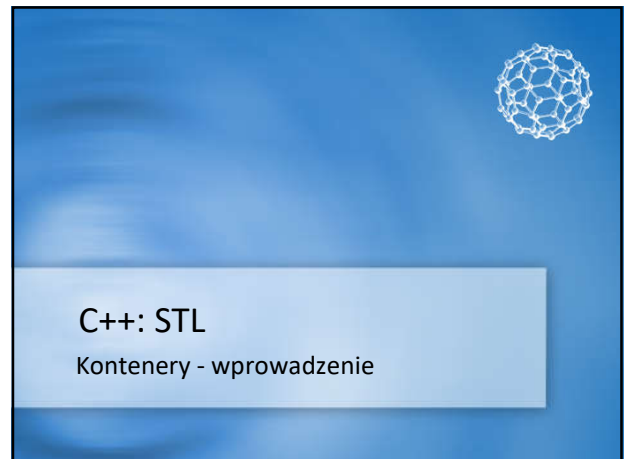
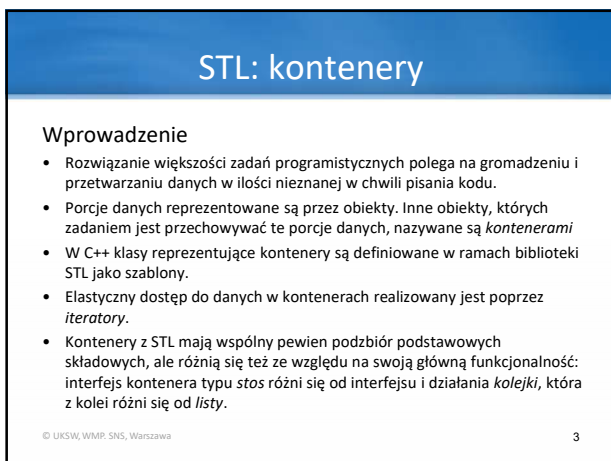




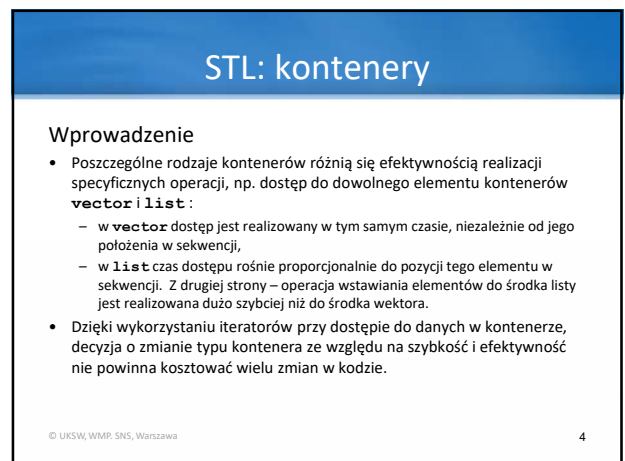
1



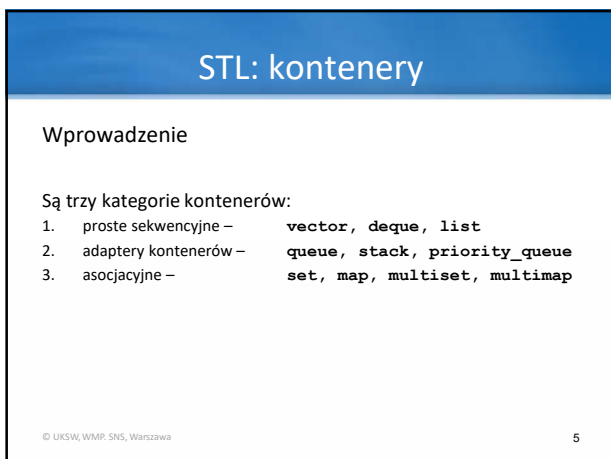
2



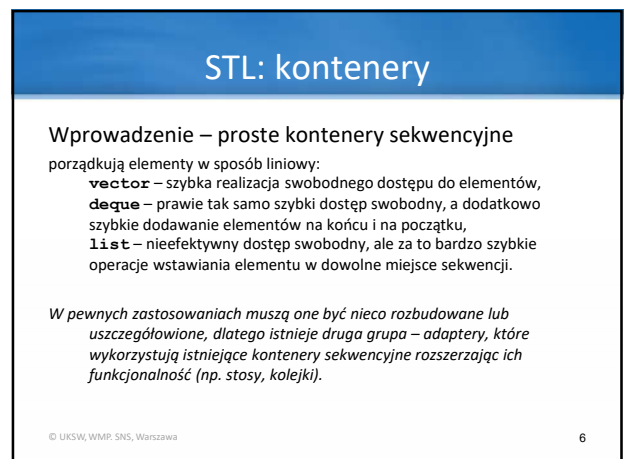
3



4



5



6

STL: kontenery

Wprowadzenie – proste kontenery sekwencyjne

- Wszystkie kontenery sekwencyjne udostępniają metodę `push_back`, a kontenery `deque` i `list` – dodatkowo `push_front`
- do elementów `vector` i `deque` można odwoływać się za pomocą []
- kontenery są właścicielami obiektów – przechowują kopie obiektów podawanych w argumentach. To oznacza, że obiekty przechowywane w kontenerach muszą implementować konstruktor kopiujący i operator przypisania a także rozsądnie działający destruktor.

Jeżeli kontenery przechowują wskaźniki do obiektów, należy pamiętać, że nie one są wtedy odpowiedzialne za zwolnienie tych obiektów przy usuwaniu kontenera (wskaźniki możemy chcieć przechowywać, kiedy chcemy np. wykorzystać polimorfizm obecny w obiektach).

© UKSW, WMP, SNS, Warszawa

7

7

STL: kontenery

```
class Shape {
public:
    virtual void draw() = 0;
    virtual ~Shape() {}
};
class Circle : public Shape {
public:
    void draw() { cout << "C::draw" << endl; } // dodajemy wskaźniki na obiekty
    ~Circle() { cout << "~Circle" << endl; }    F.push_back(new Circle);
};
class Square : public Shape {
public:
    void draw() { cout << "Square::draw" << // wywołujemy polimorficzną metodę draw
    endl; }                                  for(I j = F.begin(); j != F.end(); j++)
    ~Square() { cout << "-Square" << endl;   (*j)->draw();
    };                                       // ...
};
int main() {
    typedef std::vector<Shape*> C1;
    typedef C1::iterator I;
    C1 F;
    for(I k = F.begin(); k != F.end(); k++)
        delete *k;
}
```

© UKSW, WMP, SNS, Warszawa

8

8

C++: STL

Iteratory kontenerów



STL: kontenery

Iteratory

- Wszystkie kontenery mają swoje iteratory (za wyjątkiem adapterów kontenerów: `stack`, `queue` i `priority_queue`, ponieważ implementują one struktury nienadające się do iteracyjnego dostępu)
- Zawsze istnieją:
`<TypKontenera>::iterator;`
`<TypKontenera>::const_iterator;`
- Jeżeli obiekt kontenera został zadeklarowany jako niemodyfikowalny (`const`), metody `begin` i `end` zwracają iteratory niemodyfikujące, tj. za pośrednictwem których można odczytywać, ale nie można modyfikować zawartości kontenera.

© UKSW, WMP, SNS, Warszawa

10

10

STL: kontenery

Przykład #1: korzystanie z iteratorów

```
typedef std::vector<int> Vector; // alias dla vector<int>
```

Zaniczalimizujemy kontener-wektor za pomocą tablicy elementów typu `Vector::value_type` (alias dla parametru szablonu `vector` reprezentującego przechowywany typ danych):

```
const Vector::value_type arr[] = { 3, 4, 7, 8 };
```

Podajemy do konstruktora dwa argumenty, początek i koniec zakresu danych:

```
const Vector v (arr + 0, arr + sizeof(arr) / sizeof(*arr));
```

Wysyłamy na wyjście oryginalną zawartość wektora:

```
cout << "Przeładowujemy wektor za pomocą iteratora: \n ";
for (Vector::const_iterator i = v.begin (); i != v.end (); ++i)
    cout << *i << " ";
```

© UKSW, WMP, SNS, Warszawa

11

11

STL: kontenery

Iteratory

- Istnieją też:
`<TypKontenera>::reverse_iterator;`
`<TypKontenera>::const_reverse_iterator;`
które są adapterami iteratorów;
- Ich zadaniem jest udostępnić sekwencję w odwrotnej kolejności.

© UKSW, WMP, SNS, Warszawa

12

12

STL: kontenery

Przykład #2: korzystanie z iteratorów

```
typedef std::vector<int> Vector;
const Vector::value_type arr[] = { 3, 4, 7, 8 };
const Vector v (arr + 0, arr + sizeof(arr) / sizeof(*arr));
```

Wysyłamy na wyjście zawartość wektora **od końca (!)**

```
cout << "reverse_iterator: \n ";
for (Vector::const_reverse_iterator j = v.rbegin();
     j != v.rend(); ++j)
    cout << *j << " ";
cout << endl;
```

rbegin - zwraca iterator na ostatni element w wektorze
rend - zwraca iterator na miejsce przed pierwszym elementem w wektorze

© UKSW, WMP, SNS, Warszawa

13

13

STL: kontenery

Przykład #3: korzystanie z iteratorów

- Jak wiadomo, iterator może być używany do zwracania elementu, na który wskazuje. Może to zrobić na dwa sposoby. Np. jeżeli mamy iterator `it` i metodę `zrobto()`, będącą metodą obiektu przechowywanego w kontenerze, można tę metodę wywołać tak: `(*it).zrobto()`;

lub tak:
`it->zrobto();`

- Wszystkie kontenery mają iteratory, stąd: korzystając z tych konstrukcji możemy utworzyć szablon funkcji działający z dowolnym kontenerem. Szablon ten przyjmuje wskaźnik względny do składowej metody i wywołuje tę metodę dla każdego obiektu kontenera.

© UKSW, WMP, SNS, Warszawa

14

14

STL: kontenery

Typ danych przechowywanych w kontenerze:

```
class Z {
    int i;
public:
    Z(int ii) : i(ii) {}
    void g() { ++i; }
    friend ostream& operator<<(
        ostream& os, const Z& z);
};

ostream& operator<<(ostream& os,
    const Z& z) {
    return os << z.i;
};
```

© UKSW, WMP, SNS, Warszawa

15

15

Szablon funkcji `apply` działającej na danych w kontenerze:

```
template<typename Cont,
        typename PtrMemFun>
void apply(Cont& c, PtrMemFun F) {
    typename Cont::iterator it = c.begin();
    while (it != c.end()) {
        ((*it).*F)();
        ++it;
    }
}
```

STL: kontenery

Przykład zastosowania w `main`

```
int main() {
    vector<Z> vz;
    // uzupełniamy wektor
    for (int i = 0; i < 10; i++)
        vz.push_back(Z(i));

    // wysyłamy wektor do strumienia cout
    for (vector<Z>::iterator j = vz.begin();
         j != vz.end(); ++j) cout << *j << " ";

    // wykorzystujemy szablon apply
    apply(vz, &Z::g);

    // ponownie wysyłamy wektor do cout
    for (vector<Z>::iterator j = vz.begin();
         j != vz.end(); ++j) cout << *j << " ";
}
```

Symbol '&' wykorzystywany przy pobieraniu wskaźnika względnego nie oznacza tu pobrania bezwzględnego adresu żadnego obiektu, ale oznacza, że wartościami wskaźnika jest przesunięcie składowej względem dowolnego obiektu klasy

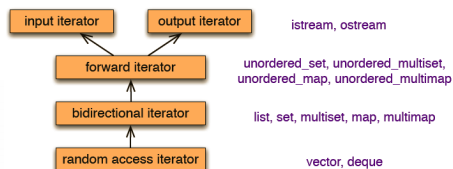
© UKSW, WMP, SNS, Warszawa

16

16

STL: kontenery

Ogólna klasyfikacja iteratorów w C++



Te kategorie mają znaczenie przy korzystaniu z nich w szablonach algorytmów (określenie dla jakiej klasy iteratorów algorytm został zaprojektowany) oraz przy tworzeniu własnych iteratorów, aby dostosowywać je do używania w istniejących szablonach algorytmów.

© UKSW, WMP, SNS, Warszawa

17

17

STL: kontenery

Iteratory wspierające dostęp do danych w strumieniu

- Iteratory wejściowe
`istream_iterator`, `istreambuf_iterator`
- Iteratory wyjściowe
`ostream_iterator`, `ostreambuf_iterator`

© UKSW, WMP, SNS, Warszawa

18

18

STL: kontenery

Iterator wejściowy

istream_iterator, **istreambuf_iterator**

służą do odczytu z sekwencji będących strumieniami.

Iterator wejściowy może zostać poddany *jednokrotnemu* wyluskaniu dla każdego elementu sekwencji, i to tylko w operacji odczytu wartości.

Mogą być przesuwane wyłącznie zgodnie z porządkiem sekwencji. Wartość graniczną końca sekwencji definiuje specjalny konstruktor oznaczający koniec strumienia wejściowego.

© UKSW, WMP, SNS, Warszawa

19

19

STL: kontenery

Iterator wejściowy – przykład:

```
double value1, value2;
istream_iterator<double> eos; // domyślnie EOF
istream_iterator<double> iit (cin); // iterator std wejścia
cout << "Podaj dwie wartości: ";
if (iit!=eos)
    value1=*iit;

iit++;
if (iit!=eos)
    value2=*iit;

cout << value1 << "*" << value2 << "=" << (value1*value2) <<
endl;
```

© UKSW, WMP, SNS, Warszawa

20

20

STL: kontenery

Iterator wyjściowy

ostream_iterator, **ostreambuf_iterator**

pozwalają wyłącznie na przypisanie wartości do wyluskanego elementu.

Iteratory wyjściowe mogą być wyluskane *tylko raz* dla każdego elementu sekwencji i to w operacji przypisania wartości do elementu; przesunięcie iteratora jest możliwe wyłącznie zgodnie z kierunkiem sekwencji. Dla tego iteratora nie ma pojęcia iteratora granicznego.

© UKSW, WMP, SNS, Warszawa

21

21

STL: kontenery

Przykład:

```
int main () {
    vector<int> V;
    for (int i=1; i<10; ++i) V.push_back(i*10);

    ostream_iterator<int> out_it (cout, " ");
    // dwa argumenty: strumień i separator

    copy ( V.begin(), V.end(), out_it ); // algorytm kopiujący,
    // szczegóły na dalszych
    // wykładach

    Wyjście: 10, 20, 30, 40, 50, 60, 70, 80, 90,
}
```

© UKSW, WMP, SNS, Warszawa

22

22

STL: kontenery

Pozostałe iteratory

2. **Iterator przedni** (*forward iterator*) – zawiera całą funkcjonalność iteratora wejściowego i wyjściowego, ponadto pozwala na wielokrotne wyluskanie wskazywanego elementu, przesuwa się jedynie zgodnie z kierunkiem sekwencji (w STL nie ma iteratorów wyłącznie przednich)
3. **iteritor dwukierunkowy** (*bidirectional operator*) – pełna funkcjonalność przedniego oraz możliwość przesuwania się w kierunku przeciwnym do kierunku sekwencji (działania: ++, --)
4. **iteritor dostępu swobodnego** (*random access operator*) – pełna funkcjonalność iteratora dwukierunkowego oraz pełna funkcjonalność wskaźników, tj. operator [], dodawanie wartości całkowitej w celu przesunięcia, porównania wartości wskaźników dla określenia „starszego” (działania: ++, --, [], +, -, +=, -=).

© UKSW, WMP, SNS, Warszawa

23

23

STL: kontenery

Zrób to sam

Niezależnie od istnienia metod robiących różne czynności, warto też implementować swoje własne szablony funkcji realizujące typowe zadania. Pomocne narzędzie do obsługi kontenerów – **szablon do czyszczenia obiektów wskazywanych przez dowolny kontener posiadający wskaźniki**.

- Jeżeli przyjęliśmy, że kontener przechowuje wskaźniki do obiektów dynamicznych, należy ustalić, kto ma prawo i obowiązek usuwać te obiekty.
- Jeżeli obiekty te mają być usuwane razem z kontenerem, dobrze jest napisać sobie narzędzie do wykonywania tej czynności.



Szablony prezentowane na następnym slajdzie zakładają, że kontener jest w pełni zgodny z konwencjami przyjętymi w STL i jest właścicielem iteratorów przynajmniej przednich (*forward iterator*).

© UKSW, WMP, SNS, Warszawa

24

24

STL: kontenery

Zrób to sam

Wersja usuwająca obiekty wskazywane przez wszystkie wskaźniki przechowywane w kontenerze

```
template<typename Seq>
void usuwanie(Seq& c) {
    typename Seq::iterator i;
    for(i = c.begin(); i != c.end(); ++i) {
        delete *i;
        *i = 0;
    }
}
```

© UKSW, WMP, SNS, Warszawa

25

25

STL: kontenery

Zrób to sam

Wersja usuwająca obiekty wskazywane przez wskaźniki z przedziału identyfikowanego przez parę iteratorów

```
template<typename InpIt>
void usuwanie(InpIt begin, InpIt end) {
    while(begin != end) {
        delete *begin;
        *begin = 0;
        ++begin;
    }
}
```



© UKSW, WMP, SNS, Warszawa

26

26

C++: STL
Typy kontenerów

27

Typy kontenerów STL
vector

29

Typy kontenerów STL
Kontenery sekwencyjne

28

STL: kontenery

Kontenery sekwencyjne: **vector**

- imituje tablicę,
- może dynamicznie zmieniać swój rozmiar
- dla zwiększenia efektywności przechowuje obiekty w ciągłym obszarze pamięci (w tablicy), co daje szybkość porównywalną z szybkością dostępu do zwykłej tablicy
- dołączenie obiektu do kontenera jest czynnością pracochłonna (dla kontenera)

© UKSW, WMP, SNS, Warszawa

30

30

STL: kontenery

Kontenery sekwencyjne: **vector**

- Pojemność wektora wyraża się przez dwa pojęcia:
 - size** – nominalny rozmiar wektora (liczba zapisanych elementów),
 - capacity** – rozmiar ciągłego obszaru pamięci zaalokowanego przez obiekt.
- Gdy w kontenerze kończy się miejsce do przechowywania obiektów, dołączenie kolejnego wiąże się z czynnością przydziału nowego, większego obszaru pamięci, skopiowania do niego dotychczasowej zawartości kontenera, usunięcia obiektów w starym obszarze pamięci i zwolnienia dotychczas zajmowanego obszaru

© UKSW, WMP, SNS, Warszawa

31

31

STL: kontenery

vector – przykład użycia #1 (prosty):

```
int main () {
    vector <int> v(10) ; // 10 -> liczba elementów

    for (int i = 0; i < 10; ++i)
        v[i] = i;      // swobodny dostęp do pól za pomocą []

    for (vector<int>::iterator p=v.begin(); p!=v.end(); ++p)
        cout << *p << ' '; // dostęp do pól za pomocą iteratora
    cout << endl ;
}
```

© UKSW, WMP, SNS, Warszawa

32

32

STL: kontenery

vector – przykład użycia #2:

- Często opłaca się tworzyć nowe klasy dziedziczące po kontenerach, aby uprościć wielokrotnie powtarzane czynności np. dostępu do plików.
- Jedyna rzecz do rozstrzygnięcia, to czy kontener ma być składową klasy, czy nowa klasa ma dziedziczyć po kontenerze.

Zadanie: Przyjmijmy, że potrzebujemy klasy, która udostępni nam zawartość pliku w formie tablicy dającej swobodny dostęp do poszczególnych wierszy tekstu pliku.

Realizacja będzie polegała na utworzeniu klasy, która w chwili tworzenia nowego obiektu tej klasy od razu odczytuje plik do wektora, po czym udostępni odczytane dane. Jednak dla użytkownika obiektu takiej klasy będzie to niewidoczne – przeniesienie danych z pliku do struktury wektora odbywa się poza jego kontrolą.

© UKSW, WMP, SNS, Warszawa

33

33

STL: kontenery

plik nagłówkowy FileEditor.h

```
class FileEditor :
public vector<std::string> {
public:
    void open(const char* filename);
    FileEditor(const char* filename)
    {
        open(filename);
    };
    FileEditor() {};
    void write(std::ostream& out =
std::cout);
};
```

© UKSW, WMP, SNS, Warszawa

34

plik definicyjny FileEditor.cpp

```
void FileEditor::open(
const char* filename) {
    ifstream in(filename);
    string line;
    while(getline(in, line))
        push_back(line);
};

void FileEditor::write(
ostream& out) {
    for(iterator w = begin();
w != end(); w++)
        out << *w << endl;
};
```

34

STL: kontenery

```
int main(int argc, char* argv[]) {
    FileEditor file;
    if(argc > 1) {
        file.open(argv[1]);
    } else {
        file.open("FEditTest.cpp"); // tu następuje przeniesienie danych
    }
    int i = 1;
    FileEditor::iterator w = file.begin();
    while(w != file.end()) {
        ostreamstream ss;
        ss << i++;
        *w = ss.str() + " : " + *w; // dopisanie numeru na początku wiersza
        ++w;
    }
    file.write(cout);
}
```

© UKSW, WMP, SNS, Warszawa

35

35

STL: kontenery

Dwa sposoby radzenia sobie z problemem kosztowności czasowej przy powiększaniu rozmiaru wektora:

- Jeżeli wiemy, jaka jest maksymalna liczba obiektów, możemy odpowiedni obszar zarezerwować wywołując metodę **reserve()**, jednak ..

nie oznacza to jeszcze, że wektor o takiej długości zostanie od razu utworzony.

Będzie on nadal taki, jak podano np. w konstruktorze, ale za to obszar ciągły pamięci zarezerwowany pod ten wektor będzie większy – w razie potrzeby powiększenia rozmiaru wektora czynność ta będzie się odbywała szybciej.

Manualne zarządzanie zajętością jest z reguły mniej efektywne – chyba, że z góry precyzyjnie znamy rozmiar danych do przechowania w wektorze.

© UKSW, WMP, SNS, Warszawa

36

36

STL: kontenery

Dwa sposoby radzenia sobie z problemem kosztowności czasowej przy powiększaniu rozmiaru wektora:

2. wykorzystanie kontenera `deque`, który czynność dołączania nowych obiektów realizuje dużo szybciej i nigdy nie kopiuje i nie usuwa elementów w przypadku zmiany rozmiaru kontenera (o tym kontenerze będzie wkrótce powiedziane więcej).

© UKSW, WMP, SNS, Warszawa

37

37

STL: kontenery

vector – unieważnienie iteratorów:

```
int main() {
    vector<int> vi(10, 0);
    vector<int>::iterator i = vi.begin();
    *i = 47;

    ostream_iterator<int> out(cout, " ");
    copy(vi.begin(), vi.end(), out); // wysłanie do strumienia cout (trick ☹)
    cout << endl;

    vi.resize(vi.capacity()+1); // wymuszenie przydziału nowego bloku pamięci

    // Od tego momentu iterator i wskazuje na niewłaściwy obszar pamięci

    *i = 48; // błąd dostępu do niezainicjalizowanej pamięci

    copy(vi.begin(), vi.end(), out); // Na pozycji vi[0] nie ma zmiany wartości
}
```

© UKSW, WMP, SNS, Warszawa

38

38

STL: kontenery

Pozostałe metody wektora:

- assign** – usuwa wszystkie dotychczasowe elementy z wektora, a następnie kopiuje określoną liczbę elementów do pustego kontenera
- at** – zwraca referencję na element w określonej komórce wektora
- back** – zwraca referencję na ostatni element w wektorze
- begin** – zwraca iterator dostępu swobodnego wskazujący na pierwszy element w kontenerze
- capacity** – zwraca liczbę elementów, jaką wektor mógłby zawierać bez realokacji zajmowanego obszaru pamięci
- clear** – usuwa wszystkie elementy z wektora
- empty** – zwraca wartość `true`, jeżeli wektor jest pusty
- end** – zwraca iterator dostępu swobodnego wskazujący na miejsce za ostatnim elementem w kontenerze

© UKSW, WMP, SNS, Warszawa

39

39

STL: kontenery

Pozostałe metody wektora:

- erase** – usuwa wskazany element (lub zakres elementów) z wektora
- front** – zwraca referencję na pierwszy element w wektorze
- get_allocator** – zwraca alokator wektora (funkcja zaawansowana)
- insert** – wstawia element (lub pewną liczbę elementów) do wektora na wskazaną pozycję
- max_size** – zwraca maksymalną długość wektora
- pop_back** – usuwa element z końca wektora
- push_back** – dodaje element na koniec wektora
- rbegin** – zwraca iterator na pierwszy element przy dostępie w odwrotnej kolejności (tj. na element ostatni)
- rend** – zwraca iterator na miejsce za ostatnim elementem przy dostępie w odwrotnej kolejności (tj. miejsce przed elementem pierwszym)

© UKSW, WMP, SNS, Warszawa

40

40

STL: kontenery

Pozostałe metody wektora:

- resize** – określa nowy rozmiar dla wektora (dodaje lub usuwa elementy)
- reserve** – rezerwuje ciągły obszar pamięci zdolny do przechowania określonej liczby elementów
- size** – zwraca liczbę elementów wektorze
- swap** – zamienia elementy między dwoma wektorami

© UKSW, WMP, SNS, Warszawa

41

41