zmiennych); nie wolno zmieniać wartości skopiowanych zmiennych.
- ▶ Zmienne przechwycone przez wartość są stałe, chyba że lambda została utworzona z atrybutem `mutable`.
- ▶ W domknięciu można umieścić listę zmiennych zewnętrznych, z których funkcja lambda może korzystać, na przykład:

```
int x, y;  
// ...  
[x, &y] (...) { return ...; }
```

Typ wyniku w wyrażeniu lambda

- ▶ Funkcja lambda określa typ zwracanego wyniku za pomocą frazy `-> TYP`.

- ▶ Przykład:

```
[](int x, int y) -> int
    { int z = x * x; return z + y + 1; }
```

- ▶ Jeśli ciało funkcji lambda składa się z jednej instrukcji `return`, to typ zwracanego wyniku będzie wydedukowany za pomocą `decltype()` (możne wtedy pominąć frazę `-> TYP`).

- ▶ Przykład:

```
[](int x, int y) // -> decltype(x*x+y+1)
    { return x * x + y + 1; }
```

Szablon `function<>`

- ▶ W pliku nagłówkowym `<functional>` jest zdefiniowany szablon klasy `function<>` - uniwersalne polimorficzne opakowanie dla funkcji.
- ▶ Instancja klasy `function<>` może przechowywać i kopiować dowolny obiekt konstruowany przez kopiowanie oraz uruchamiać zdefiniowaną funkcjonalność (funkcje, wyrażenia lambda, wyrażenia zbindowane lub inne obiekty funkcyjne).
- ▶ Lambdę można umieścić w obiekcie klasy `function<>`.
- ▶ Przykład:

```
function<int(int, int)> f =  
    [] (int x, int y) { return x * y; }
```

Szablon `function<>`

```
#include <functional>    // std::function, std::negate

// a function:
int half(int x) {return x/2;}

// a function object class:
struct third_t {
    int operator()(int x) {return x/3;}
};

...

std::function<int(int)> fn1 = half;           // function
std::function<int(int)> fn2 = &half;         // function pointer
std::function<int(int)> fn3 = third_t();     // function object
std::function<int(int)> fn4 = [](int x){return x/4;}; // lambda expression
std::function<int(int)> fn5 = std::negate<int>(); // standard function object
```

Rekurencyjne lambdy w C++

- ▶ Aby można było zrobić wywołanie rekurencyjne w lambdzie, należy w liście przechwytywanych nazw umieścić nazwę lambdy z referencją.

- ▶ Przykład:

```
function<void (int)> helloworld =  
    [&helloworld] (int count) {  
        cout << "Hello world" << endl;  
        if (count > 1) helloworld(count - 1);  
    };
```

Rekurencyjne lambdy w C++

- ▶ Niestety, nie możemy przechwycić zmiennej zadeklarowanej za pomocą `auto` w jej własnej inicjalizacji. Przykład:

```
auto fun = [&fun] (int x) -> int {  
    if (x == 0 or x == 1) return 1;  
    else return fib(x - 1) + fib(x - 2);  
}
```

- ▶ Lambdy w C++ są unikatowe i nie mają nazwy, więc problemem jest odwołanie się do funktora, który kompilator właśnie tworzy.
- ▶ Nie możemy również użyć słowa kluczowego `this` wewnątrz treści lambdy.

Rekurencyjne lambdy w C++

- ▶ Aby przechwycić zmienną zadeklarowaną za pomocą `auto` można przekazać jako parametr referencję do lambdy.

Przykład:

```
auto fib = [] (int x, const auto &f) -> int {  
    if(x == 0 || x == 1) return 1;  
    else return f(x - 1, f) + f(x - 2, f);  
};  
...  
fib(12, fib);
```

- ▶ Kod powyższy jest brzydki, ale się kompiluje. Wywołanie lambdy musi niestety zawierać dodatkowy parametr (referencję do samej lambdy).

Rekurencyjne lambdy w C++

- ▶ Aby ukryć odwołanie do tej samej lambdy można zrobić proste opakowanie (lambda w lambdzie). Przykład:

```
auto fib = [] (int64_t x) {  
    auto fi = [] (int x, const auto &f)->int {  
        if(x == 0 || x == 1) return x;  
        else return f(x - 1, f) + f(x - 2, f);  
    };  
    return fi(x, fi);  
};  
...  
fib(12);
```

- ▶ Niestety powyższy kod nadal wygląda brzydko, ale wywołanie lambdy jest dużo ładniejsze (bez dodatkowych parametrów).

Literatura [pl]

- ▶ Wyrażenia lambda C++
<https://binarnie.pl/wyrazenia-lambda-c/>
- ▶ Wyrażenie lambda λ w C++
<https://blog.artmetic.pl/wyrazenie-lambda-%CE%BB-w-c/>
- ▶ Wyrażenia lambda - użyteczna nowość C++11
<https://www.kompikownia.pl/index.php/2018/12/15/wyrazenia-lambda-uzyteczna-nowosc-c11/>
- ▶ Wyrażenia lambda C++11
<https://cpp0x.pl/kursy/Kurs-C++/Poziom-5/Wyrazenia-lambda-C++11/591>