Algorytmy i struktury danych

Wykład 12

Krzysztof M. Ocetkiewicz
Krzysztof.Ocetkiewicz@eti.pg.gda.pl

Katedra Algorytmów i Modelowania Systemów, WETI, PG

Sortowanie w C

- `stdlib.h`
- funkcja `qsort(tablica, ile, rozmiar, funkcja)`
  - tablica — wskaźnik na tablicę elementów do posortowania
  - ile — ilość elementów do posortowania
  - rozmiar — rozmiar pojedynczego sortowanego elementu
  - funkcja — funkcja porównująca dwa elementy
  - sortowanie szybkie

Sortowanie w C

```
#include<stdlib.h>

...
int porownaj(const void *a, const void *b) {
     return *(const int *)a - *(const int *)b;
     };

...
int tab[] = {6, 5, 4, 3, 2, 1};
qsort(tab, 6, sizeof(int), porownaj);
```

Wyszukiwanie w C

- `stdlib.h`
- funkcja `bsearch(klucz, tablica, ile, rozmiar, funkcja)`
  - klucz — wskaźnik na wyszukiwany klucz
  - tablica — wskaźnik na tablicę posortowanych elementów
  - ile — ilość elementów w tablicy
  - rozmiar — rozmiar pojedynczego elementu
  - funkcja — funkcja porównująca dwa elementy (jak w qsort)
  - wyszukiwanie binarne
  - wartość zwracana — wskaźnik na znaleziony element (dowolny) lub NULL

STL

- Standard Template Library
- biblioteka standardowa w języku C++
- http://cppreference.com — bardzo dobra strona z dokumentacją
- http://cplusplus.com/reference
- "magiczna" — znajomość kilku prostych recept umożliwia korzystanie nawet początkującym, jednak zrozumienie pisanego kodu wymaga głębszej znajomości języka C++ (szablony)

STL

- pliki nagłówkowe nie mają rozszerzenia .h
  - np. `#include<vector>`
  - np. `#include<algorithm>`
- zaraz za include'ami: `using namespace std;`
  - inaczej wszystkie nazwy trzeba poprzedzać `std::`
  - np. `std::sort` zamiast `sort`

STL – podstawowe koncepcje

- kontenery
- iteratory
- algorytmy

Kontenery

- kontenery to implementacje struktur danych (= struktury danych)
- udostępniają pewne operacje o określonej złożoności
- kontenery sparametryzowane są przechowywanym typem
  - np. `vector<int>` — kontener vector przechowujący liczby typu int
  - np. `vector<Data>` — kontener vector przechowujący struktury Data
- uwaga: samo vector nie jest nazwą typu danych; nie możemy napisać np. `void funkcja(vector v)`
- dopiero kontener sparametryzowany przechowywanym typem jest pełnoprawnym typem: `void funkcja(vector<int> v)` jest poprawne

Iteratory

- ograniczone “wskaźniki” na elementy wewnątrz kontenerów
- obsługują dereferencję (*) i operator -> tak jak wskaźniki
- iteratory można porównywać (==, !=)
- iteratory jednokierunkowe: obsługują dodatkowo operatory ++
- iteratory dwukierunkowe: obsługują dodatkowo operatory --
- iteratory o dostępie swobodnym: obsługują dodatkowo operatory +, -(liczba i iterator), +=, -=, <, []

Iteratory

- większość kontenerów posiada metody `begin()` i `end()`
- `begin()` zwraca iterator na pierwszy element w kontenerze
- `end()` zwraca iterator pokazujący za ostatnim elementem w kontenerze (`--end()` jest ostatnim elementem w kontenerze)
- iteratory zadeklarowane są "wewnątrz" kontenerów; aby stworzyć iterator do wektora int-ów o nazwie *nazwa* musimy napisać:
  `vector<int>::iterator nazwa;`

Algorytmy

- konwencją w STL jest przekazywanie danych nie w postaci kontenera, lecz jako pary iteratorów: pierwszy pokazuje na początek przedziału, drugi na pozycję tuż za ostatnim elementem w przekazywanym przedziale
- np. mając `vector<int> v;` sortujemy go pisząc `sort(v.begin(), v.end())`
- niektóre algorytmy mogą mieć dodatkowe wymagania co do elementów składowych kontenerów (np. powinny być porównywalne operatorem <, czy ==)

Wskaźniki jako iteratory

- z punktu widzenia STL wskaźniki są także iteratorami (w kontenerach, którymi są zwykłe tablice)
- np. całkowicie poprawny jest kod:

```
int tab[6] = {6, 5, 4, 3, 2, 1};
sort(tab, tab + 6);
```

vector

- plik nagłówkowy: `#include<vector>`
- implementacja tablicy dynamicznej (ze zamortyzowanym kosztem wstawienia $O(1)$)
- zachowuje się podobnie jak tablica (możemy używać indeksowania)

vector

- ważniejsze metody:
  - `vector<int> v;`
  - `v[index]` — dostęp do elementu na pozycji index
  - `v.at(index)` — dostęp do elementu na pozycji index + wyjątek przy błędnym indeksie
  - `v.size()` — zwraca rozmiar wektora
  - `v.empty()` — zwraca true jeżeli wektor jest pusty
  - `v.push_back(wartosc)` — dodanie na koniec wektora wartości *wartosc* ($O(1)$)
  - `v.pop_back()` — usunięcie ostatniej wartości z wektora ($O(1)$)
  - `v.front()` — pierwszy element (to samo co `v[0]`)
  - `v.back()` — ostatni element (to samo co `v[v.size()-1]`)

vector

- ważniejsze metody (c.d.):
  - `v.resize(int n)` — ustawienie rozmiaru wektora na n
  - `v.reserve(int n)` — ustawienie pojemności wektora na co najmniej n

Iteratory w vector

- iteratory o dostępie swobodnym
- dodanie lub usunięcie elementu wektora może spowodować, że iteratory przestaną być poprawne (np. gdy nastąpi realokacja lub gdy usuwamy element ze środka wektora)
- jeżeli zależy nam na zachowaniu poprawności iteratorów, musimy skorzystać z `resize` / `reserve`

Rozmiar a pojemność

- tak jak w przypadku zwykłej tablicy dynamicznej, rozmiar wektora to nie to samo co jego pojemność
- rozmiar to liczba elementów rzeczywiście przechowywanych w tablicy
- pojemność to maksymalny rozmiar, który wektor może osiągnąć bez ponownej alokacji
- rozmiar nigdy nie jest większy niż pojemność

vector

```
vector<int> v; // utworzenie pustego wektora
v[2] = 4; // BŁĄD -- wektor nie ma elementów
v.push_back(3);
v.push_back(4);
v.push_back(5);
v[2] = 7; // dobrze -- wektor ma 3 elementy
v.back() == 7; // prawda
v.pop_back();
v.back() == 4; // prawda
v.reserve(100);
v[99] = 4; // BŁĄD -- wektor ciągle ma 2 elementy
v.resize(100);
v[99] = 4; // dobrze
```

vector

```
vector<int> v(100); // wektor 100-elementowy
v[2] = 4; // dobrze -- wektor ma rozmiar 100
v.push_back(3); // teraz ma 101 elementów
v.push_back(4); // teraz ma 102 elementy
v.push_back(5); // teraz ma 103 elementy
v.reserve(4);
v[99] = 4; // dobrze -- reserve nic nie zrobiło
v.resize(4);
v[99] = 4; // ŹLE -- wektor ma tylko 4 elementy
```

vector

```
// iteracja
for(i = 0; i < v.size(); i++) {
      ...  v[i] ...
      };
// iteracja STL-owa
vector<int>::iterator it;
for(it = v.begin(); it != v.end(); ++it) {
      ...  *it ...
      };
// iteracja STL-owa
vector<int>::iterator it, kon = v.end();
for(it = v.begin(); it != kon; ++it) {
      ...  *it ...
      };
```

vector

```
for(auto it = v.begin(); it != v.end(); ++it) {
      ...  *it ...
      };
for(const &elem:  v) {
      ...  elem ...
      };
```

deque

- plik nagłówkowy: `#include<deque>`
- kontener bardzo podobny do vectora
- nieznacznie wolniejszy od vectora
- obsługuje dodatkowo wstawianie i usuwanie elementu na początku w zamortyzowanym czasie $O(1)$
- nie posiada metod `reserve`, `resize`
- wstawienie/usunięcie elementu może unieważnić iteratory

list

- plik nagłówkowy: `#include<list>`
- lista dwukierunkowa / jednokierunkowa
- iteratory dwukierunkowe
- brak indeksowania
- wstawienie w dowolne, wskazane miejsce zajmuje czas stały
- wstawianie / usuwanie elementów nie unieważnia iteratorów (poza tymi wskazującymi na elementy usuwane)

list

- `splice` — przenoszenie fragmentów listy do innej (w czasie stałym)
- `size` może mieć złożoność $O(n)$, do testowania czy pusta należy użyć `empty`

forward_list

- plik nagłówkowy: `#include<forward_list>`
- lista jednokierunkowa
- iteratory jednokierunkowe
- brak indeksowania
- wstawienie na początek, lub za wskazany element zajmuje czas stały
- wstawianie / usuwanie elementów nie unieważnia iteratorów (poza tymi wskazującymi na elementy usuwane)

forward_list

- `splice_after` — przenoszenie fragmentów listy do innej (w czasie stałym)
- `size` może mieć złożoność $O(n)$, do testowania czy pusta należy użyć `empty`

stack

- plik nagłówkowy: `#include<stack>`
- stos
- `s.push(element)` — włożenie elementu na stos
- `s.top()` — element na wierzchołku stosu
- `s.pop()` — zdjęcie elementu z wierzchołka stosu (uwaga: nie zwraca zdejmowanego elementu)
- `s.empty()` — zwraca true, jeżeli stos jest pusty
- `s.size()` — liczba elementów na stosie
- nie posiada iteratorów

queue

- plik nagłówkowy: `#include<queue>`
- kolejka
- `s.push(element)` — włożenie elementu na koniec kolejki
- `s.front()` — element na początku kolejki
- `s.back()` — element na końcu kolejki
- `s.pop()` — wyjęcie elementu z początku kolejki
  (uwaga: nie zwraca zdejmowanego elementu)
- `s.empty()` — zwraca true, jeżeli kolejka jest pusta
- `s.size()` — liczba elementów w kolejce
- nie posiada iteratorów

find

- plik nagłówkowy: `#include<algorithm>`
- `find(poczatek, koniec, wartosc)`
- `poczatek` i `koniec` to iteratory określające przedział do przeszukania
- `wartosc` to wartość do znalezienia
- zwraca pozycję (iterator) wartości `wartosc` w przedziale `< poczatek, koniec )`
- jeżeli *wartosc* nie została znaleziona, zwraca `koniec`
- wymaga aby elementy wskazywane przez iteratory były porównywalne (==)

find — przykład

```
int tab[] = {1, 2, 3, 4, 5, 6};
int *p = find(tab, tab + 6, 4);
printf(``*p=%d p-tab=%d\n'', *p, p - tab);
```

```
*p=4 p-tab=3
```

```
p = find(tab, tab + 6, 10);
printf(``p-tab=%d\n'', p - tab);
```

```
p-tab=6
```

find — przykład

```
int tab[] = {1, 2, 3, 4, 5, 6};
vector<int> v(tab, tab + 6);
vector<int>::iterator it;
it = find(v.begin(), v.end(), 4);
printf(``*it=%d it-v.begin()=%d\n'',
       *it, it - v.begin());
```

```
*it=4 it-v.begin()=3
```

```
it = find(v.begin(), v.end(), 10);
printf(``it-v.begin()=%d\n'', it - v.begin());
if(it == v.end()) printf(``nie znaleziono\n'');
```

```
it-v.begin()=6
nie znaleziono
```

binary_search

- plik nagłówkowy: `#include<algorithm>`
- `binary_search(poczatek, koniec, wartosc)`
- *poczatek* i *koniec* to iteratory określające przedział do przeszukania
- *wartosc* to wartość do znalezienia
- zwraca `true` jeżeli znaleziono wartość, `false` jeżeli nie znaleziono
- `upper_bound(poczatek, koniec, wartosc)` = pierwszy niemniejszy
- `lower_bound(poczatek, koniec, wartosc)` = pierwszy większy

Sortowanie

- `#include<algorithm>`
- `sort` i `stable_sort`
- `sort(poczatek, koniec)`
  - sortuje porównując elementy operatorem `<`
- `sort(poczatek, koniec, porownanie)`
  - sortuje porównując elementy funkcją `porownanie`

Sortowanie

```
#include<algorithm>

...
bool porownaj(int a, int b) {
     return a > b;
     };

...
int tab[] = {6, 5, 4, 3, 2, 1};
sort(tab, tab + 6);
```

```
1 2 3 4 5 6
```

```
sort(tab, tab + 6, porownaj);
```

```
6 5 4 3 2 1
```

Drzewa w STL

- `map`, `set` — drzewa czerwono-czarne
- struktura drzewa jest "niejawna" — nie mamy możliwości dostępu do poszczególnych węzłów, a co za tym idzie niemożliwe jest rozbudowanie drzewa (np. do drzewa statystyk pozycyjnych)
- `map` — pamięć skojarzeniowa; z kluczami powiązane są dane
- `set` — zbiór; klucze nie są powiązane z danymi
- `map`, `set` — klucze są unikatowe
- `multimap`, `multiset` — klucze mogą się powtarzać
- kontenery te przechowują klucze w uporządkowanej kolejności (porządek wyznacza operator $<$)

Drzewa w STL

- iteratory w mapie pokazują na pary (*klucz*, *wartosc*)
- dla `map<K, V>::iterator it`
  - klucz jest dostępny jako `it->first` (lub `(*it).first`)
  - wartość jest dostępna jako `it->second` (lub `(*it).second`)
- iteratory w zbiorze pokazują na klucze (wartości nie ma)

map

```
#include<map>
#include<string>
using namespace std;
int main() {
     map<string,int> mp;
     mp[''xyz''] = 1; mp[''abc''] = 2;
     mp[''def''] = 3;
     for(map<string,int>::iterator it = mp.begin();
          it != mp.end(); ++it) {
          cout << it->first.c_str() << '':'';
          cout << it->second << '', '';
          };
          return 0;
     };
```

```
abc:2, def:3, xyz:1,
```

map

- jeżeli klucz nie istnieje, operator [] go wstawi
- w przypadku `mp[''klucz'']` `= wartosc` jest to działanie pożądane, jednak w sytuacji `if(mp[''klucz''] ... )` nie zawsze
- do testowania istnienia klucza w mapie należy użyć metody `find`

map

- jako argument do `find` podajemy klucz, zwróconą wartością jest iterator na parę (*klucz*, *wartosc*) (o ile istnieje) np.:

```
map<string, int>::iterator it;
it = mp.find(''abc'');
if(it == mp.end()) cout << ''nie ma klucza'';
else {
     cout << it->first << ''='';
     cout << it->second << endl;
     };
```

map

- należy pamiętać, że każde indeksowanie czy `find` to wyszukanie klucza — w drzewie zrównoważonym zajmuje to $O(\log n)$
- w przypadku wykonywania wielu operacji czas ten może stać się zauważalny
- nie należy zatem stosować konstrukcji typu:

```
if(mp.find(''abc'') == mp.end())
      cout << ''nie ma klucza'';
else cout << ''abc'' << ''='' << mp[''abc''] << endl;
```

Iteratory

- iteratory w mapie i zbiorze są dwukierunkowe
- niech `it = mp.find(''def'');`
- po wykonaniu `--it`, `it->first` będzie poprzednim kluczem
- po wykonaniu `++it`, `it->first` będzie następnym kluczem

Tablice haszowane

- `unordered_map`
- użycie jak mapy, z pewnymi wyjątkami
  - elementy nie są uporządkowane
  - iteratory tylko jednokierunkowe
  - dla własnych typów trzeba napisać funkcję haszującą

string

- klasa opakowująca napis (tablicę znaków z określonym rozmiarem)
- obsługuje operatory przypisania, porównania, dodania (znaku, napisu i string-a), indeksowania (odczyt/modyfikacja konkretnego znaku)
- `size()` – długość
- `c_str()` – napis kompatybilny z C (tylko do odczytu) np. `printf(''%s'', s.c_str());`

Referencje

- korzystając z STL trzeba pamiętać o referencjach
- referencje deklarujemy korzystając ze znaku &
  (podobnie, jak z * w przypadku wskaźników)
- referencje należy zainicjować w trakcie deklaracji
- formalnie: referencja jest drugą nazwą dla zmiennej, np.
  ```
  int a;
  int &b = a;
  ```
  od tej pory a i b oznacza dokładnie to samo
- referencja nie ma "osobowości" – nie można utworzyć wskaźnika czy referencji na referencję; pobranie adresu pobierze adres "oryginalnej" zmiennej

Referencje

- parametry funkcji mogą być referencjami, np.:

```
void zamien(int &a, int &b) {
      int t = a; a = b; b = t;
      };
```

- referencja musi być związana ze zmienną lub elementem tablicy, nie może być zainicjalizowana stałą:

```
int &b = 5; // błąd
zamien(5, 4); // błąd
```

Referencje

- stałe referencje (`const int &b = a;`) zezwalają na odczyt zawartości, zabraniają modyfikacji
- dobrą praktyką programistyczną jest oznaczanie jako stałe wszystkich parametrów funkcji, które są referencjami i nie są modyfikowane w funkcji

Uwaga

- Od dzisiaj nie przekazujemy stringów ani innych kontenerów STL do funkcji przez wartość (aż do zrozumienia i opanowania semantyki przeniesienia).
- Każdy parametr, który jest referencją jest stałą referencją, chyba, że trzeba zmodyfikować zawartość parametru.

Dlaczego?

- przekazanie przez wartość (`void funkcja(string s)`) kopiuje wartość parametru, czyli:
  - tworzy nowy obiekt (parametr)
  - przepisuje zawartość argumentu do parametru
  - na koniec funkcji usuwa parametr
- przekazanie przez referencję (`void funkcja(const string &s)`) sprowadza się do przekazania wskaźnika, czyli:
  - tworzy nowy wskaźnik (parametr)
  - przepisuje adres argumentu do parametru
  - na koniec funkcji usuwa wskaźnik

Niebezpieczeństwa STL

- przekazywanie przez wartość
- komunikaty o błędach
- trzeba pamiętać o tym, co mówiliśmy na wykładzie
- `vect.push_back(vect[0]);`
-
```
list<int>::iterator it;
for(it = lista.begin(); it != lista.end(); ++it) {
      if(*it == 4) lista.erase(it);
      };
```