Podstawy programowania obiektowego

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

http://home.agh.edu.pl/~pszwed/

Aktualizacja: 02.06.2020

12. Biblioteka standardowa C++

Biblioteka standardowa

- Wielu producentów kompilatorów oferujących początkowo kontenery obiektowe rozbudowało swoje biblioteki o kontenery wykorzystujące szablony.
- Około 1995 roku pojawiła się biblioteka STL (Standard Template Library), która przerodziła się w standardową bibliotekę C++.
- Zalety:
  - w obecnej chwili biblioteka jest faktycznym standardem;
  - dostarczana jest w postaci kodu źródłowego szablonów
  - biblioteka definiuje wiele podstawowych klas: string, list, vector, map, set, itd.
  - biblioteka implementuje standardowe algorytmy (np.: sort, replace)
  - biblioteka oferuje *poprawione* wersje klas strumieni
  - kod biblioteki został zoptymalizowany

STL - wprowadzenie

- Szablony STL umieszczone są w plikach o charakterystycznych nazwach – nie mają one rozszerzenia. Nazwy plików odpowiadają nazwom szablonów.
- Wszystkie definicje STL umieszczone są w odrębnej przestrzeni nazw (ang. *namespace*) `std`.
- Stąd nazwą kontenera jest na przykład `std::vector`. Aby pominąć przedrostek std należy skorzystać z deklaracji

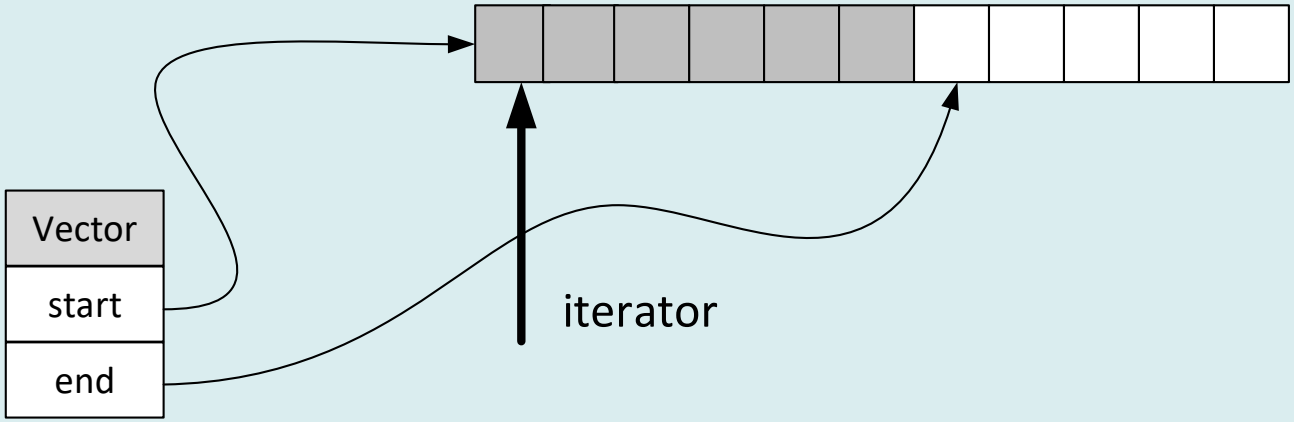
  `using namespace std;`

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

int  main()
{
    std::vector<int> v1;
    vector<int>v2;
    for(int i=0;i<20;i++)v1.push_back(i);
}
```

STL wprowadzenie

- Biblioteka STL w specyficzny sposób definiuje pojęcie iteratora. Iteratory mogą być traktowane jak wskaźniki, które podczas iteracji będą przebiegały kolejne elementy kontenera, aż do momentu, kiedy osiągną adres danych poza kontenerem.
- Aby uzyskać dostęp do elementu należy użyć operatora dereferencji *.
- Jeżeli obiekty w kontenerze należą do pewnej klasy, dostęp do pól i metod obiektu zapewni operator ->.



STL przykład iteracji

```
int  main(){
    std::vector<int> v1;
    for(int i=0;i<20;i++)v1.push_back(i);

    for(vector<int>::const_iterator it=v1.begin();
        it!=v1.end();
        ++it)
        cout<<*it<<" ";
}
```

- `vector<int>::const_iterator` – jest to typ iteratora zdefiniowany jako klasa wewnętrzna szablonu.
- `it=v1.begin()` – ustawia iterator tak, by wskazywał pierwszy element kontenera
- `it!=v1.end()` – testuje, czy iterator nie wyszedł poza sekwencję elementów w kontenerze
- `++it`– inkrementuje iterator
- `*it` – realizuje dostęp do elementu wskazywanego przez iterator

Iteracja w C++11

- Automatyczne określenie typu iteratora

```
for(auto it=v1.begin();it!=v1.end();++it)
    cout<<*it<<" ";
```

- Pętla typu *dla każdego* (range-based)
  - Wartości elementów

```
for (int e:v1) cout << e << " ";
```

  - Referencje do elementów

```
for (int& e:v1) e=-e;
```

  - Automatyczne określenie typu elementu

```
for (auto e:v1) cout << e << " ";

for (auto&e:v1) e=e*e;
```

Iteracja w C++11

```
int  main() {
    std::vector<int> v1;
    for (int i = 0; i < 20; i++)v1.push_back(i);
    for(auto it=v1.begin();it!=v1.end();++it)
        cout<<*it<<" ";
    cout<<endl;
    for (int e:v1) cout << e << " ";
    cout<<endl;
    for (int& e:v1) e=-e;
    for (auto e:v1) cout << e << " ";
    cout<<endl;
    for (auto&e:v1) e=e*e;
    for (auto e:v1) cout << e << " ";
    cout<<endl;
}
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16 -17 -18 -19
0 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361
```

Wybrane kontenery STL

Szablon <string>

- Szablon definiuje klasę `std::string` reprezentującą tablicę znakową, która może dynamicznie zmieniać swoje rozmiary.
- Poza funkcjami do zarządzania pamięcią, klasa zawiera szereg metod umożliwiających dostęp do znaków lub działania na całych tekstach (konkatenacja, usuwanie, wstawianie, zastępowanie, itd.)

Zarządzanie pamięcią

`capacity()` Zwraca rozmiar bufora.
`reserve (size_type n =0)` Zmienia rozmiary bufora. Gwarantuje, że po jej wykonaniu funkcja capacity() będzie zwracała co najmniej n.
`size()` lub `length()` Zwraca długość tekstu (liczbę znaków umieszczonych w buforze).
`resize(size_type n, E c = E())` Gwarantuje, że funkcja `size()`będzie zwracała co najmniej `n`. W razie potrzeby powiększa tablicę i wypełnia znakiem `c`.

Przykład

```
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char* argv[]){
    string s;
    ifstream is("readme.txt");
    s.reserve(40);
    for(int i=0;;i++){
        int c=is.get();
        if(c<0)break;
        s+=(char)c;  // lub: s.append(1,c);
        if(i%100==0){
            cout<<s.capacity()<<" ";
            cout<<"("<<s.size()<<") ";
        }
    }
    cout<<endl<<s;
    return 0;
}
```

40 (1) 160 (101) 320 (201) 320 (301) 640 (401) 640 (501) 640 (601) 1280 (701) 1280 (801) 1280 (901) 1280 (1001) 1280 (1101) 1280 (1201) 2560 (1301) 2560 (1401) 2560 (1501) 2560 (1601) 2560 (1701) 2560 (1801) 2560 (1901) 2560 (2001) 2560 (2101) 2560 (2201) 2560 (2301) 2560 (2401) 2560 (2501) 8135 (2601) …tekst pliku …

Metoda c_str()

Metoda zwraca stały wskaźnik do tekstu. Może być użyta tam, gdzie wymagane jest użycie danych typu `const char*`.

```
int main(){
    string s="Tekst";
    cout<<strcmp(s.c_str(),"Tekst")<<endl;
}
```

Dostęp do znaków

Użytkownik klasy string może zrealizować dostęp do znaków z wykorzystaniem `operatora []`, funkcji `at()` lub *iteratorów*.

```
string s="abcdefghij";
s[5]='F';
string::iterator it=s.begin();
*it='A';

s.at(1)='B';
for(string::const_iterator ci=s.begin();
    ci!=s.end();ci++)cout<<*ci;
cout<<endl;
```

ABcdeFghij

Dostęp do znaków

```
for(string::reverse_iterator ri=s.rbegin();
    ri!=s.rend();ri++)cout<<*ri;
cout<<endl;


string s2;
s2.resize(10);
string::reverse_iterator ri;
for(ri=s.rbegin(),it=s2.begin();ri!=s.rend();
    ri++,it++)*it=*ri;
cout<<s2<<endl;
```

Wypisywanie od końca
jihgFedcBA

Kopiowanie od końca
jihgFedcBA

Uwaga: iterator `begin()` i funkcja `c_str()` zazwyczaj zwracają wskaźnik do tego samego obszaru pamięci. Jednakże nie należy używać ich zamiennie. Wywołanie iteratora nie gwarantuje obecności znaku 0 na końcu.

Konkatenacja tekstów

Do konkatenacji (łączenia) tekstów służy metoda append( ) występująca w kilku przeciążonych wersjach lub operatory + i +=.

```
int main(int argc, char* argv[])
{
    string s("Ala ma");
    s.append(" kota");
    s+=" i psa";
    cout<<endl<<s;
    return 0;
}
```

Ekstrakcja tekstów

Do wyodrębniania fragmentów tekstu służy metoda `substr()`. Jej parametrami są indeks początkowy i długość tekstu.

```
int main(int argc, char* argv[])
{
    string s="abcdefgijklmn";
    cout<<s.substr(3,3)<<endl;
    return 0;
}
```

Rezultat: `def`

Usuwanie fragmentów tekstu

W klasie string zdefiniowano kilka przeciążonych wersji metody `erase()` pozwalającej na usuwanie fragmentów (lub całego) tekstu.

```
int main(int argc, char* argv[])
{
    string s="abcdefgijklmn";
    int n1= s.find('d');
    int n2 = 3; // ile znakow
    s.erase(n1,n2);
    cout<<s<<endl;
    s="abcdefgijklmn";
    s.erase(s.begin()+2,s.begin()+4);
    cout<<s<<endl;
    return 0;
}
```

Rezultat:
abcgijklmn
abefgijklmn

Wstawianie fragmentów tekstu

Do wstawiania fragmentów tekstu służy (przeciążona na kilka sposobów) funkcja `insert()`. Funkcja wstawia w miejsce określone przez indeks lub iterator tekst, który może być tablicą znaków, fragmentem innego obiektu klasy string lub sekwencją identycznych znaków.

```
int main(int argc, char* argv[]){
    string s("Ala ma kota");
    s.insert(strlen("Ala ma "), "psa i ");
    cout<<s<<endl;
    s="abcdefgijklmn";
    s.insert(1,2,'x');
    cout<<s<<endl;
    return 0;
}
```

Rezultat:
Ala ma psa i kota
axxbcdefgijklmn

Znajdywanie i zastępowanie fragmentów tekstu

Funkcja `find()` pozwala na wyszukanie w tekście łańcucha znaków lub wystąpienia znaku. Funkcja `replace()` pozwala na zastąpienie wskazanego fragmentu tekstu innym.

```
int main(int argc, char* argv[])
{
    string s("Witaj $user, jestes zalogowany");
    string userTag="$user";
    int start=s.find(userTag);
    s.replace(start,userTag.size(),"Piotr Szwed");
    cout<<s<<endl;
    return 0;
}
```

```
Witaj Piotr Szwed, jestes zalogowany
```

Adaptacja klasy

Klasa `string` nie jest szablonem, ale klasą powstałą w wyniku instancjacji szablonu `basic_string` typem `char`. Jest ona zdefiniowana w postaci deklaracji `typedef`:

```
typedef basic_string<char> string;
```

Intencją autorów biblioteki było stworzenie ogólnego szablonu, który pozwalałby na obsługę łańcuchów znakowych także dla innych platform i języków (Unicode, jezyków azjatyckich).
Szablon `basic_string` zdefiniowany jest jako:

```
template<class charT,
       class traits = char_traits<charT>,
       class allocator = allocator<charT> >
class basic_string;
```

Adaptacja klasy

Klasa `traits` „własności” definiuje kilkanaście funkcji statycznych określających podstawowe cechy znaków:

```
static void assign(E& x, const E& y);
static E *assign(E *x, size_t n, const E& y);
static bool eq(const E& x, const E& y);
static bool lt(const E& x, const E& y);
static int compare(const E *x, const E *y, size_t n);
static size_t length(const E *x);
static E *copy(E *x, const E *y, size_t n);
static E *move(E *x, const E *y, size_t n);
static const E *find(const E *x, size_t n, const E& y);
static E to_char_type(const int_type& ch);
static int_type to_int_type(const E& c);
static bool eq_int_type(const int_type& ch1, const int_type& ch2);
static int_type eof();
static int_type not_eof(const int_type& ch);
```

Przykład

Klasa typu string, której operator == ignoruje wielkość liter (na podst. Thinking In C++)

```
class ichar_traits :
        public  std::char_traits<char>
{
public:
    static int compare(const char* str1,
                       const char* str2, size_t n) {
        return strncasecmp(str1,str2,n);
    }
};

typedef basic_string<char, ichar_traits,
        allocator<char> > istring;

int main() {
    istring first = "tHis";
    istring second = "ThIS";
    cout << (first==second) << endl;
}
```

Lista – szablon <list>

Szablon definiuje listę dwukierunkową. Możliwe jest:

- dodawanie elementów na początku i końcu listy,
- wstawianie, usuwanie, łączenie ze sobą list.
- automatyczne sortowanie listy.

Nie jest możliwy swobodny dostęp do elementów za pośrednictwem operatora [].

Lista jest efektywna, jeżeli liczba elementów i ich miejsce wstawienia nie są z góry określone.

Lista – szablon <list>

```
#include <list>
int main()
{
    list<int> li;
    for(int i=0;i<10;i++)
        li.push_back(-rand()%100);
    for(int i=0;i<10;i++)
        li.push_front(rand()%100);
    for(list<int>::const_iterator it=li.begin();
        it!=li.end();it++) cout<<*it<<" ";
    cout<<endl;
    li.sort();
    for(auto it=li.begin();it!=li.end();it++)
        cout<<*it<<" ";
    cout<<endl;
}
```

```
32 69 66 30 93 8 5 43 51 11 -33 -43 -62 -29 0 -8 -52 -56 -56 -19
-62 -56 -56 -52 -43 -33 -29 -19 -8 0 5 8 11 30 32 43 51 66 69 93
```

Lista - sortowanie

Aby możliwe było automatyczne sortowanie elementów listy za pomocą metody `sort()`, klasa, której obiekty są umieszczone na liście musi zapewniać operatory do porównywania elementów

```
class A{
public:
    bool operator==(const A&)const{return false;}
    bool operator<(const A&)const{return false;}
};

int main(){
    list<A> ali;
    for(int i=0;i<10;i++)ali.push_back(A());
    ali.sort();
}
```

Skompiluje się, ale sortowanie nie ma sensu

Lista - sortowanie

```
class Indexes{
public:
    int row;
    int col;
    Indexes(int _r,int _c):row(_r),col(_c){}
    bool operator==(const Indexes&o)const{
        return row==o.row && col==o.col;
    }
    bool operator<(const Indexes&o)const{
        if(row<o.row)return true;
        if(row==o.row && col<o.col)return true;
        return false;
    }
};
```

Lista - sortowanie

```
int main(){
    list<Indexes> list;
    for(int i=0;i<10;i++){
        list.push_back(Indexes(rand()%3,rand()%4));
    }
    for(Indexes&idx:list){
        cout<<"("<<idx.row<<" "<<idx.col<<") ";
    }
    cout<<endl;
    list.sort();
    for(Indexes&idx:list){
        cout<<"("<<idx.row<<" "<<idx.col<<") ";
    }
}
```

(0 1) (2 2) (0 0) (0 0) (1 0) (2 3) (2 3) (2 0) (0 2) (2 1)
(0 0) (0 0) (0 1) (0 2) (1 0) (2 0) (2 1) (2 2) (2 3) (2 3)

Słownik – szablon <map>

- Słownik jest zapisem funkcji odwzorowującej *klucze* w *wartości*. Jest więc zbiorem par (*klucz*, *wartość*).
  - W słowniku może wystąpić tylko pojedyncza instancja klucza.
  - Dana wartość może być przypisana wielu kluczom.
- Szablon map jest optymalizowany, aby zapewnić dużą prędkość wyszukiwania elementów słownika, dlatego jest implementowany jako drzewo.
- Klasa może być użyta jako klucz, jeżeli zapewnia operatory do porównywania elementów == oraz < lub jest typem wbudowanym zapewniającym te operatory domyślnie.

Przykład

```cpp
#include <map>

int main()
{
    map<string,int> dict;
    dict.insert(map<string,int>::value_type("open",0));
    dict.insert(pair<string,int>("o",0));
    dict.emplace("close",1);
    dict["c"]=1;
    dict["exit"]=-1;
    map<string,int>::const_iterator i;
    for(i=dict.begin();i!=dict.end();i++){
        cout<<i->first<<" -> "<<i->second<<endl;
    }
    //...
```

Przykład

```
//...
    for(;;){
        char command[256];;
        cin.getline(command,256);
        i = dict.find(command);
        if(i==dict.end()){
            cout<<command<<" ???";continue;
        }
        cout<<i->second<<endl;
        if(i->second==-1)return 0;
    }
}
```

Rezultat:

c -> 1
close -> 1
exit -> -1
o -> 0
open -> 0
... Liczby odpowiadające poleceniom

Przykład – macierz rzadka

```
class SparseMatrix {
public:
    map<Indexes, double> map;

    void set(int r, int c, double v) {
        map.emplace(make_pair(Indexes(r, c), v));
    }

    double get(int r, int c) {
        auto it = map.find(Indexes(r, c));
        if (it == map.end())return 0;
        return it->second;
    }
    Indexes getShape() const;
};
```

Przykład – macierz rzadka

Funkcja wyznacza liczbę wierszy i kolumn.
Mapa przechowuje elementy typu pair z polami:

- first – pierwszy parametr szablonu (klucz)
- second – drugi parametr szablonu (wartość)

```
Indexes SparseMatrix::getShape()const{
    Indexes ret(-1,-1);
    for(const auto&e:map) {
        if (e.first.row > ret.row)ret.row = e.first.row;
        if (e.first.col> ret.col)ret.col = e.first.col;
    }
    ret.row++;
    ret.col++;
    return ret;
}
```

Przykład – macierz rzadka

```
int main(){
    SparseMatrix m;
    for(int i=0;i<5;i++){
        m.set(i,i,i+1);
    }
    Indexes shape= m.getShape();
    for(int i=0;i<shape.row;i++){
        for(int j=0;j<shape.col;j++){
            cout<<m.get(i,j)<<" ";
        }
        cout<<endl;
    }
}
```

```
1 0 0 0 0
0 2 0 0 0
0 0 3 0 0
0 0 0 4 0
0 0 0 0 5
```

Zbiór – szablon <set>

Zbiór jest kontenerem, który przechowuje unikalne wartości elementów. Szablon definiuje trzy podstawowe metody:

- `insert()` – dodaje element
- `empty()` – testuje czy zbiór jest pusty
- `find()` – znajduje element w zbiorze

Zbiór może być traktowany jak zdegenerowana postać słownika: zapisuje odwzorowanie *klucz→wartość*, ale to, w co odwzorowany jest klucz jest nieistotne: liczy się obecność klucza w zbiorze.

Zbiór jest implementowany jako drzewo, dlatego elementy zbioru muszą spełniać takie same wymagania dotyczące interfejsu, jak klucze dla słownika: definiować operatory == oraz <.

Przykład

```
#include <set>
int main()
{
    set<int> cont;
    cout<<(cont.empty()?"empty":"!empty")<<endl;
    for(int i=10;i>=0;i--)cont.insert(i);
    for(int i=5;i<15;i++)cont.insert(i);
    cout<<(cont.empty()?"empty":"!empty")<<endl;

    const auto it = cont.find(10);
    if(it != cont.end())cout<<"has:"<<*it<<endl;

    for(auto it=cont.begin();it!=cont.end();it++)
        cout<<*it<<" ";
    cout<<endl;
}
```

```
empty
!empty
has:10
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

Szablon priority_queue

Szablon definiuje kolejkę uporządkowanych elementów.

- `push()` – dodaje element do kolejki sortując ją
- `top()` – zwraca pierwszy (największy element)
- `pop()` – usuwa pierwszy element z kolejki

```
#include <queue>

int main(){
    priority_queue<int> k;
//  priority_queue<int,vector<int>,greater<int>> k;
    for(int i=0;i<10;i++)k.push(rand()%100);
    while(!k.empty()){
        cout << k.top()<<" ";
        k.pop();
    }
    cout<<endl;
}
```

```
62 56 56 52 43 33 29 19 8 0
// 0 8 19 29 33 43 52 56 56 62
```

Przykład

Zliczanie słów. Które występują najczęściej?

```
void read(map<string,int>&bow){
    ifstream ifs("w-pustyni.txt");
    while(ifs){
        char buf[8192];
        ifs.getline(buf,8192);
        const char*sep=" \t,.:;-?!()—\r";
        for(char*ptr=strtok(buf,sep);ptr;ptr=strtok(0,sep)){
            if(*ptr)bow[ptr]++;
        }
    }
}
```

1. Czytamy kolejne akapity za pomocą `getline()`
2. Dzielimy długie linie na słowa (`sep` to znaki będące separatorami)
3. Dodajemy do mapy `bow`

Jak posortować pary?

- Zakładamy, że pary *(słowo, liczba wystąpień)* z `bow` umieścimy w kolejce priorytetowej. Jak jednak posortować je według liczby wystąpień, czyli pola `second`.
- Opcjonalnym parametrem szablonu `priority_queue` jest komparator (obiekt funkcyjny).
- Klasa definiuje `operator()`. Jest on wołany podczas porównywania elementów.

```
class pair_comparator{
public:
    bool operator()(const pair<string,int>&a,
                    const pair<string,int>&b){
        return a.second<b.second;
    }
};
```

Porównywanie par

```
int main(){
    pair<string,int> ala("Ala",10);
    pair<string,int> adam("Adam",20);
    pair<string,int> jan("Jan",30);

    pair_comparator comp;
    cout<<ala.first<<" < "<<adam.first<<":"<<
       comp(ala,adam)<<endl;
    cout<<jan.first<<" < "<<adam.first<<":"<<
       comp(jan,adam)<<endl;
}
```

Ala < Adam:1
Jan < Adam:0

Zastosowanie

```
int main(){
    map<string,int> bow;
    read(bow);
    priority_queue<pair<string,int>,
                   vector<pair<string,int>>,
                   pair_comparator> k;

    for(const auto&e:bow)k.push(e);

    while(!k.empty()){
        const auto&e=k.top();
        cout<<e.first<<":"<<e.second<<endl;
        k.pop();
    }
}
```

i:3830
się:2740
w:2051
na:1862
nie:1778
z:1702
że:1391
do:1185
a:820
to:757
Staś:628
po:587
Nel:551
ale:530
jak:519
o:478
tak:439
od:409
za:394
mu:383
…

Algorytmy

Iteratory

Bilblioteka STL klasyfikuje iteratory ze względu na możliwość:

- odczytu (input iterator) *InIt*
- zapisu (output iterator) *OutIt*
- kierunek ruchu (w przód *FwdIt*, w przód i tył *BidIt*, random access *RanIt*)

Nową cechą zmodyfikowanej biblioteki iostream jest zdefiniowanie iteratorów umożliwiających odczyt lub zapis do strumieni (traktowanych jak kontenery).

Przykład

```
int main()
{
    list<int> li;
    istream_iterator<int> ist(cin);
    for(;ist!=istream_iterator<int>();ist++){
        li.push_back(*ist);
    }
    // istream_iterator<int>() - koniec strumienia
    for(int i=0;i<10;i++)li.push_back(rand()%10);
    for(int i=0;i<10;i++)li.push_front(rand()%10);
```

Po uruchomieniu programu wpisujemy liczby na konsoli i wprowadzamy znak końca strumienia CTRL-D (Linux) lub CTRL-Z (Windows)

Przykład

```
//...
    ostream_iterator<int> out(cout, " ");
    for(int i=0;i<10;i++){
        *out=i;
        out++;
    }
    cout<<endl;
    cout<<endl;
    copy(li.begin(),li.end(),out);
    cout<<endl;
    li.sort();
    copy(li.begin(),li.end(),out);
    cout<<endl;
}
```

```
-5 -6 -12 -23 -45
0 1 2 3 4 5 6 7 8 9

2 9 6 0 3 8 5 3 1 1 -5 -6 -12 -23 -45 3 3 2 9 0 8 2 6 6 9
-45 -23 -12 -6 -5 0 0 1 1 2 2 2 3 3 3 5 6 6 6 8 8 9 9 9
```

- `ostream_iterator<int> out(cout, " ")` – iterator umożliwiający wpisywanie do strumienia wyjściowego
- `copy()` – algorytm kopiowania

Algorytmy

- W nagłówku <algorithm> zdefiniowano szereg szablonów funkcji implementujących standardowe algorytmy działające na kontenerach różnych typów (w tym strumieniach).
- Parametrami tych funkcji są zazwyczaj iteratory (wejściowe i wyjściowe) oraz dla niektórych algorytmów *obiekty funkcyjne*.

Obiekty funkcyjne

- Obiekty funkcyjne pozwalają na adaptację działania algorytmu przez przekazanie do niego funkcji.
- Zamiast wskaźnika do funkcji (w stylu C) przekazywany jest obiekt klasy, która definiuje operator () .
- Operator () jako jedyny może przyjąć dowolną liczbę argumentów – listę formalnych parametrów funkcji.

Przykład

```
class Incrementer{
public:
    void operator()(int&t){
        t++;
    }
};
```

```
template <class T, class FO>
void callFunctionObject(T&t,FO&foo)
{
    foo(t);
}
```

```
int main()
{
    int x=10;
    Incrementer plus_plus;
    plus_plus(x);
    cout<<x<<endl;

    callFunctionObject(x,plus_plus);
    cout<<x<<endl;
}
```

```
11
12
```

Typy obiektów funkcyjnych

Algorytmy STL wykorzystują w zasadzie cztery podstawowe rodzaje funkcji (obiektów funkcyjnych):

- Funkcja jednoargumentowa (unarna)
- Funkcja dwuargumentowa (binarna)
- Predykat (jednoargumentowa funkcja zwracająca typ bool)
- Predykat dwuargumentowy (binarny)

Algorytm copy

Algorytm pozwala na przekopiowanie zawartości kontenera (wszystkie elementy lub pewien obszar) do innego kontenera w miejsce wskazane przez iterator wyjściowy.

```
#include <algorithm>
int main()
{
    list<int> li;
    for(int i=0;i<10;i++)
    {li.push_back(i);li.push_front(i);}
    list<int> li2;
    li2.resize(li.size());
    copy(li.begin(),li.end(),li2.begin());

    copy(li2.begin(),li2.end(),
         ostream_iterator<int>(cout," "));
    cout<<endl;
}
```

```
9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
```

Algorytm count

Algorytm oblicza liczbę wystąpień elementu w kontenerze.

```
int main()
{
    list<int> li;
    for(int i=0;i<10;i++)
    {li.push_back(i);li.push_front(i);}
    cout<<count(li.begin(),li.end(),0)<<endl;
}
```

Wynik: 2

Algorytm count_if

Algorytm oblicza liczbę elementów w kontenerze spełniających *predykat* przekazany jako obiekt funkcyjny.

```
class LessThen{
    int v;
public:
    LessThen(int _v){v=_v;}
    bool operator()(int k){return k<v;}
};

int main()
{
    list<int> li;
    for(int i=0;i<10;i++)
    {li.push_back(i);li.push_front(i);}
    cout<<
        count_if(li.begin(),li.end(),LessThen(3))
        <<endl;
}
```

Rezultat: 6

Algorytm for_each

Algorytm wykonuje dla każdego obiektu należącego do wskazanej sekwencji w kontenerze jednoargumentową funkcję przekazaną w postaci obiektu funkcyjnego.

```
class Dump
{
public:
    void operator()(int k){cout<<k<<" ";}
};

int main()
{
    list<int> li;
    for(int i=0;i<10;i++)
    {li.push_back(i);li.push_front(i);}
    for_each(li.begin(),li.end(),Dump());
    cout<<endl;
}
```

Rezultat:
9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9

Algorytm includes

Algorytm działa na uporządkowanych kolekcjach elementów. Pozwala na sprawdzenie, czy wszystkie elementy danego kontenera są zawarte w drugim kontenerze. Algorytm może być używany do porównywania zbiorów.

```
int main(){
    set<int> s1;
    for(int i=0;i<20;i++){s1.insert(i);}
    set<int> s2;
    for(int i=0;i<10;i++){s2.insert(i);}

    cout<<"s1 includes s2 "<<
        includes(s1.begin(),s1.end(),s2.begin(),s2.end())
        <<endl;
    cout<<"s2 includes s1 "<<
        includes(s2.begin(),s2.end(),s1.begin(),s1.end())
        <<endl;;
}
```

```
s1 includes s2 1
s2 includes s1 0
```

Operacje na zbiorach

```
int main(){
    set<int> a;
    set<int> b;
    for(int i=0;i<10;i++)a.insert(i);
    for(int i=5;i<15;i++)b.insert(i);
    vector<int> result;
    result.resize(a.size()+b.size());
    vector<int>::iterator it =
            set_union(
                    a.begin(),a.end(),
                    b.begin(),b.end(),result.begin());
    result.resize(it-result.begin());

    for (it=result.begin(); it!=result.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    return 0;
}
```

set_union	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
set_intersection	5 6 7 8 9
set_difference	0 1 2 3 4
set_symmetric_difference	0 1 2 3 4 10 11 12 13 14

Algorytm min_element

Algorytm pozwala na wyznaczenie minimalnego elementu w kontenerze. Przykłady pokazują dwie wersje wywołania algorytmu: opartą na naturalnym porządku elementów i z dostarczonym obiektem funkcyjnym (dwuargumentowym predykatem) do porównywania elementów.

```
int main()
{
    list<int> li;
    for(int i=0;i<10;i++)
    {li.push_back(i);li.push_front(i);}
    auto m = min_element(li.begin(),li.end());

    cout<<*m<<endl;
}
```

Rezultat: 0

Algorytm min_element

```
class ReverseLess
{
public:
    bool operator()(int e1,int e2){return e1>e2;}
};

int main()
{
    list<int> li;
    for(int i=0;i<10;i++)
    {li.push_back(i);li.push_front(i);}
    auto m = min_element(li.begin(),li.end(),ReverseLess());

    cout<<*m<<endl;
}
```

Rezultat: 9

Algorytm replace

Algorytm pozwala na zastąpienie w kontenerze każdego wystąpienia elementu o danej wartości nową wartością.

```
int main()
{
    list<int> li;
    for(int i=0;i<10;i++)
    {li.push_back(i);li.push_front(i);}
    replace(li.begin(),li.end(),5,-5);
    copy(li.begin(),li.end(),
         ostream_iterator<int>(cout," "));
    cout<<endl;
}
```

Rezultat:
9 8 7 6 -5 4 3 2 1 0 0 1 2 3 4 -5 6 7 8 9

Algorytm replace_if

Algorytm zastępuje nową wartością wszystkie elementy spełniające określony predykat przekazany jako obiekt funkcyjny.

```
class LessThen{
    int v;
public:
    LessThen(int _v){v=_v;}
    bool operator()(int k){return k<v;}
};

int main(){
    list<int> li;
    for(int i=0;i<10;i++)
    {li.push_back(i);li.push_front(i);}
    replace_if(li.begin(),li.end(),LessThen(5),0);
    copy(li.begin(),li.end(),
         ostream_iterator<int>(cout," "));
    cout<<endl;
}
```

Rezultat:
9 8 7 6 5 0 0 0 0 0 0 0 0 0 5 6 7 8 9

Wyrażenia lambda

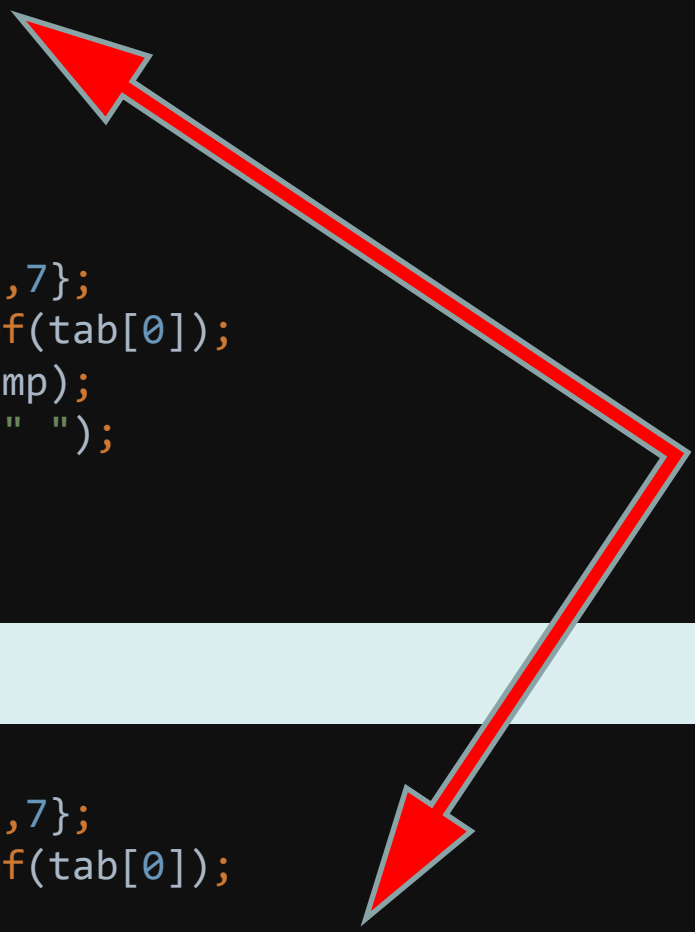
Wyrażenia lambda

- Wyrażenia lambda jest wyrażeniem umieszczanym w kodzie programu, na podstawie którego w locie generowany jest kod funkcji lub obiekt funkcyjny.
- Mówiąc ogólnie, jest to obiekt, który można wywołać (ang. *callable*) będący funkcją lub obiektem funkcyjnym.
- Wyrażenia lambda mogą być używane tam, gdzie jest oczekiwany obiekt funkcyjny lub wskaźnik do funkcji.
- Obiekty funkcyjne generowane z wyrażeń lambda nazywane są **domknięciami** (ang. *closure*), a ich klasy klasami domknięć (ang. *closure class*)
- Wyrażenia lambda mogą być traktowane, jak anonimowe fragmenty kodu
- Nie mogą być wielokrotnie wykorzystywane (nie mają nazwy)

Przykład

```
int cmp(const void*a,const void*b){
    return *(int*)a-*(int*)b;
}

int main(){
    int tab[]={10,2,3,12,90,3,4,12,8,7};
    int tab_size = sizeof(tab)/sizeof(tab[0]);
    qsort(tab,tab_size,sizeof(int),cmp);
    ostream_iterator<int> out(cout, " ");
    copy(tab,tab+tab_size,out);
}
```



```
int main(){
    int tab[]={10,2,3,12,90,3,4,12,8,7};
    int tab_size = sizeof(tab)/sizeof(tab[0]);
    qsort(tab,tab_size,sizeof(int),
                    [](auto a,auto b){return *(int*)a-*(int*)b;});
    ostream_iterator<int> out(cout, " ");
    copy(tab,tab+tab_size,out);
}
```

```
2 3 3 4 7 8 10 12 12 90
```

Przykład

- `qsort` jest biblioteczną funkcją standardowej biblioteki języka C implementującą algorytm quick-sort dla elementów dowolnego typu umieszczonych w tablicy.
- Wymagane jest dostarczenie wskaźnika do funkcji służącej do porównywania elementów.
- W C++ może to być jawnie zadeklarowana funkcja lub wyrażenie lambda:

```
…
int(*cmp)(const void*,const void*)=
      [](auto a,auto b){return *(int*)a-*(int*)b;};
qsort(tab,tab_size,sizeof(int),cmp);
…
```

Zmienna `cmp` to wskaźnik do funkcji zwracającej `int` i przyjmującej dwa argumenty typu `const void*`.

Przykład

Można także posłużyć się algorytmem sort, który występuje w dwóch wersjach

- `void sort (RandomAccessIterator first, RandomAccessIterator last)` – parametrami są iteartory wskazujące zakres do sortowania
- `void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp)` – dodatkowym parametrem jest komparator, obiekt funkcyjny, który zwraca `true`, jeżeli `a` ma być umieszczone przed `b`.

Przykład

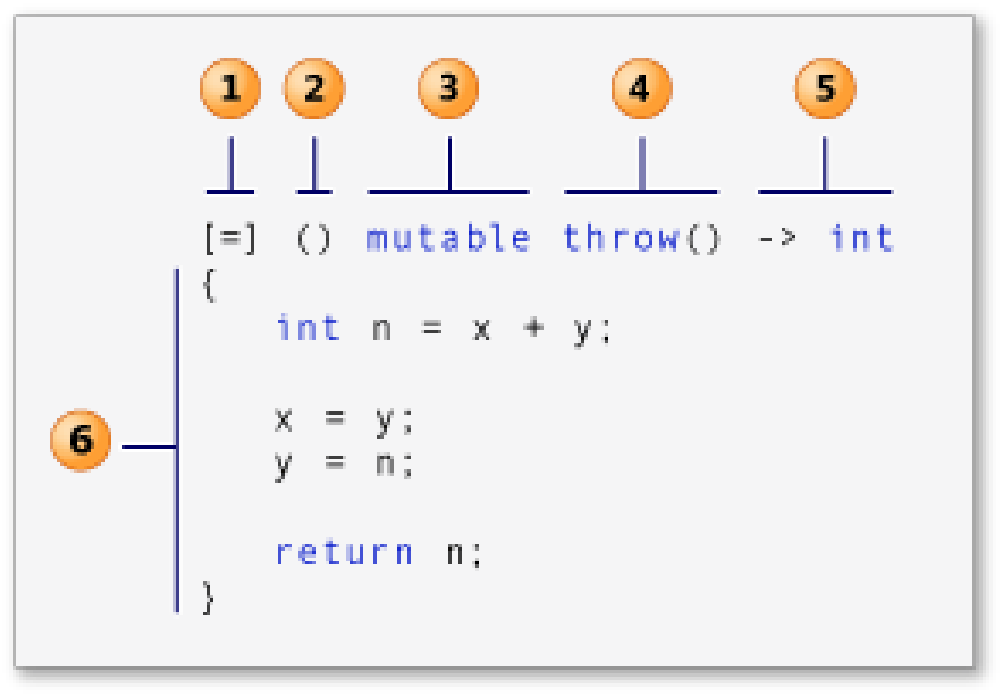
Posortowane rosnąco elementy, ale parzyste przed nieparzystymi

```
int main(){
    int tab[]={10,11,33,21,2,3,12,90,3,4,12,8,7};
    int tab_size = sizeof(tab)/sizeof(tab[0]);
    sort(tab,tab+tab_size,[](int a, int b){
                    return a%2<b%2?true:(a%2>b%2?false:a<b);});
    ostream_iterator<int> out(cout, " ");
    copy(tab,tab+tab_size,out);

}
```

```
2 4 8 10 12 12 90 3 3 7 11 21 33
```

Składnia wyrażeń lambda



https://docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=vs-2019

1. **Zapowiada wyrażenie lambda**. Niepuste, specyfikuje sposób przechwytywania (dostępu do) otaczających zmiennych
2. Opcjonalnie: lista parametrów
3. Opcjonalnie: specyfikacja `mutable` (dodatkowe opcje przechwytywania)
4. Opcjonalnie: specyfikacja wyjątków
5. Opcjonalnie: zwracana wartość
6. **Ciało funkcji**.

Wyrażenia lambda

- Zgodnie ze składnią, najprostsze wyrażenie lambda ma postać:

  `[]{}`

  Składa się wyłącznie z zapowiedzi `[]` oraz pustego ciała `{}`.
- Aby wywołać obiekt powstały kompilacji wyrażenia lambda, należy użyć operatora wywołania funkcji `()`.

```
int main(){
    // Wywołanie klasy domknięcia
    []{cout<<"Hello lambda"<<endl;}();
    // wewnętrzna nazwa klasy
    cout<< typeid([]{}).name()<<endl;
    // Dla każdego wyrażenie odrębna klasa domknięcia
    cout<< (typeid([]{})==typeid([]{}))<<endl;
}
```

```
Hello lambda
Z4mainEUlvE0_
0
```

Porównanie `typeid([]{})==typeid([]{})` zwróciło fałsz, czyli dla identycznych wyrażeń powstają osobne klasy domknięcia

Odpowiednik wyrażenia lambda

Klasą domknięcia dla wyrażenia lambda

```
[]{cout<<"Hello lambda"<<endl;}
```

jest:

```
struct anonymous{
    inline auto operator()() const{
        cout<<"Hello lambda"<<endl;
    }
};
```

W instrukcji `[]{cout<<"Hello lambda"<<endl;}();` wołany jest bezparametrowy operator wywołania funkcji `()`.

Parametry

Parametry wyrażenia lambda możemy podać jawnie lub użyć specyfikacji auto. W takim przypadku kompilator wywnioskuje, jaki jest właściwy typ obiektu

```
int main(){
    int tab[]={10,11,33,21,2,3,12,90,3,4,12,8,7};
    int tab_size = sizeof(tab)/sizeof(tab[0]);
    for_each(tab,tab+tab_size,[](auto a){cout<<a<<" ";});
    cout<<endl;
    for_each(tab,tab+tab_size,[](auto&a){a*=a;});
    for_each(tab,tab+tab_size,[](int a){cout<<a<<" ";});

}
```

```
10 11 33 21 2 3 12 90 3 4 12 8 7
100 121 1089 441 4 9 144 8100 9 16 144 64 49
```

W pierwszym wywołaniu typem a będzie `int`, w drugim referencja `int&`, co pozwoli zmodyfikować zawartość tablicy.

Przechwytywanie zmiennych

- W kwadratowych nawiasach zapowiadających można określić, które zmienne z otoczenia powinny być widoczne wewnątrz wyrażenia lambda i w jaki sposób uzyskać do nich dostęp.
- Przekazywanie zmiennych otoczenia do wyrażenia lambda nazywane jest ich przechwytywaniem (ang. capture)

```
int main(){
    int tab[]={10,11,33,21,2,3,12,90,3,4,12,8,7};
    int tab_size = sizeof(tab)/sizeof(tab[0]);
    double mean=0;
    for_each(tab,tab+tab_size,
             [&mean,tab_size](auto a){mean+=(double)a/tab_size;});
    cout<<"Mean "<<mean<<endl;

}
```

Przechwytywane zmienne to `mean` (przez referencję) i `tab_size` (przez wartość).
Obliczana jest średnia wartości w tablicy i umieszczana w zewnętrznej zmiennej `mean`.

Przechwytywanie zmiennych

Odpowiednikiem wyrażenia

```
[&mean,tab_size](auto a){mean+=(double)a/tab_size;}
```

Jest obiekt klasy domknięcia:

```
struct anonymous{
    double&mean;
    int tab_size;

    anonymous(double&_mean,int _tab_size)
        :mean(_mean),tab_size(_tab_size){}

    inline auto operator()(int a)const{
        mean+=(double)a/tab_size;
    }
};
```

Przechwytywane zmienne stają się polami klasy. Zmienna `mean` przechwytywana przez referencję zamienia się w referencję, zmienna `tab_size` jest kopiowana.

Przechwytywanie zmiennych

Składnia	Opis
`[ ](…){…}`	Żadna zmienna nie jest przechwytywana
`[=](…){…}`	Wszystkie zmienne są przechwytywane jako kopie
`[=](…){…}`	Wszystkie zmienne są przechwytywane jako referencje
`[x](…){…}`	Zmienna x jest przechwytywana jako kopia
`[&x](…){…}`	Zmienna x jest przechwytywana jako referencja
`[&,x](…){…}`	Wybrane zmienne są przechwytywane jako kopie, wszystkie pozostałe przez referencje
`[=,&x](…){…}`	Wybrane zmienne są przechwytywane przez referencje, wszystkie pozostałe jako kopie

Przechwytywanie wszystkich zmiennych nie jest zalecane, ponieważ:

- Powoduje generację ciężkich obiektów z wieloma polami
- Wyrażenia lambda są często kopiowane (copy-paste) i nie jest widoczne, które zmienne są przechwytywane. W nowym otoczeniu zmienne o tych samych nazwach mogą mieć inne typy.

Wiszące referencje…

```
auto create_lambda(int x){
    return [&](auto a){return a<x;};
}

int main(){
    cout <<"Czy 5 < 10? >> "<<
        create_lambda(10)(5)<<endl;
}
```

Czy 5 < 10? >> 0

Wyrażenie lambda przechwyciło zmienną x (parametr funkcji przez referencję). Obiekt domknięcia został zwrócony po wyjściu z funkcji `create_lambda()`, więc referencja wskazuje usunietą zmienną na stosie.

Po zmianie metody przechwytywania na `[=]` wynik jest poprawny…

```
auto create_lambda(int x){
    return [=](auto a){return a<x;};

}
```

Czy 5 < 10? >> 1

mutable

- Operator wywołania funkcji w wygenerowanej klasie domknięcia jest typu const. Oznacza to, że nie może modyfikować atrybutów klasy.
- Te z kolei odpowiadają przechwyconym zmiennym.
- Użycie modyfikatora `mutable` powoduje, że operator nie jest typu `const`.

```
struct anonymous{
    double&mean;
    int tab_size;

    anonymous(double&_mean,int _tab_size)
        :mean(_mean),tab_size(_tab_size){}

    inline auto operator()(int a)const{
        mean+=(double)a/tab_size;
    }
};
```

Mutable - przykład

```
int main(){
    int x=9,y=11;
    auto mul =
    [=]()mutable{
        int z = 0;
        while(y>0){
            if(y%2==1)z=z+x;
            x=2*x;
            y=y/2;
        }
        return z;
    };

    cout<<x<<"*"<<y<<"="<<mul()<<endl;
}
```

9*11=99

Przykład demonstracyjny. Zamiast przechwytywać zmienne, lepiej przekazać ich wartości jako parametry

Wyrażenie lambda przechwytuje (kopiuje) zmienne x i y.
Następnie stosując znany algorytm mnożenia wyznacza ich iloczyn.
Oryginalne wartości zmiennych nie są modyfikowana (lambda działa na kopiach).

Przechwytywanie w metodach

- Wyrażenie lambda wewnątrz metody klasy może przechwycić jej atrybuty oraz wskaźnik `this`.
- Ma także dostęp do atrybutów prywatnych.

```
class A{
    int x=2;
    int y=5;
public:
    void info()const{
        [=](){
            printf("this=%p x=%d, y=%d\n",this,x,y);
        }();
    }
};

int main(){
    A a;
    a.info();

}
```

this=0xffffcc18 x=2, y=5

Zwracane wartości

Na ogół nie ma potrzeby specyfikacji typów zwracanych wartości, ponieważ kompilator jest w stanie je wydedukować.

- W przypadku parametrów `auto` (których raczej nie można stosować dla zwykłych funkcji) typy zwracanych wartości mogą się zmieniać.
- Skompilowane wyrażenia lambda zachowują się wtedy jak szablony funkcji. Są to tzw. generyczne wyrażenia lambda.

```
int main(){
    auto min = [](auto a, auto b, auto c){
        return a<=b && a<=c?a : b<=a && b<=c?b:c; };

    cout<<typeid(min(1,2,3)).name()<<" ";
    cout<<typeid(min(1,2,3.0)).name();
}
```

i d

- W pierwszym przypadku zwracany jest `int`, w drugim `double`.
- Po zmianie wyrażenia lambda na:

```
[](auto a, auto b, auto c)->double{…}
```

zwracanym typem będzie `double`.

Generacja wskaźników do funkcji

- Wskaźniki do funkcji mogą być oczekiwane przez biblioteki napisane w języku C (np. qsort). Kompilator analizując wywołanie jest w stanie wydedukować jakiego typu obiekt wygenerować (funkcję czy domknięcie).
- Można jednak wygenerować wskaźnik do funkcji jawnie, poprzedzając wyrażenie lambda znakiem +.

```
int main(){
    // auto więc wymuszamy generację funkcji za pomoca +
    auto ptr= +[](int a, int b){return a<b?a:b;};
    printf("ptr  = %p\n",ptr);

    // po lewej stronie jets wskaźnik do funkcji
    // więc kompilator wygeneruje funkcję
    int (*ptr2)(int,int) = [](int a, int b){return a<b?a:b;};
    printf("ptr2 = %p\n",ptr2);
}
```

```
ptr  = 0x100404ecf
ptr2 = 0x100404f29
```

Lambdy zwracające lambdy

Wyrażenie lambda, jako funkcja, może zwracać domknięcia lambda. Dzięki temu użycie wielu standardowych algorytmów jest prostsze.

```
int main(){
    int tab[]={0,1,2,3,4,5,6,7,8,9,10};
    int tab_size = sizeof(tab)/sizeof(tab[0]);

    auto less_than = [](auto x) {
        return [x](auto y) {
            return y < x;
        };
    };
    cout<<count_if(tab,tab+tab_size,less_than(7))<<endl;
    cout<<count_if(tab,tab+tab_size,less_than(3))<<endl;

}
```

```
7
3
```

`less_then` jest zmienną (domknięciem lambda). Wywołanie `less_then(7)` zwraca predykat (funkcję logiczną) skonfigurowany tak, aby sprawdzać, czy x<7. Predykat jest użyty w algorytmie `count_if` zwracającym liczbę elementów kontenera, dla których predykat jest prawdziwy.