Funktory i lambdy

Funktory

Funktor (obiekt funkcyjny) rozszerza pojęcie funkcji. Utworzenie funktora możliwe jest dzięki zdefiniowaniu operatora wywołania - czyli nawiasów () w pewnej klasie.
Dzięki temu możemy stworzyć obiekt, który może zachowywać się jak normalna funkcja.

```
class X
{
public:
    void operator()(string str)
    {
        std::cout << "Wywolanie funktora X z parametrem " << str << std::endl;
    }
};

int main()
{
    X foo;
    foo("Hi!");
}
```

Używanie funktorów jest bardzo wygodne w wielu algorytmach biblioteki STL, gdyż często potrzebują one funkcji powównujących (komparatory), decydujących o spełnieniu jakiegoś warunku (predykat), i innych.

Przykład ze zwykłą funkcją:

Wywołanie funkcji w algorytmie może być bez nawiasów (podawana jest tylko jej nazwa - czyli wskaźnik do niej).

```
bool is_even(int a)
{
    return (a % 2) == 0;
}

std::vector<int> a = { 1, 2, 3, 4, 5 };
std::find_if(a.begin(), a.end(), is_even);   // zastowosowanie algorytmu z funkcją
```

Przykład z funktorem:

```
class Is_even
{
    bool operator() (int a)
    {
        return (a % 2) == 0;
    }
};

std::vector<int> a = { 1, 2, 3, 4, 5 };
std::find_if(a.begin(), a.end(), Is_even());    // zastosowanie algorytmu z funktorem
```

Funktory predefiniowane

- bit_and, bit_or, bit_xor
- logical_and, logical_or, logical_not
- greater, greater_equal, less, less_equal, not_equal_to

• divides, minus, modulus, multiplies, negate, plus
• ...

Wyrażenia lambda

Lambda to funkcja zdefiniowana w miejscu użycia, stąd wynika, że jest ona ograniczona do tego miejsca, i co za tym idzie - jednorazowa. Odpowiada jej operator trzech nawiasów: []()\{\} .

`[](){}` - pusta lambda
`[](){ return 2; }` - lambda nienazwana zwracająca liczbę
`[](int i){ return i >= 0; }` - lambda nienazwana określająca, czy parametr jest >= 0

`auto multiplyByTwo = [](int k){ return k * 2; };` - lambda nazwana i przypisanie jej zmiennej typu automatycznego
`int number = multiplyByTwo(4);` - zastosowanie lambdy

Zmienne zewnętrzne w lambdzie - użycie klauzuli przechwytywania

```
int a {6};
auto add5 = [=](int x){ return x + a; };
```

```
int counter {};
auto inc = [&counter] { counter++; };
```

```
int even_count = 0;
for_each(v.begin(), v.end(), [&even_count](int n)
{
    cout << n;
    if (n % 2 == 0)
        ++even_count;
});
cout << "W wektorze jest " << even_count << " parzystych liczb." << endl;
```

Wewnątrz nawiasów [] można umieścić elementy, które mają być widoczne dla lambdy w zakresie, w której jest zdefiniowana.
`[]` - nie może być użyta żadna zmienna z zakresu
`[&]` - widoczna jest każda zmienna w zakresie i przechwytywana będzie przez referencję
`[=]` - widoczna jest każda zmienna w zakresie i przechwytywana będzie przez wartość (kopię)
`[capture-list]` - widoczne są tylko zmienne wymienione w nawiasach [], jeśli któraś ma być przechwytywana przez referencję, jej nazwa musi być poprzedzona operatorem &

std::function

```
void print_num(int i)
{
    std::cout << i << std::endl;
}

std::function<void(int)> f_display = print_num;
f_display(-29);

std::function<void()> f_display_15 = [](){ print_num(15); };
f_display_15();
```

std::function przechowuje funkcję. W powyższym przykładzie jest to analogiczne do wywoływania funkcji po nowej, innej nazwie lub (tak jak w przypadku std::bind) do wywołania funkcji z wybranym z góry argumentem.

Deklaracja std::function:
std::function<*typ_zwracany*(*argumenty*)> nazwa = przypisana_funkcja;

Justyna Walkowiak