STL: algorytmy numeryczne

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

© UKSW, WMP. SNS, Warszawa

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.

© UKSW, WMP. SNS, Warszawa

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;
}
```

© UKSW, WMP. SNS, Warszawa

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;
}
```

© UKSW, WMP. SNS, Warszawa

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.

© UKSW, WMP. SNS, Warszawa

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;
}
```

© UKSW, WMP. SNS, Warszawa

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));
```

© UKSW, WMP. SNS, Warszawa

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);
```

© UKSW, WMP. SNS, Warszawa

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.

© UKSW, WMP. SNS, Warszawa

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;
}
```

© UKSW, WMP. SNS, Warszawa

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**.

© UKSW, WMP. SNS, Warszawa

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**

© UKSW, WMP. SNS, Warszawa

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
```

© UKSW, WMP. SNS, Warszawa

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
```

© UKSW, WMP. SNS, Warszawa

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`).

© UKSW, WMP. SNS, Warszawa

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).

© UKSW, WMP. SNS, Warszawa

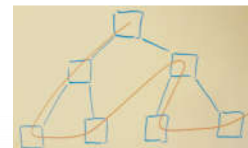
C++: STL

Implementowanie własnych algorytmów

STL: implementowanie algorytmów

Algorytm przeszukiwania w głąb

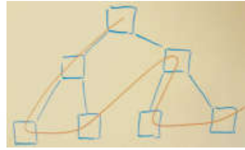
Przeszukiwanie grafu – odwiedzenie wszystkich jego wierzchołków w kolejności jak na rysunku obok:



© UKSW, WMP. SNS, Warszawa

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		

© UKSW, WMP. SNS, Warszawa

STL: implementowanie algorytmów

Algorytm przeszukiwania w głąb (*rekurencyjny*)

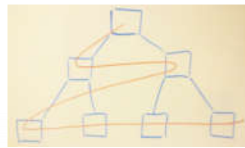
```
typedef vector<int> vi;
typedef vector<vi> vvi;
int N; // liczba wierzchołków
vvi W; // graf
vi V(N, false);  // flagi określające, czy wierzchołek został odwiedzony
void dfs(int i) {
       if(!V[i]) {
            V[i] = true; // zaznaczamy węzeł jako odwiedzony
            for_each(W[i].begin(),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

© UKSW, WMP. SNS, Warszawa

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> >`

© UKSW, WMP. SNS, Warszawa

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.

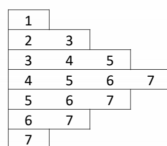
© UKSW, WMP. SNS, Warszawa

STL: implementowanie algorytmów

Algorytm przeszukiwania wszerz

Ogólna zasada działania algorytmu:

Kolejka FIFO:



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

© UKSW, WMP. SNS, Warszawa

STL: implementowanie algorytmów

Algorytm przeszukiwania wszerz

Deklaracje zmiennych:

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

© UKSW, WMP. SNS, Warszawa

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());
}
```

© UKSW, WMP. SNS, Warszawa 138

138

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**

© UKSW, WMP. SNS, Warszawa 139

139

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_);
  }
};
```

© UKSW, WMP. SNS, Warszawa 140

140

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);
  …
```

© UKSW, WMP. SNS, Warszawa 141

141

C++: STL

Obiekty funkcyjne: funktory, predykaty, adaptory

142

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*).

© UKSW, WMP. SNS, Warszawa 143

143

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

© UKSW, WMP. SNS, Warszawa

144

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

© UKSW, WMP. SNS, Warszawa

145

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*.

© UKSW, WMP. SNS, Warszawa

146

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

© UKSW, WMP. SNS, Warszawa

147

STL: obiekty funkcyjne

Predykaty – operacje logiczne:

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

© UKSW, WMP. SNS, Warszawa

148

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.

© UKSW, WMP. SNS, Warszawa

149