Sveučilište u Zagrebu
PMF – Matematički odjel

Objektno programiranje (C++)

Vježbe 01 – STL

Vinko Petričević

Općenito

- Na ovim vježbama ćemo se na brzinu prisjetiti osnova STL-a (koje su obrađene na kolegiju Računarski praktikum 1)
- Sve klase i objekti u STL smješteni su u *namespace std*. Prilikom njihovog korištenja uvijek moramo navesti std::objekt
- Ili na početku programa možemo napisati using namesapce std ili using std::string, ako želimo koristiti samo string
- Ukoliko smo koristili using namespace std, a u našem programu imamo objekt/funkciju koji se jednako zove, onda moramo naglasiti std::funkcija ili ::funkcija da ne bi dobili dvosmislenost.
- Na ovim vježbama ćemo pretpostavljati upotrebu usinga (da nepotrebno ne pišemo puno puta std::), dok na sljedećima uglavnom nećemo
- Svi spremnici rastu dinamički i sami se brinu o alokaciji potrebne memorije (o dealokaciji u nekim specijalnim slučajevima na nekim drugim vježbama)

- Za detalje pogledati: www.cplusplus.com/reference/

Tip `std::string`

- header datoteka `<string>`
- niz znakova varijabilne duljine koji ima samostalni memory management
- dovoljno efikasan za generalnu upotrebu, a ima mnoge korisne operacije
- Konstruktori:
  - `std::string s1;`
    - defaultni konstruktor – `s1` je prazan string
  - `std::string s2(s1);`
    - inicijalizacija stringa `s2` s kopijom od `s1`
  - `std::string s3("ABC");`
    - inicijalizacija stringa `s3` s kopijom string literala
  - `std::string s4(n, 'c');`
    - inicijalizacija stringa `s4` s n kopija znaka `'c'`
- String možemo istovremeno definirati i inicijalizirati na slijedeći način:
  `string s = "xyz";`
  Pitanje: Što ovakav izraz zapravo predstavlja?
  - `string s("xyz"); // dozvoljena optimizacija`
  - `string s(string("xyz")); // standard`

Podsjetnik: ulaz i izlaz

- Header datoteka: `<iostream>`
- `std::cout` - objekt klase ostream
- `std::cin` - objekt klase istream
- Primjer:

```
int i;
std::string s;
std::cin >> i;
std::cin >> s;
std::cout << "string: " << s << std::endl
          << "broj: " << i << std::endl;
```

- ignoriraju se vodeće bjeline
- učitava se niz znakova do prve sljedeće bjeline

Čitanje i pisanje stringova

- Primjer:
  - ```
    string s1, s2;
    cin >> s1 >> s2; // pročitaj prvo s1, zatim s2
    cout << s1 << s2 << endl;
    ```
- Čitanje nepoznatog broja stringova:
  - ```
    string word;
    // čitaj sve do EOF-a
    while (cin >> word)
        cout << word << endl;
    return 0;
    ```
- Čitanje cijele linije:
  - ```
    string line;
    // čitaj liniju po liniju sve do EOF-a
    while (getline(cin, line))
        cout << line << endl;
    ```

prazna linija u kojoj stisnemo:
pod Linuxom Ctrl-D,
a pod Windowsim Ctrl-Z

stanje streama kao logički uvjet

getline() učitava liniju teksta s danog ulaznog streama (sve do prelaska u novi red koji se odbacuje) u dani string

std::istream &std::getline(std::istream&, std::string&);
    ```

Operacije na stringovima

- Duljina stringa dobiva se pomoću funkcije `size()`:

```
string s("OP");
cout << "Duljina od " << s
 << " je " << s.size() << " znakova.";
```

- Napomena: `size()` vraća `string::size_type`
- Provjera da li je string prazan:

```
string s2;
if (st.size() == 0)
    // ok: prazan string
if (s2.empty())
    // ok: prazan string
```

- Relacijski operatori:
  -

```
string big = "big", small = "small";
string s1 = big;  // s1 je kopija od big
if (big == small) // neistina
    // ...
if (big <= s1)    // istina
    // ...
```

- Pridruživanje – kopiranje jednog stringa u drugi:

```
s1 = small; // kopira string small u s1
```

Operacije na stringovima

- Konkatenacija stringova:

```
string s1("hello");
string s2("world");
string s3 = s1 + ", " + s2 + "\n";
s1 += s2;
```

-
```
string spoji1(string a, string b) {
    return a + "." + b;
}
```

-
```
string spoji2(string a, string b) {
    string ret(a.size()+1+b.size(),'.');
    ret.replace(0, a.size(), a);
    // copy(b.begin(), b.end(), ret.begin()+a.size()+1);
    copy(b.begin(), b.end(), &ret[a.size()+1]);
    return ret;
}
```

Operacije na stringovima

- Pitanje: Uz pretpostavku da su s i t stringovi, koji od slijedećih izraza su legalni?
  - `s = t + "baz";`
  - `s = "baz" + t;`
  - `s = t + "baz" + "bop";`
  - `s = "baz" + t + "bop";`
  - `s = "baz" + "bop" + t;`

    (u svakoj konkatenaciji mora sudjelovati barem jedan objekt tipa string, pa `"baz" + "bop"` predstavlja grešku)

Operacije na stringovima

- Operator [] – pristup individualnim znakovima u stringu
  - ```
    string str("neki string");
    for (string::size_type i = 0; i != str.size(); ++i)
        cout << str[i] << endl;
    ```
- Operator [] vraća lvalue, tj. `str[i]` je tipa `char&`:
  - ```
    for (string::size_type i = 0; i != str.size(); ++i)
        str[i] = '*';
    ```
  - ipak, ako bi `str` bio definiran kao const objekt:
    ```
    const std::string str;
    ```
    `str[i]` će biti tipa `const char&`, pa tada nije dozvoljeno pridruživanje!
- Tip `string` i C-stil stringovi:
  ```
  std::string s;
  const char *pc = "polje znakova";
  s = pc; // ok
  char *str = s; // greška
  ```
- Funkcija `c_str()` vraća reprezentaciju stringa u C-stilu:
  ```
  char *str = s1.c_str(); // greška
  const char *str = s1.c_str(); // ok
  ```

Operacije na stringovima

- Vraćanje podstringa: `substr(pocetak, duljina)`
  - ```
    string A("Nesto"), B;
    B = A.substr(3, 2); // B="to"
    ```
- Traženje podstringa: find(stoTrazim, gdjePocinjem), vraća mjesto
  - ```
    string A("kokodako");
    int gdje = A.find("ko", 1);
    // traži "ko" počevši od 1. mjesta (ne 0.)
    // gdje=2 jer se "ko" kao podstring
    // prvi put javlja na 2.mjestu
    ```
- Ako `find()` ne uspije naći podstring, vraća `string::npos`
  - ```
    string S("kokoda"), T("kokos");
    int gdje = S.find(T, 0);
    if (gdje == string::npos)
        cout << "nema ga";
    ```
- Brisanje podstringa: `erase(pocetak, koliko)`
  - ```
    string S("nestodrugo");
    S.erase(2, 6); // sad je S="nego"
    ```

std::vector

- Generički kontejner objekata istog tipa koji predstavlja alternativu C++ poljima, ali može mijenjati veličinu
- `#include <vector>`
- Primjer:

```
vector<int> a(10);
```

Operacije nad vektorom od 10 `int`-ova korespondiraju operacijama nad poljem od 10 `int`-ova:

```
int a[10];
```

- ```
  for (int i = 0; i < a.size(); ++i)
      cout << a[i] << ' ';
  ```
- Napomena: vector kao takav nije tip podatka, već *predložak* za generiranje različitih tipova:
  - `vector<int>`
  - `vector<string>`
  - `vector<vector<double> >`
- Napomena: Kod konstruktora oblika

```
vector<T> v(n);
```

tip `T` mora biti default konstruktibilan:

- primitivni tip
- korisnički tip s defaultnim konstruktorom

(stoga `T` npr. ne može biti tip reference)
```

std::vector

- Konstruktori:
  - `vector<T> v1;`
    - defaultni konstruktor – `v1` je prazan vektor (s 0 elemenata)
  - `vector<T> v2(v1);`
    - `v2` sadrži kopije elemenata od `v1` (`v1` i `v2` moraju biti istog tipa)
  - `vector<T> v3(n, i);`
    - `v3` sadrži `n` elemenata, svaki od kojih je inicijaliziran kopijom vrijednosti `i`
  - `vector<T> v4(n);`
    - `v4` sadrži `n` elemenata, svaki od kojih je defaultno konstruiran
- Pridruživanje `v1 = v2`
  - pridružuje vektoru `v1` kopije elemenata iz `v2` (tipovi vektora `v1` i `v2` moraju biti identični)
- Uspoređivanja `==`, `!=`, `<`, `<=`, `>` i `>=`
  - svi relacijski operatori definirani tako da vektore uspoređuju leksikografski (analogno kao kod tipa string)

Operacije na vektorima

- `v.empty()`
  - vraća `true` ako je `v` prazan; inače vraća `false`
- `v.size()`
  - vraća broj elemenata u vektoru `v`
- `v.clear()`
  - brisanje svih elemenata vektora `v` `(memorija ipak neće biti oslobođena)`
- `v[n]`
  - vraća element na poziciji `n` u vektoru `v`
  - povratni tip je `T&` (ili `const T&` ako je vektor konstantan)
- `v.push_back(t)`
  - dodaje kopiju od `t` kao novi element na kraj vektora i povećava mu veličinu za 1 (može implicirati alokaciju memorije)
  - amortizirano konstantno vrijeme izvršavanja
- `v.pop_back()`
  - izbacuje element s kraja vektora

Primjeri

- Čitanje stringova sa standardnog ulaza ubacujući pritom jedan po jedan u vector:

```
vector<string> text;
string word;
while (cin >> word) {
        text.push_back(word);
        // ...
}
```

- Iteriranje kroz elemente pomoću operatora `[]`:

```
cout << "procitane rijeci:" << endl;
for (int i = 0; i < text.size(); ++i)
    cout << text[i] << ' ';
cout << endl;
```

- Iteriranje kroz elemente pomoću iteratora:

```
cout << "procitane rijeci:\n";
for(vector<string>::iterator it=text.begin();
it != text.end(); ++it)
    cout << *it << ' ';
cout << endl;
```

-
```
vector<int> v(10);
for (vector<int>::iterator iter = v.begin();
    iter != v.end(); ++iter)
        std::cin >> *iter;
```

Iteratori

- Iteratori su tipovi pridruženi svakom kontejnerskom tipu zasebno i namijenjeni su za pristupanje elementima u redoslijedu podržanom od strane kontejnera (npr. slijedno, obrnutim poretkom, itd.)
- Korištenje iteratorskih objekata sintaktički korespondira korištenju pokazivača
- Motivacija:
  - ```
    int a[10];
    int *const begin = a; // pitanje: sto znaci ova *
    int *const end = a + 10;
    for (int *iter = begin; iter != end; ++iter)
        std::cin >> *iter;
    ```

- `v.begin()` – vraća iterator koji pokazuje na početni element kontejnera
- `v.end()` – vraća iterator koji pokazuje "iza zadnjeg" elementa kontejnera
- `(v.begin() == v.end())` akko `v.empty()`
- `++` (inkrement) iteratora pozicionira ga na slijedeći element kontejnera u danom redoslijedu
- dereferenciranje iteratora (operatori * i ->) vraća referencu na objekt – element kontejnera – na kojeg iterator trenutno pokazuje. Za const_iteratore vraća const &
    ```

const_iterator

- const_iterator je iterator koji dereferenciran vraća const referencu na pripadni element kontejnera
- koristi se kad ne treba mijenjati elemente
  - ```
    for (vector<int>::const_iterator iter = v.begin();
        iter != v.end(); ++iter)
            std::cout << *iter << endl;
    ```
- const_iterator radi i na konstantnim kontejnerima
  - ```
    const vector<int> cv(42);
    vector<int>::const_iterator iter = cv.begin();
    ```
- const_iterator != const iterator
  - ```
    const vector<int>::iterator it1 = v.begin();
    *it1 = 7; // ok
    ++it1; // greska, jer jer je iterator konstantan
    ```
  - ```
    vector<int>::const_iterator it2 = v.begin();
    // greska, jer je s lijeve strane const int&
    *iter = 8;
    ```

Aritmetika iteratora

- Aritmetika iteratora – aritmetika pokazivača (ovo podržavaju iteratori na nekim spremnicima među kojima je i vector)
- `iter + n`
  - vrijednost ovog izraza je novi iterator koji pokazuje na objekt koji je `n` objekata "desno" od *iter
- `iter - n`
  - vrijednost ovog izraza je novi iterator koji pokazuje na objekt koji je `n` objekata "lijevo" od *iter
- `n` je tipa size_type ili difference_type danog kontejnera
  - difference_type je cjelobrojni tip s predznakom i javlja se kao rezultat oduzimanja iteratora (`iter1 - iter2`)
- Vrijedi: `iter1 = iter2 + (iter1 - iter2)`

Napomena: iterator može postati invalidan nakon promjene strukture kontejnera (npr. nakon poziva push_back(), pop_back(), itd. odnosno funkcija koje trebaju raditi realokaciju memorije)

Još konstruktora za vector

- Primjer:

```
#include <vector>
#include <iostream>
using namespace std;

void main() {
  int a[] = {7, 8, 9};

  // konstrukcija vektora pomocu pokazivaca
  vector<int> vi(a, a+3);

  // ili pomocu iteratora
  vector<int> vj(vi.begin(), vi.end());

  for(int i=0;i<vj.size(); ++i)
    cout << vj[i] << " ";
  cout << endl;
}
```

- Od standarda 11: `vector<double> d{1.1,1.2,1.3};`

Zadaci

- Zadatak: Napišite program koji čita stringove sa standardnog ulaza ubacujući pritom jedan po jedan u vektor, te nakon unosa EOF (^Z pod Windowsima, ili ^D pod Linuxom) ispisuje sadržaj dobivenog vektora.

- Zadatak: Napišite program koji čita stringove sa standardnog ulaza sve dok se ne učita EOF, te potom ispisuje koliko se puta pojedini string pojavio na ulazu.

- Napomena: vector<bool> ima drugačiju implementaciju od ostalih tipova

Tipovi apstraktnih spremnika

- Slijedni spremnici (sekvencijalni)
  - služe za čuvanje uređene kolekcije elemenata određenog tipa
  - osnovni tipovi: vector, list, deque, ...
  - adaptirani tipovi: stack, queue, priority_queue

- Asocijativni spremnici
  - pružaju podršku za efikasno pronalaženje elemenata na temelju ključa
  - tipovi: map, multimap, set, multiset, unordered...

Tipovi apstraktnih spremnika

- Spremnici se razlikuju po načinu pristupa elementima i po "cijeni" operacija nad elementima (čitanje, dodavanje, brisanje)
- Spremnici definiraju relativno mali osnovni broj operacija
  - Dio operacija nude svi tipovi spremnika
  - Dio operacija je specifičan za slijedne tj. asocijativne spremnike
  - Dio operacija je specifičan za konkretan spremnik
- Puno više operacija definiraju biblioteke algoritama

Slijedni spremnici (sekvencijalni)

- služe za čuvanje uređene kolekcije elemenata određenog tipa
- elemente dohvaćamo po poziciji (indeksu), npr. a[i]
- osnovni tipovi:
  - vector – "polje": brz pristup pojedinom elementu
  - list – vezana lista: brzo ubacivanje i brisanje
  - deque – red sa "dva kraja" (double-ended queue)

- adaptirani tipovi:
  - stack – stog: LIFO
  - queue – red: FIFO

  } Na temelju deque-a

  - priority_queue – prioritetni red

  } Na temelju vector-a

- Adaptori – prilagođuju slijedni spremnik koji se krije "ispod površine" tako da mu definiraju novo sučelje

Slijedni spremnici

- Zaglavlja:
  - vector – `#include <vector>`
  - list – `#include <list>`
  - deque – `#include <deque>`
  - stack – `#include <stack>`
  - queue – `#include <queue>`

- Definicija spremnika sastoji se od navođenja imena spremnika, te tipa elemenata koje želimo čuvati
  - `vector<string> svec;`
  - `list<int> ilist;`
  - `queue<float> fq;`
  - `deque<double> dd;`
  - `stack<char> cstack;`

Konstrukcija slijednih spremnika

- Inicijalizacija spremnika elementima polja

```
string words[4] = {"abc", "xyz", "foo", "bar"};
vector<string> vwords(words, words+4);
int a[6] = { 0, 1, 2, 3, 4, 5 };
list<int> ilist(a, a+6);
```

- Inicijalizacija spremnika iteratorima

```
vector<int> ivec(ilist.begin(), ilist.end());
list<int> ilist1(ilist.begin(), ilist.end());
// list<int> ilist2(ilist);
```

- Smijemo koristiti i spremnike spremnika

```
vector< list<int> > vli; //vektor liste intova
```

  - Obratite pažnju na razmak između > >
- ...i spremnike spremnikovih spremnika...

```
vector< list< deque< list <char> > > > vldli;
```

Ograničenja na podatke u spremniku

- Tip podataka koji se nalazi u slijednom spremniku mora podržavati pridruživanje (=) i kopiranje
  - Neke operacije nad spremnicima imaju i dodatne zahtjeve
    - Ako tip nije “dovoljno dobar”, moći ćemo napraviti spremnik, ali možda nećemo moći izvršavati sve operacije nad spremnikom
  - Reference ne podržavaju kopiranje, pa ne možemo imati spremnik referenci
  - Ograničenja kod asocijativnih spremnika su još veća (ključ treba imati definiran operator <, ili mu trebamo dodati funkciju za uspoređivanje)

Iteratori

- Operator ++, -- (i prefiksni i postfikni), *, ->, ==, != rade uobičajeno
- na forward_list ne radi --
- Kako doći do unutarnjih elemenata spremnika?
  - Na listi ne radi `L[3]`
- Raspon iteratora (iterator range) se interpretira kao:

  `[pocetak, kraj>`
  - Uključena je lijeva granica, ali ne i desna
- Jednakost vrijednosti na koje pokazuju iteratori (`*iter1 == *iter2`) ne mora značiti da su iteratori jednaki
- `vector` i `deque` podržavaju dodatne operacije nad iteratorima:
  - `iter + n; iter += n` pomiče iterator za n mjesta u "desno"
  - `iter - n; iter -= n;` pomiče iterator za n mjesta u "lijevo"
  - `iter1 - iter2;` vraća broj `n` takav da je `iter1 + n == iter2`
  - `< <= > >=`

    iter1 je manji od iter2 ako je pozicija iter1 ispred pozicije iter2

Tipovi u spremnicima

- Tipovi definirani u slijednim spremnicima:
  - `size_type` tip koji je dovoljno velik da podnese veličinu bilo kojeg spremnika
  - `iterator`
  - `const_iterator`
  - `reverse_iterator`
  - `const_reverse_iterator`
  - `value_type` tip podataka u spremniku

Iteratori na konstantnim spremnicima

- Za iteriranje po konstantnom spremniku potrebno je koristiti `const_iterator`

```
void even_odd(const vector<int> *pvec,
              vector<int> *pvec_even,
              vector<int> *pvec_odd) {
   vector<int>::const_iterator c_iter=pvec->begin();
   vector<int>::const_iterator c_iter_end=pvec->end();

   for ( ; c_iter != c_iter_end; ++c_iter )
       if (*c_iter % 2)
            pvec_odd->push_back(*c_iter);
       else pvec_even->push_back(*c_iter);
}
```

- `reverse_iterator` koristimo za kretanje po spremniku unazad

Operacije na spremnicima

- Iteratori:
  - `c.begin()` vraća iterator koji adresira prvi element u spremniku
  - `c.end()` vraća iterator koji adresira element iza posljednjeg elementa u spremniku
    - (ne smijemo ga dereferencirati)
  - `c.rbegin()` vraća reverse iterator ("`end() - 1`")
  - `c.rend()` vraća reverse iterator ("`begin() - 1`")

- `back, top, front, at`
- Ubacivanje elemenata
  - `c.push_back(el)`
  - `c.push_front(el)`
  - `c.insert(p, el)` //ubacuje el ispred pozicije p; vraća iterator na el
  - Operacije ubacivanja mogu uništiti iteratore

Ubacivanje elemenata

- ```
  vector<string> svec;
  list<string> slist;
  string s1("foo");
  slist.insert(slist.begin(), s1); // ili push_front(s1);
  svec.insert(svec.begin(), s1);
  ```

- ```
  string s2("bar");
  list<string>::iterator iter;
  iter = find(slist.begin(), slist.end(), s1);
  slist.insert(iter, s2);
  ```

- Poziv metode `push_back()` ekvivalentan je s:
  ```
  // slist.push_back(value);
  slist.insert(slist.end(), value);
  ```

- Možemo koristiti i način `insert(i_gdje, i_poc, i_kraj);`

Zadaci

- Zadatak: Napišite program koji učitava i stavlja na kraj liste stringove sve dok se ne učita string “kraj”
- Zadatak: Ispišite sadržaj gore dobivene liste
  - Pomoću pop_front() i front() operacija
  - Pomoću iteratora
- Zadatak: ispred svakog stringa S u listi dodajte još onoliko čvorova koliko S ima slova, u svaki od čvorova upišite po jedno slovo od S
  - npr. ako je učitana lista bila (“RP4”, “Jupi”), onda rezultantna lista treba biti (“R”, “P”, “4”, “RP4”, “J”, “u”, “p”, “i”, “Jupi”)
- Zadatak: na kraju premjestite iz liste u vektor sve stringove duljine veće od 1
- Zadatak: isti zadatak, ali nove čvorove treba dodavati iza učitanih

Zauzeće memorije

- memorijsko zauzeće vectora se ne povećava sa svakim pojedinim ubacivanjem elementa, nego se prilikom pojedinih povećavanja alocira i još nešto dodatnog prostora
- Terminologija:
  - kapacitet – ukupan broj elemenata koji se mogu nalaziti u spremniku prije opetovanog povećavanja – capacity()
  - veličina – trenutni broj elemenata u spremniku – size()
  - maksimalna veličina – maks. broj elemenata u spremniku – max_size()
- ```
  v.reserve(nova_velicina); // capacity
  v.resize(nova_velicina);  // size
  ```
- Reserve i resize mogu uništiti iteratore
- Zadatak: isprobajte kako se povećavaju size i capacity kod vektora, liste i reda prilikom dodavanja elemenata.

```
vector<int> ci;
for(int i=0; i<25; ++i) {
   ci.push_back(i*i);
   cout << "size = " << ci.size() ;
   cout << "cap  = " << ci.capacity() ;
 }
```
```

Brisanje elemenata

- `c.erase(p)` briše element na kojeg pokazuje iterator p
  - vraća iterator na element iza obrisanog
- `c.erase(b,e)` briše sve elemente između dva iteratora
- `c.clear()` briše sve elemente spremnika
- `c.pop_back()` briše zadnji element spremnika
- `c.pop_front()` briše prvi element spremnika
  - Samo za listu i deque

- ```
  string searchValue("FooBar");
  list<string>::iterator iter =
      find(slist.begin(), slist.end(), searchValue);
  if (iter != slist.end()) slist.erase(iter);
  ```

- Možemo obrisati i niz elemenata određen dvama iteratorima [>
  - ```
    list< string >::iterator first, last;
    first = find(slist.begin(), slist.end(), val1);
    last = find(slist.begin(), slist.end(), val2);
    slist.erase(first, last);
    ```

Brisanje elemenata

- Brisanje elemeneta može uništiti iteratore
- brisanje svih elemenata liste jednakih 5 – `erase`:

```
list<int> L; ... // napuni nekako L
list<int>::iterator li, ltemp;
li = L.begin();
while (li != L.end())
   if (*li == 5) {
     L.erase(li); li++; //opasno!
   }
     else li++;
```

- Pokušali smo povećati iterator koji smo upravo "uništili". Bolje bi bilo:

```
if (*li == 5) {
      li = L.erase(li);
   }
   else li++;
```

- oprez: slična stvar se može desiti i neopreznim korištenjem naredbi `push_back`, `pop_back` i `insert` na vectoru.

Spremnici – uspoređivanje

- `c1=c2` `c1 == c2;` `c1 != c2;` pridruživanje, jednakost tj. nejednakost
  - c1 i c2 moraju biti istog tipa
- `c1 < c2;` `c1 <= c2;`
- `c1 > c2;` `c1 >= c2;`
  - dozvoljeno samo ako se navedene operacije mogu izvršavati na elementima unutar spremnika
  - Uređaj je leksikografski

- Zadatak: Napišite program koji uspoređuje vektor i listu leksikografski (element po element). Sjetite se da kod liste nemate operator[ ].

- Zadatak: Isprobajte da li možete usporediti dvije “liste vektora integera”.

Izbor slijednog spremnika

- Spremnik biramo obzirom na njegove karakteristike ubacivanja, pretraživanja i brisanja elemenata
- Neki kriteriji odabira pogodnog slijednog spremnika
  - ukoliko trebamo direktan pristup elementima iz spremnika: `vector` ili `deque`
  - ukoliko unaprijed znamo broj elemenata koje trebamo spremiti: `vector` ili `deque`
  - ukoliko trebamo ubacivati ili brisati elemente na pozicijama različitim od krajeva spremnika: `list`
  - ukoliko ne trebamo ubacivati ili brisati elemente na početnom kraju spremnika: `vector`

Stack, queue

- stack

```
#include <stack>
...
stack<int> S;
S.push(3);
S.push(5);

int a = S.top();
S.pop();

if (S.empty())
  { ... }

int zz = S.size();
```

- queue

```
#include <queue>
...
queue<string> Q;
Q.push("abc");
Q.push("xy");

string a = Q.front();
Q.pop();

if (Q.empty())
  { ... }

int zz = Q.size();
```

Još elemenata C++ standardne biblioteke

- `bitset`
  - niz bitova fiksne duljine
- `complex`
  - kompleksni broj
- `pair`
  - uređeni par dva objekta

complex

- `#include <complex>`
- Omogućava korištenje kompleksnih brojeva
- `complex<double> z1(0, 7); // 0 + 7*i`
- `complex<float> z2(3); // 3 + 0*i`
- `complex<long double> zero; // 0 + 0*i`
- Može i ovo:
- `complex<int> ci(2, 2);`
- `complex<string> sc("abc", "xyz");`
- Zadatak:
- Isprobajte `cin` i `cout` na complex-u
- Isprobajte aritmetičke operacije (`+, -, *, /`) na kompleksnim brojevima
- Isprobajte uspoređivanje (`==, !=, <, >, <=`, ...)
- Napišite funkciju koja računa apsolutnu vrijednost kompleksnog broja
    - Što prima takva funkcija?
    - Koji je povratni tip funkcije?

bitset

- Potrebno je uključiti zaglavlje `<bitset>`
- `bitset` je također predložak (isto kao i `vector`), ali kao parametar prima broj
- Niz bitova fiksne duljine
  - `bitset<8> bajt; // 8 bitova`
  - `bitset<13> b_ul(19); // 13 bitova sa bin.zapisom 19`
    - `// ”višak” bitova u bitsetu se napuni nulama`
  - `bitset<19> b_s(string(“100110”));`
  - `bitset<2> b_ul2(6); // dva “desna” bita od 6 (“110”)`
- Pristup pojedinom bitu u bitsetu
  - `bitset<8> bajt(7); // “00000111”`
  - `cout << bajt[2];`
- Isprobajte slijedeće:

```
bitset<7> b_s2(“100110”);
  bajt[2] = 0;
  cout << bajt;
  bajt[2] = 5; // sve sto nije 0 je 1
  cout << bajt;
```

bitset

- Logičko “i”
  - `bitset<8> b1(7); bitset<8> b2(61);`
  - `bitset<8> r;`
  - `for (size_t i = 0; i != b1.size(); ++i)`
    - `r[i] = b1[i] && b2[i]; // 1. varijanta`
    - `if (b1[i] && b2[i]) r.set(i); // 2. varijanta`
- Jednostavnije:
  - `cout << (b1 & b2);`

- Zadatak: Pretpostavimo da igramo Loto 6/45… ne 15… u stvari 10/45. Izvučene brojeve predstavljamo bitsetom duljine 45 (46 ukoliko ne računamo 0). Napišite funkciju koja simulira izvlačenje brojeva (podsjetnik: `rand()` & `srand(time(0))`, recimo neka su brojevi dok se izvlače spremljeni u listi) i sprema ih u bitset. Napišite i funkciju koja popunjava listić – niz od 10 kombinacija koje učitavamo sa tipkovnice. Za kraj napišite i funkciju koja računa koliko ste brojeva pogodili vašim listićem.

pair

- Potrebno je uključiti zaglavlje `<utility>`
- Omogućava stvaranje uređenih parova dvaju tipova
  - `pair<string, string> student("Alan", "Ford");`
  - `pair<double, double> koordinata(1.1, 3.3);`
- Tipovi ne moraju biti isti
  - `pair<string, vector<int> > ime_vektora_i_vektor;`
- Zadatak: Napišite funkciju `min()` koja za dobiveni vector `int`-ova vraća najmanji element tog vectora, te broj njegovog pojavljivanja.

```
typedef pair<int,int> min_val_pair;
min_val_pair min(const vector<int>& ivec) {
    int minVal = ivec[0];
    int occurs = 0;
    int size = ivec.size();

    for (int i = 0; i < size; ++i) {
        if (minVal == ivec[i])
            ++occurs;
        else
            if (minVal > ivec[i]) {
                minVal = ivec[i];
                occurs = 1;
            }
    }
    return make_pair(minVal, occurs);
}
```

pair

- Par dozvoljava direktan pristup svojim koordinatama sa .first i .second, te ima sva uspoređivanja.
- Zadatak: Napišite program koji učitava matricu, te vraća koordinate maksimalnog elementa matrice (par(int,int)).
- Zadatak: Učitavajte znak po znak sa tipkovnice sve dok ne učitate točku, upitnik ili uskličnik. Ako ste pročitali slovo, spremite ga u vektor. Vektor neka se sastoji od uređenih parova <slovo, broj pojavljivanja slova>. Dakle, kada pročitate novo slovo, potrebno je pregledati da li u vektoru već postoji navedeno slovo. Ako postoji, samo treba povećati odgovarajući brojač; inače treba ubaciti novo slovo (sa odgovarajućim brojačem) na kraj vektora.

Asocijativni spremnik

- Asocijativni spremnici podržavaju gotovo sve operacije kao i slijedni spremnici:
  - Konstruktori: `C<T> c; C<T> c1(c2); C<T> c(b,e);`
  - Relacijski operatori: `== !=`
  - Tipovi, iteratori:
  - Zamjena: `c1.swap(c2);`
  - Brisanje, ubacivanje: `c.clear(); c.erase(i); c.insert(i,)`
  - Veličina: `c1.size();`

- Nepodržane operacije su:
  - `front, push_front, pop_front`
  - `back, push_back, pop_back`

- Elementi su u asocijativnim spremnicima poredani po ključu, bez obzira na redoslijed kojim ih ubacujemo u spremnik

Mapa

- `#include <map>`
- Mapa je asocijativno polje koje sadrži parove (ključ, vrijednost)
  - Ključ se upotrebljava za indeksiranje elemenata
  - vrijednost predstavlja korisni podatak koji želimo čuvati
- Primjeri:
  - `map<string, short> ocjene;`
  - `map< pair<int,int>, boja> slika;`
- Konstruktori:
  - `map<k, v> m; //mapa (ključ, vrijednost)`
  - `map<k, v> m1(m2); //mapa m1 kao kopija m2`
  - `map<k, v> m(b, e); //mapa preko iteratora`
- Tip koji koristimo kao ključ mora podržavati `operator<`
- Tipovi:
  - `map<k,v>:: key_type` ključ
  - `map<k,v>:: mapped_type` vrijednost
  - `map<k,v>:: value_type` par (const key_type, mapped_type)
  - `map<k,v>::iterator`

Mapa

- Dereferenciranje iteratora na mapu vraća par
- Dodavanje elemenata u mapu:
  - `operator[]`
    - ```
      map<string, int> word_count;
      word_count[“abc”] = 1;
      ```
- Korištenje vrijednosti koju vraća operator[]
  - `cout << word_count[“abc”];`
  - `++word_count[“abc”];`

- Oprez: operator[] ubacuje element u mapu ako on tamo još nije bio

- operacije `[]`, `find`, ... rade u logaritamskom vremenu, dok je dereferenciranje iteratora trenutno

- Zadatak: Napišite program koji učitava riječi sa ulaza te računa i ispisuje broj pojavljivanja pojedinih riječi.
      ```

Mapa

- Dodavanje elemenata u mapu sa insert:
  - `m.insert(e)`
    - Ubacuje par e u mapu m ako ključ od e još nije u mapi
    - Ako je ključ od e u mapi, onda se ne dogodi ništa
    - Vraća par (iterator, bool); Iterator pokazuje na ubačeni element dok Bool kazuje da li je element bio ubačen ili ne
  - `m.insert(b, e)`
    - svaki element u rasponu iteratora b i e se ubacuje u mapu m. Vraća void
- Primjeri:
  - `word_count.insert(make_pair("abc", 1));`
- Operacije koje provjeravaju da li je ključ u mapi:
  - `m.count(k)`
    - Vraća broj pojavljivanja ključa k u mapi m
    - `Vraća ustvari 0 ili 1`
  - `m.find(k)`
    - Vraća iterator na ključ k, ako k postoji u mapi
    - Ako ključ ne postoji u mapi, vraća end() iterator

Mapa

- Brisanje elemenata iz mape:
  - `m.erase(k)`
    - Briše elemente sa ključem k u mapi m
  - `m.erase(i)`
    - Briše element na kojeg pokazuje iterator i
- Operator [ključ] dohvaća element iz mape
  - `int occurs = word_count["xyz"];`
    - Mogući problem: ako ključ "xyz" nije postojao u mapi, operator[] ga je upravo dodao u mapu, što možda ne bismo htjeli
    - Međutim, pogodnost takvog pristupa je što operator[] u stvari vraća referencu, pa ponekad može biti brže jednom pisati `int &w = word_count["xyz"]`, nego više više puta prethodnu varijantu. Npr.
      ```
      int &w = word_count["xyz"];int i=100;while(--i>0) w++;
      ```
      Bi bilo puno bolje nego
      ```
      int i=100;while(--i>0) word_count["xyz"]++;
      ```

- Zadatak: Upišite nekoliko riječi u mapu, te izbrišite sve riječi koje se pojavljuju više od jednom.
  - Pazite: erase može pokvariti iteratore

Skup

- `#include <set>`
- Skup je kolekcija ključeva
- Primjeri:
  - `set<string> rijeci;`
  - `set< pair<int,int> > koordinate;`
- Operacije definirane na skupu su identične operacijama na mapi, osim:
  - Nema mapped_type
  - Nema operatora[]

- Zadatak: Napišite program koji računa uniju i presjek dva skupa cijelih brojeva.

Multimapa i multiskup

- Zaglavlje
  - `#include <map>`
  - `#include <set>`
- "Multi": dozvoljeno je višestruko pojavljivanje istog ključa
- Primjeri:
  - `multimap<string, int> rijeci;`
  - `multiset<string> tekst;`
- Operacije su iste kao kod mape i skupa, osim:
  - Multimap ne podržava operator[]
- Dodatne operacije specifične za "multi":
  - `m.lower_bound(k);`
  - `m.upper_bound(k);`
  - `m.equal_range();`

Zadatak: Ubacite nekoliko podataka u multimapu koja sadrži parove (autor,djelo). Npr. ("Mato Lovrak", "Vlak u snijegu"). Ubacite u multimapu nekoliko djela istog književnika. Isprobajte nove operacije

Generički algoritmi

- Generički algoritmi su algortimi koji barataju sa spremnikom putem iteratora, ne znajući pritom o kakvom se točno spremniku radi
- Unutar spremnika definirane su samo najvažnije funkcije za rad sa točno određenim spremnikom
- Operacije zajedničke svim spremnicima implementirane su odvojeno u obliku generičkih algoritama
- Zaglavlja
  - `#include <algorithm>`
  - `#include <numeric>`
- Primjeri:
  - Sort, find, merge, fill, count
  - Sort je uvijek isti, bez obzira na tip podataka koji sortiramo (potrebno je jedino da radi <)
- Najčešći oblici algoritama:
  - `Alg(beg, end, other)`
  - `Alg(beg, end, dest, other)`
  - `Alg(beg, end, beg2, other)`
  - `Alg(beg, end, beg2, end2, other)`

Generički algoritmi

- `accumulate(beg, end, val)`
- `find(beg, end, value)`
  - `find_if`
- `count`
  - `count_if`
- `fill`
  - `fill_n`
- `replace(beg, end, what, with)`
  - `replace_if(beg, end, pred, with)`
- `remove(val)`
  - `remove_if(pred)`
- `sort(beg, end)`
  - `sort(beg, end, pred)`
- Neki algoritmi rade bolje na nekim spremnicima kao članske funkcjie, npr. za listu će .sort, .reverse, .unique, merge raditi brže od ovih općenitih

- Zadatak: Isprobajte gore navedene funkcije