1

2

STL: kontenery

Wprowadzenie

- Rozwiązanie większości zadań programistycznych polega na gromadzeniu i przetwarzaniu danych w ilości nieznanej w chwili pisania kodu.
- Porcje danych reprezentowane są przez obiekty, inne obiekty, których zadaniem jest przechowywać te porcje danych, nazywane są *kontenerami*
- W C++ klasy reprezentujące kontenery są definiowane w ramach biblioteki STL jako szablony.
- Elastyczny dostęp do danych w kontenerach realizowany jest poprzez *iteratory*.
- Kontenery z STL mają wspólny pewien podzbiór podstawowych składowych, ale różnią się też ze względu na swoją główną funkcjonalność: interfejs kontenera typu *stos* różni się od interfejsu i działania *kolejki*, która z kolei różni się od *listy*.

© UKSW, WMP. SNS, Warszawa

3

STL: kontenery

Wprowadzenie

- Poszczególne rodzaje kontenerów różnią się efektywnością realizacji specyficznych operacji, np. dostęp do dowolnego elementu kontenerów **`vector`** i **`list`**:
  - w **`vector`** dostęp jest realizowany w tym samym czasie, niezależnie od jego położenia w sekwencji,
  - w **`list`** czas dostępu rośnie proporcjonalnie do pozycji tego elementu w sekwencji. Z drugiej strony – operacja wstawiania elementów do środka listy jest realizowana dużo szybciej niż do środka wektora.
- Dzięki wykorzystaniu iteratorów przy dostępie do danych w kontenerze, decyzja o zmianie typu kontenera ze względu na szybkość i efektywność nie powinna kosztować wielu zmian w kodzie.

© UKSW, WMP. SNS, Warszawa

4

STL: kontenery

Wprowadzenie

Są trzy kategorie kontenerów:

1. proste sekwencyjne – **`vector, deque, list`**
2. adaptery kontenerów – **`queue, stack, priority_queue`**
3. asocjacyjne – **`set, map, multiset, multimap`**

© UKSW, WMP. SNS, Warszawa

5

STL: kontenery

Wprowadzenie – proste kontenery sekwencyjne

porządkują elementy w sposób liniowy:

`vector` – szybka realizacja swobodnego dostępu do elementów,
`deque` – prawie tak samo szybki dostęp swobodny, a dodatkowo szybkie dodawanie elementów na końcu i na początku,
`list` – nieefektywny dostęp swobodny, ale za to bardzo szybkie operacje wstawiania elementu w dowolne miejsce sekwencji.

W pewnych zastosowaniach muszą one być nieco rozbudowane lub uszczegółowione, dlatego istnieje druga grupa – adaptery, które wykorzystują istniejące kontenery sekwencyjne rozszerzając ich funkcjonalność (np. stosy, kolejki).

© UKSW, WMP. SNS, Warszawa

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 musza 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

STL: kontenery

```
class Shape {
public:
  virtual void draw() = 0;
  virtual ~Shape() {};
};
class Circle : public Shape {
public:
  void draw(){ cout <<"C::draw"<< endl;};
  ~Circle(){ cout <<"~Circle"<< endl;};
};
class Square : public Shape {
public:
  void draw(){ cout << "Square::draw" <<
    endl; };
  ~Square(){ cout << "~Square" << endl;
    };
};
```

```
int main() {
  typedef std::vector<Shape*> C1;
  typedef C1::iterator I;
  C1 F;

  // dodajemy wskaźniki na obiekty
  F.push_back(new Circle);
  F.push_back(new Square);

  // wywołujemy polimorficzną metodę draw
  for(I j = F.begin(); j != F.end(); j++)
    (*j)->draw();
    // ...
  for(I k = F.begin(); k != F.end(); k++)
    delete *k;
}
```

© UKSW, WMP. SNS, Warszawa

8

C++: STL

Iteratory kontenerów

9

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

STL: kontenery

Przykład #1: korzystanie z iteratorów

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

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

© UKSW, WMP. SNS, Warszawa

11

STL: kontenery

Iteratory

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

© UKSW, WMP. SNS, Warszawa

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

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.

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;
};
```

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;
   }
}
```


15

STL: kontenery

Przykład zastosowania w **main**

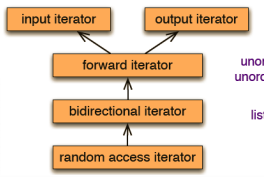
```
int main() {
   vector<Z> vz;
   // zapełniamy wektor
   for (int i = 0; i < 10; i++)
         vz.push_back(Z(i));

   // 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

16

STL: kontenery

Ogólna klasyfikacja iteratorów w C++



istream, ostream

unordered_set, unordered_multiset, unordered_map, unordered_multimap

list, set, multiset, map, multimap

vector, deque

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.

17

STL: kontenery

Iteratory wspierające dostęp do danych w strumieniu

1. Iteratory wejściowe
   istream_iterator, istreambuf_iterator
2. Iteratory wyjściowe
   ostream_iterator, ostreambuf_iterator

18

STL: kontenery

Iterator wejściowy

`istream_iterator, istreambuf_iterator`

służą do odczytu z sekwencji będących strumieniami.

Iterator wejściowy może zostać poddany *jednokrotnemu* wyłuskaniu 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

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

STL: kontenery

Iterator wyjściowy

`ostream_iterator, ostreambuf_iterator`

pozwala wyłącznie na przypisanie wartości do wyłuskanego elementu.

Iteratory wyjściowe mogą być wyłuskiwane *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

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

STL: kontenery

Pozostałe iteratory

2. **Iterator przedni** (*forward iterator*) – zawiera całą funkcjonalność iteratora wejściowego i wyjściowego, ponadto pozwala na wielokrotne wyłuskanie wskazywanego elementu, przesuwa się jedynie zgodnie z kierunkiem sekwencji (w STL nie ma iteratorów wyłącznie przednich)
3. **iterator dwukierunkowy** (*bidirectional operator*) – pełna funkcjonalność przedniego oraz możliwość przesuwania się w kierunku przeciwnym do kierunku sekwencji (działania: ++, --)
4. **iterator 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

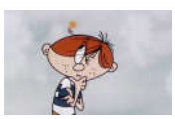
STL: kontenery

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

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

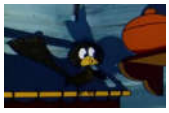
25

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

C++: STL

Typy kontenerów

27

Typy kontenerów STL

Kontenery sekwencyjne

28

Typy kontenerów STL

vector

29

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

STL: kontenery

Kontenery sekwencyjne: **`vector`**

- Pojemność wektora wyraża się przez dwa pojęcia:
  1. **size** – nominalny rozmiar wektora (liczba zapisanych elementów),
  2. **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

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

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 udostępnia odczytane dane. Jednak dla użytkownika obiektu takiej klasy będzie to niewidoczne – przeniesienie danych z pliku do struktur wektora odbywa się poza jego kontrolą.

© UKSW, WMP. SNS, Warszawa

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);
};
```

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;
};
```

© UKSW, WMP. SNS, Warszawa

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()) {
    ostringstream ss;
    ss << i++;
    *w = ss.str() + ": " + *w; // dopisanie numeru na początku wiersza
    ++w;
  }
  file.write(cout);
}
```

© UKSW, WMP. SNS, Warszawa

STL: kontenery

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

1. 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

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

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

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

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

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

Typy kontenerów STL

deque (***double-ended queue***)

STL: kontenery

Kontenery sekwencyjne: **deque** (kolejka dwustronna)

- implementacja kontenera zoptymalizowana pod kątem efektywności operacji dołączania i usuwania elementów z sekwencji na obu jej końcach,
- definiuje operator indeksowania **operator[]()** o stałym czasie dostępu
- w porównaniu z wektorem *nie przechowuje elementów w pojedynczym ciągłym obszarze pamięci* – nie ma więc potrzeby realokowania obszaru pamięci w przypadku powiększania liczby przechowywanych obiektów.
- Zmiana stanu kontenera (dodanie lub usunięcie elementu) powoduje, że iteratory mogą stać się nieważne:
  - Dodane elementu na początek – wszystkie iteratory
  - Dodanie elementu na koniec – wszystkie iteratory oraz **end()**
  - Usunięcie elementu z początku lub końca – iterator wskazujący na ten element

© UKSW, WMP. SNS, Warszawa

43

STL: kontenery

Metody z **deque** nieobecne w kontenerze **vector**:

pop_front – usuwa pierwszy element z początku sekwencji
push_front – dodaje element na początek sekwencji

Na podstawie kontenera **deque** można łatwo implementować:

1. Stos – dodawanie elementu – **push_front**, zdejmowanie elementu – **pop_front**.
2. Kolejka FIFO – dodawanie do kolejki – **push_back**, zdejmowanie z kolejki – **pop_front**,

© UKSW, WMP. SNS, Warszawa

44

STL: kontenery

Konwersja sekwencji danych między kontenerami

```
int main(int argc, char* argv[]) {
  int size = 25;
  deque<Integer> d;
  // tutaj wypełnienie kontenera określoną liczba obiektów typu Integer
  …
  cout << "\n Konwersja do wektora (1)" << endl;
  vector<Integer> v1(d.begin(), d.end());

  cout << "\n Konwersja do wektora (2)" << endl;
  vector<Integer> v2;
  v2.reserve(d.size());
  v2.assign(d.begin(), d.end());
}
```

© UKSW, WMP. SNS, Warszawa

45