Instytut Informatyki UMCS
Zakład Technologii Informatycznych

Wykład 11

Algorytmy, funktory i wyrażenia lambda

dr Marcin Denkowski

Lublin, 2019

AGENDA

1. Algorytmy
2. Funktory
3. Lambda

ALGORYTMY

- Algorytmy biblioteki STL – funkcje ogólnego przeznaczenia, działające na zakresach elementów.

- Zakres jest definiowany za pomocą iteratorów:
  `[pierwszy, ostatni)`
  gdzie `ostatni` odnosi się do elementu leżącego za ostatnim branym pod uwagę elementem.

```
ret_type algorithm(iter first, iter end, ...)
```

ALGORYTMY

- Algorytmy biblioteki STL – funkcje ogólnego przeznaczenia, działające na zakresach elementów.
- Zakres jest definiowany za pomocą iteratorów:
  `[pierwszy, ostatni)`
  gdzie `ostatni` odnosi się do elementu leżącego za ostatnim branym pod uwagę elementem.

```
ret_type algorithm(iter first, iter end, ...)
```

- Zebrane zostały w plikach nagłówkowych `<algorithm>` i `<numeric>`
- Grupy algorytmów
  - sekwencyjne, niezmieniające wartości
  - sekwencyjne zmieniające wartości
  - operacje sortowania i pokrewne
  - uogólnione operacje numeryczne

ALGORYTMY NIEMODYFIKUJĄCE

Algorytm/y	Opis
`for_each` `for_each_n`	wykonuje funktor dla każdego elementu w zakresie
`count` `count_if`	zlicza ilość wystąpień elementu spełniającego kryterium
`mismatch`	znajduje pierwszą pozycję, gdzie dwa zakresy różnią się

`find`
`find_if`
`find_if_not`
	znajduje pierwszy element w zakresie spełniający kryterium

`search` `search_n`	znajduje pierwsze wystąpienie sekwencji w zakresie spełniającej kryterium

ALGORYTMY, PRZYKŁAD

- Algorytm **`for_each`**:

```
template< class InputIt, class UnaryFunction >
UnaryFunction for_each( InputIt first, InputIt last,
                        UnaryFunction f );
```

- Przykładowa realizacja:

```
template<class InputIt, class UnaryFunction>
UnaryFunction for_each( InputIt first, InputIt last,
                        UnaryFunction fn)
{
    while( first!=last )
    {
        fn(*first);
        ++first;
    }
    return fn;
}
```

ALGORYTMY, PRZYKŁAD

- Algorytm **`for_each`**:

```
template< class InputIt, class UnaryFunction >
UnaryFunction for_each( InputIt first, InputIt last,
                        UnaryFunction f );
```

- Przykład

```
void print(int i) { cout << i << endl; }

...
std::vector<int> numbers = {10, 20, 5, 7, 11, 18};

std::for_each(numbers.begin(), numbers.end(), print);
```

ALGORYTMY, PRZYKŁAD

- Algorytm **`find`**:

```
template< class InputIt, class T >
InputIt find(InputIt first, InputIt last, const T& value );

template< class InputIt, class UnaryPredicate >
InputIt find_if(InputIt first, InputIt last, UnaryPredicate p );
```

ALGORYTMY, PRZYKŁAD

- Algorytm **`find`**:

```
template< class InputIt, class T >
InputIt find(InputIt first, InputIt last, const T& value );

template< class InputIt, class UnaryPredicate >
InputIt find_if(InputIt first, InputIt last, UnaryPredicate p );
```

- Przykład

```
std::vector<int> numbers = {10, 20, 5, 7, 11, 18};

auto it = std::find(numbers.begin(), numbers.end(), 11);

if( it != numbers.end() )
   cout << "numbers contains 11" << endl;
```

ALGORYTMY, PRZYKŁAD

- Algorytm **`find`**:

```
template< class InputIt, class T >
InputIt find(InputIt first, InputIt last, const T& value );

template< class InputIt, class UnaryPredicate >
InputIt find_if(InputIt first, InputIt last, UnaryPredicate p );
```

- Przykład

```
std::vector<int> numbers = {10, 20, 5, 7, 11, 18};
```

```
bool isOdd(int i) { return ((i%2)==1); }

it = std::find_if(numbers.begin(), numbers.end() isOdd);

cout << "The first odd value is " << *it << endl;
```

ALGORYTMY MODYFIKUJĄCE

Algorytm/y	Opis

`copy`
`copy_if`
`copy_n`
	kopiuje zakres do nowego zakresu

`move`	przenosi zakres do nowego zakresu
`fill` `fill_n`	kopiuje podaną wartość do wszystkich elementów zakresu
`generate` `generate_n`	wpisuje wynik kolejnych wywołań funktora do kolejnych elementów zakresu
`transform`	wykonuje funktor dla każdego elementu w zakresie

`remove`
`remove_if`
`remove_copy`
	przenosi na koniec zakresu elementy spełniające kryterium

`replace` `replace_if`	zastępuje wszystkie elementy w zakresie spełniające kryterium inną wartością
`unique` `unique_copy`	usuwa sąsiadujące duplikaty elementów w zakresie

ALGORYTMY, PRZYKŁAD

- Algorytm **copy**:

```
template< class InputIt, class OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt d_first );

template< class InputIt, class OutputIt, class UnaryPredicate >
OutputIt copy_if( InputIt first, InputIt last, OutputIt d_first,
                  UnaryPredicate pred );
```

- Przykład

```
bool greater10(int i) {if(i>10) return true; return false;}

...
std::vector<int> numbers = {10, 20, 5, 7, 11, 18};
std::vector<int> numbers_2(6);

std::copy_if(numbers.begin(), numbers.end(), numbers_2,
             greater10);
```

ALGORYTMY, PRZYKŁAD

- Algorytm **`generate`**:

```
template <class ForwardIt, class Generator>
void generate (ForwardIt first, ForwardIt last, Generator gen);
```

- Przykład

```
int unique_number(){
    static int i=0;
    return i+=2;
}

...
std::vector<int> numbers(10);

std::generate(numbers.begin(), numbers.end(),
              unique_number);
```

ALGORYTMY, PRZYKŁAD

- Algorytm **`transform`**:

```
template<class InIt, class OutIt, class UnaryOperation >
OutputIt transform( InIt first1, InIt last1, OutIt d_first,
                    UnaryOperation unary_op );

template<class InIt1,class InIt2,class OutIt,class BinaryOper>
OutputIt transform(InIt1 first1, InIt1 last1, InIt2 first2,
                    OutIt d_first, BinaryOper binary_op );
```

- Przykład

```
char to_upper(char c) { return c-'A'; }

...
std::string str("hello");
std::transform(str.begin(), str.end(), str.begin(),
               to_upper);
```

ALGORYTMY, PRZYKŁAD

- Algorytm **`remove`**:

```
template< class ForwIt, class T >
ForwIt remove( ForwIt first, ForwIt last, const T& value );

template< class ForwIt, class UnaryPred >
ForwIt remove_if( ForwIt first, ForwIt last, UnaryPred p );
```

- Przykład

```
bool greater10(int i) {if(i>10) return true; return false;}

...
std::vector<int> numbers = {10, 20, 5, 7, 11, 18};

auto it = std::remove_if(numbers.begin(), numbers.end(),
                         greater10);

numbers.erase(it. numbers.end());
```

ALGORYTMY SORTUJĄCE (I INNE)

Algorytm/y	Opis
`sort`	sortuje zakres w porządku rosnącym
`partial_sort`	sortuje pierwsze N elementów zakresu
`stable_sort`	kopiuje podaną wartość do wszystkich elementów zakresu
`nth_element`	częściowe sortowanie zakresu aż do sytuacji, że n-ty element
`partition` `partition_copy`	zmienia kolejność elementów w zakresie tak, że elementy spełniające kryterium znajdują się przed tymi, które go nie spełniają
`binary_search`	sprawdza czy podany element znajduje się w zakresie (w partycjonowanym)
`merge`	łączy dwa posortowane zakresy
`min_element` `max_element`	znajduje najmniejszy/największy element w zakresie

ALGORYTMY, PRZYKŁAD

- Algorytm **`sort`**:

```
template< class RandomIt >
void sort( RandomIt first, RandomIt last );

template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp );
```

- Przykład

```
bool less(int a, int b) {return a < b;}

...
std::vector<int> numbers = {10, 20, 5, 7, 11, 18};

std::sort(numbers.begin(), numbers.end(), less);
```

ALGORYTMY NUMERYCZNE

Algorytm/y	Opis
`accumulate`	„sumuje” elementy w podanym zakresie
`inner_product`	oblicza „iloczyn skalarny” dwóch zakresów
`reduce`	podobnie do accumulate ale sumowanie jest w przypadkowej kolejności
`partial_sum`	liczy „sumy” częściowe zakresu
`adjecent_difference`	liczy „różnicę” pomiędzy sąsiednimi elementami

FUNKTORY

- **Funktor** (obiekt funkcyjny, *function object*) – dowolny obiekt, którego można używać jak funkcji, za pomocą operatora()
  1) zwykłe funkcje
  2) wskaźniki na funkcje
  3) obiekty klas, w których przeciążony jest `operator()`

FUNKTORY

- **Funktor** (obiekt funkcyjny, *function object*) – dowolny obiekt, którego można używać jak funkcji, za pomocą operatora()
  1) zwykłe funkcje
  2) wskaźniki na funkcje
  3) obiekty klas, w których przeciążony jest `operator()`

- Typy funktorów:
  1) Generator – funktor bezargumentowy (***generator***)
  2) Funktor jednoargumentowy (***unary function***)
  3) Funktor dwuargumentowy (***binary function***)
  4) Predykat – funktor jednoargumentowy zwracający typ logiczny (***unary predicate***)
  5) Predykat dwuargumentowy – funktor dwuargumentowy zwracający typ logiczny (***binary predicate***)

FUNKTORY PREDEFINIOWANE

- Zdefiniowane w `<functional>`
- Funktory jedno- i dwuargumentowe

```
template<class T> struct plus{
   T operator() (const T& left, const T& right) const
       { return left+right; }
   };
```

- `template<class T> struct minus;`
- `template<class T> struct multiplies;`
- `template<class T> struct divides;`
- `template<class T> struct modulus;`
- `template<class T> struct negate;`

FUNKTORY PREDEFINIOWANE

- Zdefiniowane w `<functional>`
- Predykaty:

```
template<class T> struct equal_to{
  bool operator() (const T& left, const T& right) const
       { return left==right; }
  };
template<class T> struct not_equal_to;
template<class T> struct greater;
template<class T> struct less;
template<class T> struct greater_equal;
template<class T> struct less_equal;
template<class T> struct logical_and;
template<class T> struct logical_or;
template<class T> struct logical_not;
```

BINDER

- **`bind`** – szablon funkcji, która tworzy wrapper na wywołanie innej funkcji

```
template<clss F, class…. Args>
/*unspecified*/ bind(F&& f, Args&&… args);
```

  - `f` – funktor,
  - `args` – lista argumentów do wiązania, niewiązane argumenty można zastąpić placeholderem _1, _2, _3, ...

- Funkcja **`bind`** umożliwia wywołanie przekazanej funkcji, w taki sposób jakby miała ona inną ilość parametrów

```
...
void fun(int n1, int n2, int n3) {...}
...

auto bf = std::bind(fun, std::placeholder::_1, 4, std::placeholder::_3);
bf(2, 10); // rownowazne wywolaniu: fun(2, 4, 10);
```

WYRAŻENIA LAMBDA

- **Wyrażenia lambda** (*Lambda expressions*) – anonimowe funkcje definiowane w miejscu użycia, zdolne do operowania na zmiennych w zasięgu
- Typ wyrażenia lambda – prvalue, unikalny, nie-unia, nie-agregat, typ klasowy zwany ClosureType
- Ogólna postać:

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

`[]` – lista przechwytywanych nazw (początek definicji lambdy)

`()` – analogicznie jak przy zwykłej funkcji, argumenty, jakie ma przyjmować wyrażenie lambda

`specifiers` – atrybuty wyrażenia lambda (opcjonalne: `mutable`, `constexpr`)

`-> T` – typ zwracany wyrażenia lambda (opcjonalny)

`{}` – ciało wyrażenia lambda (równoważne ciału funkcji)

WYRAŻENIA LAMBDA

- Przykłady

1) puste wyrażenie lambda, wywołane od razu

```
[](){}
```

2) wyrażenie lambda drukujące liczbę, przypisane zmiennej `lmbda`

```
auto lmbda = [] (int x) -> void
{ std::cout << "number:" << x << std::endl; };

lmbda(13);
```

3) wyrażenie lambda, zwraca wartość powiększoną o 2

```
vector<int> vec = {…};
int x = 10;
transform(vec.begin(), vec.end(), vec.begin(),
          [](int i) { return i+2;} );
```

WYRAŻENIA LAMBDA, PRZECHWYTYWANIE

- Przechwytywanie zmiennych automatycznych do wnętrza wyrażenia lambda
- Zmienne podajemy w `[]`, po przecinku (wartość lub referencję)

```
int x = 4, y = 8;
auto lmbda = [x, &y]() { y = 2*x; };
lmbda(13);
```

- Przechwytywanie wszystkich zmiennych przez wartość

```
auto lmbda = [=]() { y*x; };
```

- Przechwytywanie wszystkich zmiennych przez referencję

```
auto lmbda = [&]() { y = 2*x; };
```