Uniformná inicializácia
Range-based for
std::initializer_list
STL algoritmy

Programovacie techniky | Pavol Marák

OBSAH

- Uniformná inicializácia
  - Stará vs. nová syntax (od štandardu C++11)
  - Príklady
- Range-based for cyklus
- std::initializer_list

OBSAH

- STL algoritmy
  - Taxonómia
  - Lambda výraz
  - Príklady

Uniformná inicializácia

- Uniformná inicializácia zavádza konzistentnú syntax pri inicializovaní objektov rôznych dátových typov.
- Pred C++11 sa rôzne dátové typy inicializovali pomocou odlišných zápisov, čo viedlo k chybám.

Uniformná inicializácia

- Nová syntax používa zápis pomocou { }.

```
type var_name{arg1, arg2, arg3, argN};
```

Stará syntax

- Pred štandardom C++11.
- Inicializácia pomocou:
  - ( ) : inicializácia konštruktorom alebo základné typy
  - { } : inicializácia agregovaných typov a poľa
  - Bez zátvoriek:
    - inicializácia premennej default konštruktorom (class typy)
    - neinicializované premenné (základné dátové typy)

Stará syntax

- Pred štandardom C++11.
- Inicializácia pomocou:
  - ( ) : inicializácia konštruktorom alebo základné typy
  - { } : inicializácia agregovaných typov a poľa
  - Bez zátvoriek:
    - inicializácia premennej default konštruktorom (class typy)
    - neinicializované premenné (základné dátové typy)

Agregovaný typ je pole alebo trieda, ktorá:

- nemá používateľom definovaný konštruktor
- nemá private/protected nestatické atribúty

Príklad

Vývojové prostredie CLion
CMakeLists.txt

```
set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++03")
```

Štandard C++03

Príklad

```
// inicializacia zakladnych (built-in) typov
int num1;
int num2 = 10;
int num3(10);
```

Neinicializovaná premenná

Príklad

```
// Agregovany typ
// Nesmie mat user-defined konstruktory
// a private/protected cleny.
struct S {
    int num;
};

// agregovany typ (pole a struktura)
int pole[4] = { 1, 2, 3, 4 };
S s1;
S s2 = { .num: 5 };
S s3[] = { {.num: 1}, {.num: 2} };
```

Príklad

```
// Ne-agregovany typ.
class C {
    int num;
public:
    C() : num(0) {}

    C(int a) : num(a) {}
};

// ne-agregovany typ (trieda s private clenom a konstruktorom)
C c1; // default konstruktor
C c2(3); // parametricky konstruktor
C c3 = 4; // konverzny konstruktor
C c4 = c3; // copy konstruktor
```

Nová syntax

- Od štandardu C++11.
- { } : inicializácia všetkých typov

Príklad

```
// zakladny (built-in) typ
int num1n{}; // zero-inicializacia
int num2n{ 10 }; // direct-inicializacia
```

Príklad

```
// Agregovany typ.
// Nesmie mat user-defined konstruktory a
// private/protected cleny.
struct S {
    int num;
};

// agregovany typ (pole a struktura)
int polen[4]{1, 2, 3, 4};
S s1n{};
S s2n{ .num: 5};
S s3n[]{{ .num: 1},{ .num: 2}};
```

Príklad

```
// Ne-agregovany typ.
class C {
    int num;
public:
    C() : num(0) {}
    C(int a) : num(a) {}
};

// neagregovany typ
C c0n; // default-inicializacia
C c1n{}; // value-inicializacia ... zavola sa default konstruktor
C c2n{3}; // direct-inicializacia
C c3n{c2n}; // copy-inicializacia
```

Príklad

```
// Ne-agregovany typ.
class C {
    int num;
public:
    C() : num(0) {}
    C(int a) : num(a) {}
};

// dynamicke pole
C* dyn_pole = new C[3]{ {1}, {2}, {3} };


(gdb) p *dyn_pole@3
$1 = {{num = 1}, {num = 2}, {num = 3}}
```

Príklad

```
int main( ) {

    // uniformna direct inicializacia pola
    double *pole1 = new double[3]{10, 20, 30};
    // uniformna zero-inicializacia pola
    double *pole2 = new double[3]{};
    // uniformna default-inicializacia pola
    double *pole3 = new double[3]; // prvky zostanu inicializovane na neurcite hodnoty

    return 0;
}
```

```
(gdb) print *pole1@3
$1 = {10, 20, 30}
(gdb) print *pole2@3
$2 = {0, 0, 0}
```

Príklad

```
// Ne-agregovany typ.
class C {
    int num;
public:
    C(int a) : num(a) {}
};

// funkcia, ktora ma na vstupe objekt triedy C
void fn1(C c) {
}

int main() {

    // inicializacia argumentov funkcie
    // fn1(C(1)); // stara syntax
    fn1( c: {1});

    return 0;
}
```

Príklad

```
// Ne-agregovany typ.
class C {
    int num;
public:
    C(int a) : num(a) {}
};

// funkcia, ktora vracia objekt triedy C
C fn2() {
    // return C(1); // stara syntax
    return {1};
}

int main() {

    // inicializacia pri navrate objektov
    C c1 = fn2();
    C c2{fn2()};
    return 0;
}
```

Príklad

```
// funkcia, ktora vracia vektor
vector<int> fnVector() {
    return {1,2,3};
}

int main() {

    // inicializacia pri navrate objektov
    vector<int> v1 = fnVector();
    vector<int> v2{fnVector()};
    return 0;
}
```

```
(gdb) print v1
$3 = std::vector of length 3, capacity 3 = {1, 2, 3}
(gdb) print v2
$4 = std::vector of length 3, capacity 3 = {1, 2, 3}
```

Príklad

```
// Ne-agregovany typ.
class C {
    int num;
public:
    C(int a) : num(a) {}
};

// funkcia, ktora vracia smernik na dynamicke pole objektov triedy C
C *fn3() {
    return new C[3]{{1},{2},{3}};
}

int main() {

    // inicializacia pri navrate objektov
    C *r1 = fn3();
    C *r2{fn3()};
    return 0;
}
```

```
(gdb) print *r1@3
$1 = {{num = 1}, {num = 2}, {num = 3}}
(gdb) print *r2@3
$2 = {{num = 1}, {num = 2}, {num = 3}}
```

Príklad

```
int a();
```

vs.

```
int a{};
```

Range-based for cyklus

- Čitateľnejší ekvivalent k tradičnému `for` cyklu.
- Je vykonaný nad "rozsahom" prvkov.
- Od štandardu C++11

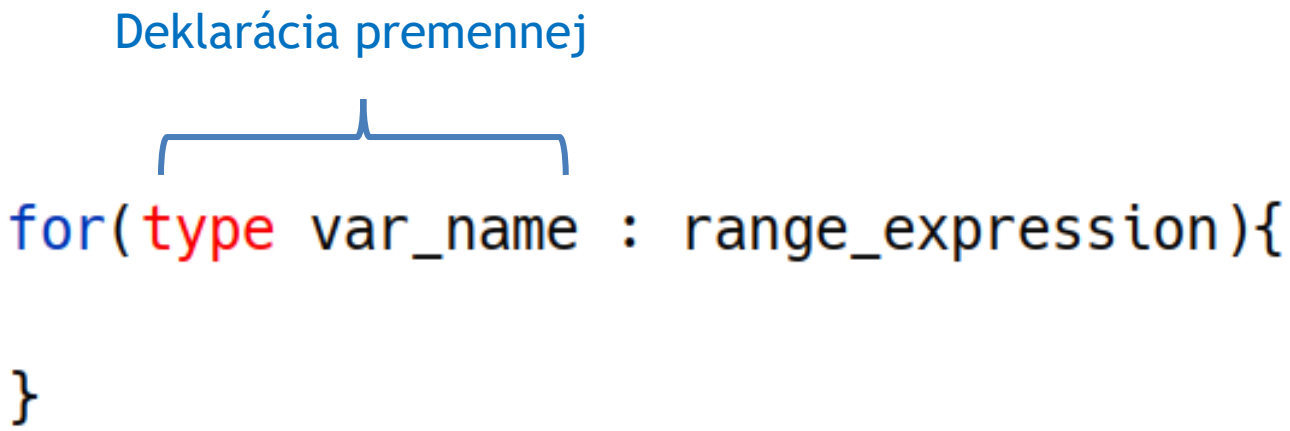
Range-based for cyklus

- Čitateľnejší ekvivalent k tradičnému `for` cyklu.
- Je vykonaný nad "rozsahom" prvkov.
- Od štandardu C++11

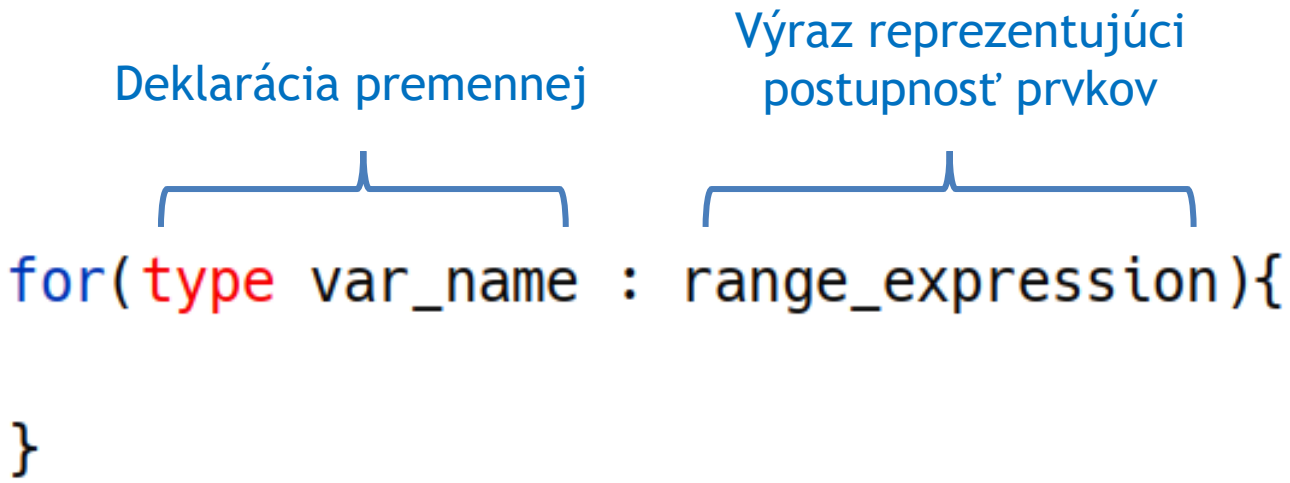
```
for(type var_name : range_expression){

}
```

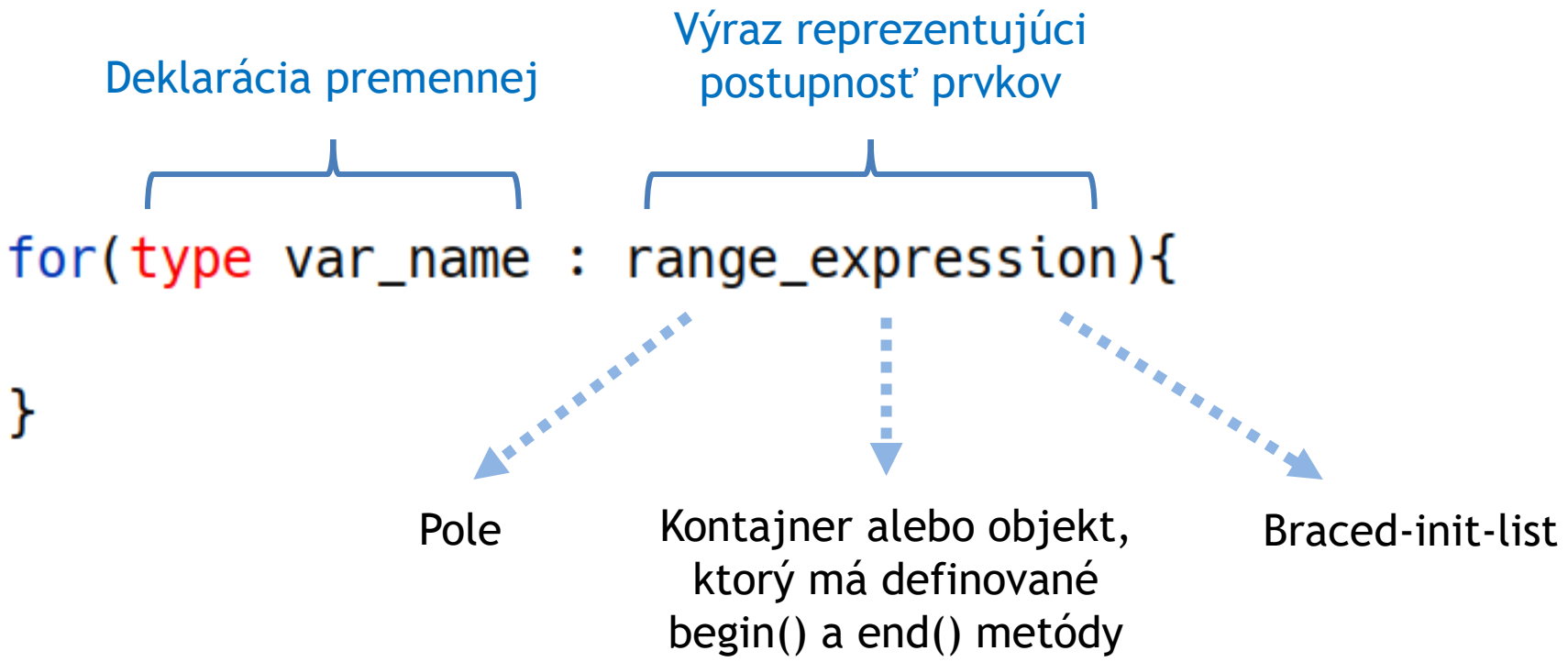
Range-based for cyklus



Range-based for cyklus



Range-based for cyklus



Príklad

```
int pole[]{1,2,3,4,5};
for (int i : pole) {
    cout << " " << i;
    // premennu 'i' mozeme zmenit,
    // ale v poli sa ziadna hodnota
    // nezmeni, nakolko 'i' je kopia
    // prvku pola
}
```

Príklad

```
int pole[]{1,2,3,4,5};  pole: [5]
for (int& i : pole) {
    i++;
    // zvysime prvky pola o 1
}
```

Príklad

```
vector<int> v {1,2,3,4,5};
for (int i : v) {
    // telo cyklu
}
```

Príklad

```
map<string,string> m{
        { x: "aaa", y: "AAA"},
        { x: "bbb", y: "BBB"},
        { x: "ccc", y: "CCC"},
};
for (pair<string,string> p : m) {
    cout << " (" << p.first << "," << p.second << ")";
}
```

Príklad

braced-init-list

```
for (int i : {5,10,15,20}) {
    // telo cyklu
}
```

std::initializer_list

- Hlavičkový súbor #include <initializer_list>
- Objekt na reprezentáciu hodnôt inicializačného zoznamu.
- Automaticky sa skonštruuje pri zápise {...}.
- Môžeme pomocou neho vytvoriť špeciálny typ konštruktora.
- Nemýliť si ho s member initializer listom (iná forma zápisu konštruktora)

Príklad

```cpp
#include <vector>
using namespace std;

class T {
    vector<int> v;
};

int main() {

    T t1{1};
    T t2{1, 2};
    T t3{1, 2, 3, 4, 5, 6, 7, 8};
    return 0;
}
```

Príklad

```cpp
#include <vector>
#include <initializer_list>
using namespace std;

class T {
    vector<int> v;
public:
    T (const initializer_list<int>& i){
        for(const int e : i){
            v.push_back(e);
        }
    }
};

int main() {

    T t1{1};
    T t2{1, 2};
    T t3{1, 2, 3, 4, 5, 6, 7, 8};
    return 0;
}
```

```
(gdb) print t1
$1 = {v = std::vector of length 1, capacity 1 = {1}}
(gdb) print t2
$2 = {v = std::vector of length 2, capacity 2 = {1, 2}}
(gdb) print t3
$3 = {v = std::vector of length 8, capacity 8 = {1, 2, 3, 4, 5, 6, 7, 8}}
```

Príklad

```
class T {
    vector<int> v;
public:
    T(int, int) {

    }

    T(const initializer_list<int> &i) {
        for (const int e : i) {
            v.push_back(e);
        }
    }
};

int main() {
    T t1{1, 2}; // zavola sa konstruktor s initializer_list
    T t2(1, 2); // zavola sa konstruktor s (int,int)
    return 0;
}
```

STL algoritmy

Hlavičkové súbory

- #include<algorithm>
- #include<numeric>

- Zbierka užitočných algoritmov pracujúcich s STL kontajnermi (aj s poliami).
- Algoritmy nikdy nemenia veľkosť kontajnera.

library

<algorithm>

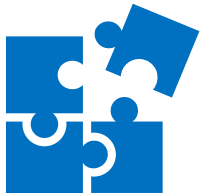
<algorithm>

Standard Template Library: Algorithms

The header <algorithm> defines a collection of functions especially designed to be used on ranges of elements.

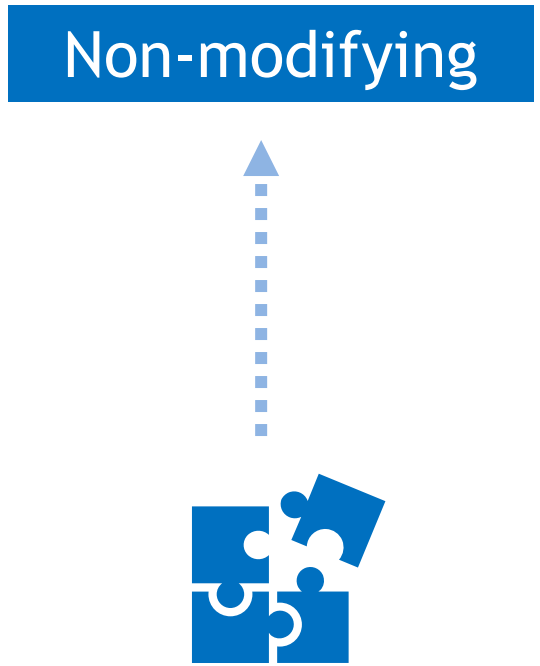
A range is any sequence of objects that can be accessed through iterators or pointers, such as an array or an instance of some of the STL containers. Notice though, that algorithms operate through iterators directly on the values, not affecting in any way the structure of any possible container (it never affects the size or storage allocation of the container).

- #include<algorithm>



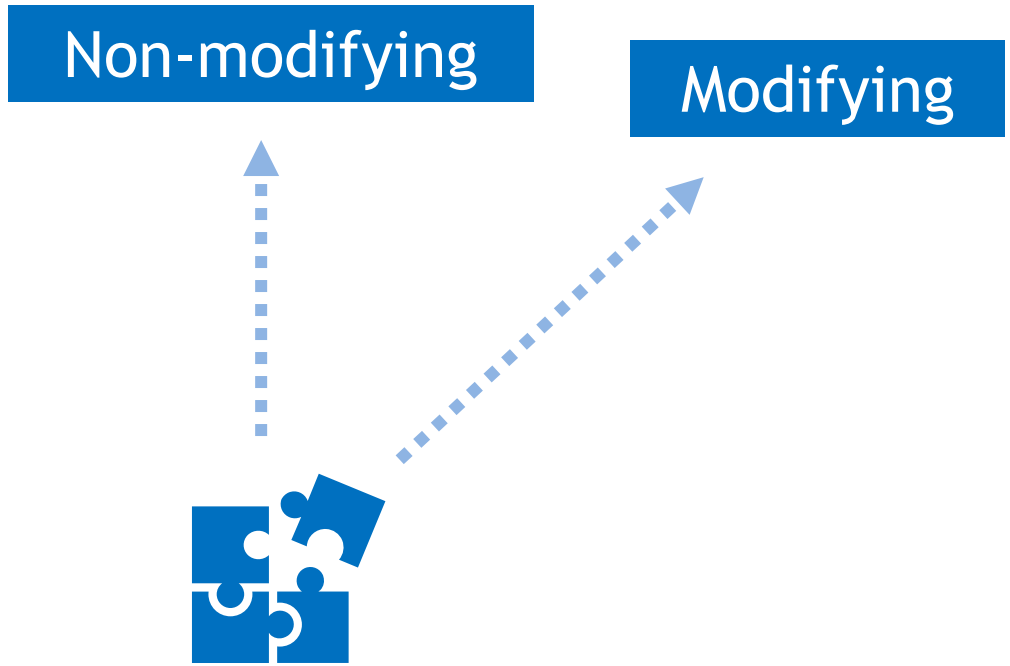
STL algoritmy

- #include<algorithm>



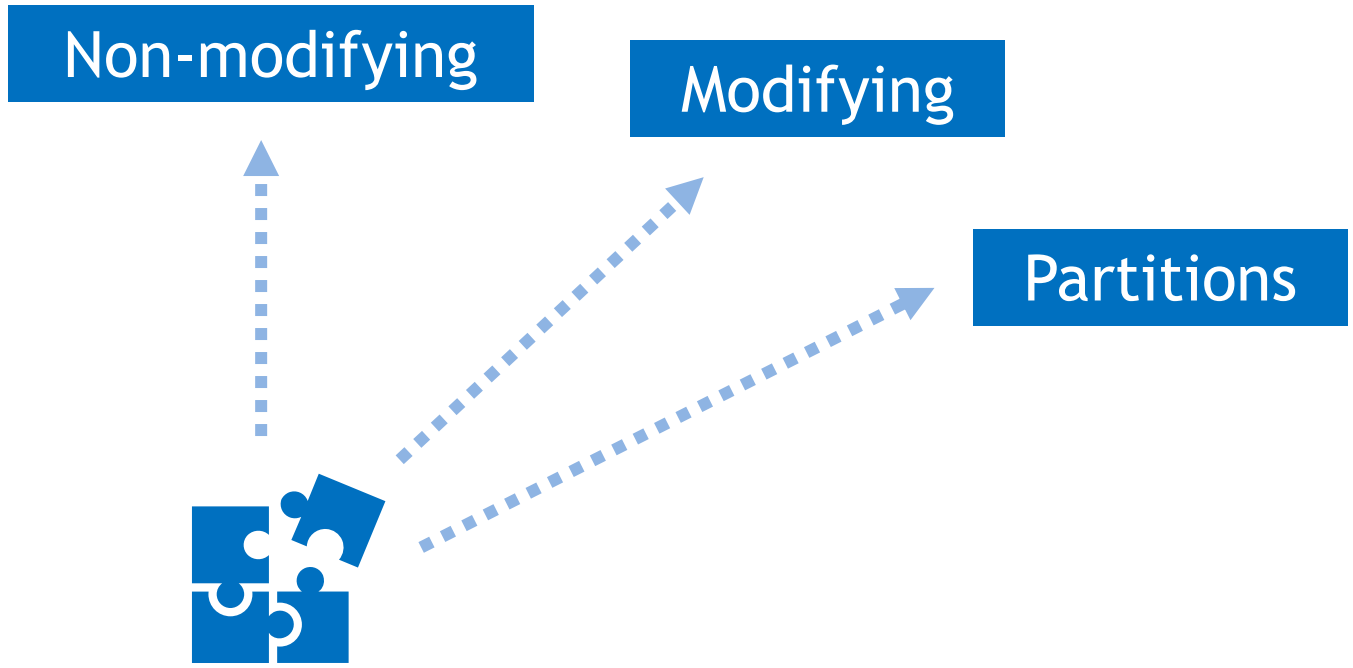
STL algoritmy

- #include<algorithm>



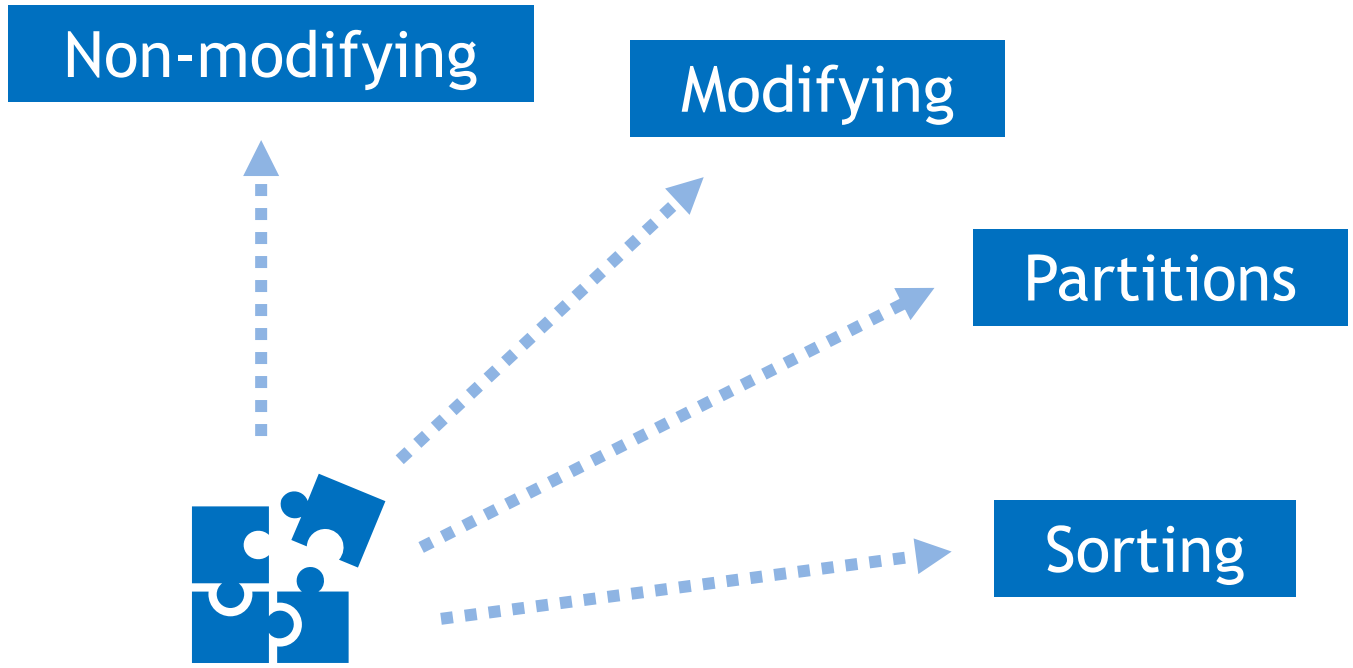
STL algoritmy

- #include<algorithm>



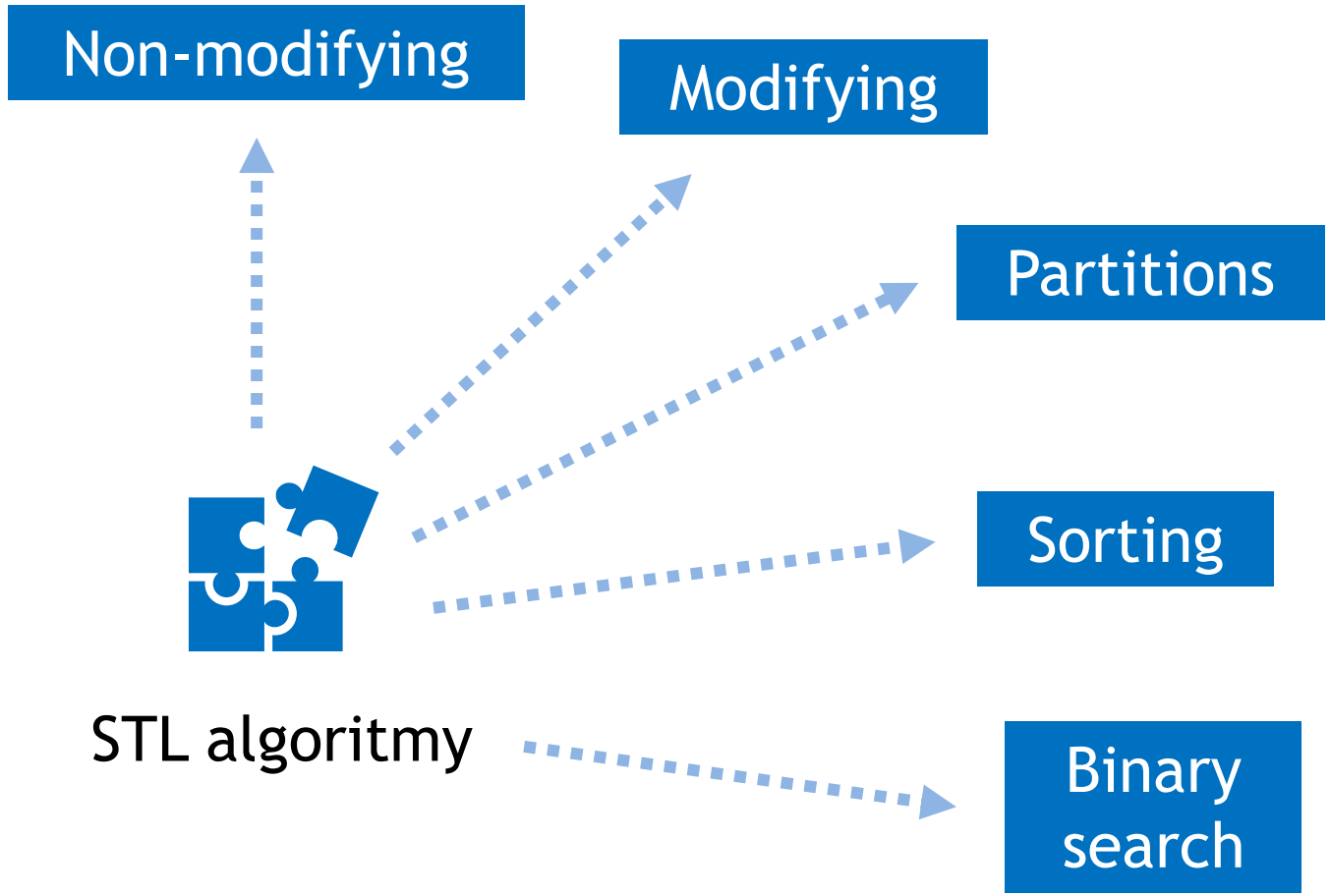
STL algoritmy

- #include<algorithm>

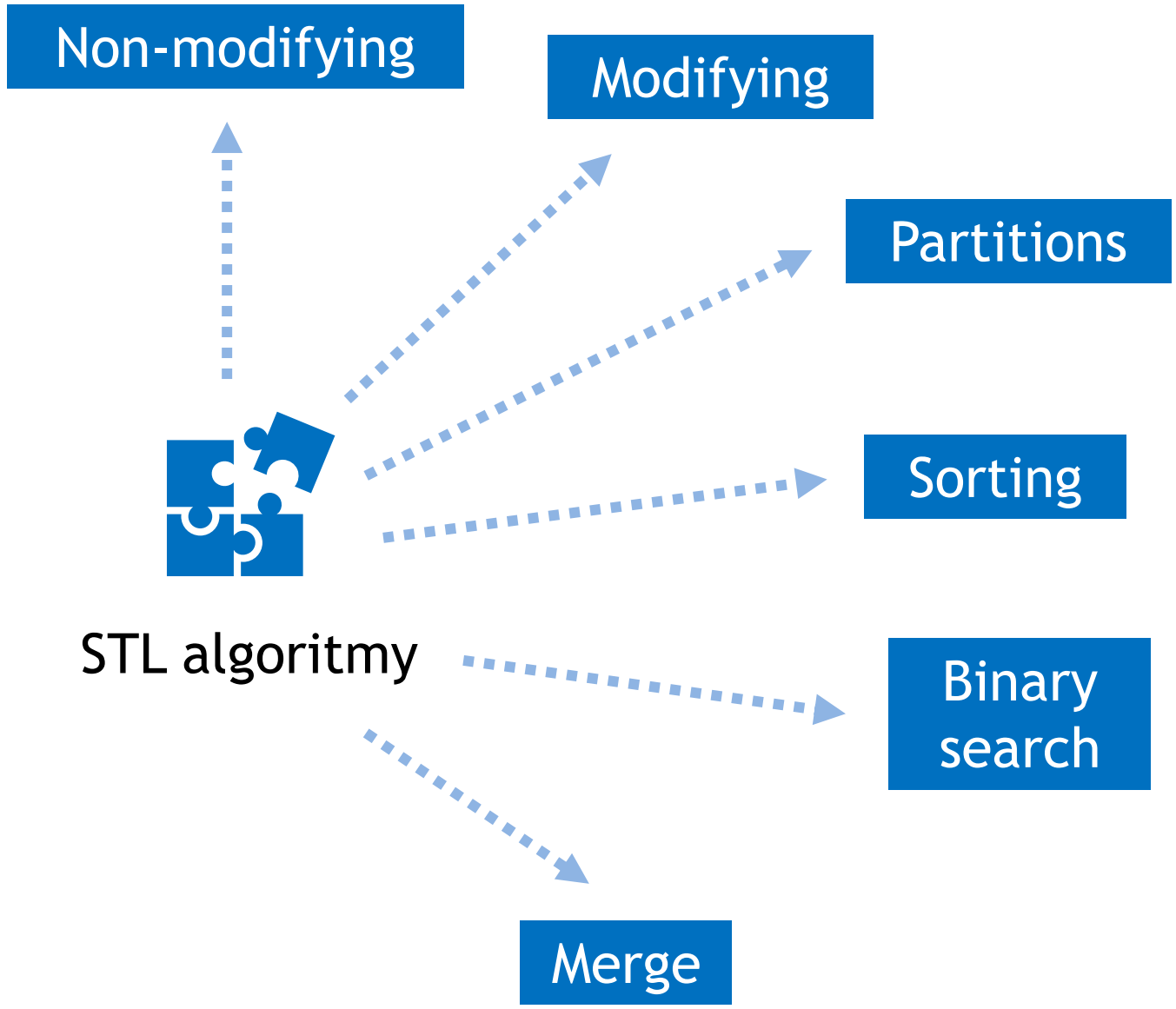


STL algoritmy

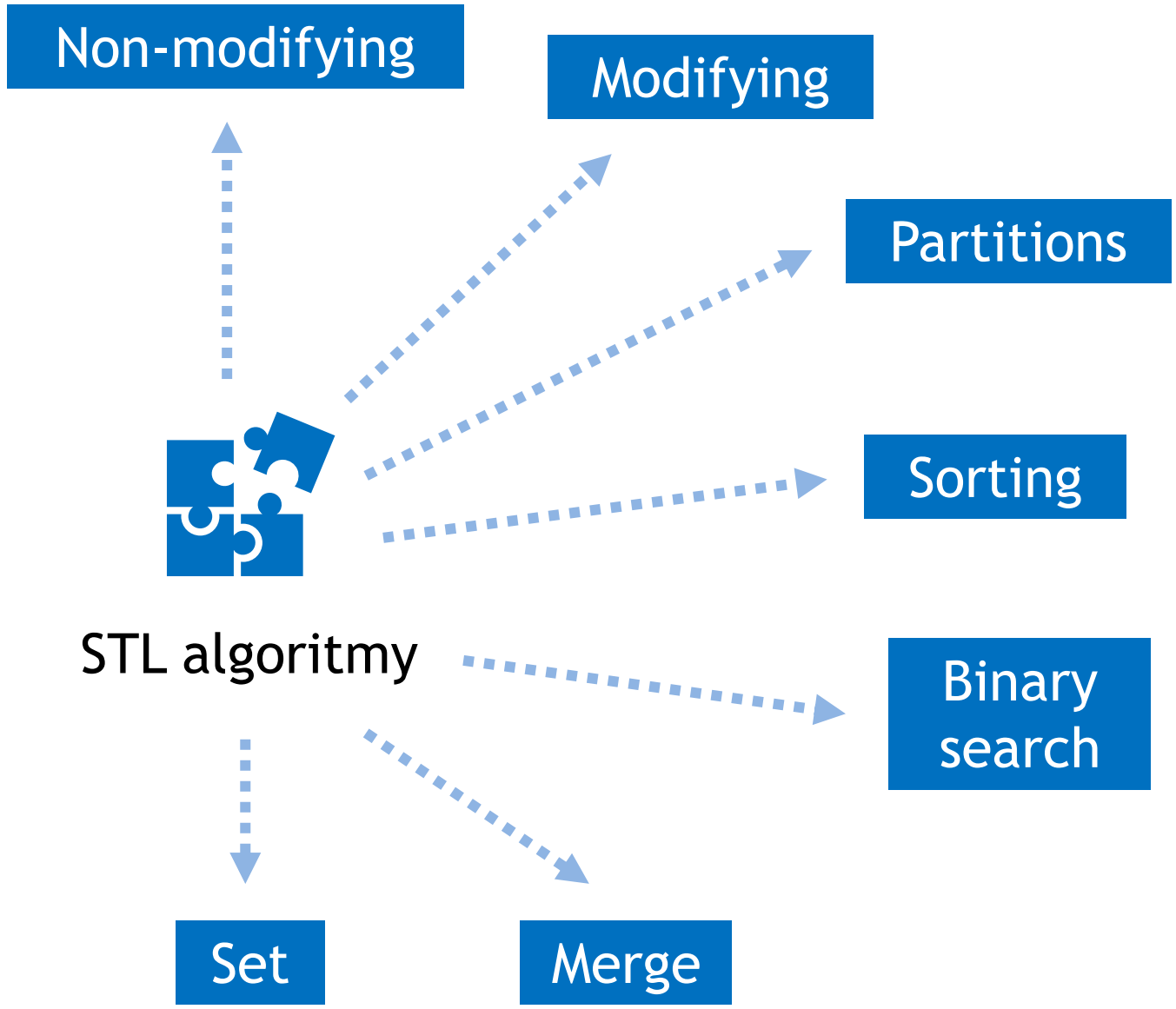
- #include<algorithm>



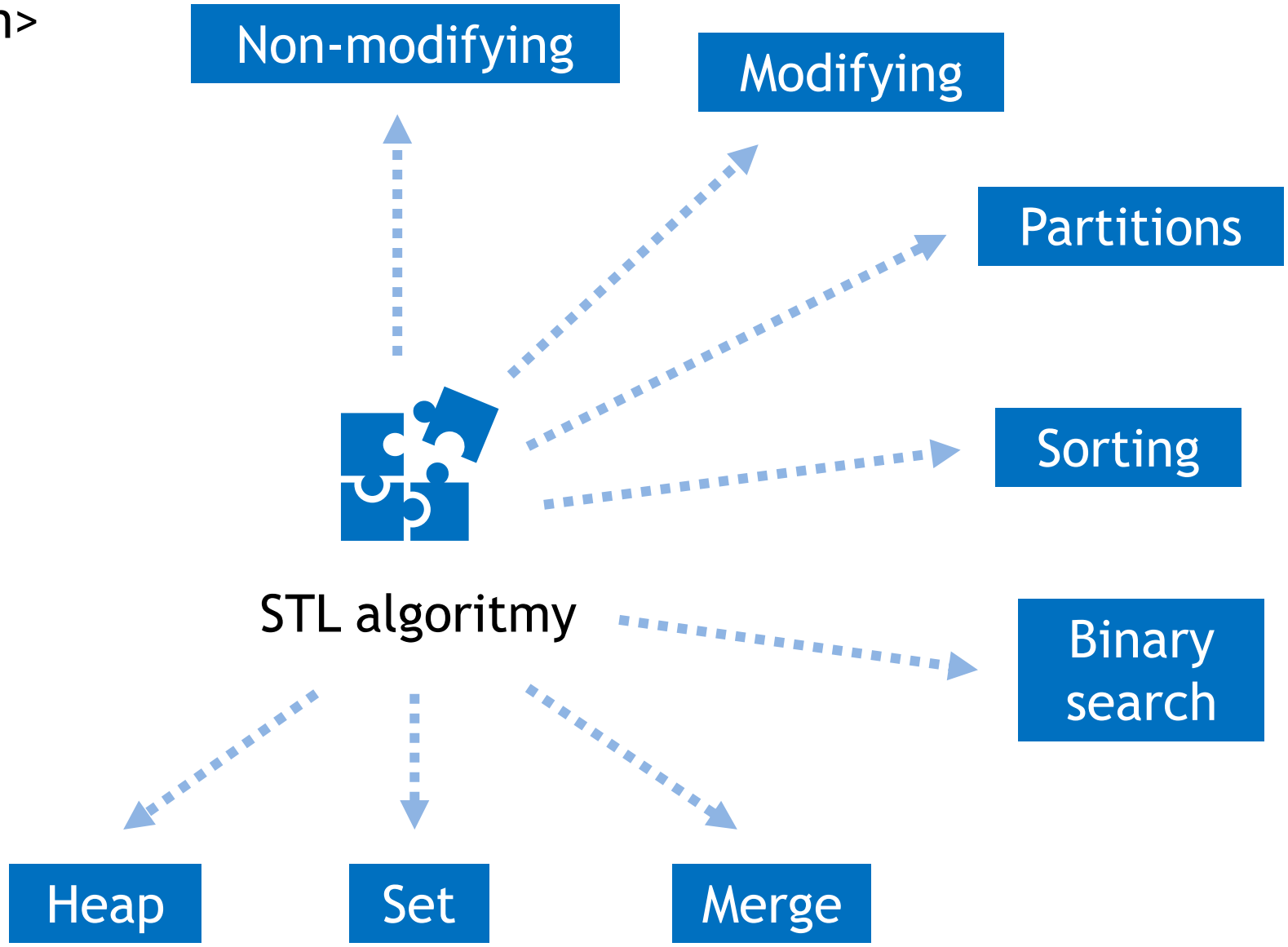
- #include<algorithm>



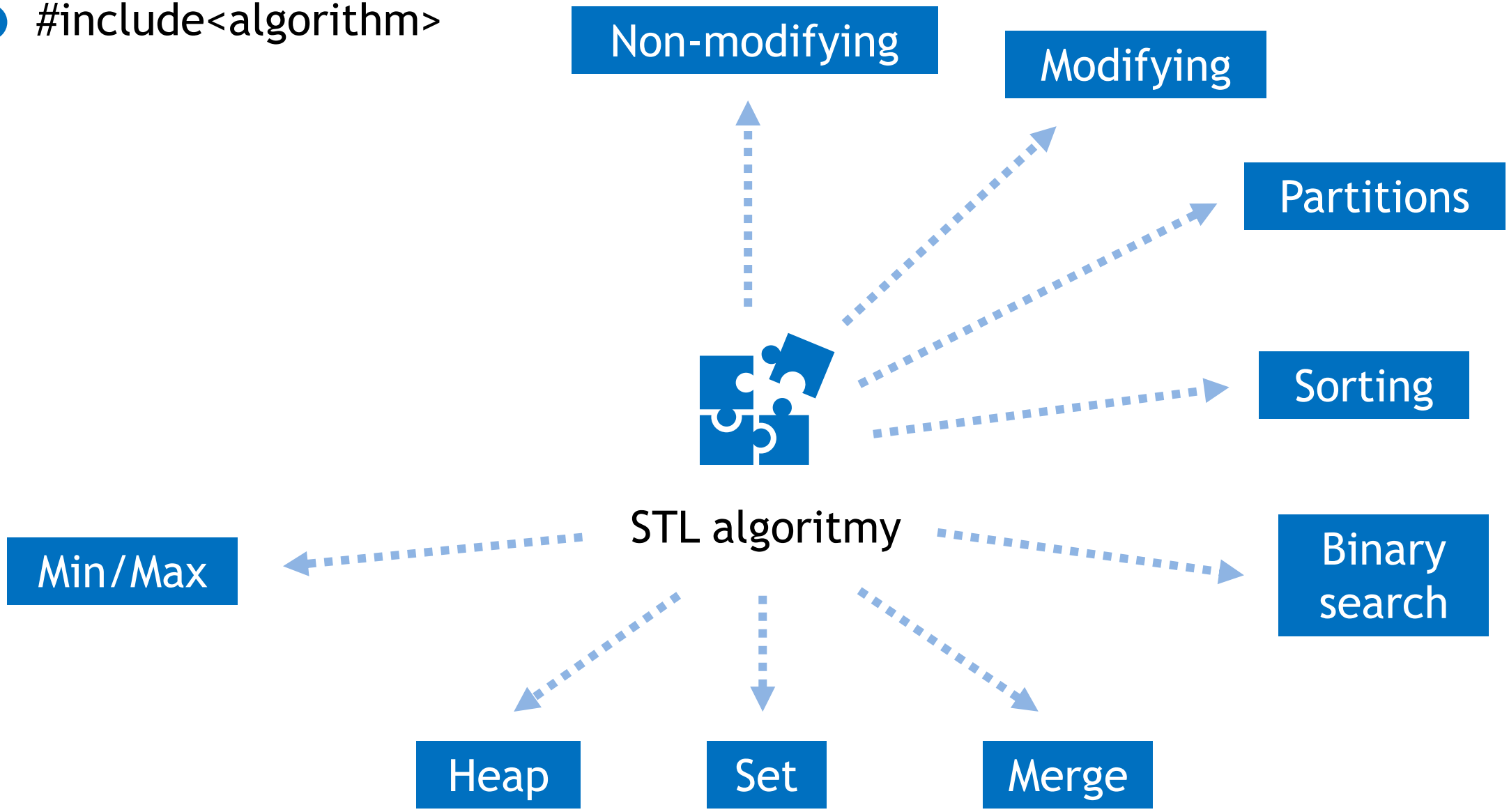
- #include<algorithm>



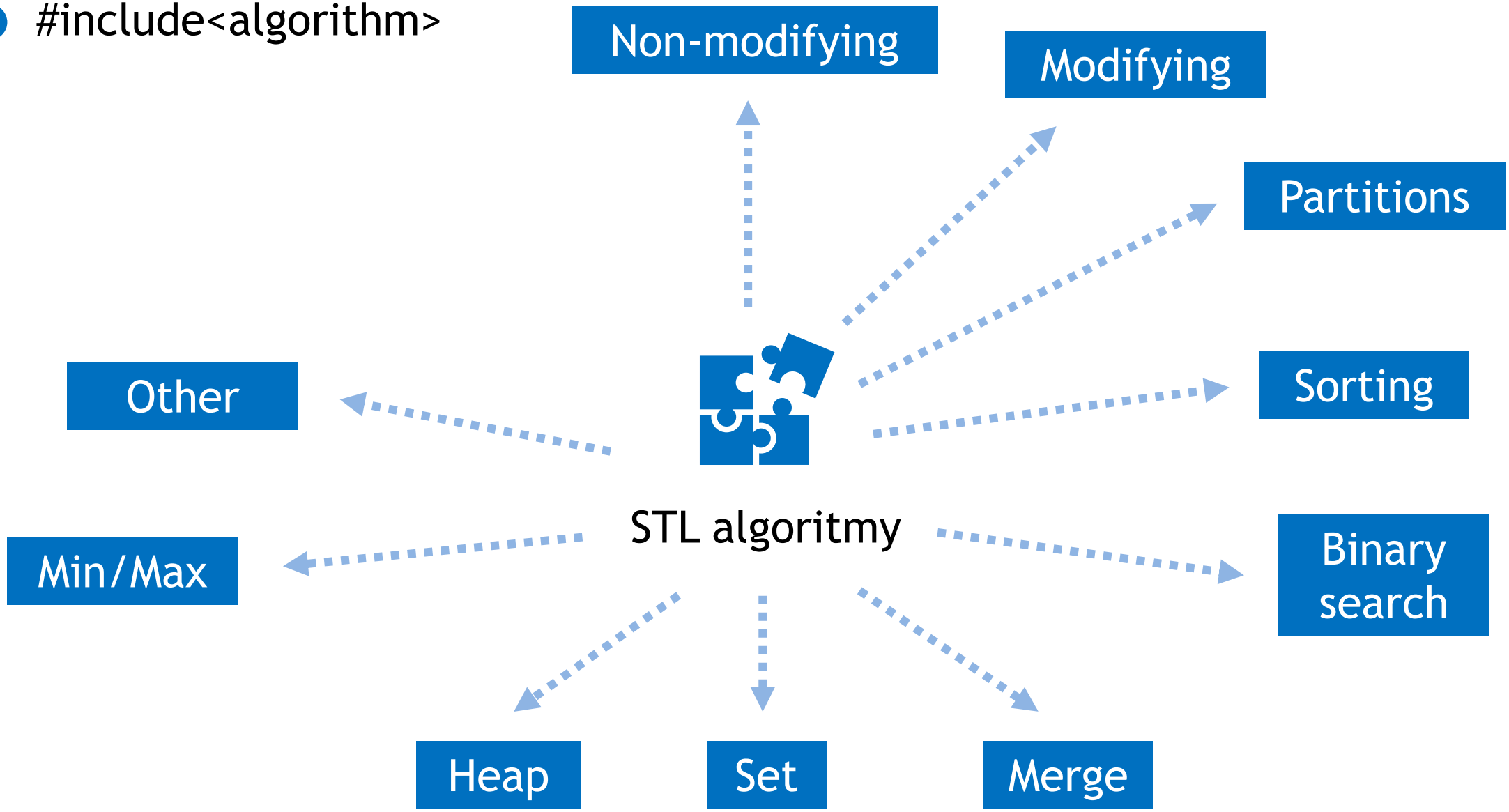
- #include<algorithm>



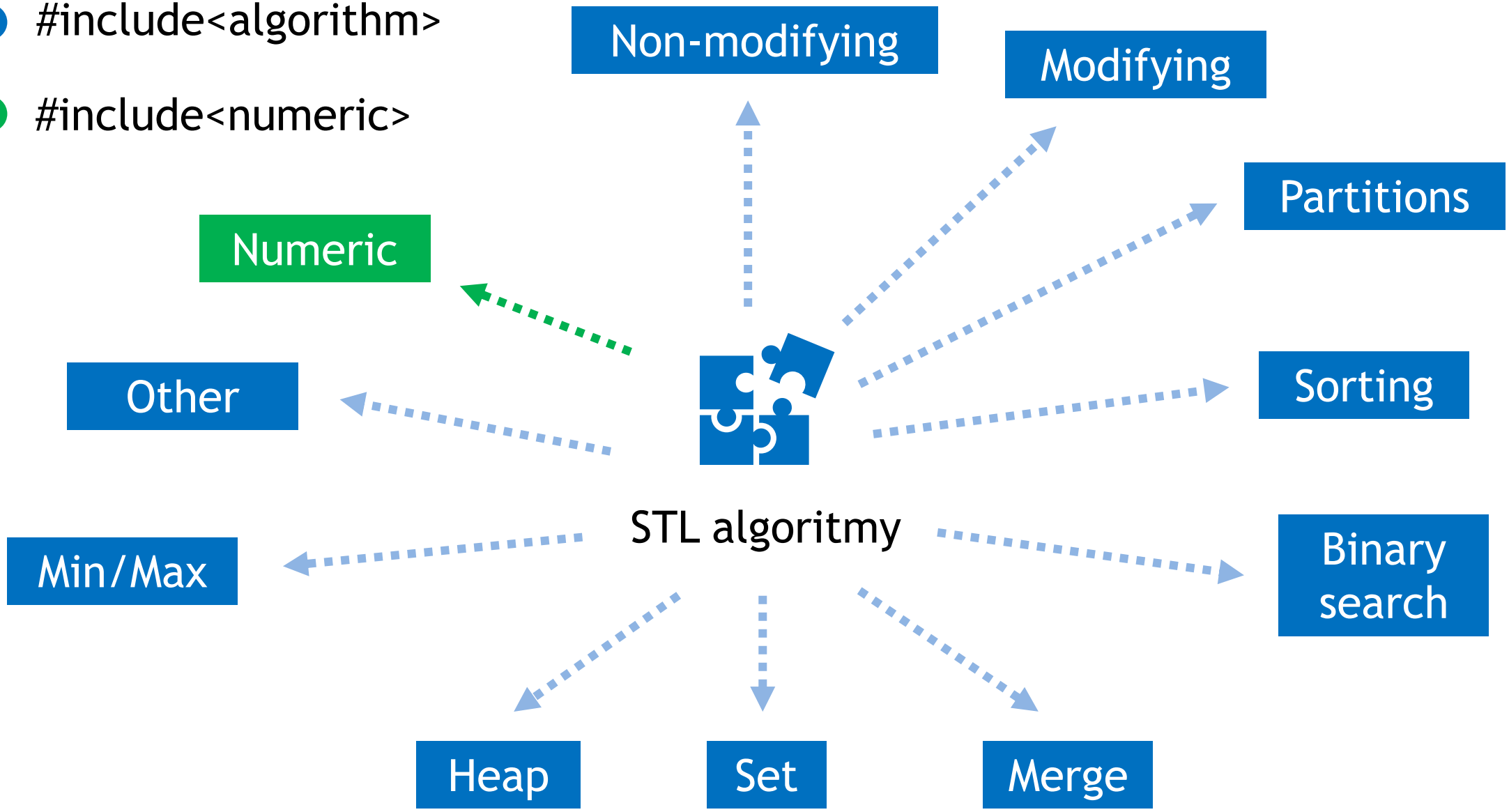
- #include<algorithm>



- #include<algorithm>



- #include<algorithm>
- #include<numeric>

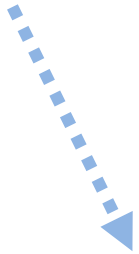


Lambda výraz

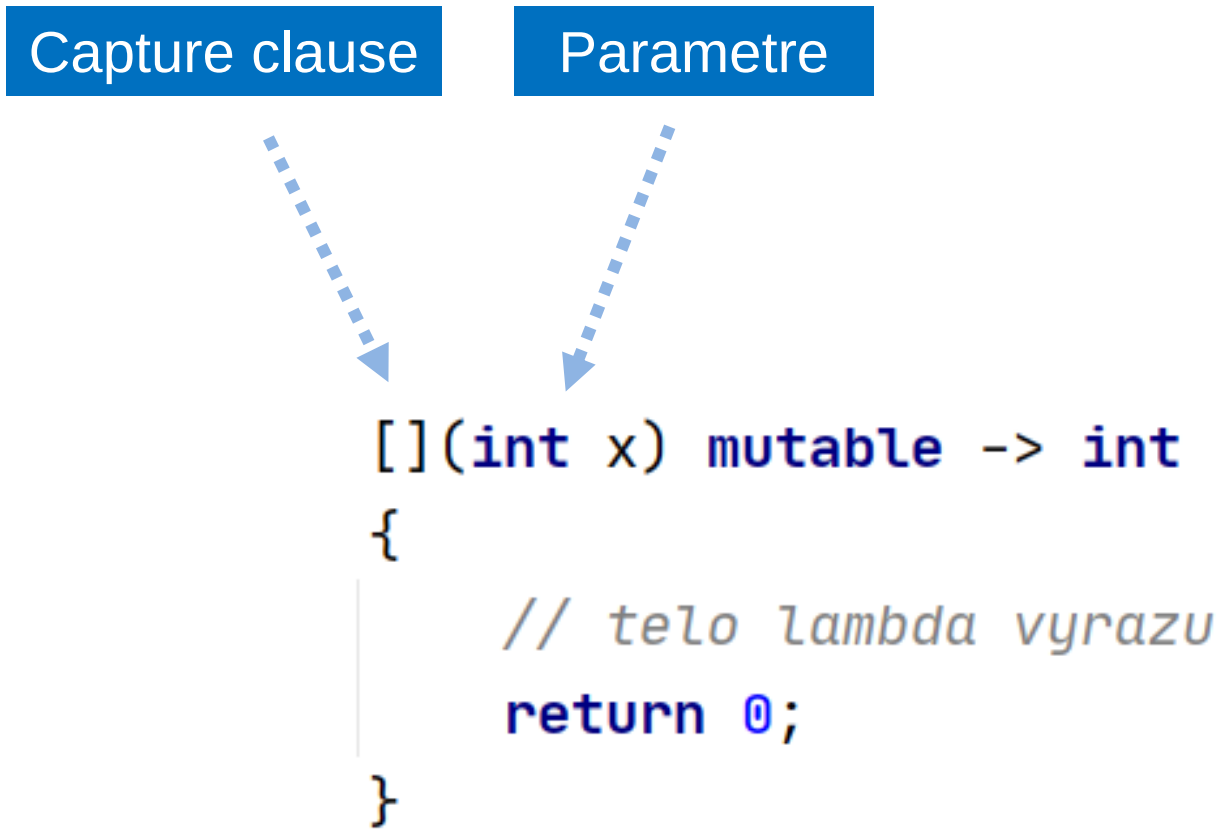
- Jedná sa o spôsob definovania anonymnej funkcie (presnejšie funktora) priamo na mieste jej zavolania (nemusíme tak vytvárať definíciu pomenovanej funkcie).
- Lambda výraz je schopný zachytávať premenné z okolia.
- Frekventovane používané s STL algoritmami.

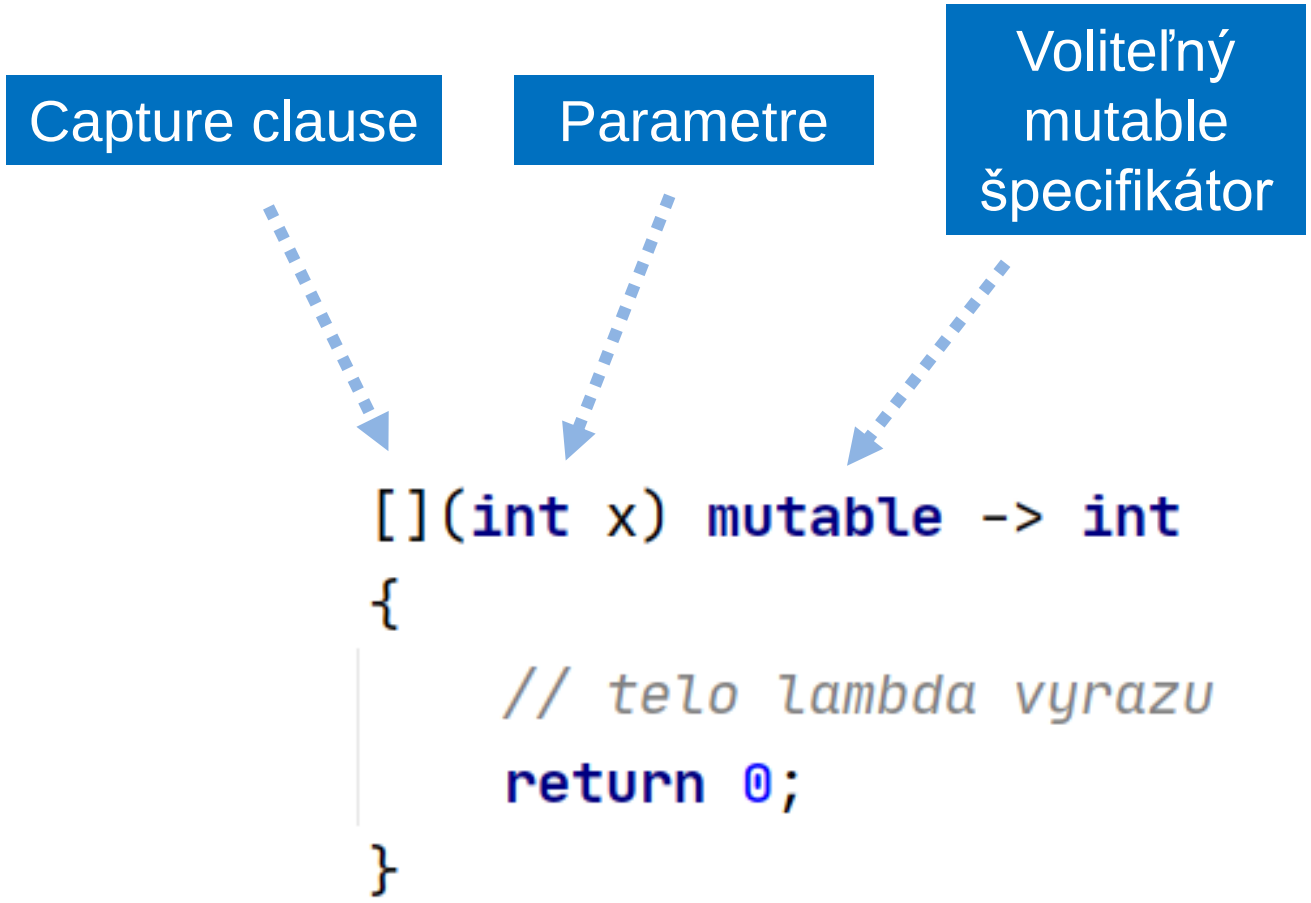
```
[](int x) mutable -> int
{
    // telo lambda vyrazu
    return 0;
}
```

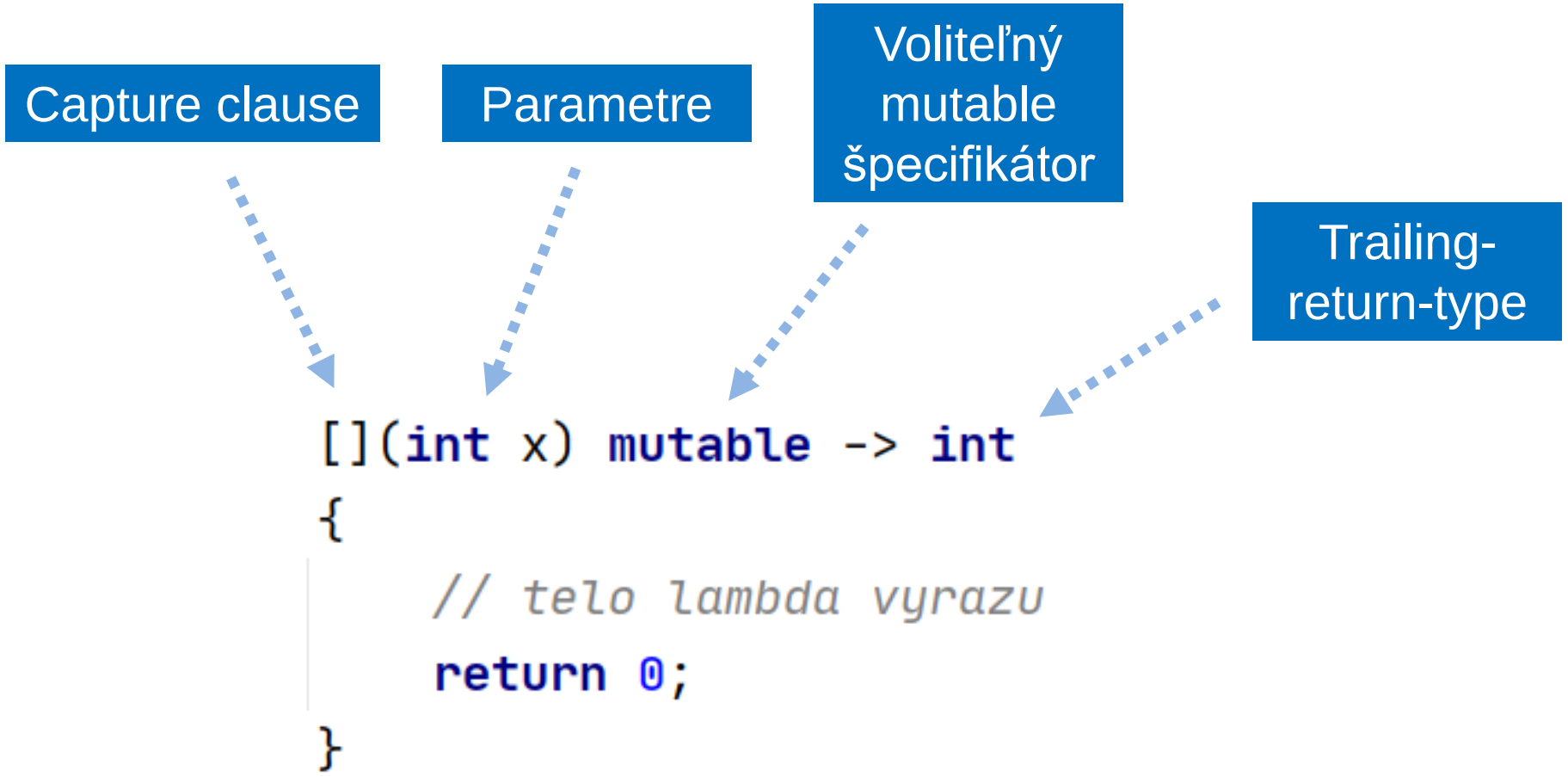
Capture clause



```
[](int x) mutable -> int
{
    // telo lambda vyrazu
    return 0;
}
```


Capture clause
Parametre
[](int x) mutable -> int
{
// telo lambda vyrazu
return 0;
}


Capture clause
Parametre
Voliteľný mutable špecifikátor
[](int x) mutable -> int
{
// telo lambda vyrazu
return 0;
}

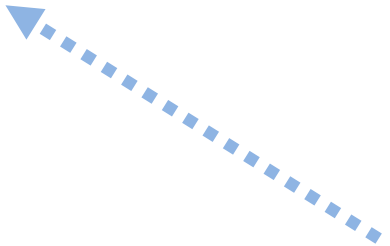

Capture clause
Parametre
Voliteľný mutable špecifikátor
Trailing-return-type
[](int x) mutable -> int
{
// telo lambda vyrazu
return 0;
}

```
[](int x) mutable -> int
{
    // telo lambda vyrazu
    return 0;
}
```

Capture clause

```
[](int x) mutable -> int
{
    // telo lambda vyrazu
    return 0;
}
```

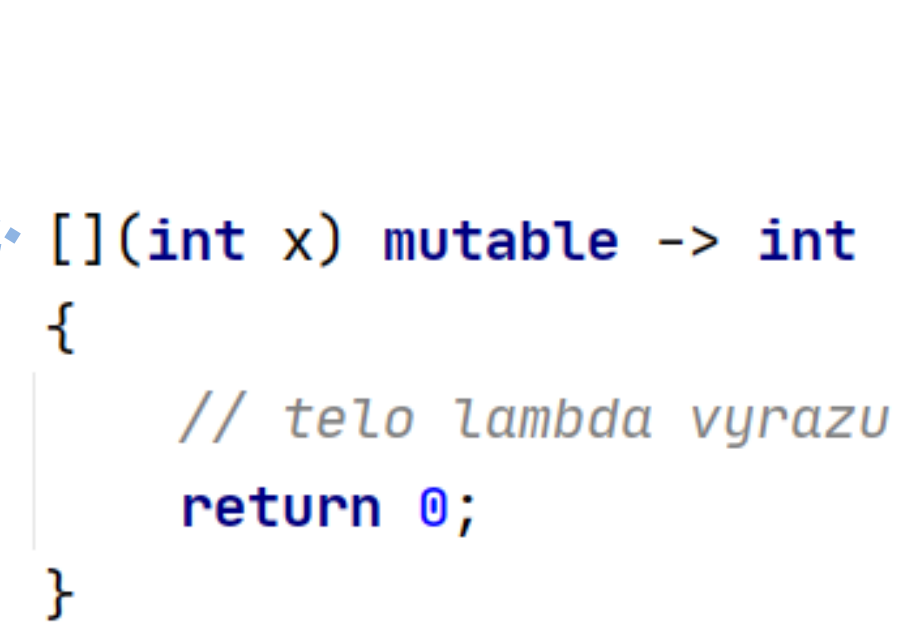
No capture `[ ]`



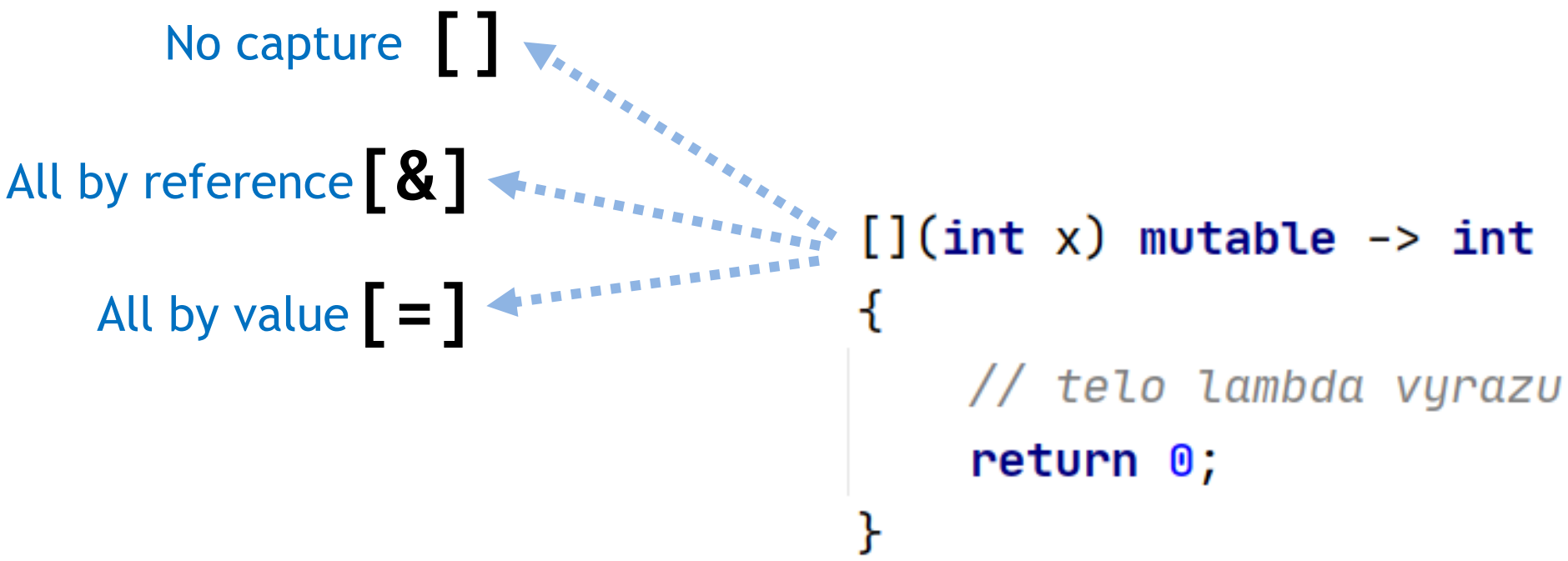
```
[](int x) mutable -> int
{
    // telo lambda vyrazu
    return 0;
}
```

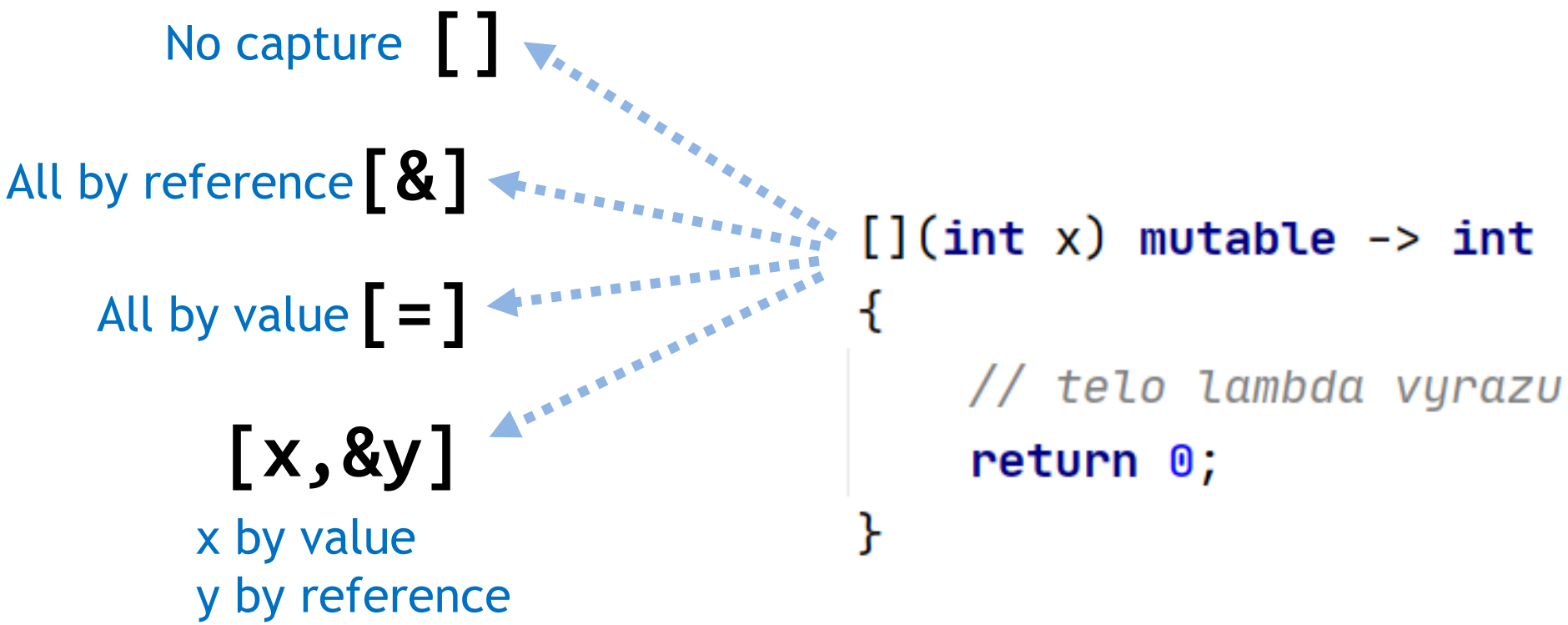
No capture `[]`

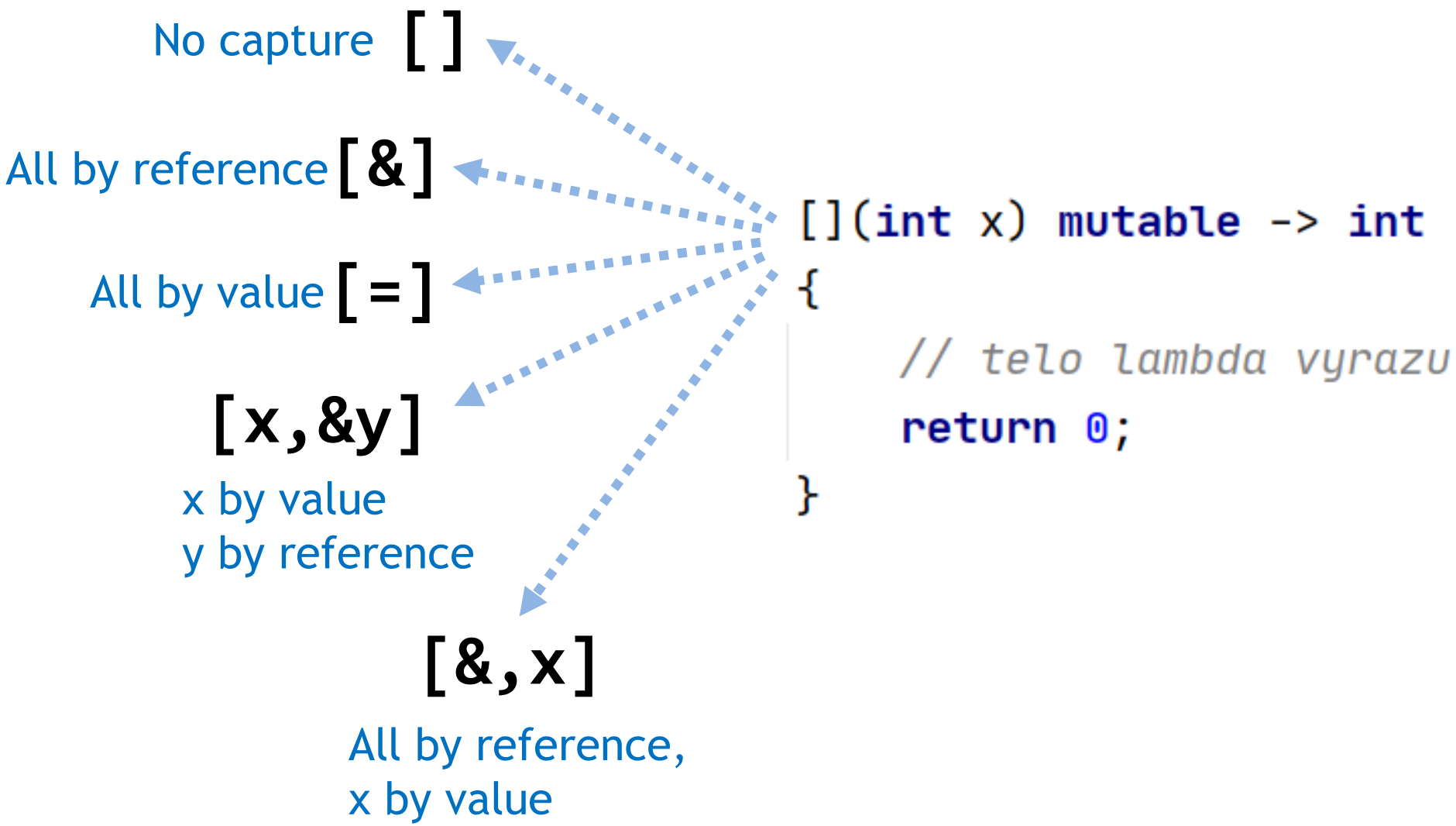
All by reference `[&]`



```
[](int x) mutable -> int
{
    // telo lambda vyrazu
    return 0;
}
```


No capture []
All by reference [&]
All by value [=]
[](int x) mutable -> int
{
// telo lambda vyrazu
return 0;
}


No capture []
All by reference [&]
All by value [=]
[x,&y]
x by value
y by reference
[](int x) mutable -> int
{
// telo lambda vyrazu
return 0;
}


No capture []
All by reference [&]
All by value [=]
[x,&y]
x by value
y by reference
[&,x]
All by reference,
x by value
[](int x) mutable -> int
{
// telo lambda vyrazu
return 0;
}

```
std::function<int(int)> lambda = [](int x) mutable -> int
{
    // telo lambda vyrazu
    return 0;
};

lambda(5);
```

```
auto lambda = [](int x) mutable -> int
{
    // telo lambda vyrazu
    return 0;
};

lambda( x: 5);
```

```
auto lambda = []() -> int (*)(int){
    return [](int a) -> int {
            return a;
    };
};
cout << lambda()(5);
```

Non-modifying

Defined in header <algorithm>

all_of (C++11) any_of (C++11) none_of (C++11)	checks if a predicate is `true` for all, any or none of the elements in a range (function template)
ranges::all_of (C++20) ranges::any_of (C++20) ranges::none_of (C++20)	checks if a predicate is `true` for all, any or none of the elements in a range (niebloid)
for_each	applies a function to a range of elements (function template)
ranges::for_each (C++20)	applies a function to a range of elements (niebloid)
for_each_n (C++17)	applies a function object to the first n elements of a sequence (function template)
ranges::for_each_n (C++20)	applies a function object to the first n elements of a sequence (niebloid)
count count_if	returns the number of elements satisfying specific criteria (function template)
ranges::count (C++20) ranges::count_if (C++20)	returns the number of elements satisfying specific criteria (niebloid)
mismatch	finds the first position where two ranges differ (function template)
ranges::mismatch (C++20)	finds the first position where two ranges differ (niebloid)
find find_if find_if_not (C++11)	finds the first element satisfying specific criteria (function template)
ranges::find (C++20) ranges::find_if (C++20) ranges::find_if_not (C++20)	finds the first element satisfying specific criteria (niebloid)

Non-modifying

`find_end`	finds the last sequence of elements in a certain range (function template)
`ranges::find_end` (C++20)	finds the last sequence of elements in a certain range (niebloid)
`find_first_of`	searches for any one of a set of elements (function template)
`ranges::find_first_of` (C++20)	searches for any one of a set of elements (niebloid)
`adjacent_find`	finds the first two adjacent items that are equal (or satisfy a given predicate) (function template)
`ranges::adjacent_find` (C++20)	finds the first two adjacent items that are equal (or satisfy a given predicate) (niebloid)
`search`	searches for a range of elements (function template)
`ranges::search` (C++20)	searches for a range of elements (niebloid)
`search_n`	searches a range for a number of consecutive copies of an element (function template)
`ranges::search_n` (C++20)	searches for a number consecutive copies of an element in a range (niebloid)

Modifying

Defined in header <algorithm>

copy **copy_if** (C++11)	copies a range of elements to a new location (function template)
ranges::copy (C++20) **ranges::copy_if** (C++20)	copies a range of elements to a new location (niebloid)
copy_n (C++11)	copies a number of elements to a new location (function template)
ranges::copy_n (C++20)	copies a number of elements to a new location (niebloid)
copy_backward	copies a range of elements in backwards order (function template)
ranges::copy_backward (C++20)	copies a range of elements in backwards order (niebloid)
move (C++11)	moves a range of elements to a new location (function template)
ranges::move (C++20)	moves a range of elements to a new location (niebloid)
move_backward (C++11)	moves a range of elements to a new location in backwards order (function template)
ranges::move_backward (C++20)	moves a range of elements to a new location in backwards order (niebloid)
fill	copy-assigns the given value to every element in a range (function template)
ranges::fill (C++20)	assigns a range of elements a certain value (niebloid)
fill_n	copy-assigns the given value to N elements in a range (function template)
ranges::fill_n (C++20)	assigns a value to a number of elements (niebloid)

Modifying

`transform`	applies a function to a range of elements, storing results in a destination range (function template)
`ranges::transform` (C++20)	applies a function to a range of elements (niebloid)
`generate`	assigns the results of successive function calls to every element in a range (function template)
`ranges::generate` (C++20)	saves the result of a function in a range (niebloid)
`generate_n`	assigns the results of successive function calls to N elements in a range (function template)
`ranges::generate_n` (C++20)	saves the result of N applications of a function (niebloid)
`remove` `remove_if`	removes elements satisfying specific criteria (function template)
`ranges::remove` (C++20) `ranges::remove_if` (C++20)	removes elements satisfying specific criteria (niebloid)
`remove_copy` `remove_copy_if`	copies a range of elements omitting those that satisfy specific criteria (function template)
`ranges::remove_copy` (C++20) `ranges::remove_copy_if` (C++20)	copies a range of elements omitting those that satisfy specific criteria (niebloid)
`replace` `replace_if`	replaces all values satisfying specific criteria with another value (function template)
`ranges::replace` (C++20) `ranges::replace_if` (C++20)	replaces all values satisfying specific criteria with another value (niebloid)

Modifying

`replace_copy` `replace_copy_if`	copies a range, replacing elements satisfying specific criteria with another value (function template)
`ranges::replace_copy` (C++20) `ranges::replace_copy_if` (C++20)	copies a range, replacing elements satisfying specific criteria with another value (niebloid)
`swap`	swaps the values of two objects (function template)
`swap_ranges`	swaps two ranges of elements (function template)
`ranges::swap_ranges` (C++20)	swaps two ranges of elements (niebloid)
`iter_swap`	swaps the elements pointed to by two iterators (function template)
`reverse`	reverses the order of elements in a range (function template)
`ranges::reverse` (C++20)	reverses the order of elements in a range (niebloid)
`reverse_copy`	creates a copy of a range that is reversed (function template)
`ranges::reverse_copy` (C++20)	creates a copy of a range that is reversed (niebloid)
`rotate`	rotates the order of elements in a range (function template)
`ranges::rotate` (C++20)	rotates the order of elements in a range (niebloid)
`rotate_copy`	copies and rotate a range of elements (function template)
`ranges::rotate_copy` (C++20)	copies and rotate a range of elements (niebloid)
`shift_left` `shift_right` (C++20)	shifts elements in a range (function template)

Modifying

random_shuffle (until C++17) **shuffle** (C++11)	randomly re-orders elements in a range (function template)
ranges::shuffle (C++20)	randomly re-orders elements in a range (niebloid)
sample (C++17)	selects n random elements from a sequence (function template)
ranges::sample (C++20)	selects n random elements from a sequence (niebloid)
unique	removes consecutive duplicate elements in a range (function template)
ranges::unique (C++20)	removes consecutive duplicate elements in a range (niebloid)
unique_copy	creates a copy of some range of elements that contains no consecutive duplicates (function template)
ranges::unique_copy (C++20)	creates a copy of some range of elements that contains no consecutive duplicates (niebloid)

Partitioning

Defined in header <algorithm>

is_partitioned (C++11)	determines if the range is partitioned by the given predicate (function template)
ranges::is_partitioned (C++20)	determines if the range is partitioned by the given predicate (niebloid)
partition	divides a range of elements into two groups (function template)
ranges::partition (C++20)	divides a range of elements into two groups (niebloid)
partition_copy (C++11)	copies a range dividing the elements into two groups (function template)
ranges::partition_copy (C++20)	copies a range dividing the elements into two groups (niebloid)
stable_partition	divides elements into two groups while preserving their relative order (function template)
ranges::stable_partition (C++20)	divides elements into two groups while preserving their relative order (niebloid)
partition_point (C++11)	locates the partition point of a partitioned range (function template)
ranges::partition_point (C++20)	locates the partition point of a partitioned range (niebloid)

Sorting

Defined in header <algorithm>

is_sorted (C++11)	checks whether a range is sorted into ascending order (function template)
ranges::is_sorted (C++20)	checks whether a range is sorted into ascending order (niebloid)
is_sorted_until (C++11)	finds the largest sorted subrange (function template)
ranges::is_sorted_until (C++20)	finds the largest sorted subrange (niebloid)
sort	sorts a range into ascending order (function template)
ranges::sort (C++20)	sorts a range into ascending order (niebloid)
partial_sort	sorts the first N elements of a range (function template)
ranges::partial_sort (C++20)	sorts the first N elements of a range (niebloid)
partial_sort_copy	copies and partially sorts a range of elements (function template)
ranges::partial_sort_copy (C++20)	copies and partially sorts a range of elements (niebloid)
stable_sort	sorts a range of elements while preserving order between equal elements (function template)
ranges::stable_sort (C++20)	sorts a range of elements while preserving order between equal elements (niebloid)
nth_element	partially sorts the given range making sure that it is partitioned by the given element (function template)
ranges::nth_element (C++20)	partially sorts the given range making sure that it is partitioned by the given element (niebloid)

Binary search

Defined in header <algorithm>

lower_bound	returns an iterator to the first element *not less* than the given value (function template)
ranges::lower_bound (C++20)	returns an iterator to the first element *not less* than the given value (niebloid)
upper_bound	returns an iterator to the first element *greater* than a certain value (function template)
ranges::upper_bound (C++20)	returns an iterator to the first element *greater* than a certain value (niebloid)
binary_search	determines if an element exists in a certain range (function template)
ranges::binary_search (C++20)	determines if an element exists in a certain range (niebloid)
equal_range	returns range of elements matching a specific key (function template)
ranges::equal_range (C++20)	returns range of elements matching a specific key (niebloid)

Merge

Defined in header <algorithm>

merge	merges two sorted ranges (function template)
ranges::merge (C++20)	merges two sorted ranges (niebloid)
inplace_merge	merges two ordered ranges in-place (function template)
ranges::inplace_merge (C++20)	merges two ordered ranges in-place (niebloid)

Set operations

Defined in header <algorithm>

includes	returns true if one sequence is a subsequence of another (function template)
ranges::includes (C++20)	returns true if one sequence is a subsequence of another (niebloid)
set_difference	computes the difference between two sets (function template)
ranges::set_difference (C++20)	computes the difference between two sets (niebloid)
set_intersection	computes the intersection of two sets (function template)
ranges::set_intersection (C++20)	computes the intersection of two sets (niebloid)
set_symmetric_difference	computes the symmetric difference between two sets (function template)
ranges::set_symmetric_difference (C++20)	computes the symmetric difference between two sets (niebloid)
set_union	computes the union of two sets (function template)
ranges::set_union (C++20)	computes the union of two sets (niebloid)

Heap operations

Defined in header <algorithm>

is_heap (C++11)	checks if the given range is a max heap (function template)
ranges::is_heap (C++20)	checks if the given range is a max heap (niebloid)
is_heap_until (C++11)	finds the largest subrange that is a max heap (function template)
ranges::is_heap_until (C++20)	finds the largest subrange that is a max heap (niebloid)
make_heap	creates a max heap out of a range of elements (function template)
ranges::make_heap (C++20)	creates a max heap out of a range of elements (niebloid)
push_heap	adds an element to a max heap (function template)
ranges::push_heap (C++20)	adds an element to a max heap (niebloid)
pop_heap	removes the largest element from a max heap (function template)
ranges::pop_heap (C++20)	removes the largest element from a max heap (niebloid)
sort_heap	turns a max heap into a range of elements sorted in ascending order (function template)
ranges::sort_heap (C++20)	turns a max heap into a range of elements sorted in ascending order (niebloid)

Min/Max operations

Defined in header <algorithm>

max	returns the greater of the given values (function template)
ranges::max (C++20)	returns the greater of the given values (niebloid)
max_element	returns the largest element in a range (function template)
ranges::max_element (C++20)	returns the largest element in a range (niebloid)
min	returns the smaller of the given values (function template)
ranges::min (C++20)	returns the smaller of the given values (niebloid)
min_element	returns the smallest element in a range (function template)
ranges::min_element (C++20)	returns the smallest element in a range (niebloid)
minmax (C++11)	returns the smaller and larger of two elements (function template)
ranges::minmax (C++20)	returns the smaller and larger of two elements (niebloid)
minmax_element (C++11)	returns the smallest and the largest elements in a range (function template)
ranges::minmax_element (C++20)	returns the smallest and the largest elements in a range (niebloid)
clamp (C++17)	clamps a value between a pair of boundary values (function template)
ranges::clamp (C++20)	clamps a value between a pair of boundary values (niebloid)

Other

Defined in header <algorithm>

equal	determines if two sets of elements are the same (function template)
ranges::equal (C++20)	determines if two sets of elements are the same (niebloid)
lexicographical_compare	returns true if one range is lexicographically less than another (function template)
ranges::lexicographical_compare (C++20)	returns true if one range is lexicographically less than another (niebloid)
lexicographical_compare_three_way (C++20)	compares two ranges using three-way comparison (function template)
is_permutation (C++11)	determines if a sequence is a permutation of another sequence (function template)
ranges::is_permutation (C++20)	determines if a sequence is a permutation of another sequence (niebloid)
next_permutation	generates the next greater lexicographic permutation of a range of elements (function template)
ranges::next_permutation (C++20)	generates the next greater lexicographic permutation of a range of elements (niebloid)
prev_permutation	generates the next smaller lexicographic permutation of a range of elements (function template)
ranges::prev_permutation (C++20)	generates the next smaller lexicographic permutation of a range of elements (niebloid)

Numeric

Defined in header <numeric>

iota (C++11)	fills a range with successive increments of the starting value (function template)
accumulate	sums up a range of elements (function template)
inner_product	computes the inner product of two ranges of elements (function template)
adjacent_difference	computes the differences between adjacent elements in a range (function template)
partial_sum	computes the partial sum of a range of elements (function template)
reduce (C++17)	similar to `std::accumulate`, except out of order (function template)
exclusive_scan (C++17)	similar to `std::partial_sum`, excludes the ith input element from the ith sum (function template)
inclusive_scan (C++17)	similar to `std::partial_sum`, includes the ith input element in the ith sum (function template)
transform_reduce (C++17)	applies an invocable, then reduces out of order (function template)
transform_exclusive_scan (C++17)	applies an invocable, then calculates exclusive scan (function template)
transform_inclusive_scan (C++17)	applies an invocable, then calculates inclusive scan (function template)

Vzorová implementácia v C/C++

Použité zdroje

Literatúra

- http://cplusplus.com/reference/
- https://en.cppreference.com/w/

Obrázky

- https://www.flaticon.com/free-icon/hacker_1106631