

Алгоритми

Однією з важливих властивостей та власною ціллю створення бібліотеки шаблонів було створення можливостей 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 пріоритетна черга автоматично сортує дані. Однак, інколи було б бажано застосовувати 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 C++ було додано дві бібліотеки `algorithms` та `numeric`, які містять функції, що реалізують деякі з відомих та потрібних у практичному програмуванні алгоритмів.

Ці бібліотеки дозволяють не витратити час на написання циклів для виконання популярних задач таких як сортування, пошук або підрахунок кількості елементів масиву. Це дозволяє також не витратити час на оптимізацію та відлагодження цих алгоритмів. Крім того деякі з цих реалізацій дозволяють використовувати розпаралелювання – можливість виконувати 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:

- немодифікуючі операції з послідовностями (`non-modifying sequence operations`);
- модифікуючі операції з послідовностями (`modifying sequence operations`);
- операції розділення (Партіції)(`Partitions`);
- сортування (`Sorting`);
- бінарний пошук (`Binary search`) на відсортованих або частково відсортованих послідовностях;
 - мінімуми/максимуми (`Min/max`);
- злиття (`Merge`) на відсортованих послідовностях;
- операції на структурі Купа (`Heap`);
- інші.

Немодифікуючі операції з послідовностями

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

Можна виділити наступні підгрупи функцій в даному класі функцій:

Функція `for_each`

Функція `for_each(pos1, pos2, fun)` виконує функцію `fun` для всіх елементів, що знаходяться між двома ітераторами `pos1` та `pos2`.

Приклад.

```
#include <iostream>    // std::cout
#include <algorithm>    // std::for_each
#include <vector>       // std::vector

template<typename T>
void myfunction (T i) { // функція над елементом послідовності
    std::cout << " " << ((i>0)?"positive ":"negative ");
}

int main () {
    double mas[] = {1.0, 2.0, -4.0, 3.0};
    std::vector<int> myvector(mas,mas+4);

    std::cout << "myvector signs:";
    std::for_each (myvector.begin(), myvector.end(), myfunction<int>);
    std::cout << "\narray signs:";
    std::for_each (mas, mas+4, myfunction<double>);

}
```

Результат:

```
myvector signs: positive positive negative positive
array signs: positive positive negative positive
```

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 об'єктів функція використовує оператор рівності(`==`).

Приклад.

```
#include <iostream>    // std::cout
#include <algorithm>    // std::find
#include <vector>       // std::vector
```

```

int main () {

    int myints[] = { 10, 20, 30, -40, 30};
    int * p;
    // пошук в масиві — результат вказівник
    p = std::find (myints, myints+5, 30);
    if (p != myints+5)
        std::cout << "Element found in myints: " << *p << '\n';
    else
        std::cout << "Element not found in myints\n";

    double mas[] = {1.0, 2.0, -4.0, 3.0};
    std::vector<int> myvector(mas,mas+4);
    std::vector<int>::iterator it;
    // пошук в векторі — результат ітератор
    it = std::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";
}

```

Element found in myints: 30
 Element not found in myvector

Для функцій пошуку також можна використовувати форми `find_if`, `find_end`, `find_first_of`, `adjacent_find`, а з C++11 також додана функція `find_if_not`.

```

// adjacent_find example
#include <iostream>    // std::cout
#include <algorithm>    // std::adjacent_find
#include <vector>       // std::vector

template<typename T>
bool myEqual (T x, T y) {
    return (x-y)*(x-y)<0.0001;
}

int main () {
    float myarr[] = {5.0f,2.f,5.0f,3.0f,2.9999f,2.0f,1.0f,1.0001f,2.0f};
    std::vector<float> myvector2 (myarr,myarr+8);
}

```

```

std::vector<float>::iterator it2;
std::vector<int> myvector3(myarr,myarr+8);// 5,2,5,3,2,2,1,1
std::vector<int>::iterator it3;

// порівняння цілих за замовченням
it3 = std::adjacent_find (myvector3.begin(), myvector3.end());

if (it3!=myvector3.end())
    std::cout << "the first value of repeated elements are: " << *it3 << '\n';

//використовуючи власне порівняння-предикат
it2 = std::adjacent_find (myvector2.begin(), myvector2.end(), myEqual<float>);

if (it2!=myvector2.end())
    std::cout << "the first value of almost repeated elements are: " << *it2 << '\n';
}
Результат:
the first value of repeated elements are: 2
the first value of almost repeated elements are: 3

```

Примітка. В усіх випадках де використовується функція як аргумент — замість неї можна (а іноді й бажано) використовувати функтор (або лямбду в C++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 об'єктів функція використовує оператор рівності(`==`).

Приклад

```

// count algorithm example
#include <iostream>    // std::cout
#include <algorithm>    // std::count
#include <vector>       // std::vector

int main () {
    // підрахуємо 10-ки серед елементів масиву:
    int myints[] = { 10, 20, 30, -40, 30, 30};
    double mas[] = {1.0, 2.0, -4.0, 3.0 };
    std::vector<int> myvector(mas,mas+4);

    int mycount = std::count (myints, myints+6, 30);
    std::cout << "10 appears " << mycount << " times.\n";
}

```

// підрахуємо 20-ки серед елементів вектору:

```
mycount = std::count (myvector.begin(), myvector.end(), 20);
std::cout << "20 appears " << mycount << " times.\n";
}
```

10 appears 3 times.

20 appears 0 times.

Також існує функція пошуку `count_if`, у формі `count_if(pos1,pos2,fun)` яка порівнює елементи за умовою, що задається предикатом `fun`.

Приклад.

// count_if

```
#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> myvector4;
```

```
    for (int i=1; i<10; i++) myvector4.push_back(i); // myvector: 1 2 3 4 5 6 7 8 9
```

```
    int mycount = count_if (myvector4.begin(), myvector4.end(), IsOdd);
```

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

```
}
```

Результат:

myvector contains 5 odd values.

Функція `mismatch`

Функція `mismatch` у формах `mismatch(pos1,pos2, col)` та `mismatch(pos1,pos2, col,fun)` знаходить елементи в двох послідовностях (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 find та find_if, шукають не лише єдиний елемент, а діапазон чи колекцію та видають результат у вигляді однонаправленого ітератору на перший знайдений елемент, або на кінцевий елемент, якщо цих елементів не було знайдено.

Інтерфейс функції: search (pos1_1, pos1_2, pos2_1, pos2_2) або Інтерфейс функції: search (pos1_1, pos1_2, pos2_1, pos2_2, predicate)

Функція search_n з інтерфейсом search_n (where_pos1, where_pos2, count, val) або search_n (where_pos1, where_pos2, count, val, predicat) шукає чи входить в даний діапазон count значень val (порівняння може відбуватись по функції у предикаті predicat). Якщо входить, то вона повертає ітератор на перше входження, інакше ітератор на кінець.

Приклад:

```

#include <bits/stdc++.h>
bool mypredicate (int i, int j) {
    return (i%2==j%2); // елементи рівні, якщо мають однакову парність
}

int main () {
    std::vector<int> haystack;
    // встановлюємо значення:    haystack: 10 20 30 40 50 60 70 80 90
    for (int i=1; i<10; i++) haystack.push_back(i*10);
    // шукаємо з предикатом по замовченню
    int needle1[] = {40,50,60,70,60,60};

```

```

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";

// пошук використовуючи власний предикат
int needle2[] = {20,30,50};
it = std::search (haystack.begin(), haystack.end(), needle2, needle2+3, mypredicate);

if (it!=haystack.end())
    std::cout << "needle2 equal parity at position " << (it-haystack.begin()) << "\n";
else
    std::cout << "needle2 equal parity not found\n";

haystack.push_back(90); haystack.push_back(80);
it = std::search_n (haystack.begin(), haystack.end(), 2,90);
if (it!=haystack.end())
    std::cout << "2 90th are found at position " << (it-haystack.begin()) << "\n";
else
    std::cout << "2 90th not found\n";
}

```

Результат:

```

needle1 found at position 3
needle2 equal parity at position 0
2 90th are found at position 8

```

Модифікуючі операції з послідовностями (Modifying sequence operations)

Функції копіювання `copy` та `copy_backward`

Функція `copy` призначена для копіювання діапазону значень з колекції у іншу колекцію. Інтерфейс функції `copy` (`pos1`, `pos2`, `result_pos`), остання позиція вказує за яку позицію потрібно копіювати. Функція `copy_backward` — яке відрізняється тим, що останній параметр вказує не на початок на кінець результату. В C++11 додали ще форми `copy_n`, яка копіює `n` чисел з заданого місця та `copy_if`, де вказується умова копіювання (результатом функції є вказівник на наступний після скопійованого блоку елемент).

```

#include <iostream>    // std::cout, ostream_iterator
#include <algorithm>   // std::copy
#include <vector>       // std::vector
#include <iterator>     // ostream_iterator
using namespace std;

```



```

template<class V,typename T> void printCont(const V & s, T val){
    copy (s.begin(), s.end(), ostream_iterator<T>(cout, " ,"));
    cout<<endl;
}
int main () {
    int myints[]={ 10,20,30,40,50,60,70};
    vector<int> myvector (7);
    vector<int> myvector2(8);

    copy( myints, myints+7, myvector.begin() );
    cout << "myvector contains:";
    printCont(myvector, myvector.at(0));
    cout<<"\n";
    copy_backward( myvector.begin()+3, myvector.end(), myvector.end()-3 );
    printCont(myvector, myvector.at(0));

    copy_n( myvector.begin()+3, 3, myvector2.begin());
    printCont(myvector2, myvector2.at(0));
    auto it = copy_if (myvector.begin(), myvector.end()-5, myvector2.begin()+3,
[] (int x){return x%20==0;} ); // використовуємо лямбду – остача від ділення на
20
    printCont(myvector2, myvector2.at(0));
    it--;
    cout<<"*it="<<*it;
}
Результат:
myvector contains:10 ,20 ,30 ,40 ,50 ,60 ,70 ,

40 ,50 ,60 ,70 ,50 ,60 ,70 ,
70 ,50 ,60 ,0 ,0 ,0 ,0 ,0 ,
70 ,50 ,60 ,40 ,0 ,0 ,0 ,0 ,
*it=40

```

Функції заміни: replace, replace_if, replace_copy, replace_copy_if

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 було модифіковано – 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';
}

```

Функції видалення `remove`, `remove_if`, `remove_copy`, `remove_copy_if`

Методи `remove`, `remove_if`, `remove_copy`, `remove_copy_if` — дозволяють видаляти елементи за значенням або по значенню предикату.

```

#include <iostream> // std::cout
#include <algorithm> // std::remove_if
#include <vector>

```

```

bool IsPower2 (unsigned i) { return (i & (i-1)) == 0; } // чи є число ступенем двійки

```

```

template <class T> // друк значень заданих інтервалом
void printIntVector(T a,
    T b){
    std::cout << "the range contains:";
    while(a!=b){
        std::cout<<*a<<" ";
        ++a;
    }
}

```

```

    }
    std::cout << '\n';
}

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

    // задаємо інтервал вказівниками
    unsigned* pbegin = myints;                       // вказівник на 1
    unsigned* pend = myints+sizeof(myints)/sizeof(int); // вказівник після 9

    pend = std::remove_if (pbegin, pend, IsPower2); // 3, 5, 6, 7, 9

    printIntVector(pbegin, pend);
    unsigned myints2[] = {10,20,30,30,20,10,10,20};

    // задаємо інтервал
    pbegin = myints2;
    pend = myints2+sizeof(myints2)/sizeof(unsigned);

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

    printIntVector(pbegin, pend);
    unsigned myints3[] = {1,2,3,4,5,6,7,8,9};
    std::vector<unsigned> myvector (9);

    std::remove_copy_if (myints,myints+9,myvector.begin(), IsPower2);
    // 3, 5, 6, 7, 9, 6, 7, 9, 0,

    printIntVector(myvector.begin(), myvector.end());
}

```

Результат:

the range contains:3, 5, 6, 7, 9,
the range contains:10, 30, 30, 10, 10,
the range contains:3, 5, 6, 7, 9, 6, 7, 9, 0,

Функції видалення дублікатів `unique` та `unique_copy`

Функції `unique` та `unique_copy` дозволяють видаляти послідовні однакові значення в послідовності

Функції обміну: `swap` та `swap_ranges`, `iter_swap`

Функції `swap` та `swap_ranges`, `iter_swap` здійснюють обмін колекціями або діапазонів значень або значеннями ітераторів.

Функція перетворення: transform

Метод transform застосовує 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';
}
```

Функції заповнення fill, fill_n та generate, generate_n

Методи fill та fill_n дозволяють швидко 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';

```

Методи `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;
}

```

Функції перестановок `reverse` та `reverse_copy`, `rotate` та `rotate_copy`, `random_shuffle`

Методи `reverse` та `reverse_copy` — дозволяють інвертувати колекцію
Методи `rotate` та `rotate_copy` — роблять циклічний зсув колекції
Метод `random_shuffle` — випадковим чином перемішує послідовність.
З C++11 додали також функцію `shuffle`, яка перемішує послідовність за допомогою даного генератору псевдовипадкових чисел

Операції розділення (`partitions`)

Функція `partition`

Функція `partition (pos1, pos2, pred)` модифікує діапазон `[pos1, pos2)` таким чином, що елементи для яких предикат `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 зобов'язаний зберігатися, якщо його потрібно зберегти використовується функція `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
std::vector<int> myvector2;

// set some values:
for (int i=1; i<10; ++i) myvector2.push_back(i); // 1 2 3 4 5 6 7 8 9

std::vector<int>::iterator bound;
bound = std::stable_partition (myvector2.begin(), myvector2.end(), IsOdd);

// print out content:
std::cout << "odd elements:";
for (std::vector<int>::iterator it=myvector2.begin(); it!=bound; ++it)
    std::cout << ' ' << *it;
std::cout << '\n';

std::cout << "even elements:";
for (std::vector<int>::iterator it=bound; it!=myvector2.end(); ++it)
    std::cout << ' ' << *it;
std::cout << '\n';

return 0;
}

```

Примітка. З C++11 додали також функції `is_partitioned`, `partition_copy`, `partition_point`

Операції сортування (Sorting)

Функція `sort`

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

`stable_sort` — працює майже так саме як `sort`, але при цьому обов'язково 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;
}
```

Функція `partial_sort` та `partial_sort_copy`

Функція часткового сортування `partial_sort` (`pos1,middle,pos2`) та `partial_sort` (`pos1, middle, pos2, comp`) — сортує елементи діапазону таким чином, що елементи перед `middle` стають найменшими елементами діапазону та відсортовані за неспаданням.

Критерій порівняння - `operator<` для першої версії та бінарний предикат `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;
}
```

Функція `partial_sort_copy` працює так саме як `partial_sort`, але зберігає невідсортовану частину.

Функція nth_element

Функція `nth_element(pos1,middle,pos2)` модифікує діапазон 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 C++11 додали функції `is_sorted`, `is_sorted_until` — які визначають чи відсортований даний діапазон або чи відсортований він до деякого елемента.

Бінарний пошук

Алгоритми бінарного пошуку працюють на відсортованих або частково відсортованих колекціях

Функція `binary_search`

Ця функція виконує бінарний пошук даного елементу 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;
}
```

Функція `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
#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;
}

```

Функції 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;
}
```

Функції злиття

Дані функції працюють для відсортованих послідовностей.

Функції merge та inplace_merge

Функція merge виконує алгоритм злиття двох відсортованих послідовностей, а функція 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;
}
```

```
}
```

Функція includes

Ця функція 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;
}
```

Функції роботи з множинами set_union, set_intersection, set_difference, set_symmetric_difference

Функції **set_union, set_intersection, set_difference, set_symmetric_difference**

- виконують відповідно об'єднання, перетин, різницю та симетричну різницю двох відсортованих 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';
}

```

Мінімум/Максимум

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

Функції `min_element`, `max_element` - повертають відповідно однонаправлений ітератор на мінімальний та максимальний елементи послідовності

З C++11 додали також функції:

Функції `minmax`, `minmax_element` — які повертають відповідно одночасно значення мінімального та максимального елементів або однонаправлений ітератор на них у вигляді пари.

Приклад

```

#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int arr[] = {10, 20, 5, 23, 42, 15};
    int n = sizeof(arr)/sizeof(arr[0]);
    vector<int> vect(arr, arr+n);
    // переверс вектору
    reverse(vect.begin(), vect.end());
}

```

```

// виведення вектору
cout << "Vector after reversing is: ";
for (int i=0; i<vect.size(); i++)
    cout << vect[i] << " ";
// тестуємо функції пошуку мінімуму та максимуму
cout << "\nMaximum element of vector is: ";
cout << *max_element(vect.begin(), vect.end())<<" or " << max({10, 20, 5, 23
,42 , 15});
cout << "\nMinimum element of vector is: ";
cout << *min_element(vect.begin(), vect.end())<<" or " << min({10, 20, 5, 23
,42 , 15});

cout << "\nmin:"<< *minmax_element(vect.begin(), vect.end()).first << " max=" <<
minmax({10, 20, 5, 23 ,42 , 15}).second;
}

```

Vector after reversing is: 15 42 23 5 20 10
 Maximum element of vector is: 42 or 42
 Minimum element of vector is: 5 or 5
 min:5 max=42

Робота з купою

Функції `make_heap`, `push_heap`, `pop_heap`, `sort_heap` – відповідно створюють структуру даних купи, 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;
}
3 Ci++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;
}

```

Інші

До інших функцій входять достатньо рідко використовувані функції `lexicographical_compare` (порівняння вмісту послідовностей лексикографічно), `next_permutation` (наступне перемішування) та `prev_permutation` (попереднє перемішування).

Алгоритми бібліотеки `numeric`

В бібліотеку `numeric` входять наступні функції:

Функція accumulate

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

// Сума елементів

```
#include <iostream>
#include <numeric> // для алгоритмів
using namespace std;
int accum1_massiv(){
    const int N = 8;
    int a[N] = {4, 12, 3, 6, 10, 7, 8, 5}, sum = 0;
    sum = accumulate(a, a+N, sum);
    cout << "Sum of all elements: " << sum << endl;
    cout << "1000 + a[2] + a[3] + a[4] = " << accumulate(a+2, a+5, 1000) << endl;
    return 0;
}
```

```
int accum2_massiv(){
    const int N = 4;
    int a[N] = {2, 10, 5, 3}, prod = 1;
    prod = accumulate(a, a+N, prod, multiplies<int>());
    cout << "Product of all elements: " << prod << endl;
    return 0;
}
```

```
class fun {
public:
    fun(){i = 1;}
    int operator()(int x, int y) {
        int u = x + i * y; i *= 2;
        return u;
    }
private:
    int i;
};
```

```
int accum3_massiv() {
    const int N = 4;
    int a[N] = {7, 6, 9, 2},
    prod = 0;
```

```

        prod = accumulate(a, a+N, prod, fun());
        cout << prod << endl;
        return 0;
    }

```

```

int main(){
    accum1_massiv();
    accum2_massiv();
    accum3_massiv();
}

```

Результат:

Sum of all elements: 55

$1000 + a[2] + a[3] + a[4] = 1019$

Product of all elements: 300

71

Функція `adjacent_difference`

Якщо x — це елемент $[first, last)$ та y — елемент в `result`, то результат цієї функції буде наступний:

$y_0 = x_0$

$y_1 = x_1 - x_0$

$y_2 = x_2 - x_1$

$y_3 = x_3 - x_2$

$y_4 = x_4 - x_3$

Функція `inner_product`

Функція рахує кумулятивний скалярний добуток інтервалу `init` де добуток рахується для пари на яку вказують числа з `first1` та `first2`. Дві операції за замовченням (додавання для результатів та множення для пар) можуть бути перевантажені бінарними функціями `binary_op1` та `binary_op2`.

Функція `partial_sum`

Якщо x представляє елемент в $[first, last)$, а y представляє елемент в `result`, то результат функції може бути представлений:

$y_0 = x_0$

$y_1 = x_0 + x_1$

$y_2 = x_0 + x_1 + x_2$

$y_3 = x_0 + x_1 + x_2 + x_3$

$y_4 = x_0 + x_1 + x_2 + x_3 + x_4$

Приклади:

```

#include <iostream>    // std::cout
#include <functional>  // std::multiplies
#include <numeric>     // std::adjacent_difference

int myop (int x, int y) {return x^y;}
int myaccumulator (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';

    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, myop);
    std::cout << '\n';

    //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';

}

```

Результат:

```

using default adjacent_difference: 1 1 1 2 4 2 1
using functional operation multiplies: 1 2 6 15 45 99 132
using custom function: 1 3 1 6 12 2 7
using default inner_product: 240
using functional operations: 70
using custom functions: 38
using default partial_sum: 1 3 6 11 20
using functional operation multiplies: 1 2 6 30 270
using custom function: 1 3 0 5 12

```

Функція `iota` (C++11)

В C++11 до цього переліку функцій також додали функцію `iota`, яка за даним інтервалом `[first,last)` повертає послідовні значення `val`, які отримуються 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';
}

```

```
}
```

Результат:

numbers: 100 101 102 103 104 105 106 107 108 109

Функтори та предикати

Багато з розглянутих алгоритмів C++ 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 безпосередньо 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 об'єкт, 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 C++ 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 C++ для подібних модифікацій функції можна використати функтори, які також звуться функціональними об'єктами (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 об'єкт або функтор – це клас який перевантажує оператор виклику `operator ()` наступним чином, що в коді

```
FunctionObjectType func;  
func();
```

вираз `func()` є викликом оператора `()` об'єкту `func`, а не викликом функції `func`. Тип функціонального об'єкту повинен бути визначеним наступним чином:

```
class FunctionObjectType {  
    public:  
    void operator() () {  
        // Код функції  
    }  
};
```

У використанні функціональних об'єктів є ряд переваг 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, вбудоване (inline) звернення до оператора () класу працює швидше, ніж функція, передана 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 Cі++, 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 об'єкт а 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 предикатом. Зокрема предикати потрібні при використанні таких функцій з бібліотеки алгоритмів, як find_if(), count_if, sort тощо.

Предикати використовуються в алгоритмах сортування, пошуку, 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

```
#include <iostream>  
#include <algorithm>
```

```
class DividedByTwo{ // чи парний даний елемент  
public:  
    bool operator()(const int x) const    {  
        return x % 2 == 0;  
    }  
};
```

```
int main(){  
    const std::size_t N = 5;  
    int A[N] = {3, 2, 5, 6, 8};  
    std::cout << std::count_if(A, A + N, DividedByTwo());  
}
```

Результат:

3

Розглянемо ще один приклад використання предикату. Для того, щоб передати предикату критерій, необхідно в тілі структури створити конструктор так, як показано на прикладі.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

struct Prd {
    int my_cnt;
    // Конструктор
    Prd(const int &t) : my_cnt(t) {}
    // Перегрузка операції () - чи більший елемент за my_cnt
    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);

    const int z = count_if(num.begin(), num.end(), Prd(4)); /* кількість елементів
більших 4 - повинно бути 3 */
    vector<int> myvector(num.size() - z); /* виділяємо вектор для елементів
менших 4 */
    /* або робимо це методами з functional C++11
    using namespace placeholders; // c++11
    const int z = count_if(num.begin(), num.end(), bind(logical_not<bool>(),
bind(Prd(4), _1))); /* кількість елементів менших 4 */
    vector<int> myvector(z); */

    vector<int>::iterator it = remove_copy_if(num.begin(), num.end(),
myvector.begin(), Prd(4)); // видаляємо елементи більші 4

    cout<< endl; // та виводимо те що залишилось
    for(vector<int>::iterator it2 = myvector.begin(); it2!=myvector.end();++it2)
        cout<<*it2<<" ";
```

```
}
```

Результат:

1, 2, 3, 4, 3, 3, 3,

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.

Огортки функцій та стандартні функтори бібліотеки `function`

Починаючи зі стандарту C++ 11 шаблонний клас `function` з бібліотеки `<function>` є поліморфною обгорткою функцій для загального використання. Об'єкти класу `function` можуть зберігати, копіювати і викликати довільні об'єкти, що можуть викликатись – 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 C++11 визначені функції-модифікатори `swap` та `assign` і оператори порівняння (`==` та `!=`) з `nullptr`. Доступ до цільового об'єкту дає функція `target`, а до його типу — `target_type`. Оператор приведення `function` до булевого типу повертає `true`, коли у класу є цільовий об'єкт.

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; }
```

```
};
```

```
// похідна через формулу різниць
```

```
std::function<double(double)> derivative(const std::function<double(double)> &f, const double h){
```

```
    return [=](double x)->double { return (f(x+h)-f(x-h))/2/h; };
```

```

}

int main(){
    // визначаємо функтор через вказівник на функцію
    std::function<void(char)> f_print_Letter = printLetter;
    f_print_Letter('Q');

    // визначаємо функтор як лямбда-вираз
    std::function<void()> f_print_Hello = [] () { std::cout << "Hello world!" << std::endl; };
    f_print_Hello();

    // визначаємо функтор через зв'язувач
    std::function<void()> f_print_Z = std::bind(printLetter, 'Z');
    f_print_Z();

    // визначаємо функтор як метод класу
    std::function<void(const A&, char)> f_printA = &A::printNumberLetter;
    A a(10);
    f_printA(a, 'A');

    // присвоєння функторів
    B b;
    std::function<void()> f_B = b;
    f_B();

    // рахуємо похідну функції як функцію
    std::function<double(double)> fd = [](double x)->double{ return x*x; } ;
    std::cout<<derivative(fd,0.001)(2.0);

}
Результат:
Q
Hello world!
Z
Number: 10 Letter: A
B()
4

```

Виключення bad_functional_call

Виключення типу `bad_functional_call` буде створено при спробі виклику функції `function::operator()`, якщо в неї буде відсутній цільовий об'єкт. 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;
}
}

```

Адаптори функторів

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

1. `bind()` - зв'язує аргументи з операцією.

Шаблонна функція `std::bind` зветься зв'язувачем та надає підтримку часткового використання функцій. Вона прив'язує деякі аргументи до функціонального об'єкту, створюючи новий функціональний об'єкт. Тобто виклик зв'язувача відповідає виклику функціонального об'єкта з деякими певними параметрами. Передавати зв'язувачу можна або безпосередньо значення аргументів, або спеціальні імена з простору імен `std::placeholders`, які вказують зв'язувачу 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` — об'єкт виклику, `bound_args` — список зв'язаних аргументів. Типом що повертається є функціональний об'єкт невизначеного типу `T`, який може бути застосовуваним в `std::function`, і для якого виконується `std::is_bind_expression<T>::value == true`. Всередині огортки міститься об'єкт типу `std::decay<F>::type`, побудованого з `std::forward<F>(f)`, а також по одному об'єкту для кожного аргументу аналогічного типу `std::decay<Arg_i>::type`.

За допомогою `bind()` можна пов'язати аргументи, що викликають об'єкт безпосередньо(вказавши, наприклад, конкретне значення) або за допомогою об'єктів, що заповнюють. Щоб не згадувати цей простір імен в програмі, необхідно використати директиву:

```
using namespace 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 шаблону, є можливість отримати ціле число при зверненні до його поля `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

2. `mem_fn()` - викликає операцію, що вказує на функцію-член об'єкту;

mem_fn

Шаблонная функція `std::mem_fn` створює клас-огортку для вказівників на члени класу. Цей об'єкт може зберігати, копіювати та викликати член класу по вказівнику.

3. Адаптори `not1` та `not2` дозволяють використовувати у адаптерах заперечення відповідно для бінарних та унарних функцій.

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 можна використати стандартний функтори, що визначений в бібліотеці `<functional>`

Тип	Назва	К-ть операндів	Тип, що повертається	Дія
Порівняння	<code>equal_to</code>	Бінарний	<code>bool</code>	<code>x == y</code>
	<code>not_equal_to</code>	Бінарний	<code>bool</code>	<code>x != y</code>
	<code>greater</code>	Бінарний	<code>bool</code>	<code>x > y</code>
	<code>less</code>	Бінарний	<code>bool</code>	<code>x < y</code>
	<code>greater_equal</code>	Бінарний	<code>bool</code>	<code>x >= y</code>
	<code>less_equal</code>	Бінарний	<code>bool</code>	<code>x <= y</code>
Логичні	<code>logical_and</code>	Бінарний	<code>bool</code>	<code>x && y</code>
	<code>logical_or</code>	Бінарний	<code>bool</code>	<code>x y</code>
	<code>logical_not</code>	Унарний	<code>bool</code>	<code>!x</code>
Арифметичні	<code>plus</code>	Бінарний	<code>T</code>	<code>x + y</code>
	<code>minus</code>	Бінарний	<code>T</code>	<code>x - y</code>
	<code>multiplies</code>	Бінарний	<code>T</code>	<code>x * y</code>
	<code>divides</code>	Бінарний	<code>T</code>	<code>x / y</code>
	<code>modulus</code>	Бінарний	<code>T</code>	<code>x % y</code>
	<code>negate</code>	Унарний	<code>T</code>	<code>-x</code>
Бітові (C++11)	<code>bit_and</code>	Бінарний	<code>T</code>	<code>x & y</code>
	<code>bit_or</code>	Бінарний	<code>T</code>	<code>x y</code>
	<code>bit_xor</code>	Бінарний	<code>T</code>	<code>x ^ y</code>
	<code>bit_not</code>	Унарний	<code>T</code>	<code>~x</code>

Приклад.

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <functional>

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;
    // функція mismatch з порівнянням за допомогою предикату
    std::pair<std::vector<double>::iterator, std::vector<double>::iterator> it;
    std::cout << "\n";
    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);
    // часткова сума з допомогою предикатів
    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';
}

```

Результат:

1 2.3 1 0

Second mismatching elements: 4 and 1

using functional operation multiplies: 1 2 2 0