# 

Функтори та алгоритми

Алгоритми

Однією з важливих властивостей та власно й цілей створення бібліотеки шаблонів було створення можливстей для стандартного використання різноманітних алгоритмів до різних типів даних. Деякі з цих алгоритмів вже містяться в деяких контейнерах. Зокрема, в контейнері Список містяться методи сортування та злиття відсортованих списків, в асоціативних контейнерах є методи пошуку, а пріорітетна черга автоматично сортує дані. Однак, інколи було б бажано засстосовувати сортування, бінарний пошук, знаходити загальну суму елементів колекції, застосовувати якусь функцію до кожного елементу контейнеру незалежно від типу контейнеру або взагалі застосувати її до звичайного масиву.

З цією метою до стандартної бібліотеки Сі++ було додано дві бібліотеки algortithms та numeric, які містять функції, що реалізують деякі з відомих та потрібних у практичному програмуванні алгоритмів.

Ці бібліотеки дозоляють не витрачати час на написання циклів для виконання популярних задач таких як сортування, пошук або підрахунок кількості елементів масиву. Це дозволяє також не витрачати час на оптимізацію та відлагдження цих алгоритмів. Крім того деякі з цих реалізацій дозволяють використовувати розпаралелювання — можливість виконивати їх в декілька потоків для пришвидшення.

Важливою особливістю стандартної бібліотеки C++ є те, що вона не тільки визначає синтаксис і семантику узагальнених алгоритмів, а й також має вимоги щодо їх продуктивності.

Функції бібліотеки algorithms

Функції бібліотеки algorithms поділяють на наступні категорії:

* Немодіфікуючі операції з послідовностями( non-modifying sequence operations)
* Модіфікуючі операції з послідовностями (modifying sequence operations)
* Операції розділення (Партіції)(Partitions)
* Сортування (Sorting)
* Бінарний пошук (Binary search) на відсортованих або частково відсортованих послідовностях
* Злиття (Merge) на відсортованих послідовностях
* Операції на структурі Купа (Heap)
* Мінімуми/Максимуми (Min/max)
* інші

Немодіфікуючі операції з послідовностями

Немодіфікуючі операції з послідовностями — це функції що виконують якусь певну дії над масивом даних але не модифкують їх змісту. Зокрема, це методи пошуку, послідовного виконання дій тощо.

Функція for\_each

Функція for\_each(pos1, pos2, fun) виконує функцію fun для всіх елементів, що знаходяться між двома ітераторами pos1 та pos2.

// for\_each example

#include <iostream> // std::cout

#include <algorithm> // std::for\_each

#include <vector> // std::vector

void myfunction (int i) { // function:

std::cout << ' ' << i;

}

int main () {

double mas[] = {1.0, 2.0, 4.0,3.0};

std::vector<int> myvector(mas,mas+4);

std::cout << "myvector contains:";

for\_each (myvector.begin(), myvector.end(), myfunction);

std::cout << '\n';

}

Функція find

Функція find(pos1, pos2, g) шукає елемент g серед всіх елементів, що знаходяться між двома ітераторами pos1 та pos2. Якщо елемент знайдений — повертається ітератор вводу, що вказує на цей елемент, якщо ні — ітератор на кінцеву позицію. Для порівняння обєктів функція використовує оператор рівності(==).

// find example

#include <iostream> // std::cout

#include <algorithm> // std::find

#include <vector> // std::vector

int main () {

// using std::find with array and pointer:

int myints[] = { 10, 20, 30, 40 };

int \* p;

p = std::find (myints, myints+4, 30);

if (p != myints+4)

std::cout << "Element found in myints: " << \*p << '\n';

else

std::cout << "Element not found in myints\n";

// using std::find with vector and iterator:

std::vector<int> myvector (myints,myints+4);

std::vector<int>::iterator it;

it = find (myvector.begin(), myvector.end(), 30);

if (it != myvector.end())

std::cout << "Element found in myvector: " << \*it << '\n';

else

std::cout << "Element not found in myvector\n";

return 0;

}

Для функцій пошуку також можна використовувати форми find\_if, find\_end, find\_first\_of, adjacent\_find, а з С++11 також додана функція find\_if\_not.

// adjacent\_find example

#include <iostream> // std::cout

#include <algorithm> // std::adjacent\_find

#include <vector> // std::vector

bool myfunction (int i, int j) {

return (i==j);

}

int main () {

int myints[] = {5,20,5,30,30,20,10,10,20};

std::vector<int> myvector (myints,myints+8);

std::vector<int>::iterator it;

// using default comparison:

it = std::adjacent\_find (myvector.begin(), myvector.end());

if (it!=myvector.end())

std::cout << "the first pair of repeated elements are: " << \*it << '\n';

//using predicate comparison:

it = std::adjacent\_find (++it, myvector.end(), myfunction);

if (it!=myvector.end())

std::cout << "the second pair of repeated elements are: " << \*it << '\n';

return 0;

}

Примітка. В усіх випадках де використвується функція як аргумент — замість неї можна (а іноді й бажано) використовувати функтор (або лямбду в Сі++11)

Функція count

Функція count(pos1, pos2, g) повертає кількість входжень g серед всіх елементів, що знаходяться між двома ітераторами pos1 та pos2. Для порівняння обєктів функція використовує оператор рівності(==).

// count algorithm example

#include <iostream> // std::cout

#include <algorithm> // std::count

#include <vector> // std::vector

int main () {

// counting elements in array:

int myints[] = {10,20,30,30,20,10,10,20}; // 8 elements

int mycount = std::count (myints, myints+8, 10);

std::cout << "10 appears " << mycount << " times.\n";

// counting elements in container:

std::vector<int> myvector (myints, myints+8);

mycount = std::count (myvector.begin(), myvector.end(), 20);

std::cout << "20 appears " << mycount << " times.\n";

return 0;

}

Також існує функція пошуку count\_if , у формі count\_if(pos1,pos2,fun) яка порівнює елементи за умовою, що задається предікатом fun.

// count\_if example

#include <iostream> // std::cout

#include <algorithm> // std::count\_if

#include <vector> // std::vector

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

int main () {

std::vector<int> myvector;

for (int i=1; i<10; i++) myvector.push\_back(i); // myvector: 1 2 3 4 5 6 7 8 9

int mycount = count\_if (myvector.begin(), myvector.end(), IsOdd);

std::cout << "myvector contains " << mycount << " odd values.\n";

return 0;

}

Функція mismatch

Функція mismatch у формах mismatch(pos1,pos2, col) та mismatch(pos1,pos2, col,fun) знаходтить елементи в двох посілідовностях (контейнерах), що не співпадають. Перша послідовність задається ітераторами початкової та кінцевої позиції, друга — параметром col. Вона повертає пару вхідних ітераторів, перший з яких вказує на першу позицію, що не співпадає в першому контейнері, а другий — в другому.

// mismatch algorithm example

#include <iostream> // std::cout

#include <algorithm> // std::mismatch

#include <vector> // std::vector

#include <utility> // std::pair

bool mypredicate (int i, int j) {

return (i==j);

}

int main () {

std::vector<int> myvector;

for (int i=1; i<6; i++) myvector.push\_back (i\*10); // myvector: 10 20 30 40 50

int myints[] = {10,20,80,320,1024}; // myints: 10 20 80 320 1024

std::pair<std::vector<int>::iterator,int\*> mypair;

// using default comparison:

mypair = std::mismatch (myvector.begin(), myvector.end(), myints);

std::cout << "First mismatching elements: " << \*mypair.first;

std::cout << " and " << \*mypair.second << '\n';

++mypair.first; ++mypair.second;

// using predicate comparison:

mypair = std::mismatch (mypair.first, myvector.end(), mypair.second, mypredicate);

std::cout << "Second mismatching elements: " << \*mypair.first;

std::cout << " and " << \*mypair.second << '\n';

return 0;

}

Функція equal

Функція equal на відміну від mismatch визначає чи співпадає вміст двох послідовностей.

// equal algorithm example

#include <iostream> // std::cout

#include <algorithm> // std::equal

#include <vector> // std::vector

bool mypredicate (int i, int j) {

return (i==j);

}

int main () {

int myints[] = {20,40,60,80,100}; // myints: 20 40 60 80 100

std::vector<int>myvector (myints,myints+5); // myvector: 20 40 60 80 100

// using default comparison:

if ( std::equal (myvector.begin(), myvector.end(), myints) )

std::cout << "The contents of both sequences are equal.\n";

else

std::cout << "The contents of both sequences differ.\n";

myvector[3]=81; // myvector: 20 40 60 81 100

// using predicate comparison:

if ( std::equal (myvector.begin(), myvector.end(), myints, mypredicate) )

std::cout << "The contents of both sequences are equal.\n";

else

std::cout << "The contents of both sequences differ.\n";

return 0;

}

Функції search, search\_n

Функції search, search\_n — на відміну від функцій пошуку findn та find\_if, шукають не лише єдиний елемент, а діапазон чи колекцію та видають реультат у вигляді однонапрвленого ітератору на перший знайдений елемент, або на кінцевий елемент, якщо цих елментів не було знайдено.

Інтерфейс функції: search (pos1\_1,pos1\_2, pos2\_1, pos2\_2)

// search algorithm example

#include <iostream> // std::cout

#include <algorithm> // std::search

#include <vector> // std::vector

bool mypredicate (int i, int j) {

return (i==j);

}

int main () {

std::vector<int> haystack;

// set some values: haystack: 10 20 30 40 50 60 70 80 90

for (int i=1; i<10; i++) haystack.push\_back(i\*10);

// using default comparison:

int needle1[] = {40,50,60,70};

std::vector<int>::iterator it;

it = std::search (haystack.begin(), haystack.end(), needle1, needle1+4);

if (it!=haystack.end())

std::cout << "needle1 found at position " << (it-haystack.begin()) << '\n';

else

std::cout << "needle1 not found\n";

// using predicate comparison:

int needle2[] = {20,30,50};

it = std::search (haystack.begin(), haystack.end(), needle2, needle2+3, mypredicate);

if (it!=haystack.end())

std::cout << "needle2 found at position " << (it-haystack.begin()) << '\n';

else

std::cout << "needle2 not found\n";

Функція search\_n з інтерфейсом search\_n (where\_pos1,where\_pos2, count, val) шукає чи входить в даний діапазон count значень val. Якщо входить, то вона повертає ітератор на перше входження, інакше ітеатор на кінець.

// search\_n example

#include <iostream> // std::cout

#include <algorithm> // std::search\_n

#include <vector> // std::vector

bool mypredicate (int i, int j) {

return (i==j);

}

int main () {

int myints[]={10,20,30,30,20,10,10,20};

std::vector<int> myvector (myints,myints+8);

std::vector<int>::iterator it;

// using default comparison:

it = std::search\_n (myvector.begin(), myvector.end(), 2, 30);

if (it!=myvector.end())

std::cout << "two 30s found at position " << (it-myvector.begin()) << '\n';

else

std::cout << "match not found\n";

// using predicate comparison:

it = std::search\_n (myvector.begin(), myvector.end(), 2, 10, mypredicate);

if (it!=myvector.end())

std::cout << "two 10s found at position " << int(it-myvector.begin()) << '\n';

else

std::cout << "match not found\n";

return 0;

}

Modifying sequence operations

# Функція copy

# Функція copy призначена для копіювання діапазону значеньв колекції у іншу колекцію:

Інтерфейс: copy ( pose1, pos2, result\_pos), остання позиція вказує за яку позицію потрібно копіювати.

// copy algorithm example

#include <iostream> // std::cout

#include <algorithm> // std::copy

#include <vector> // std::vector

int main () {

int myints[]={10,20,30,40,50,60,70};

std::vector<int> myvector (7);

std::copy ( myints, myints+7, myvector.begin() );

std::cout << "myvector contains:";

for (std::vector<int>::iterator it = myvector.begin(); it!=myvector.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

return 0;

}

# Функція copy\_backward — буде копіювати в зворотньому порядку. В Сі++11 додали ще форми copy\_n та copy\_if

Методи swap та swap\_ranges, iter\_swap здійснюють обмін колекціями або діапазонів значень або значеннями ітераторів.

Метод transform застосовє дану функцію до колекції(діапазону) або двох колекцій(діапазонів) та модифікує при цьому одну з них.

// transform algorithm example

#include <iostream> // std::cout

#include <algorithm> // std::transform

#include <vector> // std::vector

#include <functional> // std::plus

int op\_increase (int i) { return ++i; }

int main () {

std::vector<int> foo;

std::vector<int> bar;

// set some values:

for (int i=1; i<6; i++)

foo.push\_back (i\*10); // foo: 10 20 30 40 50

bar.resize(foo.size()); // allocate space

std::transform (foo.begin(), foo.end(), bar.begin(), op\_increase);

// bar: 11 21 31 41 51

// std::plus adds together its two arguments:

std::transform (foo.begin(), foo.end(), bar.begin(), foo.begin(), std::plus<int>());

// foo: 21 41 61 81 101

std::cout << "foo contains:";

for (std::vector<int>::iterator it=foo.begin(); it!=foo.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

return 0;

}

# Методи replace, replace\_if, replace\_copy, replace\_copy\_if змінюють вказане значення в колекції (у формах replace\_copy, replace\_copy\_if одночасно модифікується і те значення яким модифіковане -тобто відбувається обмін значеннями)

// replace\_copy example

#include <iostream> // std::cout

#include <algorithm> // std::replace\_copy

#include <vector> // std::vector

int main () {

int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };

std::vector<int> myvector (8);

std::replace\_copy (myints, myints+8, myvector.begin(), 20, 99);

std::cout << "myvector contains:";

for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

std::vector<int> foo,bar;

// set some values:

for (int i=1; i<10; i++) foo.push\_back(i); // 1 2 3 4 5 6 7 8 9

bar.resize(foo.size()); // allocate space

std::replace\_copy\_if (foo.begin(), foo.end(), bar.begin(), IsOdd, 0);

// 0 2 0 4 0 6 0 8 0

std::cout << "bar contains:";

for (std::vector<int>::iterator it=bar.begin(); it!=bar.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

}

Методи fill та fill\_n дозволяють шиводко ініціалізувати колекцію даними.

std::vector<int> myvector (8,10); // myvector: 10 10 10 10 10 10 10 10

std::fill\_n (myvector.begin(),4,20); // myvector: 20 20 20 20 10 10 10 10

std::fill\_n (myvector.begin()+3,3,33); // myvector: 20 20 20 33 33 33 10 10

std::cout << "myvector contains:";

for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

std::fill (myvector.begin(),myvector.begin()+4,5); // myvector: 5 5 5 5 0 0 0 0

std::fill (myvector.begin()+3,myvector.end()-2,8); // myvector: 5 5 5 8 8 8 0 0

std::cout << "myvector contains:";

for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

Метод generate та generate\_n дозволяють ініціалізувати чи модифікувати колекцію за допомогою даної функції.

// generate algorithm example

#include <iostream> // std::cout

#include <algorithm> // std::generate

#include <vector> // std::vector

#include <ctime> // std::time

#include <cstdlib> // std::rand, std::srand

// function generator:

int RandomNumber () { return (std::rand()%100); }

// class generator:

struct c\_unique {

int current;

c\_unique() {current=0;}

int operator()() {return ++current;}

} UniqueNumber;

int main () {

std::srand ( unsigned ( std::time(0) ) );

std::vector<int> myvector (8);

std::generate (myvector.begin(), myvector.end(), RandomNumber);

std::cout << "myvector contains:";

for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

std::generate (myvector.begin(), myvector.end(), UniqueNumber);

std::cout << "myvector contains:";

for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

return 0;

}

Методи remove, remove\_if, remove\_copy, remove\_copy\_if — дозволяють видаляти елементи за значенням або по значенню предікату.

// remove\_if example

#include <iostream> // std::cout

#include <algorithm> // std::remove\_if

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

int main () {

int myints[] = {1,2,3,4,5,6,7,8,9}; // 1 2 3 4 5 6 7 8 9

// bounds of range:

int\* pbegin = myints; // ^

int\* pend = myints+sizeof(myints)/sizeof(int); // ^ ^

pend = std::remove\_if (pbegin, pend, IsOdd); // 2 4 6 8 ? ? ? ? ?

// ^ ^

std::cout << "the range contains:";

for (int\* p=pbegin; p!=pend; ++p)

std::cout << ' ' << \*p;

std::cout << '\n';

int myints[] = {10,20,30,30,20,10,10,20}; // 10 20 30 30 20 10 10 20

// bounds of range:

int\* pbegin = myints; // ^

int\* pend = myints+sizeof(myints)/sizeof(int); // ^ ^

pend = std::remove (pbegin, pend, 20); // 10 30 30 10 10 ? ? ?

// ^ ^

std::cout << "range contains:";

for (int\* p=pbegin; p!=pend; ++p)

std::cout << ' ' << \*p;

std::cout << '\n';

int myints[] = {1,2,3,4,5,6,7,8,9};

std::vector<int> myvector (9);

std::remove\_copy\_if (myints,myints+9,myvector.begin(),IsOdd);

std::cout << "myvector contains:";

for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

Функції unique та unique\_copy дозволяють видаляти послідовні однакові значення в послідовності

int myints[] = {10,20,20,20,30,30,20,20,10}; // 10 20 20 20 30 30 20 20 10

std::vector<int> myvector (myints,myints+9);

// using default comparison:

std::vector<int>::iterator it;

it = std::unique (myvector.begin(), myvector.end()); // 10 20 30 20 10 ? ? ? ?

// ^

myvector.resize( std::distance(myvector.begin(),it) ); // 10 20 30 20 10

// using predicate comparison:

std::unique (myvector.begin(), myvector.end(), myfunction); // (no changes)

// print out content:

std::cout << "myvector contains:";

for (it=myvector.begin(); it!=myvector.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n'

// using default comparison:

std::vector<int>::iterator it;

it=std::unique\_copy (myints,myints+9,myvector.begin()); // 10 20 30 20 10 0 0 0 0

// ^

std::sort (myvector.begin(),it); // 10 10 20 20 30 0 0 0 0

// ^

// using predicate comparison:

it=std::unique\_copy (myvector.begin(), it, myvector.begin(), myfunction);

// 10 20 30 20 30 0 0 0 0

// ^

myvector.resize( std::distance(myvector.begin(),it) ); // 10 20 30

// print out content:

std::cout << "myvector contains:";

for (it=myvector.begin(); it!=myvector.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

Методи reverse та reverse\_copy — дозволяють інвертувати колекцію

Методи rotate та rotate\_copy — роблять циклічний зсув колекції

# Метод random\_shuffle — випадковим чином пермішує послідовність.

З Сі++11 додали також функцію shuffle, яка перемішує послідовність за допомогою даного генератору псевдовипадкових чисел

Операції розділення (partitions)

# partition

Функція partition (pos1,pos2, pred) модифікує діяапазон [pos1,pos2) таким чином, що елменти для яких предікат pred повертає true передують тим, де він повертає false. Результат — ітератор, що вказує на перший елемент, який повертає false в новому діапазоні   
Відносний порядок при цьому не зобвязаний зберігатися, якщо його потрібно зберегти використовується функція [stable\_partition](http://cplusplus.com/stable_partition) .

// partition algorithm example

#include <iostream> // std::cout

#include <algorithm> // std::partition

#include <vector> // std::vector

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

int main () {

std::vector<int> myvector;

// set some values:

for (int i=1; i<10; ++i) myvector.push\_back(i); // 1 2 3 4 5 6 7 8 9

std::vector<int>::iterator bound;

bound = std::partition (myvector.begin(), myvector.end(), IsOdd);

// print out content:

std::cout << "odd elements:";

for (std::vector<int>::iterator it=myvector.begin(); it!=bound; ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

std::cout << "even elements:";

for (std::vector<int>::iterator it=bound; it!=myvector.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

return 0;

}

// stable\_partition example

#include <iostream> // std::cout

#include <algorithm> // std::stable\_partition

#include <vector> // std::vector

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

int main () {

std::vector<int> myvector;

// set some values:

for (int i=1; i<10; ++i) myvector.push\_back(i); // 1 2 3 4 5 6 7 8 9

std::vector<int>::iterator bound;

bound = std::stable\_partition (myvector.begin(), myvector.end(), IsOdd);

// print out content:

std::cout << "odd elements:";

for (std::vector<int>::iterator it=myvector.begin(); it!=bound; ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

std::cout << "even elements:";

for (std::vector<int>::iterator it=bound; it!=myvector.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

return 0;

}

З Сі++11 додали також функції is\_partitioned, partition\_copy, partition\_point

**Sorting**

# Функція sort — сортує за зростанням вказаний інтервал за вказаним бінарним предікатом (за замовченням — це стандартний або перевантажений оператор <)

# stable\_sort — працює майже так саме як sort але при цьому обовязково зберігається взаємний порядок елементів, для яких критерій порівняння визначав еквівалентність.

// stable\_sort example

#include <iostream> // std::cout

#include <algorithm> // std::stable\_sort

#include <vector> // std::vector

bool compare\_as\_ints (double i,double j)

{

return (int(i)<int(j));

}

int main () {

double mydoubles[] = {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62, 2.58};

std::vector<double> myvector;

myvector.assign(mydoubles,mydoubles+8);

std::cout << "using default comparison:";

std::stable\_sort (myvector.begin(), myvector.end());

for (std::vector<double>::iterator it=myvector.begin(); it!=myvector.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

myvector.assign(mydoubles,mydoubles+8);

std::cout << "using 'compare\_as\_ints' :";

std::stable\_sort (myvector.begin(), myvector.end(), compare\_as\_ints);

for (std::vector<double>::iterator it=myvector.begin(); it!=myvector.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

return 0;

}

# partial\_sort

Функція часткового сортування partial\_sort (pos1,middle,pos2) та partial\_sort (pos1, middle, pos2, comp) — сортує елементи діапазону таким чином, що елементи перед middle сатють найменшими елементами діапазону та відсортовані за неспадінням.  
Критерій порівняння - operator< для першої версії та бінарний предікат comp для другої.

// partial\_sort example

#include <iostream> // std::cout

#include <algorithm> // std::partial\_sort

#include <vector> // std::vector

bool myfunction (int i,int j) { return (i<j); }

int main () {

int myints[] = {9,8,7,6,5,4,3,2,1};

std::vector<int> myvector (myints, myints+9);

// using default comparison (operator <):

std::partial\_sort (myvector.begin(), myvector.begin()+5, myvector.end());

// using function as comp

std::partial\_sort (myvector.begin(), myvector.begin()+5, myvector.end(),myfunction);

// print out content:

std::cout << "myvector contains:";

for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

return 0;

}

# partial\_sort\_copy — таке саме як partial\_sort, але зберігає невдсортовану частину

# nth\_element

# nth\_element(pos1,middle,pos2) модифікує діапазое таким чином, що елемент на позиції middle займає ту позицію, яка в нього була би в відсортованому масиві.

// nth\_element example

#include <iostream> // std::cout

#include <algorithm> // std::nth\_element, std::random\_shuffle

#include <vector> // std::vector

bool myfunction (int i,int j) { return (i<j); }

int main () {

std::vector<int> myvector;

// set some values:

for (int i=1; i<10; i++) myvector.push\_back(i); // 1 2 3 4 5 6 7 8 9

std::random\_shuffle (myvector.begin(), myvector.end());

// using default comparison (operator <):

std::nth\_element (myvector.begin(), myvector.begin()+5, myvector.end());

// using function as comp

std::nth\_element (myvector.begin(), myvector.begin()+5, myvector.end(),myfunction);

// print out content:

std::cout << "myvector contains:";

for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

return 0;

}

З Сі++11 додали is\_sorted, is\_sorted\_until — які визначають чи відсортований даний діапазон або чи відсортований він до деякого елементу.

Бінарний пошук

Алгоритми бінарного пошуку працюють на відсортованих або частково відсортованих колекціях

# binary\_search

Виконує бінарний пошук даного елемента в послідовності.

// binary\_search example

#include <iostream> // std::cout

#include <algorithm> // std::binary\_search, std::sort

#include <vector> // std::vector

bool myfunction (int i,int j) { return (i<j); }

int main () {

int myints[] = {1,2,3,4,5,4,3,2,1};

std::vector<int> v(myints,myints+9); // 1 2 3 4 5 4 3 2 1

// using default comparison:

std::sort (v.begin(), v.end());

std::cout << "looking for a 3... ";

if (std::binary\_search (v.begin(), v.end(), 3))

std::cout << "found!\n"; else std::cout << "not found.\n";

// using myfunction as comp:

std::sort (v.begin(), v.end(), myfunction);

std::cout << "looking for a 6... ";

if (std::binary\_search (v.begin(), v.end(), 6, myfunction))

std::cout << "found!\n"; else std::cout << "not found.\n";

return 0;

}

# equal\_range

Функція equal\_range(pos1,pos2, val) повертає піддіапазон тих елементів діапазону (як пару на початок та кінець піддіапазону), що еквівалентні val.

// equal\_range example

// equal\_range example

#include <iostream> // std::cout

#include <algorithm> // std::equal\_range, std::sort

#include <vector> // std::vector

bool mygreater (int i,int j) { return (i>j); }

int main () {

int myints[] = {10,20,30,30,20,10,10,20};

std::vector<int> v(myints,myints+8); // 10 20 30 30 20 10 10 20

std::pair<std::vector<int>::iterator,std::vector<int>::iterator> bounds;

// using default comparison:

std::sort (v.begin(), v.end()); // 10 10 10 20 20 20 30 30

bounds=std::equal\_range (v.begin(), v.end(), 20); // ^ ^

// using "mygreater" as comp:

std::sort (v.begin(), v.end(), mygreater); // 30 30 20 20 20 10 10 10

bounds=std::equal\_range (v.begin(), v.end(), 20, mygreater); // ^ ^

std::cout << "bounds at positions " << (bounds.first - v.begin());

std::cout << " and " << (bounds.second - v.begin()) << '\n';

return 0;

}

# lower\_bound, upper\_bound

# Функції lower\_bound(pos1,pos2, val), upper\_bound(pos1,pos2, val) повертають відповідно ітератори на перший та останній елемент піддіапазону еквівалентних val значень.

/ lower\_bound/upper\_bound example

#include <iostream> // std::cout

#include <algorithm> // std::lower\_bound, std::upper\_bound, std::sort

#include <vector> // std::vector

int main () {

int myints[] = {10,20,30,30,20,10,10,20};

std::vector<int> v(myints,myints+8); // 10 20 30 30 20 10 10 20

std::sort (v.begin(), v.end()); // 10 10 10 20 20 20 30 30

std::vector<int>::iterator low,up;

low=std::lower\_bound (v.begin(), v.end(), 20); // ^

up= std::upper\_bound (v.begin(), v.end(), 20); // ^

std::cout << "lower\_bound at position " << (low- v.begin()) << '\n';

std::cout << "upper\_bound at position " << (up - v.begin()) << '\n';

return 0;

}

Злиття

Дані функції працюють для відсортованих послідовностей.

# merge

Функція merge виконує алгоритм злиття двох відсортованих послідовностей

# inplace\_merge

Виконує злиття діапазонів [first,middle) та [middle,last), поміщуючи результат в діапазон [first,last).

Метод зберігає відносне розташування еквівалентних елементів

// inplace\_merge example

#include <iostream> // std::cout

#include <algorithm> // std::inplace\_merge, std::sort, std::copy

#include <vector> // std::vector

int main () {

int first[] = {5,10,15,20,25};

int second[] = {50,40,30,20,10};

std::vector<int> v(10);

std::vector<int>::iterator it;

std::sort (first,first+5);

std::sort (second,second+5);

it=std::copy (first, first+5, v.begin());

std::copy (second,second+5,it);

std::inplace\_merge (v.begin(),v.begin()+5,v.end());

std::cout << "The resulting vector contains:";

for (it=v.begin(); it!=v.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

return 0;

}

# includes

Перевіряє, чи входить одна відсортована послідовність в іншу

// includes algorithm example

#include <iostream> // std::cout

#include <algorithm> // std::includes, std::sort

bool myfunction (int i, int j) { return i<j; }

int main () {

int container[] = {5,10,15,20,25,30,35,40,45,50};

int continent[] = {40,30,20,10};

std::sort (container,container+10);

std::sort (continent,continent+4);

// using default comparison:

if ( std::includes(container,container+10,continent,continent+4) )

std::cout << "container includes continent!\n";

// using myfunction as comp:

if ( std::includes(container,container+10,continent,continent+4, myfunction) )

std::cout << "container includes continent!\n";

return 0;

}

# set\_union, set\_intersection, set\_difference, set\_symmetric\_difference

- виконують відповідно обєднання, перетин, різницю та симетричну різницю двох відсоротваних послідовностей.

// set\_symmetric\_difference example

#include <iostream> // std::cout

#include <algorithm> // std::set\_symmetric\_difference, std::sort

#include <vector> // std::vector

int main () {

int first[] = {5,10,15,20,25};

int second[] = {50,40,30,20,10};

std::vector<int> v(10); // 0 0 0 0 0 0 0 0 0 0

std::vector<int>::iterator it;

std::sort (first,first+5); // 5 10 15 20 25

std::sort (second,second+5); // 10 20 30 40 50

it=std::set\_symmetric\_difference (first, first+5, second, second+5, v.begin());

// 5 15 25 30 40 50 0 0 0 0

v.resize(it-v.begin()); // 5 15 25 30 40 50

std::cout << "The symmetric difference has " << (v.size()) << " elements:\n";

for (it=v.begin(); it!=v.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

Робота з купою

# make\_heap, push\_heap, pop\_heap, sort\_heap — відповідно створюють структуру даних купа, додають елемент до купи, видаляють елемент з купи та сортують її.

// range heap example

#include <iostream> // std::cout

#include <algorithm> // std::make\_heap, std::pop\_heap, std::push\_heap, std::sort\_heap

#include <vector> // std::vector

int main () {

int myints[] = {10,20,30,5,15};

std::vector<int> v(myints,myints+5);

std::make\_heap (v.begin(),v.end());

std::cout << "initial max heap : " << v.front() << '\n';

std::pop\_heap (v.begin(),v.end()); v.pop\_back();

std::cout << "max heap after pop : " << v.front() << '\n';

v.push\_back(99); std::push\_heap (v.begin(),v.end());

std::cout << "max heap after push: " << v.front() << '\n';

std::sort\_heap (v.begin(),v.end());

std::cout << "final sorted range :";

for (unsigned i=0; i<v.size(); i++)

std::cout << ' ' << v[i];

std::cout << '\n';

return 0;

}

З Сі++11 додали також функції is\_heap(), is\_heap\_until

// is\_heap example

#include <iostream> // std::cout

#include <algorithm> // std::is\_heap, std::make\_heap, std::pop\_heap

#include <vector> // std::vector

int main () {

std::vector<int> foo {9,5,2,6,4,1,3,8,7};

if (!std::is\_heap(foo.begin(),foo.end()))

std::make\_heap(foo.begin(),foo.end());

std::cout << "Popping out elements:";

while (!foo.empty()) {

std::pop\_heap(foo.begin(),foo.end()); // moves largest element to back

std::cout << ' ' << foo.back(); // prints back

foo.pop\_back(); // pops element out of container

}

std::cout << '\n';

return 0;

}

Мінімум/Макксимум

# min, max — повертають відповідно значення мінімально та максимального елементу послідовності

# min\_element, max\_element - повертають відповідно однонаправлений ітератор на мінімальний та максимальний елементи послідовності

З Сі++11 додали також методи

# Minmax, minmax\_element — які повертають відповідно одночасно значення мінімального та максимального елементів або однонаправлений ітератор на них у вигляді пари.

Інші

# lexicographical\_compare

Lexicographical less-than comparison

true if the range [first1,last1) compares *lexicographically less* than the range [first2,last2).  
  
A *lexicographical comparison* is the kind of comparison generally used to sort words alphabetically in dictionaries; It involves comparing sequentially the elements that have the same position in both ranges against each other until one element is not equivalent to the other. The result of comparing these first non-matching elements is the result of the lexicographical comparison.  
  
If both sequences compare equal until one of them ends, the shorter sequence is *lexicographically less* than the longer one.  
  
The elements are compared using operator< for the first version, and comp for the second. Two elements, a and b are considered equivalent if (!(a<b) && !(b<a)) or if (!comp(a,b) && !comp(b,a))

// lexicographical\_compare example

#include <iostream> // std::cout, std::boolalpha

#include <algorithm> // std::lexicographical\_compare

#include <cctype> // std::tolower

// a case-insensitive comparison function:

bool mycomp (char c1, char c2)

{ return std::tolower(c1)<std::tolower(c2); }

int main () {

char foo[]="Apple";

char bar[]="apartment";

std::cout << std::boolalpha;

std::cout << "Comparing foo and bar lexicographically (foo<bar):\n";

std::cout << "Using default comparison (operator<): ";

std::cout << std::lexicographical\_compare(foo,foo+5,bar,bar+9);

std::cout << '\n';

std::cout << "Using mycomp as comparison object: ";

std::cout << std::lexicographical\_compare(foo,foo+5,bar,bar+9,mycomp);

std::cout << '\n';

return 0;

}

# next\_permutation, prev\_permutation

# 

Transform range to next permutation

[first,last) into the next [*lexicographically*](http://cplusplus.com/lexicographical_compare) *greater* permutation.  
  
A permutation is each one of the N! possible arrangements the elements can take (where N is the number of elements in the range). Different permutations can be ordered according to how they compare [*lexicographicaly*](http://cplusplus.com/lexicographical_compare) to each other; The first such-sorted possible permutation (the one that would compare *lexicographically smaller* to all other permutations) is the one which has all its elements sorted in ascending order, and the largest has all its elements sorted in descending order.  
  
The comparisons of individual elements are performed using either operator< for the first version, or comp for the second.  
  
If the function can determine the next higher permutation, it rearranges the elements as such and returns true. If that was not possible (because it is already at the largest possible permutation), it rearranges the elements according to the first permutation (sorted in ascending order) and returns false.

// next\_permutation exprev\_permutation

ample

#include <iostream> // std::cout

#include <algorithm> // std::next\_permutation, std::sort

int main () {

int myints[] = {1,2,3};

std::sort (myints,myints+3);

std::cout << "The 3! possible permutations with 3 elements:\n";

do {

std::cout << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';

} while ( std::next\_permutation(myints,myints+3) );

std::cout << "After loop: " << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';

int myints[] = {1,2,3};

std::sort (myints,myints+3);

std::reverse (myints,myints+3);

std::cout << "The 3! possible permutations with 3 elements:\n";

do {

std::cout << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';

} while ( std::prev\_permutation(myints,myints+3) );

std::cout << "After loop: " << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';

return 0;

}

Алгоритми бібліотеки numeric

В бібліотеку numeric входять наступні функції

accumulate

Функція accumulate(pos1,pos2,init) або accumulate(pos1,pos2,init, fun) виконує послідовне сумування елемнтів послідовності або послідовне виконання заданої функції fun від першого до останнього починаючи з деякого значення init

// accumulate example

#include <iostream> // std::cout

#include <functional> // std::minus

#include <numeric> // std::accumulate

int myfunction (int x, int y) {return x+2\*y;}

struct myclass {

int operator()(int x, int y) {return x+3\*y;}

} myobject;

int main () {

int init = 100;

int numbers[] = {10,20,30};

std::cout << "using default accumulate: ";

std::cout << std::accumulate(numbers,numbers+3,init);

std::cout << '\n';

std::cout << "using functional's minus: ";

std::cout << std::accumulate (numbers, numbers+3, init, std::minus<int>());

std::cout << '\n';

std::cout << "using custom function: ";

std::cout << std::accumulate (numbers, numbers+3, init, myfunction);

std::cout << '\n';

std::cout << "using custom object: ";

std::cout << std::accumulate (numbers, numbers+3, init, myobject);

std::cout << '\n';

return 0;

}

# adjacent\_difference

Якщо x — це елемент [first,last) та y — елемент в result, то рецзультат цієї функції буде наступний:  
y0 = x0   
y1 = x1 - x0   
y2 = x2 - x1   
y3 = x3 - x2   
y4 = x4 - x3

// adjacent\_difference example

#include <iostream> // std::cout

#include <functional> // std::multiplies

#include <numeric> // std::adjacent\_difference

int myop (int x, int y) {return x+y;}

int main () {

int val[] = {1,2,3,5,9,11,12};

int result[7];

std::adjacent\_difference (val, val+7, result);

std::cout << "using default adjacent\_difference: ";

for (int i=0; i<7; i++) std::cout << result[i] << ' ';

std::cout << '\n';

std::adjacent\_difference (val, val+7, result, std::multiplies<int>());

std::cout << "using functional operation multiplies: ";

for (int i=0; i<7; i++) std::cout << result[i] << ' ';

std::cout << '\n';

std::adjacent\_difference (val, val+7, result, myop);

std::cout << "using custom function: ";

for (int i=0; i<7; i++) std::cout << result[i] << ' ';

std::cout << '\n';

return 0;

}

# inner\_product

Функція рахує комулятивний скалярний добуток інтервалу

init де добуток рахується як пара на якй вказують числа з first1 та first2. Дві операції за замовченням (додавання для результатів та множення дляпар) можуть бути перевантажені бінарними функціями binary\_op1 та binary\_op2.

// inner\_product example

#include <iostream> // std::cout

#include <functional> // std::minus, std::divides

#include <numeric> // std::inner\_product

int myaccumulator (int x, int y) {return x-y;}

int myproduct (int x, int y) {return x+y;}

int main () {

int init = 100;

int series1[] = {10,20,30};

int series2[] = {1,2,3};

std::cout << "using default inner\_product: ";

std::cout << std::inner\_product(series1,series1+3,series2,init);

std::cout << '\n';

std::cout << "using functional operations: ";

std::cout << std::inner\_product(series1,series1+3,series2,init,

std::minus<int>(),std::divides<int>());

std::cout << '\n';

std::cout << "using custom functions: ";

std::cout << std::inner\_product(series1,series1+3,series2,init,

myaccumulator,myproduct);

std::cout << '\n';

return 0;

}

# partial\_sum

Якщо x предстваляє елемент в [first,last) та y представляє елемент в result, то результат функції може бути представлений:  
y0 = x0   
y1 = x0 + x1   
y2 = x0 + x1 + x2   
y3 = x0 + x1 + x2 + x3   
y4 = x0 + x1 + x2 + x3 + x4

// partial\_sum example

#include <iostream> // std::cout

#include <functional> // std::multiplies

#include <numeric> // std::partial\_sum

int myop (int x, int y) {return x+y+1;}

int main () {

int val[] = {1,2,3,4,5};

int result[5];

std::partial\_sum (val, val+5, result);

std::cout << "using default partial\_sum: ";

for (int i=0; i<5; i++) std::cout << result[i] << ' ';

std::cout << '\n';

std::partial\_sum (val, val+5, result, std::multiplies<int>());

std::cout << "using functional operation multiplies: ";

for (int i=0; i<5; i++) std::cout << result[i] << ' ';

std::cout << '\n';

std::partial\_sum (val, val+5, result, myop);

std::cout << "using custom function: ";

for (int i=0; i<5; i++) std::cout << result[i] << ' ';

std::cout << '\n';

return 0;

}

В Сі++11 до цього переліку функцій також додали функцію iota, яка за даним інтервалом [first,last) повертає послідовні значення val, які отрмуються послідовним застосуванням інкременту ++val до кожного елементу.

// iota example

#include <iostream> // std::cout

#include <numeric> // std::iota

int main () {

int numbers[10];

std::iota (numbers,numbers+10,100);

std::cout << "numbers:";

for (int& i:numbers) std::cout << ' ' << i;

std::cout << '\n';

return 0;

}

Функтори, предікати та лямбда(Сі++11)

Багато з розглянутих алгоритмів Сі++ мають у якості аргументу функцію, наприклад, for\_each, generate, find\_if.

Крім того, функція може бути аргументом конструктору для шаблонів класів set, multiset, map, multimap для задання шляху сортування ключів цих класів.

Задати цю функцію можна як безпосереньо функцію, тобто приписавши реалізацію або декларацію цієї функції до виклику її як аргумент функції чи конструктору, це фактично означає, що ми використовуємо вказівник на цю функцію. Але інколи було б бажано використати цю функцію як змінну або обєкт, тобто мати її як конкретний екземпляр деякого класу.

Зокрема, розглянемо функцію, яка приймає лише один аргумент але під час виклику цієї функції нам потрібно передати, наприклад, ще параметр. Однак на Сі++ це неможливо, оскільки функція приймає лише один параметр. Що можна зробити? Очевидною відповіддю можуть бути глобальні змінні. Однак практика гарного кодування не виступає за використання глобальних змінних і стверджує, що їх слід використовувати лише тоді, коли немає іншої альтернативи.

На Сі++ для подібних модифікацій функції можна використати функтори, які також звуться функціональними обєктами (Зверніть увагу, що функтори — це не те само що функції !!).

Функтори - це об’єкти, які можна обробляти так, ніби вони є функцією або вказівником на функції.

Функціональний обєкт або функтор — це клас який перевантажує оператор виклику operator () нустпним чином, що в коді

FunctionObjectType func;

func();

вираз func() є викликом оператору () обєкту func, а не викликом функції func. Тип функціонального обєкту повинен бути визначеним наступним чином:

class FunctionObjectType {

public:

void operator() () {

// Код функції

}

};

У використання функціональних об'єктів є ряд переваг [5] перед використанням функцій, а саме:

1. Функціональний об'єкт може мати стан. Фактично може бути два об'єкта одного і того ж функціонального типу, що знаходяться в різних станах в одне і теж час, що неможливо для звичайних функцій. Також функціональний об'єкт може забезпечити операції попередньої ініціалізації даних.

2. Кожен функціональний об'єкт має тип, а отже є можливість передати цей тип як параметр шаблону для вказівки певної поведінки. Наприклад, типи контейнерів з різними функціональними об'єктами відрізняються.

3. Об'єкти-функції часто виконуються швидше ніж вказівники на функції. Наприклад, вбудуване (inline) звернення до оператора () класу легше, ніж функція, передану за вказівником.

Функтори найчастіше використовуються разом із STL у такому сценарії:

//class Comparator

struct Classcomp {

bool operator() (const int& lhs, const int& rhs) const

{return lhs>rhs;}

};

int main (){

int mas[]= {1,2,3,4,5};

set<int,Classcomp> fifth\_set; // class as Compare

for(int i=0;i<5;++i){

fifth\_set.insert(mas[i]);

}

}

Таким чином, функтор (або функціональний об’єкт) - це клас Сі++, який діє як функція. Функтори викликаються з використанням синтаксису виклику функції. Для того, щоб створити функтор, потрібно створити об'єкт MyFunctor, який перевантажує оператор “круглі дужки” (operator()).

Тоді код:

MyFunctor(10);

Еквівалентний коду

MyFunctor.operator()(10);

Ще один приклад використання функтору — в алгоритмі transform.

class increment{

int num;

public:

increment(int n): num(n) {}

int operator(int k){

return num+k;

}

};

Тоді рядок

transform(arr, arr+n, arr, increment(to\_add));

буде еквівалентним наступному коду:

// Creating object of increment

increment obj(to\_add);

// Calling () on object

transform(arr, arr+n, arr, obj);

Тобто обєкт a створений таким, що перевантажує operator(). Таким чином, функтори достатньо ефективні при використанні в C++ STL.

Предикати

Зауважимо, що частковий випадок функторів, які повертають логічний тип bool зветься предікатом. Зокрема предікати потрібні при використанні таких функцій з бібліотеки алгоритмів, як find\_if(), count\_if, sort тощо.

Предикати використовуються в алгоритмах сортування, пошука, а також в усіх інших, що мають в кінці \_if. Сенс у тому, що об’єкт-функція у випадку використання предикату повертає true або false у залежності від виконання необхідної умови. Це або факт відповідності об’єктом деяких властивостей, або результат порівняння двох об’єктів за визначеною ознакою.

Приклад використання предикатів

**class** DividedByTwo

{

**public**:

bool operator()(**const** int x) **const** {

**return** x % 2 == 0;

}

};

int main(){

**const** std::size\_t N = 3;

int A[N] = {3, 2, 5};

std::cout << std::count\_if(A, A + N, DividedByTwo());

}

Розглянемо приклад використання предикату на прикладі алгоритму count\_if. Для того, щоб передати предикату критерій, необхідно в тілі структури створити конструктор так, як показано на прикладі.

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

struct Prd {

int my\_cnt;

// Конструктор

Prd(const int &t) : my\_cnt(t) {}

// Перегрузка операції ()

bool operator() (const int & v) {

return v > my\_cnt;

}

};

int main() {

int mas[] = {1, 2, 3, 4, 5, 3, 7, 3, 9, 3};

vector<int> num(mas, mas + 10);

cout << count\_if(num.begin(), num.end(), Prd(3)) << endl;

return 0;

}

Зрозуміти, як працює програма можна, якщо взяти за увагу, що викликається функція, як аргумент іншої функції, тобто operator()(arg\_prd, Prd(arg\_crt)), де

arg\_prd, що передається предикату, елемент масиву, а arg\_crt - аргумент-критерій методу Prd(). Але оскільки ми маємо справу з класом, то ми кажемо, що другим аргументом викликається наш конструктор. Ось у цьому й розкривається перевага функторів, у порівнянні з звичайними функціями.

### std::function

#### Починаючи зі стандарту C ++ 11 шаблонний клас std :: function є поліморфною обгорткою функцій для загального використання. Об'єкти класу std :: function можуть зберігати, копіювати і викликати довільні викликаються об'єкти - функції, лямбда-вирази, вирази зв'язування і інші функціональні об'єкти. Взагалі кажучи, в будь-якому місці, де необхідно використовувати вказівник на функцію для її відкладеного виклику, або для створення функції зворотного виклику, замість нього може бути використаний std :: function, який надає користувачеві велику гнучкість в реалізації.

#### Визначення класу:

template<class> class function; // undefined

template<class R, class... ArgTypes> class function<R(ArgTypes...)>;

Також в стандарті С++11 визначені функції-модифікатори swap та assign і оператори порівняння (== та !=) з nullptr. Доступ до цільового обєкту жає функція target, а а до його типу — target\_type. Оператор приведення function до булевого типу повертає true, коли у класа є цьовий обєкт.

#### Приклад:

#include <iostream>

#include <functional>

struct A {

A(int num) : num\_(num){}

void printNumberLetter(char c) const {std::cout << "Number: " << num\_ << " Letter: " << c << std::endl;}

int num\_;

};

void printLetter(char c)

{

std::cout << c << std::endl;

}

struct B {

void operator() () {std::cout << "B()" << std::endl;}

};

int main()

{

// Содержит функцию.

std::function<void(char)> f\_print\_Letter = printLetter;

f\_print\_Letter('Q');

// Содержит лямбда-выражение.

std::function<void()> f\_print\_Hello = [] () {std::cout << "Hello world!" << std::endl;};

f\_print\_Hello();

// Содержит связыватель.

std::function<void()> f\_print\_Z = std::bind(printLetter, 'Z');

f\_print\_Z();

// Содержит вызов метода класса.

std::function<void(const A&, char)> f\_printA = &A::printNumberLetter;

A a(10);

f\_printA(a, 'A');

// Содержит функциональный объект.

B b;

std::function<void()> f\_B = b;

f\_B();

}

Результат:

Q

Hello world!

Z

Number: 10 Letter: A

B()

#### std::bad\_functional\_call

[В](https://ru.wikipedia.org/wiki/Исключение_(программирование))иключення типу bad\_functional\_call буде створено при спробі виклику функції function::operator(), якщо в неє буде відсутній цільовий обєкт. Клас bad\_functional\_call є нащадком std::exception, і в нього є доступним віртуальний метод what() для отримання тексту помилки.

Приклад:

#include <iostream>

#include <functional>

int main()

{

std::function<void()> func = nullptr;

try {

func();

} catch(const std::bad\_function\_call& e) {

std::cout << e.what() << std::endl;

}

}

Більшість функторів і предикатів за кількістю операндів - бінарні, тому дуже часто доводиться використовувати функційні адаптери. Адаптери - також функтори, але вони назначені для зв’язки функторів й аргументів. До стандартних функційних адаптерів відносяться наступні:

1. bind() - зв’язує аргументи з операцією;
2. mem\_fn() - викликає операцію, що вказує на функцію-член об’єкту;
3. not1() - унарне заперечення;
4. not2() - бінарне заперечення.

За допомогою bind() можна пов’язати аргументи, що викликають об’єкт безпосередньо(вказавши, наприклад, конкретне значення) або за допомогою об’єктів, що заповнюють. Ці об’єкти - (\_1,\_2,\_3, ...) визначені в просторі імен placeholders.

Щоб не згадувати цей простір імен в програмі, необхідно використати директиву:

using namespace placeholders;

### std::mem\_fn

Шаблоная функція std::mem\_fn створює клас-гортку для вказівників на члени класу. Цей обєкт може зберігати, копіювати та кикликати член класу по вказівнику.

Звязувачи

### std::bind

Шаблонна функція std::bind зветься з*вязувачем* та предоставляє пвідтримку часткового використання функцій. Вона прив'язує деякі аргументи до функціонального об'єкту, створюючи новий функціональний об'єкт. Тобто виклик звязувача відповідає виклику функціонального об'єкта з деякими певними параметрами. Передавати звязувачу можна або безпосередньо значення аргументів, або спеціальні імена з простору імен std :: placeholders, які вказують звязувачу на те, що даний аргумент буде пов'язаний, і визначають порядок аргументів який повертається в функціональний об'єкт.

### Визначення функції:

template<class F, class... BoundArgs>

unspecified bind(F&& f, BoundArgs&&... bound\_args);

template<class R, class F, class... BoundArgs>

unspecified bind(F&& f, BoundArgs&&... bound\_args);

Тут f — обєкт виклику, bound\_args — список звязаних аргументів. Типом що повертається є функциональний обєкт невизначеного типу T, який може бути застосуваваним в std::function, і для якого виконується std::is\_bind\_expression<T>::value == true. Всередині огортка містить обєкт типу std::decay<F>::type, побудованого з std::forward<F>(f), а такоже по одному обєкту для кожного аргументу аналогичного типу std::decay<Arg\_i>::type.

### std::placeholders

У просторі імен std :: placeholders містяться спеціальні об'єкти \_1, \_2, ..., \_N, де число N залежить від реалізації. Вони використовуються в функції bind для завдання порядку вільних аргументів. Коли такі об'єкти передаються у вигляді аргументів на функцію bind, то для них генерується функціональний об'єкт, в якому, при виклику з непов'язаними аргументами, кожен заповнювач \_N буде замінений на N-й за рахунком непов'язаний аргумент. Для отримання цілого числа k з заповнювач \_K передбачений допоміжний шаблонний клас std :: is\_placeholder. При передачі йому заповнювач, як параметра шаблона, є можливість отримати ціле число при зверненні до його полю value. Наприклад, is\_placeholder <\_3> :: value поверне 3.

### Приклад

#include <iostream>

#include <functional>

int myPlus (int a, int b) {return a + b;}

int main()

{

std::function<int (int)> f(std::bind(myPlus, std::placeholders::\_1, 5));

std::cout << f(10) << std::endl;

}

Результат:

15

Стандартні функтори

Інколи потрібно використати функтор який є достатньо стандартним — наприклад, для того щоб порівнювати два аргументи числового типу або використати додавання, множення і т.п. Для цього можні використати стандартний функтори, що визначений в бібліотеці <functional>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип | Назва | К-ть операндів | Тип, що повертається | Дія |
| Порівняння | equal\_to | Бінарний | bool | x == y |
| not\_equal\_to | Бінарний | bool | x!= y |
| greater | Бінарний | bool | x > y |
| less | Бінарний | bool | x < y |
| greater\_equal | Бінарний | bool | x >= y |
| less\_equal | Бінарний | bool | x <= y |
| Логичні | logical\_and | Бінарний | bool | x && y |
| logical\_or | Бінарний | bool | x || y |
| logical\_not | Унарний | bool | !x |
| Арифметичні | plus | Бінарний | T | x + y |
| minus | Бінарний | T | x - y |
| multiplies | Бінарний | T | x \* y |
| divides | Бінарний | T | x / y |
| modulus | Бінарний | T | x % y |
| negate | Унарний | T | -x |
| Бітові (Сі++11) | bit\_and | Бінарний | T | x & y |
| bit\_or | Бінарний | T | x | y |
| bit\_xor | Бінарний | T | x ^ y |
| bit\_not | Унарний | T | ~x |

Приклад

#include <iostream>

#include <vector>

#include <algorithm>

#include <numeric>

#include <functional>

//using namespace std;

int main(){

std::vector<double> v1, v2(4);

double mas1[] ={1.0,2.3,1.0,4.1};

v1.assign(1.0,4);

std::copy(mas1,mas1+3,v2.begin());

for (std::vector<double>::iterator it = v2.begin(); it!=v2.end(); ++it)

std::cout << ' ' << \*it;

// using predicate comparison:

std::pair<std::vector<double>::iterator, std::vector<double>::iterator> it;

it = std::mismatch (v1.begin(), v1.end(), v2.begin(), std::less<double>() );

std::cout << "Second mismatching elements: " << \*it.first;

std::cout << " and " << \*it.second << '\n';

std::vector<int> val(v2.begin(),v2.end());

std::vector<int> result(4);

std::partial\_sum (val.begin(), val.end(), result.begin(), std::multiplies<int>());

std::cout << "using functional operation multiplies: ";

for (int i=0; i<4; i++) std::cout << result[i] << ' ';

std::cout << '\n';

}