Kurs języka C++

14. Obiekty funkcyjne i lambdy

Spis treści

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`.

Algorytm 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 lambd 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;
```

Wyrażenia lambda

- Wyrażenie lambda jest wygodnym sposobem definiowania anonimowego obiektu funkcji w miejscu, w którym jest wywoływana lub przenoszona jako argument do funkcji.
- Wyrażenia lambda są używane do hermetyzacji kilku wierszy kodu, które są przesyłane do algorytmów lub metod.
- Najprostsza lambda: `[]{}`

Wyrażenia lambda

- Programista często chciałby zdefiniować predykatowe funkcje w pobliżu wywołań takich funkcji, jak na przykład pochodzących ze standardowej biblioteki `<algorithm>` (szczególnie `sort` i `find`) - oczywistym rozwiązaniem jest zdefiniowanie w takim miejscu funkcji lambda (określanej też jako lambda-wyrażenie).
- Funkcje lambda to anonimowe obiekty funkcyjne.
- Lambdy nie posiadają ani konstruktora domyślnego ani operatora przypisania.
- Główne zastosowanie funkcji lambda to ich użycie jako argumentu sterującego obliczeniami w innych funkcjach.
- Przykład:

```
auto f = [](int x, int y) { return x + y; }
```

Wyrażenia 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; }
```

Wyrażenia 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.
- W domknięciu można umieścić listę zmiennych zewnętrznych, z których funkcja lambda może korzystać, na przykład:

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

Wyrażenia lambda

- Można utworzyć obiekt funkcyjny anonimowego typu reprezentujący lambdę :

```
auto lambda = [](…)->…{ … };
```

Do takiej lambdy 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);
```

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