Алгоритмизация и программирование

Лекция 4

Перечисления

enum (перечисления)

```
void set_color(/*какой тип?*/ color){
    /* Код */
}

void set_color(int color){
    /* Код */
}

void set_color(std::string color){
    /* Код */
}
```

enum

```
enum Color
{
    // Список перечислителей
    color_black,
    color_red,
    color_blue,
    color_green,
    color_white,
    color_cyan,
    color_yellow,
    color_magenta, //<- запятую можно оставить
}; // <- точкой с запятой

// Определяем несколько переменных перечислимого типа Color
Color paint = color_white;
Color house(color_blue);
Color apple { color_red };
```

enum

```
void set_color(Color color){
    /* Код */
}

set_color(color_red); // нормально
set_color(1);         // ошибка
```

enum

```
enum Color
{
  red,
  blue, // blue помещается в глобальное пространство имен
  green
};

enum Feeling
{
  happy,
  tired,
  blue // ошибка, blue уже использовался в enum Color в глобальном пространстве имен
};
```

enum

```
// определяем новое перечисление с именем Animal
enum Animal
{
    animal_cat = -3,
    animal_dog, // присвоено -2
    animal_pig, // присвоено -1
    animal_horse = 5,
    animal_giraffe = 5, // имеет то же значение, что и animal_horse
    animal_chicken // присвоено 6
};

int pet = animal_pig;
Animal mypet = animal_pig;

std::cout << mypet; // перед передачей в std::cout вычисляется как int
std::cin >> mypet;  // вызовет ошибку компилятора
```

enum

```
// Использовать в качестве базы для перечисления
// 8-битный целочисленный тип без знака.
enum Color : std::uint_least8_t
{
    color_black,
    color_red,
    // ...
};
```

enum class

```
enum class Color
{
  red,
  blue, // blue помещается в пространство имен Color
  green
};

enum class Feeling
{
  happy,
  tired,
  blue // нормально, blue помещается в пространство имен Feeling
};

Color color = Color::blue;
Color color = blue; // ошибка
```

Объединения

union (объединения)

```
// Структура
struct building
{
    std::string owner;
    std::string city;
    int amountRooms;
    float price;
};

building apartment;
apartment.owner = "John";
apartment.city = "NY";
apartment.amountRooms = 10;
apartment.price = 100;
```

```
// Объединение
union building
{
    std::string owner;
    std::string city;
    int amountRooms;
    float price;
};

building apartment;
// Только что-то одно
```

Объединение занимает в памяти столько места, сколько занимает самое большое его поле.
После инициализации менять тип данных перечисления нельзя (хотя иногда можно)

Указатели

Указатель

Составной тип данных предназначенный для хранения адреса некоторой программной сущности:

```
int a;                  // Хранит целое число
int* ptr_a = nullptr; // Хранит адрес переменной типа int

std::string str;        // Хранит строку
std::string* ptr_str; // Хранит адрес переменной типа std::string

double*  ptr_d;         // Хранит адрес переменной типа double
double** ptr_ptr_d;   // Хранит адрес переменной типа double*
```

& | Оператор взятия адреса

Позволяет узнать адрес в памяти, по которому находится программная сущность:

```
int a;
int* ptr_a = &a;

std::string str;
std::string* ptr_str = &str;

double d;
double*  ptr_d = &d;
double** ptr_ptr_d = &ptr_d;

ptr_d = &a;      // Не соответствие типов
ptr_a = ptr_str; // Не соответствие типов

ptr_a = a;       // Не соответствие типов
a = ptr_a;       // Не соответствие типов
```

* | Оператор разыменования (dereference)

Позволяет получить доступ к программной сущности по адресу:

```
int a;
int* ptr_a = &a;
int b = *ptr_a;

double d;
double*  ptr_d = &d;
double** ptr_ptr_d = &ptr_d;
double c = **ptr_ptr_d;

*ptr_a = 10;  // Значение a изменилось на 10
```

Адресная арифметика

Позволяет получить доступ к другому адресу:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int* ptr_a = &a[0];
int* end_a = &a[9];

ptr_a = ptr_a + 1; // a[1]
ptr_a = ptr_a + 2; // a[3]
ptr_a++;           // a[4]
ptr_a--;           // a[3]
ptr_a = ptr_a - 3; // a[0]

int size = end_a - ptr_a; // 10
```

Ссылки

Ссылка (на l-value)

Альтернативное имя для переменной:

```
int a;            // Хранит целое число
int& fer_a = a;   // Ничего не хранит, просто имя

std::string str;              // Хранит строку
std::string& fef_str = str; // Ничего не хранит, просто имя

int& fer_a2 = fer_a;  // Тоже ссылка на a

int& fer_a;           // Нельзя
int&& fer_fer_a;      // Другой тип ссылки
int&&& fer_fer_fer_a; // Не существует
```

Псевдонимы

Псевдоним | альтернативное имя

```
// typedef
typedef int integer;
integer a = 1;

typedef int vector[10];
vector vect;
vector matrix[5]; // arr[5][10]

typedef int (*foo)(int a, int b);

// using
using integer = int;

using vector = int[10];
using matrix = vector[5];

using foo = int (*)(int a, int b);
```

Типы данных из STL

std::string

#include <string>

h/cpp hackingcpp.com

```
string s = "I'm sorry, Dave.";
```

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 indices

non-mutating

s.size()	→ 16	(number of characters)
s[2]	→ 'm'	(character at index 2)
s.find("r")	→ 6	(first match from start)
s.rfind("r")	→ 7	(first match from end)
s.find("X")	→ string::npos	(not found, invalid index)
s.find(' ', 5)	→ 10	(first match after index ≥ 5)
s.substr(4, 6)	→ string{"sorry,"}	
s.contains("sorry")	→ *true*	(C++23)
s.starts_with('I')	→ *true*	(C++20)
s.ends_with("Dave.")	→ *true*	(C++20)
s.compare("I'm sorry, Dave.")	→ 0	(identical)
s.compare("I'm sorry, Anna.")	→ > 0	(same length, but 'D' > 'A')
s.compare("I'm sorry, Saul.")	→ < 0	(same length, but 'D' < 'S')

mutating

	s += " I'm afraid I can't do that."	⇒	s = "I'm sorry, Dave. I'm afraid I can't do that."
	s.append("..")	⇒	s = "I'm sorry, Dave..."
size	s.clear()	⇒	s = ""
size	s.resize(3)	⇒	s = "I'm"
size	s.resize(20, '?')	⇒	s = "I'm sorry, Dave.????";
index based	s.insert(4, "very ")	⇒	s = "I'm very sorry, Dave."
index based	s.erase(5, 2)	⇒	s = "I'm sry, Dave."
index based	s[15] = '!'	⇒	s = "I'm sorry, Dave!"
index based	s.replace(11, 5, "Frank")	⇒	s = "I'm sorry, Frank"
iterator based	s.insert(s.begin(), "HAL: ")	⇒	s = "HAL: I'm sorry, Dave."
iterator based	s.insert(s.begin()+4, "very ")	⇒	s = "I'm very sorry, Dave."
iterator based	s.erase(s.begin()+5)	⇒	s = "I'm srry, Dave."
iterator based	s.erase(s.begin(), s.begin()+4)	⇒	s = "sorry, Dave."

Constructors

```
string {'a','b','c'}                  ➡ a b c
string (4, '$')                       ➡ $ $ $ $
string (@firstIn, @lastIn)            ➡ e f g h
       source  iterator range
  b c d e f g h i j
string ( a b c d )   copy/move        ➡ a b c d
         source string object
```

Obtain Iterators or Reverse Iterators

```
.begin()  → @first
  a b c d e f
.end()    → @one_behind_last
  don't use to access elements!
  a b c d e f
```

```
.rbegin() → reverse@last .base()
  a b c d e f
.rend()   → reverse@one_before_first
  don't use to access elements!   .base()
  a b c d e f
```

String → Number Conversion

```
int         stoi (●,●,●);
long        stol (●,●,●);
long long   stoll(●,●,●);

unsigned long       stoul (●,●,●);
unsigned long long  stoull(●,●,●);

float       stof (●,●,●);
double      stod (●,●,●);
long double stold(●,●,●);
```

- ● = `const string&` input string
- ● = `std::size_t* p = nullptr` output for number of processed characters
- ● = `int base = 10` base of target system; default: decimal

Number → String Conversion

```
string to_string( ● );
```

● = int | long | long long | unsigned | unsinged long | unsigned long long | float | double | long double

v1.6

Последовательные контейнеры

C++ Standard Library **Sequence Containers**

`array<T,size>`

fixed-size array

`#include <array>`

```
std::array<int,6> a {1,2,3,4,5,6};
cout << a.size();    // 6
cout << a[2];        // 3
a[0] = 7;            // 1st element ⇒ 7
```

contiguous memory; random access; fast linear traversal

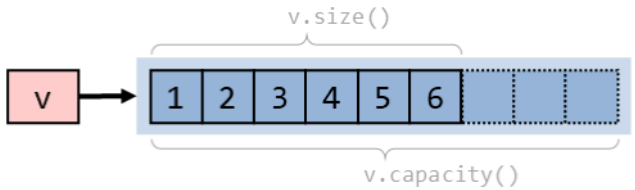
`vector<T>`

dynamic array

C++'s "default" container

`#include <vector>`

```
std::vector<int> v {1,2,3,4,5,6};
v.reserve(9);
cout << v.capacity();      // 9
cout << v.size();          // 6
v.push_back(7);            // appends '7'
v.insert(v.begin(), 0);    // prepends '0'
v.pop_back();              // removes last
v.erase(v.begin()+2);      // removes 3rd
v.resize(20, 0);           // size ⇒ 20
```



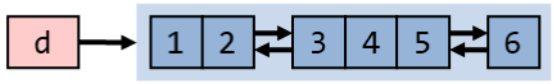
contiguous memory; random access;
fast linear traversal; fast insertion/deletion at the ends

`deque<T>`

double-ended queue

`#include <deque>`

```
std::deque<int> d {1,2,3,4,5,6};
// same operations as vector
// plus fast growth/deletion at front
d.push_front(-1);   // prepends '-1'
d.pop_front();      // removes 1st
```



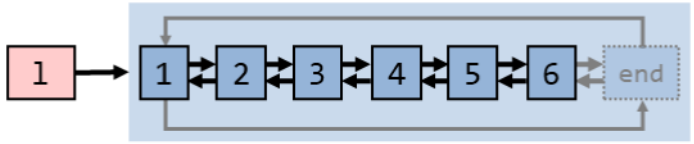
fast insertion/deletion at both ends

`list<T>`

doubly-linked list

`#include <list>`

```
std::list<int> l {1,5,6};
std::list<int> k {2,3,4};
// O(1) splice of k into l:
l.splice(l.begin()+1, std::move(k))
// some special member function algorithms:
l.reverse();
l.sort();
```



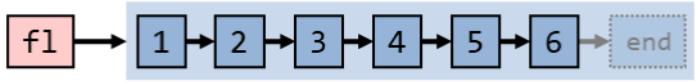
fast splicing; many operations without copy/move of elements

`forward_list<T>`

singly-linked list

`#include <forward_list>`

```
std::forward_list<int> fl {2,2,4,5,6};
fl.erase_after(begin(fl));
fl.insert_after(begin(fl), 3);
fl.insert_after(before_begin(fl), 1);
```



lower memory overhead than std::list; only forward traversal

std::vector<*ValueType*>

C++'s "default" dynamic array

`#include <vector>`

h/cpp hackingcpp.com

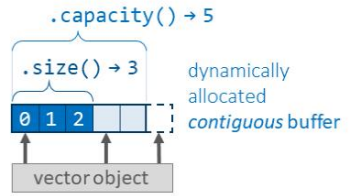
Construct A New Vector Object

```
vector<int> v1 {2,9,1,8,5,4}
vector<int> v2 (begin(v1)+3, end(v1))
vector<int> v3 (5,3)
vector<int> deep_copy_of_v1 (v1)
```

C++17 value type deducable from argument type

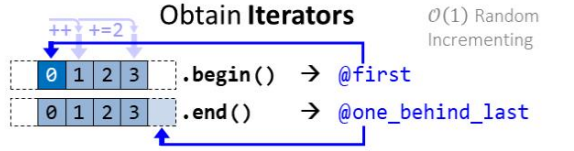
```
vector w {7,4,2};   // vector<int>
```

Typical Memory Layout

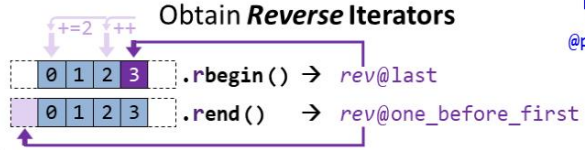


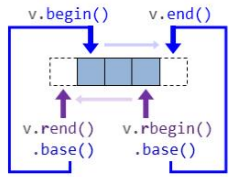
Obtain Iterators

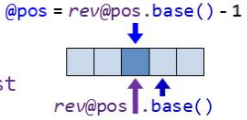
$\mathcal{O}(1)$ Random Incrementing



Obtain *Reverse* Iterators





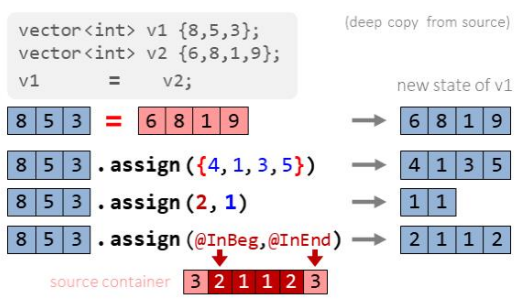


Assign New Content To An Existing Vector

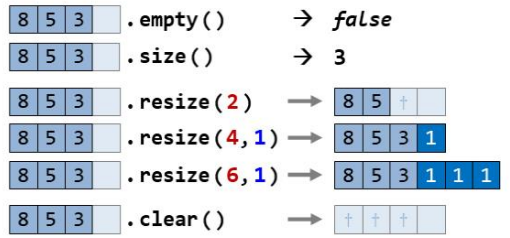
```
vector<int> v1 {8,5,3};
vector<int> v2 {6,8,1,9};
v1      =     v2;
```

(deep copy from source)

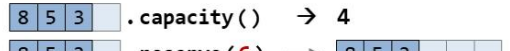
new state of v1



Query/Change Size (= Number of Elements)



Query/Grow Capactiy (= Memory Buffer Size)



Avoid expensive memory allocations: `.reserve` capacity before appending / inserting if you know the (approximate) number of elements to be stored in advance!

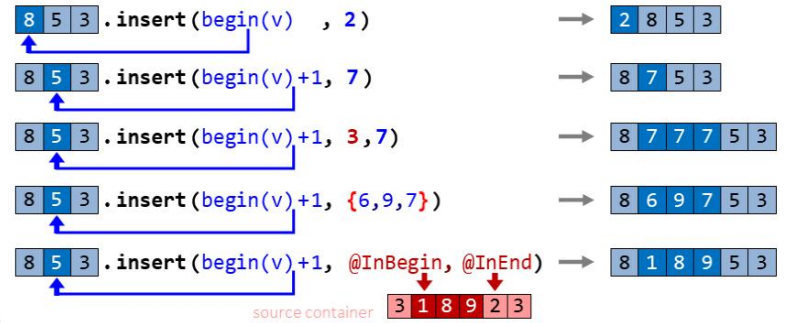
Append Elements

$\mathcal{O}(1)$ Amortized Complexity

Insert Elements at Arbitrary Positions

$\mathcal{O}(n)$ Worst Case

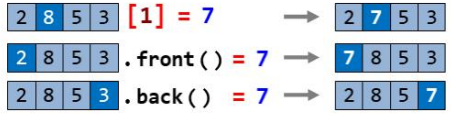
```
vector<int> v {8,5,3};
```



Get Element Values

$\mathcal{O}(1)$ Random Access

Change Element Values

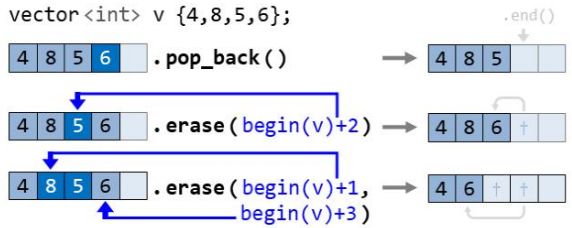


Out of Bounds Access

Erase Elements

$\mathcal{O}(n)$ Worst Case

```
vector<int> v {4,8,5,6};
```



Shrink The Capacity

(might be inefficient)

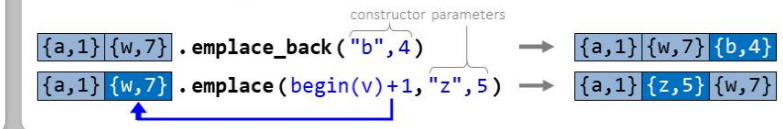
Erasing, resizing or clearing will not shrink the capacity!

```
vector<int> v (1024,0);   // capacity is at least 1024
v.resize(40);             // capacity unchanged!
v.shrink_to_fit();        // may shrink (not guaranteed)
v.swap(vector<int>(v));   // shrinks but has copy overhead
```

Insert & Construct Elements in Place

$\mathcal{O}(n)$ Worst Case

```
vector<pair<string,int>> v {{"a",1},{"w",7}};
```



v1.1

Ассоциативные контейнеры

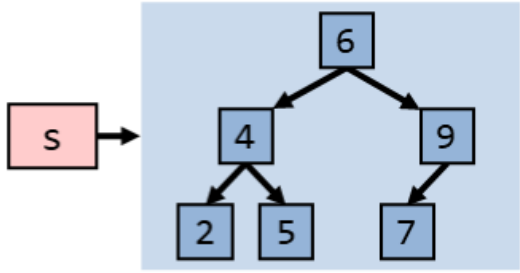
set<*Key*>

unique, ordered keys

multiset<*K*>

(non-unique) ordered keys

```
std::set<int> s;
s.insert(7); …
s.insert(5);
auto i = s.find(7);  // → iterator
if(i != s.end())     // found?
  cout << *i;        // 7
if(s.contains(7)) {…}     C++20
```



usually implemented as balanced binary tree (red-black tree)

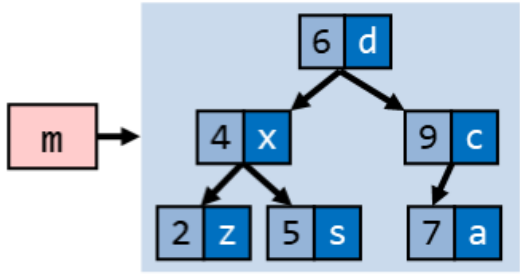
map<*Key,Value*>

unique key → value-pairs; ordered by keys

multimap<*K,V*>

(non-unique) key → value-pairs, ordered by keys

```
std::map<int,char> m;
m.insert({7,'a'});  …
m[4] = 'x';              // insert 4→x
auto i = s.find(7);  // → iterator
if(i != s.end())     // found?
  cout << i->first   // 7
       << i->second; // a
if(s.contains(7)) {…}     C++20
```



usually implemented as balanced binary tree (red-black tree)

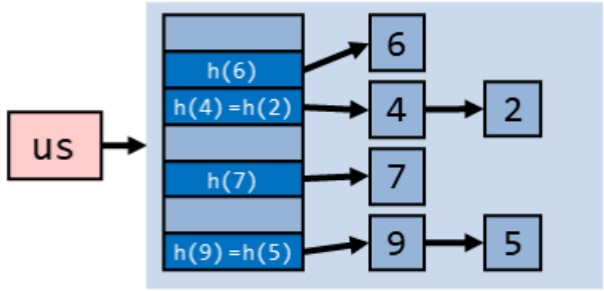
unordered_set<*Key*>

unique, hashable keys

unordered_multiset<*Key*>

(non-unique) hashable keys

```
std::unordered_set<int> us;
us.insert(7); …
us.insert(5);
auto i = us.find(7); // → iterator
if(i != us.end())    // found?
  cout << *i;        // 7
if(s.contains(7)) {…}     C++20
```



hash table for key lookup, linked nodes for key storage

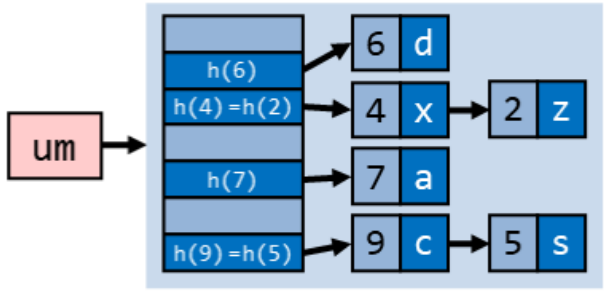
unordered_map<*Key,Value*>

unique key → value-pairs; hashed by keys

unordered_multimap<*Key,Value*>

(non-unique) key → value-pairs; hashed by keys

```
unordered_map<int,char> um;
um.insert({7,'a'});  …
um[4] = 'x';             // insert 4→x
auto i = um.find(7); // → iterator
if(i != um.end())    // found?
  cout << i->first   // 7
       << i->second; // a
if(s.contains(7)) {…}     C++20
```

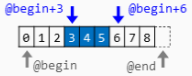


hash table for key lookup, linked nodes for (key, value) pair storage

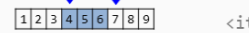
Некоторые стандартные алгоритмы

C++ Standard Library Algorithms

Iterator Ranges



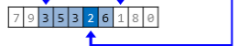
`distance(@begin, @end) → element_count`



<iterator>

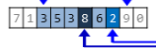
Non-Modifying Sequence Operations

`min_element(@begin, @end) → @minimum`



`minmax_element(@begin, @end) → {@minimum, @maximum}`

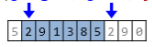
C++11



`any_of (@begin, @end, f(○)↦bool) →`
`all_of (@begin, @end, f(○)↦bool) →`
`none_of(@begin, @end, f(○)↦bool) →`

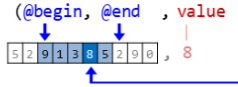
- true, if *f* yields true for *any*, *all* or *none* elements in the input range
- false otherwise

C++11



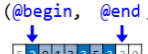
`find_if(@begin, @end , f(○)↦bool)→`
`find (@begin, @end , value )→`

- @1st match
- @end if no match

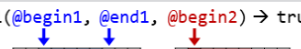


`count_if(@begin, @end , f(○)↦bool) →`
`count (@begin, @end , value ) →`

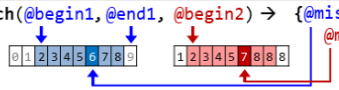
number of occurrences



`equal(@begin1, @end1, @begin2) → true` if all elements in both ranges are equal

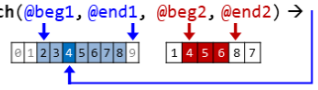


`mismatch(@begin1, @end1, @begin2) → {@mismatch_in1, @mismatch_in2}`



`search(@beg1, @end1, @beg2, @end2) →`

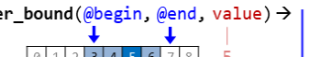
- @1st occurrence of sequence 2 inside sequence 1
- @end1 otherwise



Binary Search On *Sorted* Sequences ⇒ $\mathcal{O}(\log n)$

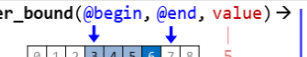
`lower_bound(@begin, @end, value) →`

- @1st element not < value
- @end if no such element exists



`upper_bound(@begin, @end, value) →`

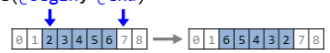
- @1st element > value
- @end if no such element exists



`equal_range(@begin, @end, value) → {@1st item not < value or @end if none such found), @1st item > value or @end if none such found)}`

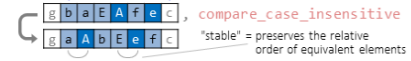
Reordering Elements

`reverse(@begin, @end)`

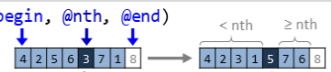


`sort(@begin, @end, f(○,○)↦bool)`
`sort(@begin, @end, std::less )`

`stable_sort(@begin, @end, compare(○,○)↦bool)`
`=○<○`



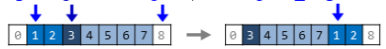
`nth_element(@begin, @nth, @end)`



element at nth position → element that would be in that position in a sorted sequence

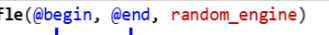
`partition(@begin, @end, f(○)↦bool) → @part2`

`rotate(@begin, @newfst, @end) → @old_begin`



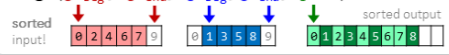
`next_permutation(@begin, @end) → true` if new permutation is lexicographically greater

`shuffle(@begin, @end, random_engine)`

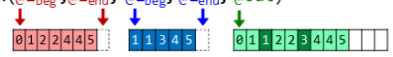


Manipulate *Sorted* Sequences ⇒ $\mathcal{O}(n)$

`merge(@1beg, @1end, @2beg, @2end, @out)`

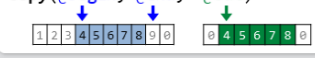


`set_union(@1beg, @1end, @2beg, @2end, @out)`



Changing Values

`copy(@begin, @end, @out)`

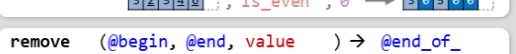


`transform(@begin, @end, @out, f(○)↦■)`

`generate(@begin, @end, f()↦■)`

`replace(@begin, @end, old, new)`

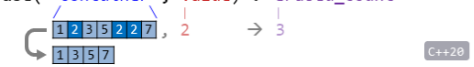
`replace_if(@begin, @end, f(○)↦bool, new)`



`remove (@begin, @end, value ) → @end_of_`
`remove_if(@begin, @end, f(○)↦bool) → remaining`

`unique(@begin, @end) → @end_of_remaining`

`erase( container , value) → erased_count`



C++20

Numeric Algorithms

<numeric>

`reduce(@begin, @end, ω = ●{}, ⊕ = ○+○)`
`reduce(@begin, @end, ω, ⊕(□,○)↦■) } → ω+●2+●0+●n +⋯+●1+●3`

C++17

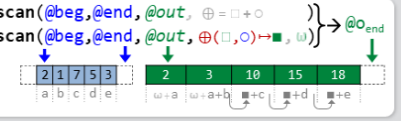
arbitrary evaluation order!

`transform_reduce(@●,@●, @♦,ω,⊕ = □+○ ,⊗ = ○×◊) }→ R⊗`
`transform_reduce(@●,@●, @♦,ω,⊕(□,Δ)↦■,⊗(○,◊)↦▲) }`
`transform_reduce(@●,@●, ω,⊕(□,Δ)↦■,f(○)↦▲) → Rf`

arbitrary evaluation order!

C++17 $R_f = \omega + f(\bullet_0) + f(\bullet_1) + \cdots$ $R_\otimes = \omega + (\bullet_0 \times \blacklozenge_0) + (\bullet_1 \times \blacklozenge_1) + \cdots$

`inclusive_scan(@beg,@end,@out, ⊕ = □+○ ) }→ @oend`
`inclusive_scan(@beg,@end,@out, ⊕(□,○)↦■, ω) }`



C++17

Sequence Queries

all_of	(C++11)
any_of	(C++11)
none_of	(C++11)
count	count_if
find	find_if
find_if_not	(C++11)
find_end	
find_first_of	
adjacent_find	
for_each	
for_each_n	(C++17)
sample	(C++20)
equal	
mismatch	
search	
search_n	
lexicographical_compare	
lexicographical_compare_three_way	(C++20)

Reordering Elements

reverse	reverse_copy
rotate	rotate_copy
shift_left (C++20)	shift_right (C++20)
shuffle (C++11)	
swap	
swap_ranges	
iter_swap	

Partitioning

is_partitioned	
partition	
stable_partition	
partition_copy	
partition_point	

Permutations

is_permutation (C++11)	
next_permutation	prev_permutation

Sorting

sort	
stable_sort	
partial_sort	
partial_sort_copy	
is_sorted	
is_sorted_until	
nth_element	

Changing Elements

copy	
copy_backward	
copy_if	(C++11)
copy_n	(C++11)
move	(C++11)
move_backward	(C++11)
fill	
fill_n	
generate	
generate_n	
transform	
replace	replace_if
replace_copy	replace_copy_if
remove	remove_if
remove_copy	remove_copy_if
unique	
unique_copy	

Binary Search on Sorted Ranges

binary_search	
lower_bound	upper_bound
equal_range	
includes	

Merging of Sorted Ranges

merge	
inplace_merge	
set_union	
set_intersection	
set_difference	
set_symmetric_difference	

Heaps

make_heap	
sort_heap	
push_heap	
pop_heap	
is_heap	(C++11)
is_heap_until	(C++11)

Minimum/Maximum

min	max
min_element	max_element
minmax	(C++11)
minmax_element	(C++11)
clamp (C++17)	

Numeric

#include <numeric>

accumulate	
adjacent_difference	
inner_product	
partial_sum	
iota	(C++11)
reduce	(C++17)
inclusive_scan	(C++17)
exclusive_scan	(C++17)
transform_reduce	(C++17)
transform_inclusive_scan	(C++17)
transform_exclusive_scan	(C++17)

h/cpp hackingcpp.com

v3.0