



Typy kontenerów STL

list

46

STL: kontenery

Kontenery sekwencyjne: **list** (lista dwukierunkowa)

- przystosowany do szybkiego wstawiania elementów w dowolne miejsce sekwencji,
- nie nadaje się do efektywnej realizacji operacji swobodnego dostępu, dlatego brak implementacji operatora indeksowania `[]`,
- najlepiej sprawdza się przy iterowaniu wzdłuż sekwencji lub w jej przeciwnym kierunku,
- ma większy narzut pamięciowy niż pozostałe dwa kontenery – musi przechowywać jeszcze wskaźniki *nast* i *poprz*, dlatego efektywniej jest w nim przechowywać małą liczbę dużych obiektów niż wiele drobnych,
- przemieszczanie obiektów wewnątrz listy polega na przełączaniu wskaźników.

© UKSW, WMP, SNS, Warszawa

47

47

STL: kontenery

Sortowanie listy (*rosnąco, na dwa sposoby*)

1. `void sort();` // metoda bezargumentowa
2. `template <typename Compare> void sort (Compare comp);` // metoda z predykatem

- Elementy są porównywane za pomocą operatora `<` lub `comp`.
- `comp` – typ obiektu funkcyjnego (zwany *predykatem*, może być zrobiony przez programistę); musi zwracać `true`, jeżeli pierwszy argument jest mniejszy od drugiego (sortowanie rosnąco), a `false` – w przeciwnym przypadku.
- Identyczne elementy po posortowaniu zachowują swoje położenie względem siebie.
- wywołanie `sort` nie powoduje wywołania konstruktorów, destruktorów ani kopiowania obiektów w liście.

© UKSW, WMP, SNS, Warszawa

48

48

STL: kontenery

```
list<int> c1;
list<int>::iterator ci_iter;
c1.push_back( 20 );
c1.push_back( 10 );
c1.push_back( 30 );
cout << "Before sorting: c1 =";
for ( ci_iter = c1.begin( ); ci_iter != c1.end( ); ci_iter++ )
    cout << " " << *ci_iter;
cout << endl;
c1.sort( );
cout << "After sorting c1 =";
for ( ci_iter = c1.begin( ); ci_iter != c1.end( ); ci_iter++ )
    cout << " " << *ci_iter;
cout << endl;
c1.sort( greater<int>( ) );
cout << "After sorting with 'greater than' operation, c1 =";
for ( ci_iter = c1.begin( ); ci_iter != c1.end( ); ci_iter++ )
    cout << " " << *ci_iter;
cout << endl;
```

Obiekt funkcyjny służący do porównywania: zwraca `true`, jeżeli pierwszy argument jest większy – sortowanie będzie malejące

© UKSW, WMP, SNS, Warszawa

49

49

STL: kontenery

Obiekt funkcyjny **greater** służący do porównania

```
C++98:
template <class T>
struct greater : binary_function<T,T,bool> {
    bool operator() (const T& x, const T& y) const {
        return x > y;
    }
};

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

© UKSW, WMP, SNS, Warszawa

50

50

STL: kontenery

Obiekt funkcyjny **greater** służący do porównania

```
C++11:
template <class T>
struct greater {
    bool operator() (const T& x, const T& y) const {
        return x > y;
    };
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef bool result_type;
};
```

© UKSW, WMP, SNS, Warszawa

51

51

STL: kontenery

reverse – odwrócenie kolejności elementów w liście

```
list<int> c1;
list<int>::iterator c1_iter;
c1.push_back( 10 );
c1.push_back( 20 );
c1.push_back( 30 );
cout << "c1 = ";
for ( c1_iter = c1.begin(); c1_iter != c1.end(); c1_iter++ )
    cout << " " << *c1_iter;
cout << endl;
c1.reverse();
cout << "Reversed c1 = ";
for ( c1_iter = c1.begin(); c1_iter != c1.end(); c1_iter++ )
    cout << " " << *c1_iter;
cout << endl;
```

© UKSW, WMP, SNS, Warszawa

52

52

STL: kontenery

Przenoszenie elementów listy z jednego kontenera do drugiego

1. void splice (iterator position, list& x);
// cała lista
2. void splice (iterator position, list& x, iterator i);
// pojedynczy element
3. void splice (iterator position, list& x, iterator first, iterator last);
// zakres elementów w liście

Działanie w/w metod wywołanych na rzecz obiektu-kontenera: przenosi

1. wszystkie elementy z kontenera **x**
 2. element **i** z kontenera **x**
 3. elementy z zakresu [**first**, **last**) z kontenera **x**
- do swojego kontenera i wstawia przed miejsce wskazywane przez **position**.

Rozmiary (tj. wynik wywołania "size") obydwu kontenerów ulegają zmianie.

© UKSW, WMP, SNS, Warszawa

53

53

STL: kontenery

Przenoszenie elementów listy

```
list<int> c1, c2, c3, c4;
list<int>::iterator c1_iter,
c2_iter, w_iter, f_iter, l_iter;
c1.push_back( 10 );
c1.push_back( 11 );
c2.push_back( 12 );
c2.push_back( 20 );
c2.push_back( 21 );
c3.push_back( 30 );
c3.push_back( 31 );
c4.push_back( 40 );
c4.push_back( 41 );
c4.push_back( 42 );

w_iter = c2.begin();
w_iter++;
c2.splice( w_iter, c1 );
cout << „Po przeniesieniu c1 do c2: c2=";
for ( c2_iter = c2.begin();
      c2_iter != c2.end(); c2_iter++ )
    cout << " " << *c2_iter;
cout << endl;

Wyjście:
Po przeniesieniu c1 do c2: c2 = 12 10 11 20 21
```

© UKSW, WMP, SNS, Warszawa

54

54

STL: kontenery

Przenoszenie elementów listy

```
list<int> c1, c2, c3, c4;
list<int>::iterator c1_iter,
c2_iter, w_iter, f_iter, l_iter;
c1.push_back( 10 );
c1.push_back( 11 );
c2.push_back( 12 );
c2.push_back( 20 );
c2.push_back( 21 );
c3.push_back( 30 );
c3.push_back( 31 );
c4.push_back( 40 );
c4.push_back( 41 );
c4.push_back( 42 );

f_iter = c3.begin();
c2.splice( w_iter, c3, f_iter );
cout << " Po przeniesieniu pierwszego " <<
" z c3 do c2: c2 = ";
for ( c2_iter = c2.begin();
      c2_iter != c2.end(); c2_iter++ )
    cout << " " << *c2_iter;
cout << endl;

c2: 12 10 11 20 21

Wyjście:
Po przeniesieniu pierwszego z c3 do c2: c2 = 12 10 11 30 20 21
```

© UKSW, WMP, SNS, Warszawa

55

55

STL: kontenery

Przenoszenie elementów listy

```
list<int> c1, c2, c3, c4;
list<int>::iterator c1_iter,
c2_iter, w_iter, f_iter, l_iter;
c1.push_back( 10 );
c1.push_back( 11 );
c2.push_back( 12 );
c2.push_back( 20 );
c2.push_back( 21 );
c3.push_back( 30 );
c3.push_back( 31 );
c4.push_back( 40 );
c4.push_back( 41 );
c4.push_back( 42 );

// kontynuacja; c2: 12 10 11 30 20 21
f_iter = c4.begin();
l_iter = c4.end();
l_iter--;
c2.splice( w_iter, c4, f_iter, l_iter );
cout << " Po przeniesieniu zakresu z c4 "
<< "do c2: c2 = ";
for ( c2_iter = c2.begin();
      c2_iter != c2.end(); c2_iter++ )
    cout << " " << *c2_iter;
cout << endl;

c2: 12 10 11 30 20 21

Wyjście:
Po przeniesieniu zakresu z c4 do c2: c2 = 12 10 11 30 40 41 20
21
```

© UKSW, WMP, SNS, Warszawa

56

56

STL: kontenery

Usuwanie z listy

void remove (const value_type& val);

- usuwa z listy wszystkie elementy, których wartość jest równa **val**,
- na rzecz usuwanych elementów wywoływane są destruktory a rozmiar listy jest redukowany.

© UKSW, WMP, SNS, Warszawa

57

57

STL: kontenery

remove – przykład:

```
list<int> c1;
list<int>::iterator c1_iter, c2_iter;
c1.push_back( 5 );
c1.push_back( 100 );
c1.push_back( 5 );
c1.push_back( 200 );
c1.push_back( 5 );
c1.push_back( 300 );

list<int> c2 = c1;
c2.remove( 5 );

cout << „Po usunięciu elementów " <<
      "z wartością 5, lista c2 =";
for ( c2_iter = c2.begin( );
      c2_iter != c2.end( ); c2_iter++ )
    cout << " " << *c2_iter;
cout << endl;

cout << "Początkowa zawartość listy"
      << " c1 =";
for ( c1_iter = c1.begin( );
      c1_iter != c1.end( ); c1_iter++ )
    cout << " " << *c1_iter;
cout << endl;
```

© UKSW, WMP, SNS, Warszawa

58

58

STL: kontenery

Łączenie posortowanych rosnąco list

1. `void merge(list& x);` // metoda bezargumentowa
 2. `template <typename Compare> void merge(list& x, Compare comp);` // metoda z predykatem
- wprowadza dane z kontenera `x` do listy z zachowaniem ich posortowania w taki sposób, aby posortowanie całości zostało zachowane (obydwa kontenery powinny być uprzednio posortowane wg tego samego kryterium).
 - po zakończeniu kontener `x` jest pusty,
 - Elementy są porównywane za pomocą operatora `<` lub `comp`.
 - `comp` – typ obiektu funkcyjnego; dla dwóch wartości tego samego typu zwraca `true`, jeżeli pierwszy argument jest mniejszy od drugiego, a `false` – w przeciwnym przypadku.
 - wywołanie `merge` nie powoduje nie powoduje wywołania konstruktorów, destruktorów ani kopiowania obiektów.

© UKSW, WMP, SNS, Warszawa

59

59

STL: kontenery

merge – przykład:

```
list<int> c1, c2;
list<int>::iterator c2_iter;

c1.push_back( 3 );
c1.push_back( 6 );
c2.push_back( 2 );
c2.push_back( 4 );

c2.merge( c1 ); // domyślnie połączenie następuje w porządku rosnącym
cout << "Po wykonaniu merge c2 =";
for ( c2_iter = c2.begin( ); c2_iter != c2.end( ); c2_iter++ )
    cout << " " << *c2_iter;
cout << endl;

Wyjście:
Po wykonaniu merge c2 = 2 3 4 6
```

© UKSW, WMP, SNS, Warszawa

60

60

STL: kontenery

Usuwanie powtórzeń

1. `void unique();`
2. `template <typename BinaryPredicate> void unique (BinaryPredicate binary_pred);`

usuwa wszystkie elementy poza pierwszymi ze znalezionych szeregów identycznych elementów w liście (metoda specjalnie dedykowana dla list posortowanych),

- elementy są porównywane za pomocą operatora `=` lub `binary_pred`,
- `binary_pred` – typ obiektu funkcyjnego; dla dwóch wartości tego samego typu zwraca `true`, jeżeli są sobie równe, a `false` – w przeciwnym przypadku,
- dla wszystkich par elementów w kontenerze zostanie wywołany `binary_pred(*i, *(i-1))` i jeżeli zwróci `true`, to usunie element wskazywany przez `i`.

© UKSW, WMP, SNS, Warszawa

61

61

STL: kontenery

```
list<int> c1;
list<int>::iterator c2_iter;
c1.push_back( -10 );
c1.push_back( 10 );
c1.push_back( 10 );
c1.push_back( 20 );
c1.push_back( 20 );
c1.push_back( -10 );
list<int> c2 = c1;
c2.unique( ); // zakładamy, że lista jest posortowana (duplikaty sąsiadują)
cout << "Po usunięciu duplikatów: c2 =";
for ( c2_iter = c2.begin( ); c2_iter != c2.end( ); c2_iter++ )
    cout << " " << *c2_iter;
cout << endl;

Wyjście:
Po usunięciu duplikatów: c2 = c2 = -10 10 20 -10
```

© UKSW, WMP, SNS, Warszawa

62

62

STL: kontenery

Zamiana zawartości kontenerów

```
void swap (list& x);
```

zamienia zawartość kontenera, na rzecz którego została wywołana (wskazywanego przez `this`), z zawartością kontenera `x`.

© UKSW, WMP, SNS, Warszawa

63

63

STL: kontenery

```
list<int> c1,c2;
list<int>::iterator c1_iter, c2_iter;
c1.push_back( 1 );
c1.push_back( 2 );
c1.push_back( 3 );
c2.push_back( 10 );
c2.push_back( 20 );
c1.swap( c2 );
cout << "Po zamianie z c2, lista c1:";
for ( c1_iter = c1.begin( ); c1_iter != c1.end( ); c1_iter++ )
    cout << " " << *c1_iter;
cout << endl;
cout << "Lista c2 po zamianie:";
for ( c2_iter = c2.begin( ); c2_iter != c2.end( ); c2_iter++ )
    cout << " " << *c2_iter;
cout << endl;
```

Wyjście:
Po zamianie z c2, lista c1: 10 20
Lista c2 po zamianie: 1 2 3

© UKSW, WMP, SNS, Warszawa

64

64



Typy kontenerów STL Adaptery kontenerów

STL: kontenery

Adapter kontenera

- wykorzystuje jeden z kontenerów sekwencyjnych do przechowywania własnych danych (*uwaga: w trybie kompozycji, a nie dziedziczenia*),
- domyślnie wykorzystuje zawsze kontener sekwencyjny deque, ale programista ma możliwość wskazać inny kontener sekwencyjny,
- adapter nie ma własnych iteratorów (*ponieważ korzysta z kontenera w trybie kompozycji*).

© UKSW, WMP, SNS, Warszawa

66

66



Typy kontenerów STL Adapter: stack

STL: kontenery

Adapter kontenera: **stack** (stos)

```
template <typename T, typename C1 = deque<T>>
class stack;
```

- daje bardzo niewielki zbiór metod, tj. tylko te właściwe dla operacji na stosie (połóż, zdejmij); m.in. nie da się iterować wzdłuż elementów stosu (LIFO),
- kontener C1 powinien dostarczać następujących metod:
 - empty
 - size
 - back
 - push_back
 - pop_back
- klasy podstawowych kontenerów sekwencyjnych **vector**, **deque** oraz **list** spełniają te wymagania.

© UKSW, WMP, SNS, Warszawa

68

68

STL: kontenery

Adapter kontenera: **stack** (stos)

- push** – wstawia fizycznie element na szczyt stosu; przyjmuje jako argument wartość, która posłuży do zainicjalizowania nowego obiektu, tj. argument dla metody **push_back**,
- pop** – usuwa fizycznie ze szczytu stosu jeden, ostatnio dodany obiekt, tj. wywołany jest na jego rzecz destruktor.

© UKSW, WMP, SNS, Warszawa

69

69

STL: kontenery

Adapter kontenera: **stack** (stos) – przykład:

```
int main () {
    stack < string , vector <string> > str_stack ;
    // albo np.:      stack <string, list<string> > str_stack;
    // albo też po prostu: stack <string> str_stack;

    string quote [3] = {"Ala \n", "ma\n", "kota \n"};
    for ( int i = 0; i < 3; ++i)
        str_stack.push(quote[i]); // umieszcza kopię na szczycie stosu

    while (!str_stack.empty()) {
        cout << str_stack.top(); // zwraca referencję do elementu na szczycie
        str_stack.pop();        // zdejmuję szczytowy element ze stosu
    }
}
```

© UKSW, WMP, SNS, Warszawa

70



Typy kontenerów STL

Adapter: **queue**

STL: kontenery

Adapter kontenera: **queue** (kolejka)

```
template <typename T, typename C1 = deque<T>>
class queue;
```

- ograniczona wersja kontenera **deque**: obsługuje wstawianie do kolejki jedynie pojedynczych elementów na jeden jej koniec i pobieranie pojedynczych elementów z drugiego końca (FIFO),
- nie ma żadnych metod dodatkowych charakterystycznych dla kontenerów sekwencyjnych; nie można iterować wzdłuż kolejki.

© UKSW, WMP, SNS, Warszawa

72

STL: kontenery

Adapter kontenera: **queue** (kolejka)

```
template <typename T, typename C1 = deque<T>>
class queue;
```

- kontener C1 powinien dostarczać następujących metod:
 - **empty**
 - **size**
 - **back**
 - **push_back**
 - **pop_front**
- klasy podstawowych kontenerów sekwencyjnych **vector**, **deque** oraz **list** spełniają te wymagania.

© UKSW, WMP, SNS, Warszawa

73

STL: kontenery

Adapter kontenera: **queue** (kolejka)

- **push** – wstawia fizycznie element na koniec kolejki; przyjmuje jako argument wartość, która posłuży do zainicjalizowania nowego obiektu, tj. argument dla metody **push_back**,
- **pop** – usuwa fizycznie następny obiekt (najstarszy w kolejce), tj. wywołany jest na jego rzecz destruktor.

© UKSW, WMP, SNS, Warszawa

74

STL: kontenery

Adapter kontenera: **queue** (kolejka) – przykład:

```
queue <int> q1;
q1.push( 10 );
q1.push( 20 );
q1.push( 30 );
queue <int>::size_type i;
i = q1.size();
cout << "Długość kolejki wynosi " << i << "." << endl;
int& ii = q1.back(); // odczyt najnowszego elementu w kolejce
int& iii = q1.front(); // odczyt najstarszego elementu w kolejce
cout << "Liczba z tyłu kolejki q1 to " << ii << "." << endl;
cout << "Liczba stojąca na czele kolejki q1 to " << iii << "." << endl;
q1.pop();
i = q1.size();
cout << "Po zdjęciu elementu długość kolejki wynosi " << i << "." << endl;
```

© UKSW, WMP, SNS, Warszawa

75



Typy kontenerów STL

Adapter: `priority_queue`

76

STL: kontenery

Adapter kontenera: `priority_queue` (kolejka priorytetowa)

```
template < typename T,
            typename C1 = vector<T>,
            typename Compare = less<typename C1::value_type>>
class priority_queue;
```

- działa jak zwykła kolejka, ale wywołanie metody `push` w celu wstawienia obiektu oznacza wstawienie elementu w określonym miejscu kolejki, zależnie od priorytetu obiektu;
- jest to więc **kolejka elementów posortowanych wg określonej wartości (element o wartości najwyższej jest na pierwszym miejscu w kolejce)**.
- sortowanie odbywa się z wykorzystaniem funkcji lub obiektu funkcyjnego definiującego relację 'mniejsze niż' dla typu elementów przechowywanych w kolejce – domyślnie jest to szablon elementu funkcyjnego `less`

© UKSW, WMP, SNS, Warszawa 77

77

STL: kontenery

Obiekt funkcyjny `less` służący do porównania

```
template <class T> struct less : binary_function<T,T,bool> {
    bool operator() (const T& x, const T& y) const {
        return x<y;
    };
};

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

© UKSW, WMP, SNS, Warszawa

78

78

STL: kontenery

Adapter kontenera: `priority_queue` – przykład:
kolejka liczb posortowanych rosnąco (element o wartości najwyższej jest na pierwszym miejscu w kolejce):

```
int main() {
    priority_queue<int> pqi;
    srand(time(0)); // uruchamiamy generator liczb losowych

    for(int i = 0; i < 100; i++)
        pqi.push(rand() % 25);

    while(!pqi.empty()) {
        cout << pqi.top() << ' ';
        pqi.pop();
    }
}
```

© UKSW, WMP, SNS, Warszawa

79

79

STL: kontenery

Adapter kontenera: `priority_queue` – przykład:
wskazanie własnego kontenera (`vector`) i własnego obiektu funkcyjnego (szablon elementu funkcyjnego `greater` – sortowanie od największego do najmniejszego):

```
int main() {
    priority_queue<int, vector<int>, greater<int>> pqi;
    srand(time(0)); // uruchamiamy generator liczb losowych

    for(int i = 0; i < 100; i++)
        pqi.push(rand() % 25);

    while(!pqi.empty()) {
        cout << pqi.top() << ' ';
        pqi.pop();
    }
}
```

© UKSW, WMP, SNS, Warszawa

80

80

STL: kontenery

Podsumowanie: po co używać adapterów `stack`, `queue` i `priority_queue`, jeżeli tylko opakowują kontener sekwencyjny `deque` i ograniczają zbiór dostępnych metod?

- ✓ Aby ograniczyć i uczynić bardziej jednoznacznym kod naszego programu. To poprawia bezpieczeństwo i intuicyjność.
Np. dla kogoś, kto nie wie, że używamy `deque` aby symulować `stos`, odgadywanie tego ze sposobu jego wykorzystania w kodzie może prowadzić do nieporozumień, a dalsze rozwijanie tego kodu daje większe ryzyko błędów. Natomiast kiedy w kodzie wykorzystywany jest `stack`, jest jasne do czego służy, a ograniczenie jego zbioru metod do tych niezbędnych ogranicza też możliwości popełnienia błędów w razie modyfikacji kodu.

© UKSW, WMP, SNS, Warszawa

81

81



Typy kontenerów STL

Kontenery asocjacyjne

82

STL: kontenery

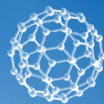
Kontenery asocjacyjne – **set**, **map**, **multiset** i **multimap**

- wraz z wartościami przechowują klucze dla tych wartości (**map** i **multimap**), lub traktują klucze jako przechowywane wartości (**set** i **multiset**)
- w przypadku **set** podstawowymi operacjami są polecenia wstawiania elementu do kontenera oraz sprawdzania czy taki element istnieje w kontenerze
- w przypadku **map** wyszukiwanie po kluczu odbywa się w ten sposób, że najpierw szuka w kontenerze klucza i dopiero jeżeli żądany klucz istnieje, następuje odwołanie do wartości skojarzonej z tym kluczem

© UKSW, WMP, SNS, Warszawa

83

83



Typy kontenerów STL

set

84

STL: kontenery

Kontenery asocjacyjne – **set**

Modeluje tradycyjny zbiór matematyczny, w którym nie mogą pojawić się identyczne wartości. Zawiera składową metodę **insert** która dodaje element do zbioru, pomijając te wartości, które już w zbiorze są.

Kontenery typu **set** można poddawać operacji sumy zbiorów, iloczynu, różnicy albo weryfikacji zawierania się jednego zbioru w drugim.

Ten kontener można wykorzystać do zupełnie innego zadania, np. policzenia wszystkich słów, jakie choć raz pojawiły się w określonym tekście.

© UKSW, WMP, SNS, Warszawa

85

85

STL: kontenery

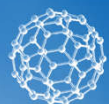
```
void wordSet(const char* fileName) {
    ifstream source(fileName);
    string word;
    set<string> words;
    while(source >> word)
        words.insert(word);
    cout << "Liczba różnych słów w tekście: " << words.size() <<
    endl;
};

int main(int argc, char* argv[]) {
    if(argc > 1)
        wordSet(argv[1]);
    else
        wordSet("WordSet.cpp");
}
```

© UKSW, WMP, SNS, Warszawa

86

86



Typy kontenerów STL

map

87

STL: kontenery

Kontenery asocjacyjne – map

Przykład:

Kontener przechowujący dane studentów:

```
typedef map<string, Student> Studenci; //Studenci - mapa elementów
```

Opis studenta – pola obiektu typu Student:

1. Nazwisko (pole kluczowe)
2. Wiek (pole opisowe)

© UKSW, WMP, SNS, Warszawa

88

88

STL: kontenery

Kontenery asocjacyjne – map

```
class Student {
    string Nazwisko;
    int Wiek;
public:
    Student(): Nazwisko("Nowy student"), Wiek(19) {};
    Student(const string& nazwisko, const int wiek):
        Nazwisko(nazwisko), Wiek(wiek) {};
    Student(const Student& temp):
        Nazwisko(temp.PobierzNazwisko()), Wiek(temp.PobierzWiek()) {};
    ~Student() {};
    void UstawNazwisko(const string& nazwisko) { Nazwisko = nazwisko; };
    string PobierzNazwisko() const { return Nazwisko; };
    void UstawWiek(const int wiek) { Wiek = wiek; };
    int PobierzWiek() const { return Wiek; };
    Student& operator=(const Student& temp);
};
```

© UKSW, WMP, SNS, Warszawa

89

89

STL: kontenery

Kontenery asocjacyjne – map

```
Student& Student::operator=(const Student& temp) {
    Nazwisko = temp.PobierzNazwisko();
    Wiek = temp.PobierzWiek();
    return *this;
};

ostream& operator<<(ostream& osoba, const Student& temp) {
    osoba << temp.PobierzNazwisko() << " ma " << temp.PobierzWiek() << " lat";
    return osoba;
};

template<typename T, typename A>
void ShowMap(const map<T, A>& v) {
    for (typename map<T, A>::const_iterator i = v.begin(); i != v.end(); ++i)
        cout << i->first << ": " << i->second << "\n";
    cout << endl;
};
```

© UKSW, WMP, SNS, Warszawa

iter->first wskazuje klucz, a iter->second wskazuje na wartość (obiekt klasy Student)

90

90

STL: kontenery

```
typedef map<string, Student> Studenci; //Studenci - mapa elementów
int main() {
    Student Kowalski("Kowalski", 19);
    Student Nowak("Nowak", 20);
    Student Adamczewski("Adamczewski", 21);
    Student Kowalczyk("Kowalczyk", 16);
    Studenci Rocznik;
    // dodajemy nowe elementy i nadajemy im wartości
    Rocznik[Kowalski.PobierzNazwisko()] = Kowalski;
    Rocznik[Nowak.PobierzNazwisko()] = Nowak;
    Rocznik[Adamczewski.PobierzNazwisko()] = Adamczewski;
    Rocznik[Kowalczyk.PobierzNazwisko()] = Kowalczyk;
    cout << "Rocznik:\n";
    ShowMap(Rocznik);
    // próba odwołania się do nieistniejącego klucza bez przypisania wartości
    cout << "Wiemy że " << Rocznik["Zeromski"].PobierzNazwisko()
        << " ma " << Rocznik["Zeromski"].PobierzWiek() << " lat\n";
}
```

© UKSW, WMP, SNS, Warszawa

91

91

STL: kontenery

Odwołanie się do nieistniejącego klucza

```
Rocznik["Zeromski"].PobierzNazwisko()
```

- w przypadku tablicy odwołanie się do nieistniejącego elementu spowoduje błąd, natomiast..
- w przypadku mapy nieistniejący element (tj. przypisanie) po prostu się stworzy z wykorzystaniem konstruktora domyślnego:
 - Nazwisko: Nowy student
 - Wiek: 19

Dlatego operator [] nie nadaje się do wyszukiwania istniejących elementów.

```
cout << "Wiemy że " << Rocznik["Zeromski"].PobierzNazwisko()
    << " ma " << Rocznik["Zeromski"].PobierzWiek() << " lat\n";
```

© UKSW, WMP, SNS, Warszawa

Napis w oknie konsoli:

Wiemy, że Nowy student ma 19 lat

92

92

STL: kontenery

Składowe first i second

elementu wskazywanego przez iterator przechowują odpowiednio klucz i wartość elementu, co wynika z definicji typu map::value_type jako synonimu odpowiedniej specjalizacji typu std::pair:

```
typedef pair<const Key, T> value_type;
```

Szablon tworzy kopie wartości składowych, dlatego potrzebny jest konstruktor kopiujący i destruktor dla typu przechowywanych danych.

```
template<typename Type1, typename Type2>
struct pair {
    typedef Type1 first_type;
    typedef Type2 second_type;
    Type1 first;
    Type2 second;

    pair();
    pair(const Type1& _Val1, const Type2& _Val2);
    template<typename Other1, typename Other2>
    pair(const pair<Other1, Other2>& _Right);
};
```

© UKSW, WMP, SNS, Warszawa

93

93

STL: kontenery

Kontener **map**: przykład – zadanie:

Jak policzyć liczby *różnych* słów występujących w tekście?

Rozwiązanie:

Potrzebny jest kontener, który będzie zawierał listę różnych słów (tj. listę bez powtórzeń), która dla każdego słowa będzie miała przypisany licznik wystąpień tego słowa.

© UKSW, WMP, SNS, Warszawa

94

94

STL: kontenery

```
typedef map<string,int> WMap;
void wordMap(const char* fileName) {
    ifstream source(fileName);
    string word;
    WMap words;
    while(source >> word)
        words[word]++; // (!)
};
int main(int argc, char* argv[]) {
    if(argc > 1)
        wordMap(argv[1]);
    else
        wordMap("WordMap.cpp");

    for (WMap::iterator w = words.begin(); w!=words.end(); w++)
        cout << w->first << ":" << w->second << endl;
    cout << "Liczba różnych słów w tekście:" << words.size() << endl;
}
```

© UKSW, WMP, SNS, Warszawa

95

95

STL: kontenery

words[word]++;

Wyrażenie inkrementuje wartość typu **int** skojarzoną z wartością zmiennej **word**.

Jeżeli słowo o wartości **word** nie występuje w kontenerze, do kontenera jest wstawiana automatycznie para *klucz-wartość* o kluczu równym **word** i składowej wartości inicjalizowanej przez wywołanie pseudo-konstruktora **int()**, zwracającego wartość zero.

Jeżeli wiemy, że w tekście wystąpiło słowo np. **typedef** i chcemy tylko dowiedzieć się, ile razy wystąpiło, można napisać:

```
cout << words["typedef"] << endl;
```

Jeżeli jednak tego słowa nie było w zbiorze słów, poleceniem powyżej zostanie ono dodane.

© UKSW, WMP, SNS, Warszawa

96

96