Programowanie obiektowe C++
Programowanie zorientowane obiektowo

Wykład 6

Witold Dyrka
witold.dyrka@pwr.wroc.pl

12/11/2012

Prawa autorskie itp.

Wiele slajdów do tego wykładu powstało w oparciu o slajdy Bjarne Stroustrupa do kursu Foundations of Engineering II (C++) *prowadzonego w Texas A&M University* http://www.stroustrup.com/Programming (tym razem większość wykładu:-)

Program wykładów

1. Wprowadzenie. Obsługa błędów. Klasy (8/10)
2. Abstrakcja i enkapsulacja: klasy i struktury. (15/10)
3. Polimorfizm: przeładowanie funkcji i operatorów. Konwersje (22/10)
4. Zarządzanie pamięcią: konstruktor, destruktor, przypisanie (29/10)
5. Dziedziczenie. Polimorfizm dynamiczny (5/11)
6. **Programowanie uogólnione: szablony (12/11)**
7. Kolokwium zaliczeniowe (19/11)

Plan na dziś

- Szablony funkcji
  - definicja, konkretyzacja, specjalizacja
- Szablony klas
- Programowanie uogólnione
- Standardowa biblioteka szablonów STL
- Iteratory

Klasa **vector** liczb **double**

```
class vector {          // an almost real vector of doubles
    int sz;                         // the size
    double* elem;                   // a pointer to the elements
    int space;                      // size+free_space
public:
    vector() : sz(0), elem(0), space(0) { }          // default constructor
    explicit vector(int s) :sz(s), elem(new double[s]) , space(s) { }  // constructor
    vector(const vector&);                           // copy constructor
    vector& operator=(const vector&);                // copy assignment
    ~vector() { delete[ ] elem; }                    // destructor

    double& operator[ ](int n) { return elem[n]; }   // access: return reference
    int size() const { return sz; }                  // current size

    void resize(int newsize);                    // grow (or shrink)
    void push_back(double d);                    // add element

    void reserve(int newalloc);                      // get more space
    int capacity() const { return space; }           // current available space
};
```

Szablony

- **vector** liczb **double** jest świetny...
- Ale potrzebujemy wektorów elementów dowolnego typu:
  - **vector<double>**
  - **vector<int>**
  - **vector<Month>**
  - **vector<Record*>** *// vector of pointers*
  - **vector< vector<Record> >** *// vector of vectors*
  - **vector<char>**
- Chcemy
  - by typ elementu był parametrem **vector**-a
  - by **vector** mógł przechowywać elemeny typów wbudowanych oraz typów użytkownika
- Nie jest to żadna magia kompilatora
  - możemy definiować swoje własne typy i funkcje sparametryzowane, czyli <u>szablony</u>

Zacznijmy od szablonów funkcji

Przykład 1 – max dwóch zmiennych:

- opcja 1: polimorfizm

```
long maks(long a, long b) { return (a>b?a:b); }

double maks(double a, double b) { return (a>b?a:b); }

char maks(char a, char b) { return (a>b?a:b); }
```

- opcja 2: szablon (template)

```
template <class T>
T maks(T a, T b) { return (a>b ? a : b); }
```

Szablony funkcji - składnia

Definicja szablonu:

```
template <class T>
T maks(T a, T b) { ... }
```

lub

```
template <typename T>
T maks(T a, T b) { ... }
```

Konkretyzacja:

```
cout << maks('a','b');

cout << maks(1,2);

cout << maks<int>(1,2);
```

UWAGI:

T może być dowolnym typem:

- prostym
- strukturą
- klasą
- szablonem

class i **typename** używamy zamiennie

Konkretyzacja szablonu funkcji

Przykłady:

```
cout << maks('a','b');              // b
cout << maks(1,2);                  // 2
cout << maks(1.1,2.2);              // 2.2

cout << maks(1,2.2);                // Błąd: niejednoznaczny typ
cout << maks<double>(1,2.2);        // 2.2

cout << maks("leon","jan");         // wynik zależny od... adresów łańcuchów!
```

Specjalizacja szablonu funkcji

```
template <class T>
T maks(T a, T b) { return (a>b?a:b); }

template <>
char* maks(char* a, char* b) {
  return (strcmp(a,b)>0 ? a : b);
}


cout << maks("leon","jan");                 // bez zmian!
                                            // Czy ktoś ma pomysł?

cout << maks<char*>("leon","jan");          // "leon" – działa!
                                            // kompilator odróżnia
                                            // const char* od char*
```

Szablony funkcji

- Kiedy?
  - te same operacje na różnych typach danych przeważnie służące do operowania na zbiorach, np. sortowanie, porównywanie itp.
  - szeroko stosowane w standardowej bibliotece szablonów STL
- Dlaczego?
  - elastyczne i szybkie
  - krótszy, łatwiejszy do pielęgnacji kod
  - bezpieczniejsze niż rzutowanie typów

Wróćmy do klasy **vector**, czyli szablony klas

```
// an almost real vector of Ts:
template <class T> class vector {        // for all types T
    int sz;              // the size
    T* elem;             // a pointer to the elements
    int space;           // size+free_space
public:
    vector() : sz(0), elem(0), space(0) { }                        // default constructor
    explicit vector(int s) :sz(s), elem(new T[s]), space(s) { }    // constructor
    vector(const vector&);                                         // copy constructor
    vector& operator=(const vector&);                              // copy assignment
    ~vector() { delete[ ] elem; }                                  // destructor

    T& operator[ ] (int n) { return elem[n]; }                     // access: return reference
    int size() const { return sz; }                                // the current size

    // ...
};
```

Szablon **vector**-a i jego konkretyzacja

```
// an almost real vector of Ts:
template<class T> class vector {
    // …
};

vector<double> vd;                     // T is double
vector<int> vi;                        // T is int
vector< vector<int> > vvi;             // T is vector<int>
                                       //         in which T is int
vector<char> vc;                       // T is char
vector<double*> vpd;                   // T is double*
vector< vector<double>* > vvpd;        // T is vector<double>*
                                       //         in which T is double
```

vector<double> to po prostu

```
// an almost real vector of doubles:
class vector {
    int sz;                 // the size
    double* elem;           // a pointer to the elements
    int space;              // size+free_space
public:
    vector() : sz(0), elem(0), space(0) { }                  // default constructor
    explicit vector(int s) :sz(s), elem(new double[s]), space(s) { } // constructor
    vector(const vector&);                                   // copy constructor
    vector& operator=(const vector&);                        // copy assignment
    ~vector() { delete[ ] elem; }                            // destructor

    double& operator[ ] (int n) { return elem[n]; }          // access: return reference
    int size() const { return sz; }                          // the current size

    // …
};
```

A vector<char> to:

```
// an almost real vector of chars:
class vector {
    int sz;                  // the size
    char* elem;              // a pointer to the elements
    int space;               // size+free_space
public:
    vector() : sz(0), elem(0), space(0) { }                      // default constructor
    explicit vector(int s) :sz(s), elem(new char[s]), space(s) { } // constructor
    vector(const vector&);                                       // copy constructor
    vector& operator=(const vector&);                            // copy assignment
    ~vector() { delete[ ] elem; }                                // destructor

    char& operator[ ] (int n) { return elem[n]; }                // access: return reference
    int size() const { return sz; }                              // the current size

    // …
};
```

Parametry szablonów klas

kilka parametrów typu

inne parametry (nie typu)

```
template <typename T, typename S = char, int rozmiar=10>
class dwuLista {

 T kolumna1[rozmiar];
 S kolumna2[rozmiar];

public:
 void wloz(int poz, T pole1, S pole2) {
  if (poz>=0&&poz<rozmiar) {
   kolumna1[poz] = pole1;
   kolumna2[poz] = pole2;
  }
  else throw runtime_error("Poza zakresem!");
 };

 T wez1(int poz) {
  if (poz>=0&&poz<rozmiar) return(kolumna1[poz]);
  else throw runtime_error("Poza zakresem!");
 };

 S wez2(int poz) {
  if (poz>=0&&poz<rozmiar) return(kolumna2[poz]);
  else throw runtime_error("Poza zakresem!");
 };
};
```

parametry „nie typu”
w klasie są stałe

wszystkie parametry mogą
posiadać wartości domyślne

Inne cechy szablonów klas

- Szablony klas można specjalizować:
  - szablon ogólny: `template <typename T> class element{ … };`
  - specjalizacja: `template <> class element<char> { … };`
- Szablony klas można częściowo specjalizować:

```
template <class gatunek1, class gatunek2> class krzyzowka  { … };
template <class gatunek1> class krzyzowka<gatunek1, ecoli> { … };
```

...jeśli kompilator pozwala...

- Każda wygenerowana w wyniku instancjacji klasa ma oddzielną kopię składników statycznych:

`static int element<char>::ileObiektow` **oraz** `static int element<double>::ileObiektow`

są od siebie niezależne!!

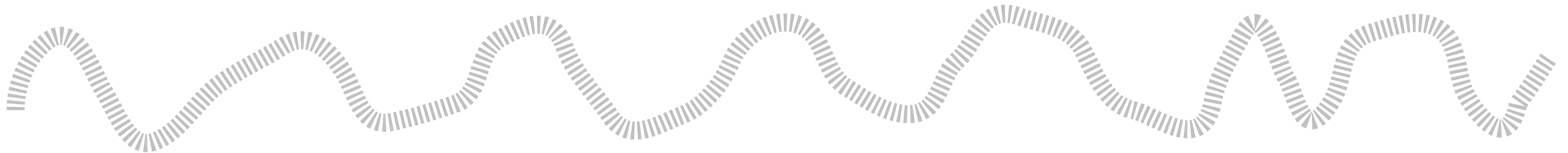
Szablony klas

- Kiedy?
  - tworzenie klas będących konteneramі (pojemnikami) obiektów lub innych danych

- Przykłady:
  - zbiór łańcuchów dowolnego typu
  - stos elementów dowolnego typu
  - kolejki, wektory, listy, tablice ...

Szablony klas (2)

- sparametryzowana klasa
- w momencie podstawienia parametrów (konkretyzacji szablonu) kompilator generuje klasę lub jej metodę
- szablony klas można specjalizować dla parametru



- stosować do pisania klas obsługujących kontenery (pojemniki) obiektów różnych typów
- krótszy i łatwiejszy do pielęgnacji kod
- w zasadzie komplementarne do obiektowości

Każda róża ma kolce

- Problemy
  - kiepska diagnostyka błędów
    - często naprawdę kiepskie komunikaty o błędach
    - często dopiero w momencie konsolidacji
  - wszystkie szablony muszą być w pełni zdefiniowane w każdej jednostce translacji
    - umieszczaj definicje szablonów w plikach nagłówkowych

- Bjarne S. radzi
  - używaj bibliotek opartych na szablonach
    - takich jak standardowa biblioteka szablonów (STL) C++
      - *np..,* **vector**, **sort()**
  - początkowo pisz tylko proste szablony
    - aż nie zdobędziesz doświadczenia

Idźmy dalej...

- Umiemy już pisać klasy i funkcje, które zachowują się bardzo podobnie niezależnie od używanych typów danych
  - używanie **int**-a nie różni się zbyt wiele od używania **double**-a
  - używanie **vector<int>** nie różni się zbyt wiele od używania **vector<string>**
- Chcielibyśmy pisać programy w taki sposób,
  że nie musielibyśmy powtarzać pracy za każdym razem,
  kiedy stworzymy nowy rodzaj kontenera
  albo nieco zmodyfikujemy sposób interpretacji danych

Ideały

- Chcielibyśmy pisać programy w taki sposób,
  że nie musielibyśmy powtarzać pracy za każdym razem,
  kiedy stworzymy nowy rodzaj kontenera
  albo nieco zmodyfikujemy sposób interpretacji danych
  - szukanie wartości w wektorze nie różni się zbyt wiele od szukania wartości w liście lub w tablicy
  - szukanie łańcucha bez względu na wielkość liter nie różni się zbyt wiele od szukania łańcucha z uwzględnieniem wielkości liter
  - rysowanie dokładnych wartości danych eksperymentalnych nie różni się zbyt wiele od rysowania wartości zaokrąglonych
  - Kopiowanie pliku nie różni się zbyt wiele od kopiowania wektora

Ideały (2)

- Kod ma być:
  - łatwy do czytania i łatwy do modyfikacji
  - regularny, krótki, szybki
- Spójny dostęp do danych
  - niezależnie jak są przechowywane, niezależnie od ich typu
  - bezpieczeństwo typów
- Zwarte przechowywanie danych
- Szybkie operacje
  - dostęp do danych, dodawanie i usuwanie danych
- Standardowe wersje najbardziej popularnych algorytmów
  - kopiowanie, szukanie, sortowanie, sumowanie itp.

Programowanie ogólne

Pisanie kodu, który może działać z różnymi typami przekazywanymi do niego jako argumenty, pod warunkiem, że typy te spełniają określone wymagania syntaktyczne I semantyczne

- Celem (dla użytkownika takiego kodu) jest
  - większa poprawność
    - dzięki lepszej specyfikacji
  - szerszy zakres zastosowań
    - możliwość ponownego wykorzystania
  - lepsza wydajność
    - dzięki szerszemu użyciu optymalnych bibliotek
    - pozbycie się niepotrzebnie wolnego kodu
- Od konkretu do abstrakcji
  - w przeciwnym wypadku kod często puchnie

Przykład uogólniania (konkretne algorytmy)

```
double sum(double array[], int n)      // one concrete algorithm (doubles in array)
{
    double s = 0;
    for (int i = 0; i < n; ++i ) s = s + array[i];
    return s;
}

struct Node { Node* next; int data; };

int sum(Node* first)                   // another concrete algorithm (ints in list)
{
    int s = 0;
    while (first) {
        s += first->data;
        first = first->next;
    }
    return s;
}
```

Przykład uogólniania (abstrakcyjna struktura danych)

```
// pseudo-code  for a more general version of  both algorithms

int sum(data)              // somehow parameterize with the data structure
{
    int s = 0;                      // initialize
    while (not at end) {            // loop through all elements
        s = s + get value;          // compute sum
        get next data element;
    }
    return s;                       // return result
}
```

- Potrzebujemy trzech operacji na strukturze danych:
  - sprawdzenie czy doszliśmy do końca (***not at end***)
  - pobieranie wartości (***get value***)
  - pobieranie kolejnego elementu (***get next data element***)

Przykład uogólniania (wersja STL)

// Concrete STL-style code for a more general version of both algorithms

```
template<class Iter, class T>        // Iter should be an Input_iterator
                                     // T should be something we can + and =
T sum(Iter first, Iter last, T s)    // T is the "accumulator type"
{
    while (first!=last) {
        s = s + *first;
        ++first;
    }
    return s;
}
```

- Niech użytkownik inicjuje akumulator:

```
float a[] = { 1,2,3,4,5,6,7,8 };
double d = 0;
d = sum(a,a+sizeof(a)/sizeof(*a),d);
```

Standardowa biblioteka szablonów
***S**tandard **T**emplate **L**ibrary*

- Część biblioteki standardowej C++
- Główny cel
  - najbardziej ogólna, wydajna i elastyczna reprezentacja koncepcji (idei, algorytmów)
  - oddzielne koncepcje oddzielnie w kodzie
  - dowolne łączenie koncepcji tam gdzie ma to sens
- Aby programowanie było jak matematyka
  - w myśl zasady: „dobre programowanie jest matematyką”
  - twórcą jest Rosjanin, Aleksander Stiepanow

STL

- ok. 10 kontenerów i 60 algorytmów

		Sequence containers					Associative containers				Unordered associative containers				Container adaptors		
Headers		<array>	<vector>	<deque>	<forward_list>	<list>	<set>		<map>		<unordered_set>		<unordered_map>		<stack>	<queue>	
		array	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
	(constructor)	(implicit)	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
	(destructor)	(implicit)	~vector	~deque	~forward_list	~list	~set	~multiset	~map	~multimap	~unordered_set	~unordered_multiset	~unordered_map	~unordered_multimap	~stack	~queue	~priority_queue
	operator=	(implicit)	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=
	assign	N/A	assign	assign	assign	assign	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Iterators	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	N/A	N/A	N/A
	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin			
	end	end	end	end	end	end	end	end	end	end	end	end	end	end	N/A	N/A	N/A
	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend			
	rbegin	rbegin	rbegin	rbegin	N/A	rbegin	rbegin	rbegin	rbegin	rbegin	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	crbegin	crbegin	crbegin	crbegin		crbegin	crbegin	crbegin	crbegin	crbegin							
	rend	rend	rend	rend	N/A	rend	rend	rend	rend	rend	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	crend	crend	crend	crend		crend	crend	crend	crend	crend							

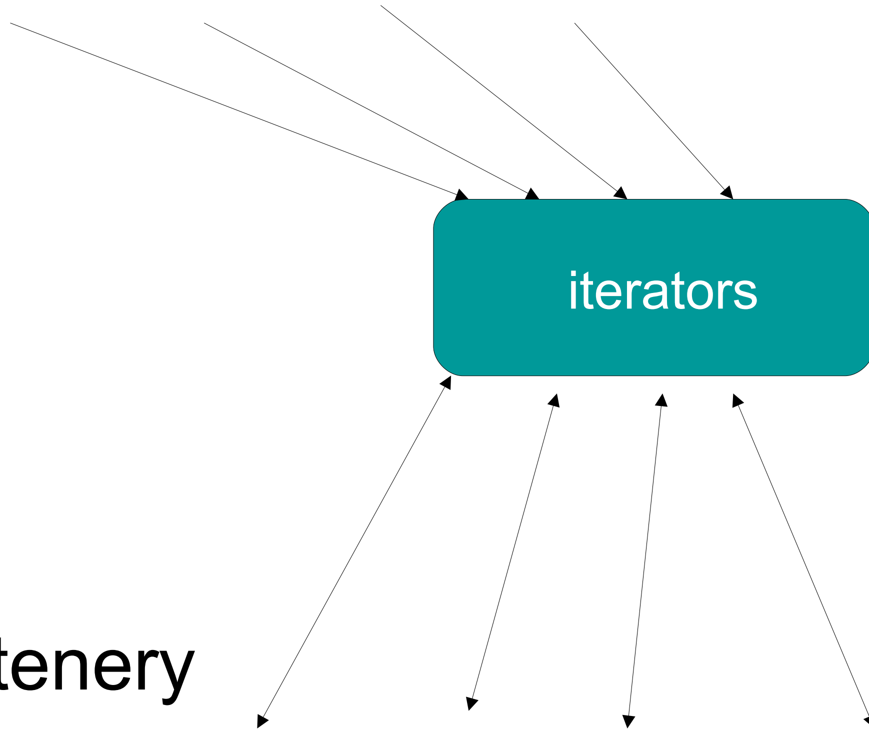
http://en.cppreference.com/w/cpp/container

- plus dodatkowe kontenery i algorytmy
  - Boost.org, Microsoft, SGI, …
- dokumentacja:
  - BOOST: http://www.boost.org/
  - Microsoft: http://msdn.microsoft.com/en-us/library/y097fkab.aspx
  - SGI: http://www.sgi.com/tech/stl/ (jasny opis)

Podstawowy model STL

- Algorytmy

sort, find, search, copy, …



- Kontenery

vector, list, map, hash_map, …

Separacja

- **Algorytmy** operują na danych, ale nie wiedzą nic o kontenerach
- **Kontenery** przechowują dane, ale nie wiedzą nic o algorytmach
- Algorytmy i kontenery współpracują poprzez **iteratory**
  - Każdy kontener ma swoje własne typy iteratorów

Iteratory

- Para iteratorów definiuje sekwencję
  - Początek (wskazuje na pierwszy element – jeśli istnieje)
  - Koniec (wskazuje na element za ostatnim elementem)

Iterator jest typem, który udostępnia operacje iteratorowe:

++ idź do następnego elementu

* pobierz wartość

== sprawdź czy dwa iteratory wskazują na ten sam element

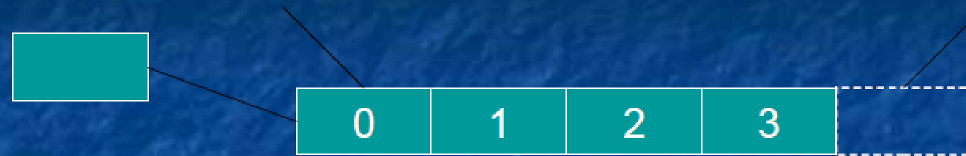
Niektóre iteratory udostępniają więcej operacji (np. --, + i [ ])

Containers

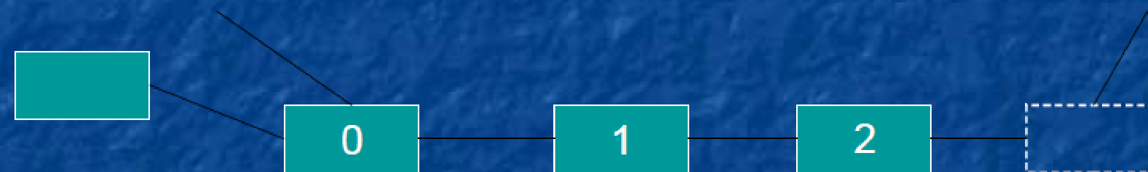
(hold sequences in difference ways)

Parasol
Smarter computing.
Texas A&M University

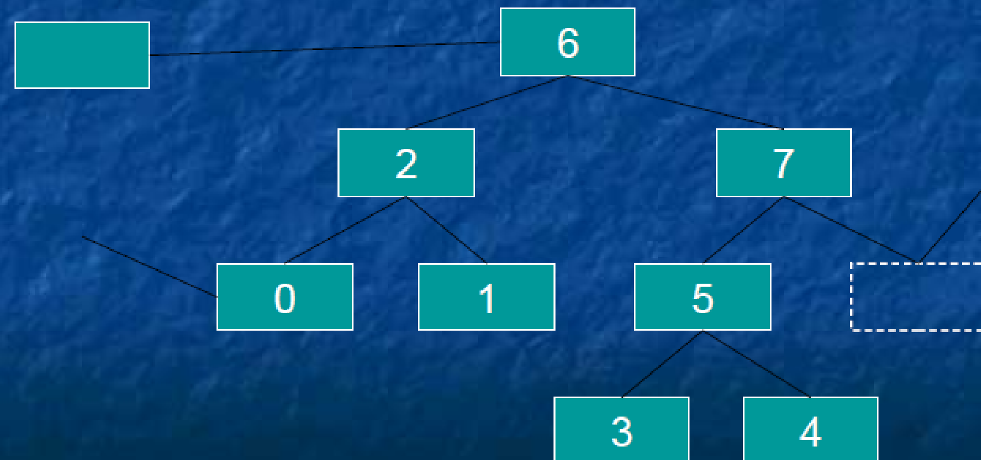
- **vector**



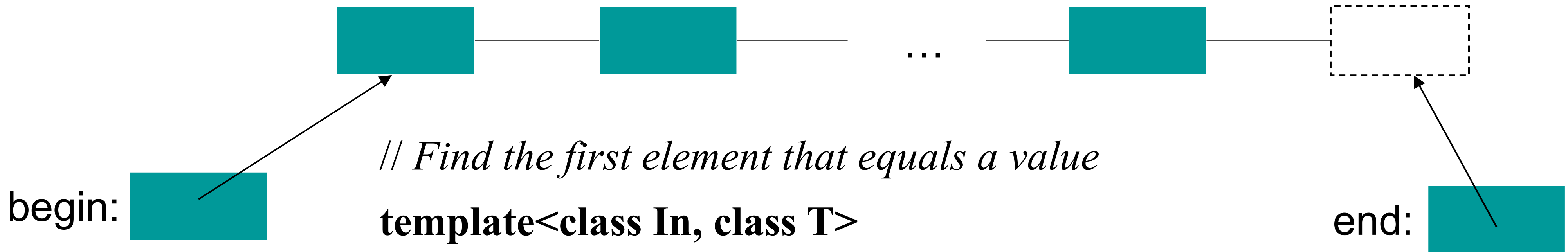
- **list**
  (doubly linked)



- **set**
  (a kind of tree)



Najprostszy algorytm: **find()**




```
// Find the first element that equals a value
template<class In, class T>
In find(In first, In last, const T& val)
{
    while (first!=last && *first != val) ++first;
    return first;
}

void f(vector<int>& v, int x)        // find an int in a vector
{
    vector<int>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // …
}
```

Możemy pominąć różnice pomiędzy kontenerami

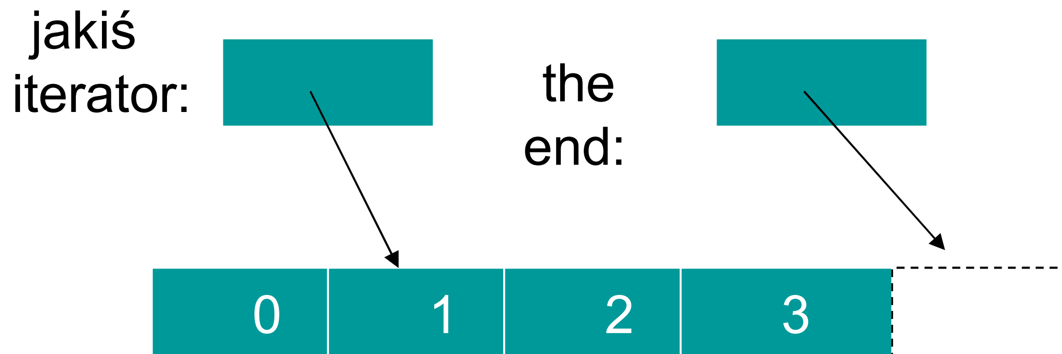
find()

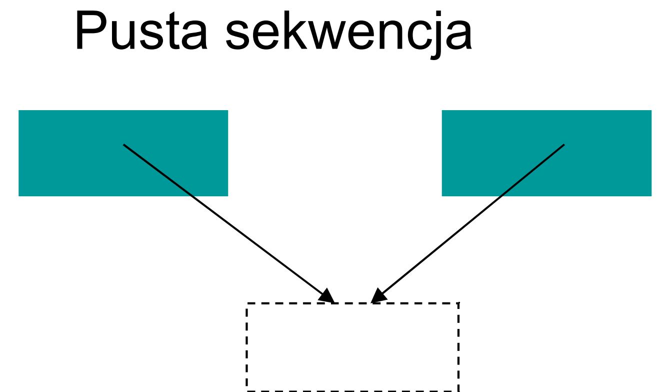
ogólny zarówno dla typu elementów jak i kontenera

```
void f(vector<int>& v, int x)                    // works for vector of ints
{
    vector<int>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // …
}

void f(list<string>& v, string x)                // works for list of strings
{
    list<string>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // …
}

void f(set<double>& v, double x)                 // works of set of doubles
{
    set<double>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // …
}
```

Algorytmy i iteratory

- Iterator wskazuje (odnosi się do, oznacza) element w sekwencji
- Koniec sekwencji to jeden za ostatnim elementem
  - ***nie*** "ostatni element"
  - konieczne by elegancko reprezentować pustą sekwencję
  - jeden-za-ostatnim-elementem nie jest elementem
    - możesz porównać iterator odnoszący się do niego
    - nie możesz wyłuskać go (przeczytać jego wartości)
- Zwrócenie końca sekwencji oznacza nieznalezienie lub niepowodzenie operacji





Dziś najważniejsze było...

- Szablony funkcji i klas – pojęcia:
  - parametryzacja, konkretyzacja, specjalizacja
- Programowanie ogólne
  - piękno matematyki w programowaniu
- Używaj bibliotek opartych na szablonach
  - np. STL
- Iteratory

Kolokwium

- I termin
  - poniedziałek, 19/11/2012 (w terminie wykładu)
    - osoby o nazwiskach na A-Ł, godz. 13.15
    - osoby o nazwiskach na M-Ż, godz. 14.15
- II termin
  - poniedziałek, 26/11/2012 (w terminie wykładu)
    - wszyscy godz. 13.15