Nowoczesne Techniki Programowania
Funktory

molinari*
Wydział Fizyki i Informatyki Stosowanej
Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Listopad 2021

After learning lambdas

*GitHub

Spis treści

1 Funktory

Funktor, inaczej **obiekt funkcyjny**, to każdy obiekt, którego można użyć jak funkcji.

- wskaźniki do funkcji,
- obiekt klasy z przeładowanym operatorem (),
- wyrażenia lambda.

1.1 Wskaźnik do fukcji

```
#include <iostream>
#include <string>

int comp_int(const void * num1, const void * num2) {
    return *(const int*)num1 - *(const int *)num2;
}

int (*comp)(const void*, const void *) = comp_int; // pointer to
    function

int arr[] = {37,4,56,7,0,0,21,-1};

const int size = sizeof(arr)/sizeof(int);

qsort(arr, size, sizeof(int), comp); // using functor
for (auto num : arr) std::cout << num << " ";
std::cout << std::endl;
```

Listing 1: Kod w katalogu: Wskazniki

Output:
-1 0 0 4 7 21 37 56

1.2 Obiekt klasy z przeładowanym operatorem ()

Właściwości

- można zdefiniować blisko jego użycia (np. wewnątrz innej funkcji lub innej klasy)
- posiada stan

```
/// Converts given unsigned to binary or hexadecimal
class Convert
{
public:
    /// ctor sets default type converting to to_bin
    Convert() : _conv{&Convert::to_bin} {}
    ~Convert() {};


    // switching functions

    /// Switches converter to 'to binary' mode
    void bin()
    {
        _conv = &Convert::to_bin;
    }
    /// Switches converter to 'to hexadecimal' mode
    void hex()
    {
        _conv = &Convert::to_hex;
    }


    /// Overloaded operator() allows us to use object as function
    /// @param num Number to convert
    /// @returns String representing converted number
    std::string operator()(const unsigned num)
    {
        return (this->*_conv)(num);
    }

private:
    /// Pointer to function that actually does converting
    std::string (Convert::*_conv)(const unsigned);

    // converting functions
    std::string to_bin(const unsigned num)
    {
        std::string str{""};
        for (unsigned temp = num ; temp != 0 ; temp /= 2)
            str.insert(str.begin(), ((temp%2 == 1) ? '1' : '0'));
        return str;
    }

    std::string to_hex(const unsigned num)
    {
        std::string str{""};
        for (unsigned temp = num ; temp != 0 ; temp /= 16)
            str.insert(str.begin(), ((temp%16<10) ? static_cast<char>((
    temp%16)+48) : static_cast<char>((temp%16)+55)));
        return str;
    }

};
```

```
Convert convert;
const int unsigned = 51966;
// dec to bin
std::cout << "Using default state, decimal to binary.\n\t";
std::cout << convert(num) <<std::endl;

// dec to hex
convert.hex();
std::cout << "Switching to hex mode.\n\t";
std::cout << convert(num) <<std::endl;

// dec to bin
convert.bin();
std::cout << "Back to bin mode.\n\t";
std::cout << convert(num) <<std::endl;
```

Listing 2: Kod w katalogu: Klasa

Output:

```
Using default state, decimal to binary.
    1100101011111110
Switching to hex mode.
    CAFE
Back to bin mode.
   1100101011111110
```

1.3 Wyrażenia lambda (funkcja anonimowa)

1.3.1 Wstęp

Właściwości

- można zdefiniować bardzo blisko jego użycia
- przechwytywanie
- posiada stan

Sposób użycia:

```
[ captures ] ( params ) lambda-specifiers -> T { body }
```

[] - początek wyrażenia lambda, przechwytywanie nazw,
() - między okrągłe nawiasy podajemy argumenty jak w zwykłej funkcji, opcjonalne, atrybuty wyrażenia lambda np. `mutable`, opcjonalne,
-> T - zwracany typ, opcjonalne,
{} - analogicznie jak w zwykłych funkcjach, ciało funkcji.

```
// #include <vector> // std::vector
#include <numeric> // std::iota
#include <algorithm> // std::transform

std::vector<int> foo(10);
std::vector<int> bar(10);

std::iota(foo.begin(), foo.end(), 0); // fill vec with 0, 1, 2, ...

for (auto num : foo)
    std::cout << num << " ";
std::cout << std::endl;

// lambda returns every element of foo raised to the power of 2
std::transform(foo.begin(), foo.end(), bar.begin(), [](int x){return x*x
    ; });

for (auto num : bar)
    std::cout << num << " ";
std::cout << std::endl;
```

Listing 3: Kod w katalogu: Lambda

Output:

```
0 1 2 3 4 5 6 7 8 9
0 1 4 9 16 25 36 49 64 81
```

1.3.2 Przechwytywanie nazw

Zmienne automatyczne można przechwytywać na dwa sposoby, przez **wartość**, lub **referencję**. W pierwszym przypadku taka zmienna będzie (*prawie* zawsze) **stała**.

```
int x = 10, y = 3;
std::cout <<[ x, & y ] { return x + y; }();
```

Listing 4: Kod w katalogu: Lambda

Output:
13

Zmienna `x` jest przechwycona przez wartość, a `y` przez referencję.

Domyślne przechwytywanie:
`[=]` - wszystkie zmienne będą przechwytywane przez wartość,
`[&]` - wszystkie zmienne będą przechwytywane przez referencję,
`[=, &x]` - wszystkie zmienne będą przechwytywane przez wartość, ale `x` przez referencję,
`[&, y, z]` - wszystkie zmienne będą przechwytywane przez referencję, ale `y` i `z` przez wartość.

1.3.3 Atrybuty

Zmienne przechwytywane przez wartość są stałe. Można to zmienić używając atrybutu `mutable`.

```
int a = 5;
std::cout << [a]{a=10; return a;}(); // error
```

Listing 5: Kod w katalogu: Lambda

Output:

```
input_line_16:3:19: error: cannot assign to a variable captured by copy
  in a non-mutable lambda
std::cout << [a]{a=10; return a;}(); // error
                  ~^
```

```
int a = 5;
std::cout << [a]() mutable {a=10; return a;}();
```

Listing 6: Kod w katalogu: Lambda

Output:
10

1.3.4 Typ

Typ wyrażenia lambda jest oficjalnie nieokreślony.

```
auto lam = [](int x){return ++x;};
int x = 0;
std::cout << lam(x);
```

Listing 7: Kod w katalogu: Lambda

Output:
1

Lambda nie jest funkcją, jest funktorem i posiada stan.

```
int i = 0;
int j = 0;
auto lambda = [i, &j]()mutable
{
    ++i;
    ++j;
    std::cout << "Inside lambda: \ti = " << i << ", j = " << j << std::
    endl;
};
lambda();
lambda();
std::cout << "In main: \ti = " << i << ", j = " << j << std::endl;
```

Listing 8: Kod w katalogu: Lambda

Output:

```
Inside lambda:  i = 1, j = 1
Inside lambda:  i = 2, j = 2
In main:        i = 0, j = 2
```

```
int i = 0;
int j = 0;
auto lambda = [&, i]()mutable
{
    ++i;
    ++j;
    std::cout << "Inside lambda: \ti = " << i << ", j = " << j << std::
    endl;
};
lambda();
i = 5;
j = 5;
std::cout << "In main: \ti = " << i << ", j = " << j << std::endl;
lambda();
std::cout << "In main: \ti = " << i << ", j = " << j << std::endl;
```

Listing 9: Kod w katalogu: Lambda

Output:

```
Inside lambda:  i = 1, j = 1
In main:        i = 5, j = 5
Inside lambda:  i = 2, j = 6
In main:        i = 5, j = 6
```

Od C++14, zmienną w stanie lambdy można rówież utworzyć (należy użyć atrybutu `mutable`).

```
auto lambda = [ how_many = 0]() mutable
{
    std::cout << "Lambda called: " << ++how_many << " time(s)" << std::
    endl;
};
lambda();
lambda();
```

Listing 10: Kod w katalogu: Lambda

Output:

```
Lambda called: 1 time(s)
Lambda called: 2 time(s)
```

1.3.5 Przekazywanie do funkcji

Lambdę można przekazać do funkcji w miejsce, w którym oczekuje ona wskaźnika do funkcji (pod warunkiem że typ się zgadza).

```
int g(int x, int (*f)(int))
{
    return 2*f(x);
}

std::cout << g(2, [](int x){return 2*x;}) << std::endl;
```

Listing 11: Kod w katalogu: Lambda2

Output:
8

```
int g(int x, int (*f)(int))
{
    return 2*f(x);
}

std::cout << g(2, [](int x){return 2.*x;}) << std::endl; // in lambda:
    double instead of int, error!!
```

Listing 12: Kod w katalogu: Lambda2

```
main.cpp:33:19: error: invalid user-defined conversion from
  ‘main()::<lambda(int)>’ to ‘int (*)(int)’ [-fpermissive]
   33 |     std::cout << g(2, [](int x){return 2.*x;}) << std::endl;
      |                  ~^~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
int g(int x, int (*f)(int))
{
    return 2*f(x);
}

std::cout << g(2, [](int x) -> int {return 2.*x;}) << std::endl;
```

Listing 13: Kod w katalogu: Lambda2

Output:
8

Co gdy wysłaną do funkcji lambdą chcemy wykonać przechwytywanie?

```
int g(int x, int (*f)(int))
{
    return 2*f(x);
}

int y = 50;
std::cout << g(2, [&](double x){return y*x;}) << std::endl; // error!!
```

Listing 14: Kod w katalogu: Lambda2

```
main.cpp:46:19: error: cannot convert ‘main()::<lambda(int)>’ to ‘int
 (*)(int)’
   46 |     std::cout << g(2, [&](int x){return y*x;}) << std::endl;
      |                  ~^~~~~~~~~~~~~~~~~~~~~~~~~~
      |                   |
      |                   main()::<lambda(int)>
```

```
#include <functional>
int g(int x, std::function< int(int) > f)
{
    return 2*f(x);
}

int y = 50;
std::cout << g(2, [&](double x){return y*x;}) << std::endl;
```

Listing 15: Kod w katalogu: Lambda2

Output:

200

Do `std::function` można również przypisać obiekt klasy z przeładowanym operatorem ().

```
struct PrintInt
{
    void operator()(int x)
    {
        std::cout << x << std::endl;
    }
};

PrintInt printInt;
std::function<void(int)> print_int = printInt;
print_int(7);
```

Listing 16: Kod w katalogu: Lambda2

Output:
7

1.4 Kilka pojęć związanych z funktorami

1.4.1 Generatory

Generatory to funktory, które można wywołać bez żadnych argumentów.

```
std::vector<int> v(10);
std::generate(v.begin(), v.end(), [i = 1]()mutable{return ++i;});
for (auto n : v)
    std::cout << n << " ";
```

Listing 17: Kod w katalogu: Funktorypp

Output:
2 3 4 5 6 7 8 9 10 11

1.4.2 Predykaty

Predykaty to funktory zwracające wartość logiczną.

```
std::cout << std::count_if(v.begin(), v.end(), [](int x){return x
    %2==0;}) << std::endl;
```

Listing 18: Kod w katalogu: Funktorypp

Output:
5

1.4.3 Funktory predefiniowane

Funktory predefiniowane - w bibliotece STL znajduje się kilka już zdefiniowanych funktorów, które mają na celu ułatwić pracę z funkcjami tej biblioteki.

```
std::negate<int> neg; // creating object of predefined negate functor
std::vector<int> v2(10);
std::transform(v.begin(), v.end(), v2.begin(), neg);
for (auto n : v2)
    std::cout << n << " ";
```

Listing 19: Kod w katalogu: Funktorypp

Output:
-2 -3 -4 -5 -6 -7 -8 -9 -10 -11

Kilka innych przykładów:

Operator	Funktor
+	plus
-	minus
*	multiplies
/	divides
%	modulus
-	negate
==	equal_to
!=	not_equal_to
>	greater
<	less
>=	greater_equal
<=	less_equal
&&	logical_and
||	logical_or
!	logical_not