NAUJOS C++ KALBOS KONSTRUKCIJOS. C++11, C++14, C++17 STANDARTAI

(Tęsinys 1)

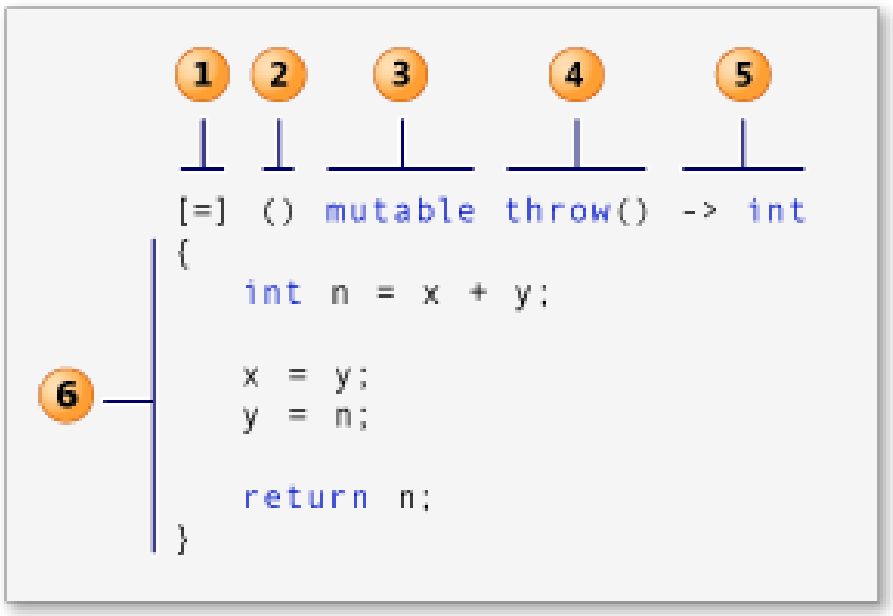
FUNKCINIO PROGRAMAVIMO ELEMENTAI C++: LAMBDA

Šie elementai įvesti nuo C++11 ir su tam tikromis modifikacijomis galioja vėlesniuose standartuose. *Lambda išraiškos* (angl. *lambda expressions*), dažnai vadinamos *lambda*, yra patogus būdas apibrėžti anoniminės funkcijos objektą (vadinamąjį *uždarymą*, angl. *closure*) tiesiai toje vietoje, kur ji yra naudojama (arba perduodama kaip funkcijos argumentas). Paprastai *lambdos* apima kelias kodo eilutes, kurios perduodamos algoritmams ar asinchroniniams metodams. ISO C++ standarte parodyta kaip naudoti *lambda* funkcijose. Čia *lambda* perduodama kaip trečiasis argumentas į STL *sort()* funkciją:

```
#include <algorithm>
#include <cmath>

void abssort(float* x, unsigned n) {
   std::sort(x, x + n,
         // Lambda expression begins
       [ ] (float a, float b) { return (std::abs(a) < std::abs(b)); }
         // end of lambda expression
   );
}
```

Lambda sintaksė yra tokia:



Čia

1. Kintamųjų fiksavimo (gavimo iš matomumo srities) sakinys, angl. *capture clause* (*lambda-introducer* in the Standard syntax)
2. Nebūtinas elementas – parametrų sąrašas, angl. *parameter list* (also known as the *lambda declarator*), optional.
3. Nebūtinas elementas –, angl. *mutable specification. Mutable* specifikacija leidžia lambda išraiškos kodui keisti kintamuosius, kurie yra gauti (tik pagal reikšmę).
4. Nebūtinas elementas – išimties išmetimo specifikacija, angl. *exception-specification*. Taip pat galima nurodyti *noexcept* išimties specifikacija, rodanti, kad lambda išraiškos neapima jokių išimčių.
5. Nebūtinas elementas –> gražinamos reikšmės tipas, angl. *trailing-return-type*.
6. Būtinas elementas – lambda kodo sekcija, angl. *lambda body*.

Lambda leidžia ne tik įvesti naujus kintamuosius savo parametrų sąraše ( ), bet taip pat pasiekti (gauti, užfiksuoti) kintamuosius iš aplinkinės matomumo srities (C++14). Pasiekti kintamuosius galima arba pagal vertę *[kint1]*, arba pagal nuorodą *[&kint2]*. Tuščias fiksavimo sakinys [ ] rodo, kad lambda išraiškos nenaudoja kintamųjų iš išorės (aplinkinės matomumo srities). Taip pat galima nurodyti vieną iš dviejų numatytų režimų: *[&]* reiškia, kad visi srities kintamieji yra prieinami pagal nuorodą, ir *[=]* reiškia, kad jie yra prieinami pagal reikšmę.

Lambda parametrų sąrašas (*lambda declarator*) iš esmės panašus į funkcijų parametrų sąrašą ir suteikia įprastą duomenų gavimo į lambda būdą:

```
auto y = [ ] (int first, int second) {  return first + second;  };
```

C++14 atveju, jei parametro tipas yra standartinis, galima naudoti tipo specifikatorių *auto*. Tai nurodys kompiliatoriui sukurti funkcijos kvietimo operatorių kaip šabloną. Kiekvienas *auto* specifikatorius parametrų sąraše atitinką atskirą parametro tipą.

```
auto y = [ ] (auto first, auto second)
{
   return first + second;
};
```

Pavyzdys 12.5. Lambda išraiškos kvietimas.

```
#include <iostream>
int main()
{
  using namespace std;
  int n = [] (int x, int y) { return x + y; }(5, 4);
  cout << n << endl;
}
```

Lambda išraiškos gali būti sudėtinės (lizdinės struktūros), t. y. talpinti savyje kitas lambda išraiškas (angl. *Nested Lambda Expressions*). Pvz. žemiau pateiktame pavyzdyje vidinė lambda išraiška yra:

```
[ ](int y) { return y * 2; }:
```

Pavyzdys 12.6. Sudėtinė (angl. *nested*) lambda išraiška.

```
// sudetine lambda
#include <iostream>

int main()
{
  using namespace std;
  //A nested lambda expression.
 int kart_2_plius_3 = [](int x) { return [](int y) { return y * 2; }(x) + 3; }(5);

cout << kart_2_plius_3 << endl;
}
```

Rezultatas: 13

Daugelis modernių programavimo kalbų palaiko aukštesnės eilės funkcijų koncepciją. Aukštesnės eilės funkcija – tai lambda išraiška, kuri naudoja kitą lambda išraišką kaip savo argumentą arba grąžina lambda išraišką (angl. *Higher-order lambda expressions*).

Naujuose C++ standartuose galima naudoti STL objektą-funkciją, kad lambda išraiška veiktų kaip aukštesnės eilės funkcija. Žemiau pateiktame pavyzdyje naudojama lambda išraiška, kuri grąžina objektą-funkciją ir lambda išraišką, kurios argumentas yra objektas-funkcija.

Pavyzdys 12.7. Sudėtinė (angl. *higher-order*) lambda išraiška.

```
// higher_order_lambda_expression.cpp
```

```
// compile with: /EHsc /W4
#include <iostream>
#include <functional>

int main()
{
    using namespace std;

    //A lambda expression that returns another lambda expression
    // that adds two numbers.
    auto addtwointegers = [ ](int x) -> function<int(int)> {
        return [=](int y) { return x + y; };
    };              //The returned lambda expression captures parameter x by value.

// A lambda expression that takes another lambda expression as its argument.
// The lambda expression applies the argument z to the function f and multiplies
// by 2.
    auto higherorder = [ ](const function<int(int)>& f, int z) {
        return f(z) * 2;
    };

    // Call the lambda expression that is bound to higherorder.
    auto answer = higherorder(addtwointegers(7), 8);

    // Print the result, which is (7+8)*2.
    cout << answer << endl;
}
```

https://docs.microsoft.com/en-us/cpp/cpp/examples-of-lambda-expressions

Išimčių naudojimas lambda išraiškose. Lambda išraiškos atitinka struktūrizuoto išimčių tvarkymo (SEH) ir C++ išimčių tvarkymo taisykles. Galima apdoroti išimtį, sugeneruotą lambda išraiškos kode arba atidėti išimties tvarkymą iki išorinio kodo vykdymo. Pavyzdyje 12.8 lambda išraiška naudojama *for_each()* funkcijoje tam, kad galima būtų užpildyti vektorinį objektą kito vektoriaus reikšmėmis. Jis naudoja *try / catch* bloką, kad galėtumėte tvarkyti neteisingą prieigą prie pirmojo vektoriaus.

Pavyzdys 12.8. Lambda išraiškos ir išimtys.

```
// eh_lambda_expression.cpp
// compile with: /EHsc /W4
```

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
   // Create a vector that contains 3 elements.
   vector<int> elements(3);

   // Create another vector that contains index values.
   vector<int> indices(3);
   indices[0] = 0;
   indices[1] = -1; //This subscript will trigger an exception.
   indices[2] = 2;

   // Use the values from the vector of index values to
   // fill the elements vector. This example uses a
   // try/catch block to handle invalid access to the
   // elements vector.
   try
   {
      for_each(indices.begin(), indices.end(), [&](int index) {
         elements.at(index) = index;
      });
   }
   catch (const out_of_range& e)
   {
      cerr << "Caught '" << e.what() << "'." << endl;
   };
}
```

Kadangi čia *try{} - catch{}* blokas apdoroja neteisingą prieigą prie pirmojo vektoriaus, bus atspausdinta:

```
Caught 'invalid vector<T> subscript'.
```