STL: algorytmy numeryczne

1. accumulate
2. partial_sum
3. inner_product
4. adjacent_difference

114

STL: algorytmy numeryczne

```
template<class InputIterator, class Type>
Type accumulate( InputIterator _First, InputIterator _Last,
                                                Type _Val );
```

- sumuje wartości zawarte w sekwencji [*_First*,*_Last*) do wartości początkowej sumy zawartej w *_Val*. Zwraca wynik sumowania.
- chociaż domyślną operacją jest sumowanie, istnieje wersja z dodatkowym, ostatnim parametrem szablonu reprezentującym funkcję lub obiekt funkcyjny służący do wykonywania dowolnych innych operacji.

115

STL: algorytmy numeryczne

```
#include <iostream>     // std::cout
#include <functional>   // std::minus
#include <numeric>      // std::accumulate
int main () {
  int init = 100;
  int numbers[] = {10,20,30};

  std::cout << "Domyślna suma: ";
  std::cout << std::accumulate(numbers,numbers+3,init);
  std::cout << '\n';              // Domyślna suma: 160

  std::cout << "Minus z biblioteki <functional>: ";
  std::cout << std::accumulate (numbers, numbers+3, init, std::minus<int>());
  std::cout << '\n';              // Minus z biblioteki <functional>: 40

  return 0;
}
```


116

STL: algorytmy numeryczne

```
#include <iostream>     // std::cout
#include <functional>   // std::minus
#include <numeric>      // std::accumulate
int MojaFun (int x, int y) {return x+2*y;} // y - element z zakresu
struct MojaKlasa {
   int operator()(int x, int y) {return x+3*y;} // y - element z zakresu
} MObj;
int main () {
  int init = 100;
  int numbers[] = {10,20,30};
  std::cout << "MojaFun: ";
  std::cout << std::accumulate (numbers, numbers+3, init, MojaFun);
  std::cout << '\n';              // MojaFun: 220
  std::cout << "MObj : ";
  std::cout << std::accumulate (numbers, numbers+3, init, MObj);
  std::cout << '\n';              // Mobj: 280
  return 0;
}
```


117

STL: algorytmy numeryczne

```
template<class InputIterator, class OutIt>
OutputIterator partial_sum( InputIterator _First,
        InputIterator _Last, OutputIterator _Result );
```

- oblicza uogólnioną sumę częściową zbioru elementów z sekwencji [*_First*,*_Last*). Polega to na utworzeniu sekwencji wynikowej, której pierwszy element jest wskazywany przez *_Result*, zawierającej wartości będące sumą elementów poprzedzających każdą z tych wartości w sekwencji wejściowej.
- np. dla sekwencji wejściowej **1,2,3,4,5** sumy w sekwencji wynikowej będą miały wartości: **1, 1+2, 1+2+3, 1+2+3+4, 1+2+3+4+5**
- chociaż domyślną operacją jest sumowanie istnieje wersja z dodatkowym, ostatnim parametrem szablonu reprezentującym funkcję lub obiekt funkcyjny służący do wykonywania dowolnych innych operacji.

118

STL: algorytmy numeryczne

```
#include <iostream>     // std::cout
#include <functional>   // std::multiplies
#include <numeric>      // std::partial_sum
int myop (int x, int y) {return x+y+1;}     // y - element z zakresu
int main () {
  int val[] = {1,2,3,4,5};
  int result[5];
  std::partial_sum (val, val+5, result, std::multiplies<int>());
  std::cout << "using functional operation multiplies: ";
  for (int i=0; i<5; i++) std::cout << result[i] << ' ';
  std::cout << '\n';
  std::partial_sum (val, val+5, result, myop);
  std::cout << "using custom function: ";
  for (int i=0; i<5; i++) std::cout << result[i] << ' ';
  std::cout << '\n';
  return 0;
}
```


119

STL: algorytmy numeryczne

```
template<class InputIterator1, class InputIterator2, class
   Type>
Type inner_product(
   InputIterator1 _First1, InputIterator1 _Last1,
   InputIterator2 _First2, Type _Val );
```

- oblicza iloczyn stanowiący odpowiednik iloczynu skalarnego dwóch wektorów reprezentowanych sekwencjami [*_First1*,*_Last1*) i *_First2*. Wartością takiej operacji jest suma iloczynów odpowiadających sobie elementów sekwencji powiększona o wartość bazową *_Val*:
  *_Val + (*_First1) * (*_First2) + ..*
- Dwie domyślne operacje *dodawania* oraz *mnożenia* można zastąpić dwoma funkcjami lub obiektami funkcyjnymi, odpowiednio **op1** (*domyślnie - dodawanie*) i **op2** (*domyślnie - mnożenie*) :

```
_Val = _Val + (_First1)*(*_First2);
_Val = op1 (_Val, op2(*_First1,*_First2));
```


120

STL: algorytmy numeryczne

```
#include <iostream>     // std::cout
#include <functional>   // std::minus, std::divides
#include <numeric>      // std::inner_product
int myaccumulator (int x, int y) {return x-y;}
int myproduct (int x, int y) {return x+y;}
double diffPow2(double x, double y) { return pow((x - y), 2.0); };
int main () {
  int init = 100;
  int series1[] = {10,20,30};
  int series2[] = {1,2,3};
  std::cout << std::inner_product(series1,series1+3,series2,init,
                                  std::minus<int>(),std::divides<int>());
  std::cout << '\n';
  std::cout << std::inner_product(series1,series1+3,series2,init,
                                  myaccumulator,myproduct);
  std::cout << '\n'; // odległość Euklidesowa miedzy series1 i series2:
  std::cout << std:: inner_product(series1,series1+3,series2,0,0,
                                   std::plus<double>(), diffPow2);
```


121

STL: algorytmy numeryczne

```
template<class InputIterator, class OutIterator>
OutputIterator adjacent_difference(
      InputIterator _First, InputIterator _Last,
      OutputIterator _Result );
```

- oblicza różnicę sąsiadujących elementów sekwencji źródłowej. Np. dla sekwencji **1,2,3,4,5** sekwencja wynikowa reprezentuje wartości: **1, 2-1, 3-2, 4-3, 5-4**,
- domyślną operację odejmowania można zastąpić funkcją lub obiektem funkcyjnym.

122

STL: algorytmy numeryczne

```
#include <iostream>     // std::cout
#include <functional>   // std::multiplies
#include <numeric>      // std::adjacent_difference
int myop (int x, int y) {return x+y;}
int main () {
  int val[] = {1,2,3,5,9,11,12};
  int result[7];
  std::adjacent_difference (val, val+7, result, std::multiplies<int>());
  std::cout << "using functional operation multiplies: ";
  for (int i=0; i<7; i++) std::cout << result[i] << ' ';
  std::cout << '\n';
  std::adjacent_difference (val, val+7, result, myop);
  std::cout << "using custom function: ";
  for (int i=0; i<7; i++) std::cout << result[i] << ' ';
  std::cout << '\n';
  return 0;
}
```


123

STL: algorytmy - podsumowanie

Ogólna konwencja dotycząca parametrów algorytmów:

- **alg(beg, end, *other args*);**
- **alg(beg, end, dest, *other args*);**
- **alg(beg, end, beg2, *other args*);**
- **alg(beg, end, beg2, end2, *other args*);**

beg, end – zakres pierwszego zbioru przechowującego dane wejściowe, na których działa algorytm **alg**

dest – iterator wskazujący miejsce, gdzie ma być zapisany wynik działania algorytmu **alg**. Algorytm zakłada, że miejsce to jest bezpieczne, tj. np. dostatecznie duże (nie wykonuje kroku weryfikacji).

beg2, end2 – zakres drugiego zbioru przechowującego dane wejściowe; algorytmy, korzystające tylko z **beg2** zakładają, że liczba elementów drugiej sekwencji jest co najmniej tak duża jak zakres **beg, end**.

124

STL: algorytmy - podsumowanie

Ogólna konwencja dotycząca parametrów algorytmów:

Algorytmy używające predykatów występują w dwóch wersjach, np.:

```
1. unique(beg, end);        // używa == do porównywania
2. unique(beg, end, comp); // używa comp do porównywania
```

Pierwsza – używa przeciążonego operatora logicznego < lub == zdefiniowanego dla typu danych przechowywanych w kontenerze.

Druga – używa obiektu funkcyjnego, tj. predykatu **comp**

125

STL: algorytmy - podsumowanie

Ogólna konwencja dotycząca parametrów algorytmów:

Algorytmy mające w nazwie **if** występują w dwóch wersjach, np.:

```
1. find(beg, end, val); // znajduje pierwsze wystąpienie val
2. find_if(beg, end, pred);  // znajduje pierwszą wartość,
                             // gdzie pred==true
```

Algorytmy mające w nazwie **_copy** występują w dwóch wersjach, np.:

```
1. reverse(beg, end);  // odwraca porządek elementów
2. reverse_copy(beg, end, dest); // kopiuje w odwrotnym
                                 // porządku do dest
```


STL: algorytmy - podsumowanie

Algorytmy kontenera **`list`**:

W przeciwieństwie do innych kontenerów, kontener **`list`** definiuje kilka algorytmów jako swoje własne metody. Np. dla obiektu **`lst`** można wywołać:

```
lst.merge(lst2);          // używa operatora <
lst.merge(lst2, comp);    // używa comp
lst.remove(val);          // używa operatora ==
lst.remove_if(pred);      // używa pred
lst.reverse();
lst.sort();               // używa operatora <
lst.sort(comp);           // używa comp
lst.unique();             // używa operatora ==
lst.unique(pred);         // używa pred
```


STL: algorytmy - podsumowanie

Algorytmy kontenera **`list`**:

Algorytm **`splice`** – może wystąpić z różnymi listami argumentów, np. dla **`lst.splice`**:

1. **`(p, lst2)`**,
   gdzie **`p`** – iterator na element w **`lst`**. Przenosi elementy z kontenera **`lst2`** do **`lst`** tuż przed **`p`** usuwając je z **`lst2`** (**`lst`** i **`lst2`** nie mogą być tym samym kontenerem i muszą być tego samego typu),
2. **`(p, lst2, p2)`**,
   gdzie **`p`** – iterator na element w **`lst`**, a **`p2`** – iterator na element w **`lst2`**. Przenosi element **`p2`** do **`lst`** w miejsce tuż przed **`p`**. (**`lst`** i **`lst2`** mogą być tym samym kontenerem),
3. **`(p, lst2, b, e)`**,
   gdzie **`p`** – iterator na element w **`lst`**, a **`b`** i **`e`** określają przedział w kontenerze **`lst2`**. Przenosi elementy z przedziału do **`lst`** w miejsce tuż przed **`p`**. (**`lst`** i **`lst2`** mogą być tym samym kontenerem, ale **`p`** nie może należeć do przedziału **`b,e`**).

STL: algorytmy - podsumowanie

Algorytmy kontenera **`list`**:

Algorytmy zadeklarowane jako metody ***są szybsze*** od algorytmów generycznych:

- algorytmy generyczne zamieniają przechowywane wartości (dodają w jednych i usuwają w innych odpowiednich miejscach w liście),
- algorytmy zadeklarowane jako metody działają na wskaźnikach reprezentujących linki między elementami kontenera (mają dostęp do składowych chronionych i prywatnych i wykorzystują to efektywnie).

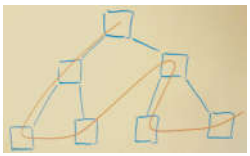
C++: STL

Implementowanie własnych algorytmów

STL: implementowanie algorytmów

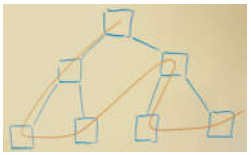
Algorytm przeszukiwania w głąb

Przeszukiwanie grafu – odwiedzenie wszystkich jego wierzchołków w kolejności jak na rysunku obok:



STL: implementowanie algorytmów

Algorytm przeszukiwania w głąb



Reprezentacja grafu w programie

1. Wierzchołki są identyfikowane przez liczby całkowite.
2. Dla każdego z wierzchołków przechowujemy listy numerów wierzchołków, z którymi jest połączony krawędzią:
   `vector< vector<int> >`

1	2	3
2	4	5
3	6	7
4		
5		
6		
7		

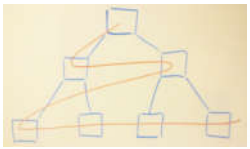
STL: implementowanie algorytmów

Algorytm przeszukiwania w głąb (*rekurencyjny*)

```
typedef vector<int> vi;
typedef vector<vi> vvi;
int N; // liczba wierzchołków
vvi W; // graf
vi V(N, false);  // flagi określające, czy wierzchołek został odwiedzony
void dfs(int i) {
       if(!V[i]) {
            V[i] = true; // zaznaczamy węzeł jako odwiedzony
            for_each(W[i].begin(), W[i].end(), dfs); // rekurencyjne wywołanie
       }
}
bool check_graph_connected_dfs() {
       int start_vertex = 0;
       V = vi(N, false);
       dfs(start_vertex);
       return (find(V.begin(), V.end(), 0) == V.end());
}
```


STL: implementowanie algorytmów

Algorytm przeszukiwania wszerz



Przeszukiwanie grafu – odwiedzenie wszystkich jego wierzchołków w kolejności jak na rysunku obok:

Reprezentacja grafu w programie

Wierzchołki są identyfikowane przez liczby całkowite.

Dla każdego z wierzchołków przechowujemy listy numerów wierzchołków, z którymi jest połączony krawędzią: `vector< vector<int> >`

STL: implementowanie algorytmów

Algorytm przeszukiwania wszerz

Ogólna zasada działania algorytmu:

Do przeszukiwania wszerz stosowana jest kolejka FIFO o nazwie Q.

1. Najpierw do Q trafia pierwszy wierzchołek z listy W
2. Następnie jest on wyjmowany z kolejki, po czym trafiają do niej wszystkie węzły sąsiadujące z węzłem początkowym.
3. Następnie ponownie pobieramy kolejny węzeł z kolejki Q i wkładamy do kolejki wszystkie jego wierzchołki sąsiednie (jeszcze nieodwiedzone).

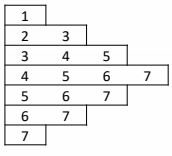
Proces jest kontynuowany póki kolejka nie jest pusta.

STL: implementowanie algorytmów

Algorytm przeszukiwania wszerz

Ogólna zasada działania algorytmu:

Kolejka FIFO:



STL: implementowanie algorytmów

Algorytm przeszukiwania wszerz

Deklaracje zmiennych:

```
typedef vector<int> vi;
typedef vector<vi> vvi;
int N; // liczba wierzchołków
vvi W; // graf
```

STL: implementowanie algorytmów

Algorytm przeszukiwania wszerz

```
bool check_graph_connected_bfs() {
      int start_vertex = 0;
      vi V(N, false); // flagi określające, czy wierzchołek został odwiedzony
      queue<int> Q;   // kolejka FIFO
      Q.push(start_vertex); // pierwszy węzeł grafu trafia do kolejki
      V[start_vertex] = true;
      while(!Q.empty()) {
            int i = Q.front();
            Q.pop(); // zdejmij element z kolejki
            for(vi::iterator it = W[i].begin(); it != W[i].end(); it++) {
                  if(!V[*it]) {
                        V[*it] = true;
                        Q.push(*it); // dodaj na koniec kolejki
                  }
            }
      }
      return (find(V.begin(), V.end(), 0) == V.end());
}
```


STL: implementowanie algorytmów

Statystyka

Moment centralny *r*-tego stopnia (*r*-tego rzędu)

$$m_r = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^r$$

gdzie:

- r=1,2,.. - rząd, stopień momentu,
- x_i – poszczególne obserwacje,
- $\bar{x}$ – średnia,
- n – liczba obserwacji.

Moment centralny **drugiego** rzędu - **wariancja**

STL: implementowanie algorytmów

Statystyka

```
template<int N, class T>
T nthPower(T x) {
  T ret = x;
  for (int i=1; i < N; ++i)
        ret *= x;
  return ret;
};
```

```
template<class T, int N>
struct SumDiffNthPower {
  T mean_;

  SumDiffNthPower(T x) : mean_(x) { };

  T operator()(T sum, T current) {
    return sum +
        nthPower<N>(current - mean_);
  }
};
```


STL: implementowanie algorytmów

Statystyka

```
template<class T, int N, class Iter_T>
T nthMoment(Iter_T first, Iter_T last, T mean)  {
  size_t cnt = distance(first, last);
  return accumulate(first, last,T(),SumDiffNthPower<T,N>(mean))/cnt;
}
template<class T, class Iter_T>
T computeVariance(Iter_T first, Iter_T last, T mean) {
  return nthMoment<T, 2, Iter_T>(first, last, mean);
}
void main() {  // tutaj utworzenie kontenera z danymi
  size_t cnt = distance(first, last);
  double sum = accumulate(first, last, 0.0);
  double mean = sum / cnt;
  double var = computeVariance(first, last, mean);
  …
```


C++: STL

Obiekty funkcyjne: funktory, predykaty, adaptory

STL: obiekty funkcyjne

Wprowadzenie

Przy okazji prezentacji algorytmów pojawiły się na różnych slajdach wywołania gotowych funktorów* z biblioteki **<functional>**:

```
std::transform (V.begin(), V.end(), W.begin(), V.begin(), std::plus<int>());
std::cout << std::accumulate (numbers, numbers+3, init, std::minus<int>());
std::cout << std::inner_product(series1,series1+3,series2,init,
                                std::minus<int>(),std::divides<int>());
std::adjacent_difference (val, val+7, result, std::multiplies<int>());
```

*) Funktory – obiekty utworzone na podstawie konkretyzacji szablonów klas wyposażonych w operator wywołania (**operator()**). Obiekt takiego typu staje się obiektem *wywoływalnym* (*callable object, function object, functor*).

STL: obiekty funkcyjne

Przykład – szablon klasy **`plus`**:

```
// STRUCT TEMPLATE plus
template<class _Ty = void>
  struct plus
  {// functor for operator+
        _CXX17_DEPRECATE_ADAPTOR_TYPEDEFS typedef _Ty first_argument_type;
        _CXX17_DEPRECATE_ADAPTOR_TYPEDEFS typedef _Ty second_argument_type;
        _CXX17_DEPRECATE_ADAPTOR_TYPEDEFS typedef _Ty result_type;

        constexpr _Ty operator()(const _Ty& _Left, const _Ty& _Right) const
        {// apply operator+ to operands
                return (_Left + _Right);
        }
};
```

Wersja z biblioteki VS2017

STL: obiekty funkcyjne

Funktory arytmetyczne (dwu- i jednoargumentowe)

`plus<Typ>`	wynik = arg1 + arg2
`minus<Typ>`	wynik = arg1 - arg2
`multiplies<Typ>`	wynik = arg1 * arg2
`divides<Typ>`	wynik = arg1 / arg2
`modulus<Typ>`	wynik = arg1 % arg2
`negate<Typ>`	wynik = - arg

- argumenty i wynik są tego samego (podanego w deklaracji) typu
- działają pod warunkiem, że dla typu, dla którego funktor jest konkretyzowany, zdefiniowany jest odpowiedni operator arytmetyczny

STL: obiekty funkcyjne

Predykaty

Przy okazji algorytmu **`sort`** zaprezentowany został też funktor **`greater`**:

```
c1.sort( greater<int>( ) );
```

Podstawowe cechy:

- argumenty (jeden lub dwa) tego samego (podanego w deklaracji) typu, wynik typu **`bool`**,
- działa pod warunkiem, że dla typu, dla którego funktor jest konkretyzowany, zdefiniowany jest odpowiedni operator relacyjny lub logiczny,
- Funktory o takich cechach nazywane są *predykatami.*

STL: obiekty funkcyjne

Predykaty – porównania:

`equal_to<Typ>`	wynik = arg1 == arg2
`not_equal_to<Typ>`	wynik = arg1 != arg2
`less<Typ>`	wynik = arg1 < arg2
`greater<Typ>`	wynik = arg1 > arg2
`less_equal<Typ>`	wynik = arg1 <= arg2
`greater_equal<Typ>`	wynik = arg1 >= arg2

STL: obiekty funkcyjne

Predykaty – operacje logiczne:

`logical_or<Typ>`	wynik = arg1 || arg2
`logical_and<Typ>`	wynik = arg1 && arg2
`logical_not<Typ>`	wynik = ! arg

STL: obiekty funkcyjne

Adaptory funktorów

Zdarza się, że potrzebujemy funktora, którego nie ma w bibliotece **`functional`**, ale jest podobny, który w znacznej części implementuje potrzebną funkcję. Do wykonania brakującej modyfikacji w działaniu funktora można wykorzystać wtedy jedną z gotowych *helper functions* .

Typy adaptorów:

1. wiążące – zmieniają funktor dwuargumentowy w jednoargumentowy,
2. negujące – stosowane do predykatów, tworzą predykat obliczający negację pierwotnego,
3. adaptory zwykłych funkcji i metod.

STL: obiekty funkcyjne

Adaptory wiążące (*binders*)

zmieniają funktor dwuargumentowy w jednoargumentowy

`bind1st(fun,val)` - wiąże pierwszy argument z wartością **`val`**
`bind2nd(fun,val)` - wiąże drugi argument z wartością **`val`**

(*Deprecated in C++11 and removed in C++17*)

Przykłady:

#1: skalowanie współrzędnych punktu **P** i zapisanie ich w **R**: **R = P*s**:

```
transform(P.begin(), P.end(), R.begin(),
                    bind2nd(multiplies<double>(), s));
```

#2: aby znaleźć w kolekcji elementy o wartości **v<7** :

```
bind2nd(less<int>(),7)
```


STL: obiekty funkcyjne

Adaptory negujące

stosowane do predykatów, tworzą predykat obliczający negację pierwotnego

`not1(fun)` - neguje wynik predykatu jednoargumentowego
`not2(fun)` - neguje wynik predykatu dwuargumentowego

Przykład:

Policzenie liczb parzystych w kolekcji **v**:

```
liczba = count_if(v.begin(),v.end(),
                  not1(bind2nd(modulus<int>(),2)));
```

Uwaga: **`modulus`** nie jest predykatem, ponieważ zwraca wynik operatora modulo, ale liczbę typu **`int`** można traktować jak wartość logiczną, więc nadal działa

STL: obiekty funkcyjne

Adaptory zwykłych funkcji i metod

tworzą funktor na podstawie:

`ptr_fun(fun)` - zwykłej funkcji lub metody statycznej
`mem_fun_ref(fun)` - metody (niestatycznej) obiektu
`mem_fun(fun)` - metody obiektu dostępnego przez wskaźnik

(*Deprecated in C++11 and removed in C++17*)

- istnieją jedynie adaptory metod i funkcji jednoargumentowych i dwuargumentowych ponieważ tylko takie są używane w bibliotece STL,
- jawne zamknięcie metod w funktorach jest niezbędne ponieważ:
  - wywołanie metody jest składniowo inne niż zwykłej funkcji (czyli inne niż funktora),
  - aby możliwe było zastosowanie adaptorów wiążących i negujących.

STL: obiekty funkcyjne

Przykład:

Mamy funkcję, która dla elementu zwraca informacje, czy należy on do pewnej klasy, czy nie (wartość **`true`** lub **`false`**). Należy policzyć elementy, które nie należą do tej klasy.

```
bool parzysta(int a) { return !(a&1); }

void main()
{
  int tab[] = { 2, 5, 4, 9, 12, 3, 8 };
  vector<int> v(tab,tab+sizeof(tab)/sizeof(int));
  vector<int>::iterator it;
  //int liczba = count_if(v.begin(),v.end(),not1(parzysta)); // źle!
  int liczba = count_if(v.begin(),v.end(),not1(ptr_fun(parzysta)));
};
```


STL: obiekty funkcyjne

Przykład:

Policzyć długości słów przechowywanych w wektorze **`vector<string>`**.
Dla typu **`string`** można wywołać metodę **`length`**

```
vector<string> numbers;         // wektor słów

numbers.push_back("raz");
numbers.push_back("dwa");
numbers.push_back("trzy");
numbers.push_back("cztery");

vector <int> lengths (numbers.size());   // wektor długości słów

transform (numbers.begin(), numbers.end(), lengths.begin(),
                                    mem_fun_ref(&string::length));
```


STL: obiekty funkcyjne

Przykład:

Policzyć długości słów przechowywanych w wektorze **`vector<string*>`**.
Dla typu **`string`** można wywołać metodę **`length`**

```
vector<string*> numbers;        // wektor słów

numbers.push_back("raz");
numbers.push_back("dwa");
numbers.push_back("trzy");
numbers.push_back("cztery");

vector <int> lengths (numbers.size());   // wektor długości słów

transform (numbers.begin(), numbers.end(), lengths.begin(),
                                    mem_fun(&string::length));
```

STL: obiekty funkcyjne

Adaptory – podsumowanie:

```
1. bind1st(fun,val)(arg)          fun(val,arg)
2. bind2nd(fun,val)(arg)          fun(arg,val)

3. not1(fun)(arg)                 !fun(arg)
4. not2(fun)(arg1,arg2)           !fun(arg1,arg2)

5. ptr_fun(fun)(arg)              fun(arg)
6. ptr_fun(fun)(arg1,arg2)        fun(arg1,arg2)
7. mem_fun(fun)(arg)              arg->fun()
8. mem_fun(fun)(arg1,arg2)        arg1->fun(arg2)
9. mem_fun_ref(fun)(arg)          arg.fun()
10.mem_fun_ref(fun)(arg1,arg2)    arg1.fun(arg2)
```


STL: obiekty funkcyjne

Adaptory – podsumowanie

- funktory standardowe są szablonami klas, dlatego przy ich użyciu musimy podać parametry szablonu (dla adaptorów jako szablonów funkcji nie musimy)
- adaptory funktorów są szablonami funkcji, których argumentami są inne funkcje lub funktory
- adaptory funktorów rzadko wywołujemy jawnie, zwykle przekazujemy jako parametr do algorytmów, dlatego wywołania z dwoma parami nawiasów praktycznie się nie zdarzają (poza samym kodem biblioteki STL)

STL: obiekty funkcyjne

Tworzenie własnych funktorów

definiowane jako publiczne klasy (struktury) pochodne od struktur

1. **`unary_function`** – jednoargumentowe
2. **`binary_function`** – dwuargumentowe

```
#include <functional>
```

```
template <class Arg, class Result>
struct unary_function {
  typedef Arg argument_type;
  typedef Result result_type;
};

template <class Arg1, class Arg2,
                          class Result>
struct binary_function {
  typedef Arg1 first_argument_type;
  typedef Arg2 second_argument_type;
  typedef Result result_type;
};
```


STL: obiekty funkcyjne

Tworzenie własnych funktorów – przykład 1

jednoargumentowy, dziedziczy po **`unary_function`**

```
template<class TYPE>
struct modulo2 : public unary_function<TYPE, void> {
    void operator() (TYPE& x) {
        x = x%2;
    }
};
```


STL: obiekty funkcyjne

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
template<class TYPE> struct modulo2 : public unary_function<TYPE, void> {
    void operator() (TYPE& x) { x = x%2; }
};
int main(int argc, char *argv[])
{
  int myints[] = {1, 2, 3, 4, 5, 6, 7, 8};
  vector<int> v(myints,myints+8);
  cout << "before: ";
  copy( v.begin(), v.end(), ostream_iterator<int>( cout, " " ) );
  cout << endl;
  for_each( v.begin(), v.end(), modulo2<int>() );
  cout << "after : ";
  copy( v.begin(), v.end(), ostream_iterator<int>( cout, " " ) );
  cout << endl;
  return 0;
}
```


STL: obiekty funkcyjne

Tworzenie własnych funktorów – przykład 2

- jednoargumentowy, dziedziczy po **`unary_function`**,
- przyjmuje w argumentach szablonu informacje potrzebne do poprawnego działania,
- przyjmijmy, że jest zapisany w pliku **`Przedzial.h`**

```
template <class T,T dol, T gora>
struct Przedzial: unary_function<T,bool> {
  bool operator()(T x) const {
    return x>=dol && x<=gora;
  }
};
```

STL: obiekty funkcyjne

Tworzenie własnych funktorów – przykład 2

```
#include <iostream>
#include <vector>
#include <algorithm>
#include "przedzial.h"
using namespace std;
void main()
{
int tab[] = { 2, 5, 4, 9, 12, 3, 8 };
vector<int> v(tab,tab+sizeof(tab)/sizeof(int));
int c;
c=count_if(v.begin(),v.end(),Przedzial<int,5,9>());
cout << c << endl ;
}
```


STL: obiekty funkcyjne

Tworzenie własnych funktorów – przykład 3

- jednoargumentowy, dziedziczy po **`unary_function`**,
- Przyjmuje w konstruktorze i przechowuje w polach informacje potrzebne do poprawnego działania,
- przyjmijmy, że jest zapisany w pliku **`Przedzial.h`**

```
template <class T>
class Przedzial: public unary_function<T,bool>
{
    T dol, gora;
  public:
    Przedzial(T d, T g) : dol(d), gora(g) {}
    bool operator()(T x) const { return x>=dol && x<=gora; }
};
```


STL: obiekty funkcyjne

Tworzenie własnych funktorów – przykład 3

```
#include <iostream>
#include <vector>
#include <algorithm>
#include "przedzial.h"
using namespace std;
void main()
{
int tab[] = { 2, 5, 4, 9, 12, 3, 8 };
vector<int> v(tab,tab+sizeof(tab)/sizeof(int));
int c;
c=count_if(v.begin(),v.end(),Przedzial<int>(5,9));
cout << c << endl ;
}
```


STL: obiekty funkcyjne

Tworzenie własnych funktorów

- dziedziczenie funktorów po **`unary_function`** i **`binary_function`** nie dodaje im żadnej nowej funkcjonalności,
- jedynym zadaniem dziedziczenia jest nadanie uniwersalnych etykiet (za pomocą **`typedef`**) typom danych, jakie są przekazywane do funktora; informacja ta jest wykorzystywana w kodzie biblioteki **`functional`**.

STL: obiekty funkcyjne

Tworzenie własnych funktorów

Przyjmijmy, że mamy funkcję i nie chcemy z niej korzystać poprzez **`ptr_fun`**, ale zrobić z niej funktor. Zasada tworzenia funktora z funkcji jest identyczna dla **`unary_function`** i **`binary_function`** :

Krok 1 (początek: Zamieniamy dwuargumentową funkcję na funktor):
Przyjmijmy, że mamy funkcję dwuargumentową, która przyjmuje dwie wartości typu **`string`** i **`int`** i zwraca **`bool`**.

```
class MyOtherFunction {
public:
  bool operator() (const string& str, int val) const {
  /* Do something, return a bool. */
  }
};
```


STL: obiekty funkcyjne

Tworzenie własnych funktorów

Ponieważ funkcja jest dwuargumentowa, sięgamy do szablonu **`binary_function`**, którego nagłówek wygląda następująco:

```
template <typename Param1Type, typename Param2Type, typename ResultType>
class binary_function;
```

Krok 2 (ostatni):
dodajemy dziedziczenie po **`binary_function`**

```
class MyOtherFunction: public binary_function<string, int, bool> {
public:
  bool operator() (const string& str, int val) const {
    /* Do something, return a bool. */
  }
};
```