



C++17 i STL

Obiekty funkcyjne i lambdy

Funkcje jako argumenty algorytmów

- ▶ Niektóre algorytmy umożliwiają przekazywanie funkcji pomocniczych zdefiniowanych przez użytkownika – funkcje te są następnie wewnętrznie wywoływane przez te algorytmy.
- ▶ Najpopularniejszym przykładem jest algorytm `for_each()`, wywołujący funkcję zdefiniowaną przez użytkownika wobec każdego elementu z podanego zakresu.

- ▶ Przykład:

```
// funkcja wypisująca przekazany argument
void print (int elem) {
    cout << elem << ' ';
}
int main() {
    vector<int> coll;
    // wstaw elementy od 1 do 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }
    // wypisz wszystkie elementy
    for_each(coll.cbegin(), coll.cend(), print);
    cout << endl;
}
```

Funkcje jako argumenty algorytmów

- ▶ Wykorzystanie argumentów funkcyjnych:
 - ▶ kryterium wyszukiwania,
 - ▶ kryterium sortowania,
 - ▶ definicja operacji, która ma być wykonana na elementach kolekcji.

- ▶ Przykład:

```
int square (int value) {
    return value*value;
}
int main() {
    set<int> coll1;
    vector<int> coll2;
    // wstaw do kolekcji coll1 elementy od 1 do 9
    for (int i=1; i<=9; ++i) {
        coll1.insert(i);
    }
    PRINT_ELEMENTS(coll1,"wartości początkowe: ");
    // transformuj każdy element z kolekcji coll1 do coll2
    transform (coll1.cbegin(),coll1.cend(),
               back_inserter(coll2),
               square);
    PRINT_ELEMENTS(coll2,"podniesione do kwadratu: ");
}
```



Predykaty



- ▶ Predykat (ang. *predicate*) jest funkcją zwracającą wartość boolowską.
- ▶ Predykaty są często używane do określenia kryterium sortowania lub wyszukiwania.
- ▶ W zależności od przeznaczenia predykaty mogą być jedno- lub dwuargumentowe.
- ▶ Biblioteka *STL* wymaga, aby predykaty były bezstanowe, to znaczy dla tej samej wartości zawsze zwracały ten sam wynik – wyklucza to funkcje, które w trakcie wywołania modyfikują swój wewnętrzny stan.

Preedykaty jednoargumentowe

- ▶ Preedykaty jednoargumentowe sprawdzają określoną właściwość pojedynczego argumentu.
- ▶ Typowy przykład stanowi funkcja wykorzystywana jako kryterium wyszukiwania do znalezienia pierwszej liczby pierwszej:

```
bool isPrime (int number) {  
    // liczby 0, 1 i ujemne nie są pierwsze  
    if (number <= 1)  
        return false;  
    // znajdź dzielnik  
    for (int div = 2; div * div <= number; div++)  
        if (number % div == 0)  
            return false; // jest dzielnik  
    return true; // nie ma dzielnika  
}  
int main() {  
    list<int> coll;  
    ...  
    auto pos = find_if (coll.cbegin(), coll.cend(),  
                        isPrime);  
    if (pos != coll.end()) // znaleziono  
        cout << *pos << " to liczba pierwsza" << endl;  
    else // nie znaleziono  
        cout << "nie znaleziono liczby pierwszej" << endl;  
}
```


Predykaty dwuargumentowe

- ▶ Predykaty dwuargumentowe porównują określoną właściwość dwóch argumentów.
- ▶ Aby na przykład posortować elementy zgodnie z własnym kryterium sortowania, możemy podać własną funkcję predykatową:

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

bool sortCriterion (const Person& p1, const Person& p2) {
    // osoba jest 'mniejsza' niż druga osoba jeśli
    // - nazwisko jest 'mniejsze'
    // - nazwisko jest 'równe' oraz imię jest 'mniejsze'
    return p1.lastname() < p2.lastname() ||
        (p1.lastname() == p2.lastname() &&
         p1.firstname() < p2.firstname());
}

int main() {
    deque<Person> coll;
    ...
    sort(coll.cbegin(), coll.cend(), // zakres
          sortCriterion); // kryterium sortowania
}
```




Obiekty funkcyjne

- ▶ Argumenty funkcyjne algorytmów nie muszą być funkcjami – mogą to być również obiekty, które zachowują się jak funkcje.
- ▶ Taki obiekt nazywamy **obiektem funkcyjnym** (ang. *function object*) lub inaczej **funktorem**.
- ▶ Możemy zdefiniować obiekt funkcyjny jako obiekt klasy udostępniającej operator wywołania funkcji.
- ▶ Było to możliwe również przed C++11.



Obiekty funkcyjne

- ▶ Obiekty funkcyjne stanowią kolejny przykład możliwości programowania ogólnego i koncepcji czystej abstrakcji – można powiedzieć, że funkcją jest wszystko to, co zachowuje się jak funkcja (jeśli więc zdefiniujemy obiekt zachowujący się jak funkcja, będzie on mógł zostać użyty jako funkcja).
- ▶ Zachowanie funkcyjne jest czymś, co można wywołać z wykorzystaniem nawiasów zwykłych i przekazaniem argumentów.
- ▶ Wszystko, co musimy zrobić, to zdefiniować operator() z odpowiednimi typami parametrów.
- ▶ Standard C++ używa pojęcia obiektu funkcyjnego w odniesieniu do wszystkich obiektów, których da się użyć jako funkcji – pojęcie to obejmuje więc wskaźniki do funkcji, obiekty klas przeciążających operator (), obiekty klas definiujących konwersję na wskaźnik do funkcji oraz lambdy.



Obiekty funkcyjne

- ▶ Obiekty funkcyjne są jednak czymś więcej niż funkcje:
 - ▶ Obiekty funkcyjne są jakby funkcjami ze stanem.
 - ▶ Obiekty funkcyjne mogą zawierać inne funkcje składowe oraz atrybuty. Oznacza to, że obiekty funkcyjne posiadają stan.
 - ▶ Inną zaletę obiektów funkcyjnych stanowi możliwość ich inicjalizacji podczas wykonywania przed ich użyciem (wywołaniem).
 - ▶ Każdy obiekt funkcyjny posiada swój własny typ.
 - ▶ Zwykle funkcje posiadają różne typy tylko wtedy, gdy różne są ich sygnatury. Obiekty funkcyjne natomiast mogą posiadać różne typy nawet w przypadku, gdy ich sygnatury są takie same. Fakt ten stanowi istotne usprawnienie z punktu widzenia programowania ogólnego wykorzystującego szablony, ponieważ pozwala na przekazanie zachowania funkcyjnego jako parametru szablonu.
 - ▶ Istnieje również możliwość zaprojektowania hierarchii obiektów funkcyjnych tak, żeby na przykład utworzyć różne specjalne warianty jednego ogólnego kryterium.
 - ▶ Obiekty funkcyjne działają zwykle szybciej od zwykłych funkcji.
 - ▶ Koncepcja szablonów pozwala zwykle na lepszą optymalizację, ponieważ na etapie kompilacji zdefiniowanych może być więcej szczegółów. Przekazywanie obiektów funkcyjnych zamiast zwykłych funkcji daje więc często lepszą wydajność.

Obiekty funkcyjne – przykłady

- ▶ Załóżmy, że do wszystkich elementów kolekcji chcemy dodać określoną wartość. Jeśli na etapie kompilacji znamy wartość, którą chcemy dodać, możemy użyć zwykłej funkcji:

```
void add10 (int& elem) {  
    elem += 10;  
}  
...  
for_each (coll.begin(), coll.end(), add10);
```

Obiekty funkcyjne – przykłady

- ▶ Jeśli potrzebujemy różnych wartości znanych na etapie kompilacji, w zamian możemy użyć szablonu:

```
template <int theValue>
void add (int& elem) {
    elem += theValue;
}
for_each (coll.begin(), coll.end(), add<36>);
```

Obiekty funkcyjne – przykłady

- ▶ Ponieważ obiekt może posiadać stan, można go zainicjalizować prawidłową wartością:

```
class AddValue {
private:
    int theValue; // wartość do dodania
public:
    // konstruktor
    AddValue(int v) : theValue(v) { }
    // operator wywołania funkcji
    void operator() (int& elem) const {
        elem += theValue;
    }
};
...
for_each (coll.begin(), coll.end(), AddValue(36));
```

Stosowanie lambd

- ▶ **Lambdy** są anonimowymi obiektami funkcyjnymi i definiują sposób dookreślania zachowania wewnątrz wyrażenia albo instrukcji – skutkiem tego można definiować obiekty reprezentujące zachowanie funkcyjne i przekazywać te obiekty jako tworzone w miejscu wywołania algorytmów predykaty albo inne dookreślenia.
- ▶ Na przykład w poniższej instrukcji:

```
// przekształć wszystkie wartości na sześciiany  
transform (coll.begin(), coll.end(), // źródło  
          coll.begin(), // przeznaczenie  
          // lambda jako obiekt funkcyjny  
          [] (double d) { return d * d * d; }  
);
```
- ▶ Lambdy nie posiadają ani konstruktora domyślnego, ani operatora przypisania.

Stosowanie lambd – zalety

- ▶ Zamierzamy wyszukać w kolekcji pierwszy element o wartości z zakresu od x do y:

```
deque<int> coll = { 1, 3, 19, 5, 13, 7, 11, 2, 17 };  
int x = 5;  
int y = 12;  
auto pos = find_if (  
    coll.cbegin(), coll.cend(), // zakres  
    [=] (int i) { return i > x && i < y; } // kryterium  
);  
cout << "pierwszy element >5 i <12: " << *pos << endl;
```
- ▶ Określenie wciąganych symboli zewnętrznych poprzez [=] to dla kompilatora informacja, że w ciele lambdy mają być widoczne wszystkie symbole bieżącego zasięgu i że będą tam dostępne przez wartość.
- ▶ Użycie [&] oznaczałoby, że symbole bieżącego zasięgu będą dostępne przez referencję, co pozwalałoby na ich modyfikowanie z wnętrza lambdy.

Stosowanie lambd – zalety

- ▶ W starym C++ zamiast predykatorowej lambdy można było stosować funkcje:

```
bool pred (int i) {  
    return i > x && i < y;  
}
```

...

```
pos = find_if (coll.begin(), coll.end(),  
             pred);
```

- ▶ W nowym C++11 eliminujemy problem dookreślenia zachowania algorytmu poza miejscem użycia algorytmu.

Stosowanie lambda – zalety

- ▶ W starym C++ zamiast predykatorowej lambda można było stosować obiekty funkcyjne:

```
class Pred
{
    int x, y;
public:
    Pred (int xx, int yy) : x(xx), y(yy) { }
    bool operator() (int i) const {
        return i > x && i < y;
    }
};

...
pos = find_if (coll.begin(), coll.end(),
              Pred(x, y)
            );
```

- ▶ W starym C++ dostęp do parametrów sterujących zachowaniem (x i y) jest doprawdy uciążliwy i nieelegancki.

Stosowanie lambda – zalety

- ▶ W starym C++ zamiast predykatorowej lambda można było stosować obiekty wiążące:

```
pos = find_if (coll.begin(), coll.end(),
              bind(logical_and<bool>(),
                  bind(greater<int>(), _1, x),
                  bind(less<int>(), _1, y)
                  )
              );
```

- ▶ W starym C++ taki zapis zachowania jest zdecydowanie nieczytelny.

Stosowanie lambda – kryteria sortowania

- ▶ W ramach kolejnego przykładu użyjemy wyrażenia lambda do określenia kryterium sortowania kontenera elementów typu Person:

```
class Person {
public:
    string firstname() const;
    string lastname() const;
    ...
};
...
deque<Person> coll;
...
// posortuj kolekcję coll według nazwiska i imienia:
sort(coll.begin(), coll.end(), // zakres przeszukiwany
    [] (const Person &p1, const Person &p2) {
        return p1.lastname() < p2.lastname() ||
            (p1.lastname() == p2.lastname() &&
             p1.firstname() < p2.firstname());
    }
);
```

Stosowanie lambd – ograniczenia

- ▶ Weźmy za przykład lambdę określającą kryterium sortowania dla kontenerów asocjacyjnych:

```
auto cmp = [] (const Person& p1, const Person& p2)
{
    return p1.lastname() < p2.lastname() ||
        (p1.lastname() == p2.lastname() &&
         p1.firstname() < p2.firstname());
};
...
set<Person, decltype(cmp)> coll(cmp);
```

- ▶ Ponieważ w deklaracji kontenera set trzeba podać kryterium sortowania, musimy uciec się do użycia konstrukcji decltype, zwracającego typ obiektu lambda; obiekt lambda musi też być przekazany do konstruktora kontenera coll.
- ▶ Inny problem z lambdą polega na braku możliwości posiadania wewnętrznego stanu, zachowywanego pomiędzy wywołaniami lambda. Jeśli taki stan jest potrzebny, trzeba przechowywać go w zmiennej zewnętrznej, zadeklarowanej w zasięgu, w którym definiowana jest lambda, i wciągniętej do lambda przez referencję.