Paradygmaty programowania

Programowanie generyczne w C++

Dr inż. Andrzej Grosser

Częstochowa, 2016

Spis treści

1. Zadanie 3

1.1. Wprowadzenie

Biblioteka algorytmów pozwala na parametryzowanie algorytmów za pomocą funktorów. Rolę funktora może pełnić wskaźnik do funkcji, wyrażenie lambda oraz klasa z przeładowanym operatorem nawiasów okrągłych.

1.2. Obiekty funkcyjne

1.2.1. Wyrażenia lambda

Wyrażenia lambda dają możliwość utworzenia domknięcia: obiektu anonimowej funkcji, która ma możliwość przechwytywania zmiennych z zasięgu ją otaczającego.

Podstawowa składnia wyrażeń lambda jest nieskomplikowana:

```
[lista-przechwytywanych-zmiennych](parametry)
{ciało-wyrażenia-lambda}
```

W nawiasach kwadratowych są przekazywane zmienne z zasięgu otaczającego wyrażenie lambda, dzięki tej konstrukcji jest możliwe użycie zmiennych wewnątrz ciała wyrażenia lambda. Zmienne mogą być przechwytywane przez wartość (sama nazwa zmiennej) lub przez referencję (nazwa zmienne poprzedzona `&`). Oprócz wybranych zmiennych można wciągnąć do wyrażenia lambda wszystkie zmienne przez referencję lub przez kopiowanie. Na przykład:

- `[x,y](){}` - do wyrażenia lambda przekazywane są kopie zmiennych x i y, nie można ich modyfikować,
- `[&x,&y](){}` - do wyrażenia lambda przekazywane są zmienne x i y za pomocą referencji, można te zmienne modyfikować,
- `[x,&y](){}` - do wyrażenia lambda przekazywane są zmienne x i y, x jest kopiowane, zaś y przekazywane za pomocą referencji,
- `[&](){}` - przechwytuje wszystkie zmienne z zasięgu otaczającego przez referencje,

- `[=](){}` - kopiuje wszystkie zmienne z zasięgu otaczającego,
- `[](){}` - nic nie jest przekazywane do wyrażenia lambda.

Parametry oznaczają parametry wywołania funkcji. W tej formie wyrażenie lambda może zawierać jedynie prostą instrukcję powrotu `return`(C++11). Na podstawie wyrażenia używanego z return kompilator jest w stanie wywnioskować typ wartości zwracanej z funkcji.

Na podstawie tego krótkiego wprowadzenia można przedstawić kilka wyrażeń lambda:

```
auto fun = [](double x, double y){return x > y;};
std::cout << std::boolalpha << fun(4, 6) << std::endl;
```

W ten sposób utworzono wyrażenie lambda, które nie przechwytuje zmiennych z zewnątrz (pusta lista zmiennych), oczekuje na wejściu dwóch parametrów x i y i zwraca wynik ich porównania (bool).

```
double a = 5;
auto fun2 = [a](double x){return x > a;};
std::cout << std::boolalpha << fun2(6) << std::endl;
```

Z kolei w tym przypadku do zasięgu wyrażenia lambda przekazywana jest zmienna a, która jest porównywana z przekazywanym do funktora parametrem.

Typ wartości zwracanej W niektórych sytuacjach kompilator nie jest w stanie wywnioskować typu wartości zwracanej, dlatego trzeba go wtedy jawnie poinformować, jaki będzie typ wartości zwracanej. Zapis jest następujący:

```
[lista-przechwytywanych-zmiennych](parametry) -> typ
{ciało-wyrażenia-lambda}
```

Na przykład:

```
double tab[] = {1,2,3};
auto fun2 = [tab]() -> double {
    double sum = 0.0;
    for(auto elem : tab)
        sum += elem;
    return sum;
};
std::cout << fun2() << std::endl;
```

1.2.2. Klasy funktorów

Oprócz wyrażeń lambda język C++ pozwala na zdefiniowanie funktorów, klas, które zachowują się jak funkcja. Składnia jest następująca:

```
class Pred{
    std::string nazwiskoPoszukiwane;
public:
    Pred(std::string naz):nazwiskoPoszukiwane(naz){}
    bool operator()(const Osoba& o)
    {
        return o.nazwisko == nazwiskoPoszukiwane;
    }
};
//...
Osoba osoba;
//Konstruktor funktora
Pred pred("Nocul");
//Operator funkcyjny
std::cout << pred(osoba);
```

To, że definiujemy funktor pokazuje słowo kluczowe `operator` z następującymi po nim nawiasami okrągłymi () wewnątrz nich nie wpisuje się nazwy argumentów, parametry wpisuje się w drugiej parze nawiasów okrągłych. Klasy funktorów mogą zawierać dane, funkcje składowe (szczególnie używane są konstruktory).

Zwróćcie uwagę na dwa aspekty wykorzystania funktora - jego utworzenie z zainicjowaniem za pomocą konstruktora i jego wykorzystanie (wywołanie operatora funkcyjnego). Ma to szczególne znaczenie, przy ich wykorzystywaniu, gdy wywołanie funktora będzie zaszyte wewnątrz szablony, jedyne co jest widoczne to utworzenie funktora. Będzie to lepiej widoczne na następującym przykładzie:

```
class Generator {
public:
    Generator(int licz) : liczba(licz)
    {}
    int operator()() {
        return liczba;
    }
private:
    int liczba;
};
```

```
//Można wyobrazić sobie następujący szablon
template<typename Iter, typename Gen>
void generate(Iter b, Iter e, Gen g) {
   for(;b!=e;++b)
//Użycie operatora funkcyjnego
      *b = g();
}
//...
int[10] tab;
//Tu jest wołany konstruktor
Gen g(12);
generate(tab, tab + 10, g);
//To samo, ale z użyciem anonimowego obiektu
generate(tab, tab+10, Gen(12));
```

Obiekt klasy Generator jest przekazywany do szablonu funkcji generate - jest zainicjowany konstruktorem. Operator funkcyjny jest wykorzystywany wewnątrz szablonu (co robi ten funktor?). Ten aspekt wykorzystania obiektu funkcyjnego jest niewidoczny w przypadku korzystania z funkcji bibliotecznych.

1.3. Algorytmy

Jak wspomniano funktory są wykorzystywane prze algorytmy ogólne z biblioteki STL. Algorytmy ogólne to szablony funkcji przygotowane dla wielu zadań takich jak sortowanie, wyszukiwanie wartości itd. Oczywiście różne algorytmy mają różne wymagania odnośnie funktorów - co mają zwracać jakie wartości pobierać, na przykład (przykłady z wyrażeniami lambda):

- Jeśli coś generujemy to funktor nie powinien pobierać żadnych argumentów i zwracać generowany obiekt.
- Funktory używane do wyszukiwania elementów powinny zwracać wartość logiczną (bool) oraz pobierać jeden argument.

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main() {
```

```
    vector<double> vec = {1,2,3,4,5};
    //Znajduje iterator do pierwszego elementu spełniającego
       predykat
    auto iter = find_if(vec.begin(), vec.end(),
        [](double x) {return x > 4;});
    cout << *iter;
}
```

– Funktory używane do nadawania porządku elementom (np. w przypadku sortowania) powinny zwracać wartość logiczną i pobierać dwa argumenty, które chcemy porównywać.

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main() {
    vector<double> vec = {1,2,3,4,5};
    //Sortuje elementy wektora w porządku malejącycm
    sort(vec.begin(), vec.end(),
        [](double x, double y) {return x > y;});
}
```

1.4. Wskazówki do zadania

Wektor vector<T> to odpowiednik tablicy, lista list<T> to lista dwukierunkowa. Zbiór set<T> to struktura uporządkowana przechowująca klucze. Początki sekwencji jak widać na podstawie załączonych przykładów są pokazywane za pomocą iteratorów (dla zwykłej tablicy wystarczają wskaźniki na początek i pierwszy element tablicy - tak jak we wprowadzeniu) - są one uzyskiwane za pomocą metod begin() i end().

Klasa `ostream_iterator` wymaga przeciążenia operatora wysyłania do strumienia - inaczej dostaniecie kilka ekranów błędów (kompilator stara się być pomocny i próbuje ustalić co mógłby programista użyć w danym momencie). Oczywiście przeciążenia operatora jest wymagane dla klas i struktur zdefiniowanych przez użytkownika.

W celu użycia klas i szablonów należy dołączyć nagłówki, odpowiednio:

– `vector<T> - <vector>`,

– `list<T> - <list>`,

– `set<T> - <set>`,

– `sort,remove_if,generate - <algorithm>`,

– `ostream_iterator<T> - <iterator>`,

Zwrócić uwagę na różnice pomiędzy przekazywaniem funktora jako parametru szablonu oraz jako parametru dla funkcji. Opisy algorytmów http:

– `http://www.sgi.com/tech/stl/`,

– `http://www.cplusplus.com/reference/`.