Računske vježbe 10

Programiranje II

Uvod

Za programski jezik C++ definisana je bogata standardna biblioteka gotovih klasa i funkcija za često korišćene složene strukture podataka (nizovi, liste, skupovi itd.) kao i za postupke (pretraživanja, uređivanja itd.). Pokazuje se da su većina tih klasa i funkcija generičke klase, odnosno funkcije. Zbog toga se često i koristi pojam standardna biblioteka šablona (engl. *STL - Standard Template Library*). Svrha ove biblioteke jeste da pruži efikasne realizacije za često korišćene strukture podataka i postupke, a koje ne zavise od tipova podataka od kojih se struktura sastoji odnosno koji se obrađuju. Jedan jednostavan primjer:

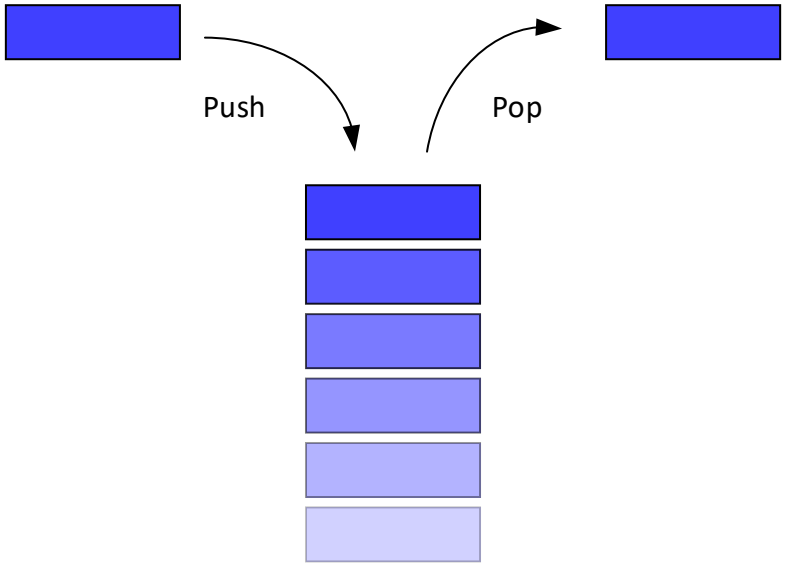
```
#include <iostream>
#include <algorithm>

using namespace std;

class Actor
{
public:
  char* name;
  Actor(char* newName)
  {
    name = new char[strlen(newName) + 1];
    strcpy(name, newName);
  }
};

int main()
{
  char brad[] = "BradPitt";
  Actor BradPitt(brad);
  char mima[] = "MimaKaradzic";
  Actor MimaKaradzic(mima);
  swap(BradPitt, MimaKaradzic);
  cout << "Brad Pitt se sada zove: " << BradPitt.name << endl;
  cout << "Mima Karadzic se sada zove: " << MimaKaradzic.name << endl;
}
```

nam pokazuje koliko je ova biblioteka moćna. Naime, upotrebom generičke funkcije **swap** zamijenili smo sadržaj dva objekta. Što je bilo u jednom sada je u drugom i obratno. Ovu funkciju nije teško realizovati za jednu klasu. Ukoliko bismo ih imali na desetine u projektu morali bismo značajno vrijeme da alociramo samo za realizaciju zamjene vrijednosti. Vrši se duboko kopiranje, o tome ne moramo da brinemo.

1. Napisati program koji provjerava da li je zadati izraz ispravan. Izraz se sastoji od otvorenih i zatvorenih oblih (), vitičastih {} i uglastih zagrada [ ].

```
#include <iostream>
#include <string>
#include <stack>

using namespace std;

bool brackets(string);

int main()
{
    string exp = "{{{{([])}}}}";
    if (brackets(exp))
        cout << "Izraz je validan!" << endl;
    else
        cout << "Izraz nije validan!" << endl;
}

bool brackets(string exp)
{
    stack<char> s;
    for (int i = 0; i < exp.length(); i++)
    {
        if (exp[i] == '(' || exp[i] == '[' || exp[i] == '{')
            s.push(exp[i]); // vrsimo upis sve dok su zagrade otvorene
        else
        {
            if (s.empty()) // provjeravamo da li je stek prazan
                return false;
            if (exp[i] == ')' && s.top() != '(')
                return false;
            else if (exp[i] == ']' && s.top() != '[')
                return false;
            else if (exp[i] == '}' && s.top() != '{')
                return false;
            s.pop(); // izbacujemo element s vrha
        }

    }
    return s.empty();
}
```

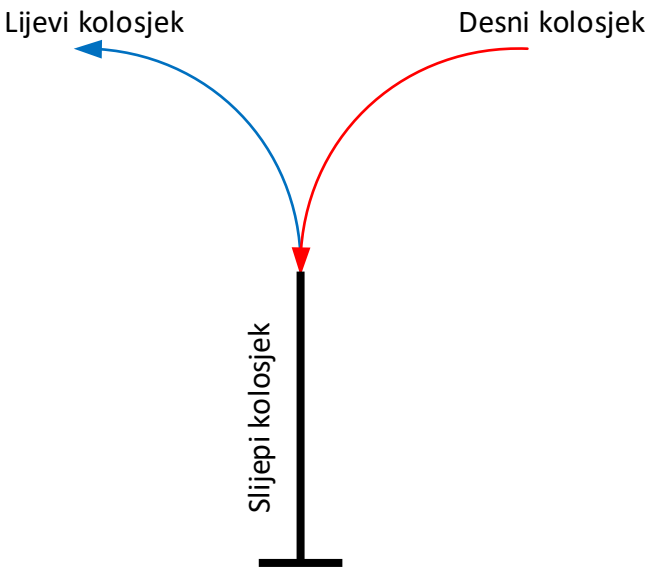
U našem zadatku ćemo se upoznati sa stekom (engl. *stack*) koji predstavlja apstraktni tip podatka odnosno strukturu podataka koja se zasniva na LIFO principu (engl *last in, first out*). Stek se može zamisliti kao gomila tanjira naslaganih jedan na drugi gdje je očigledno da ćemo tanjiru na vrhu lako pristupiti dok nam je pristup tanjirima ispod njega otežan. Elementima steka se ne može pristupiti preko indeksa, moguće je pristupiti samo elementu na vrhu. Grafička reprezentacija steka se može vidjeti na slici 1. U našem zadatku smo koristili stek kako bismo provjerili izraz koji se sastoji od otvorenih i zatvorenih zagrada. Izraz je predstavljen stringom i to ne stringom nizom karaktera već tipom podatka **string** koji nam pruža standardna biblioteka, a sa čijim prednostima smo se upoznali na predavanjima. Na stek slažemo otvorene zagrade sve dok ne dođemo do prve zatvorene. Kako je na vrhu steka poslednja otvorena zagrada, provjeravamo da li se ta otvorena zagrada poklapa sa posmatranom zatvorenom. Ukoliko se ne poklapaju onda izraz sigurno nije ispravan, a ukoliko se poklapaju otvorena zagrada se miče sa steka i prelazi se na sljedeću zatvorenu. Uočite da smo koristili još jedan novi tip podatka **bool** koji može imati vrijednosti



Slika 1: Grafička reprezentacija steka. Operacija *push* smješta neki element na vrh steka dok operacija *pop* miče element sa steka. Često se nad stekovima implementira još jedna operacija *top* odnosno *peek*. Pomoću nje možemo vidjeti koji je trenutni element na vrhu steka.

true i *false* i koji je dio C++ standarda već dugi niz godina.

2. Ka slijepom kolosjeku dolazi voz sa strane označene kao desni kolosjek:



Dozvoljeno je od voza otkačiti jedan ili više vagona (moguće i cio voz) sa njegove prednje strane i uvesti ih na slijepi kolosjek. Vagone je moguće izvesti sa slijepog kolosjeka na lijevi kolosjek, ali ne i vratiti ih na desni kolosjek. Nakon toga je moguće ponovo uvesti vagone sa desnog kolosjeka. Vagone sa lijevog kolosjeka nije moguće vratiti na slijepi kolosjek. Poznato je kojim redom dolaze vagoni. Potrebno je provjeriti da li je moguće reorganizovati voz tako da su vagoni na lijevom kolosjeku poređani kao 1, 2, ... , N.

```
#include <iostream>
#include <stack>
#include <list>
```

```
4
5  using namespace std;
6
7  bool reorganize(list<int>);
8
9  int main()
10 {
11     int N;
12     int temp;
13     list<int> track;
14     cout << "Unesite broj vagona na kolosjeku 1:" << endl;
15     cin >> N;
16     cout << "Unesite vagone na kolosjeku 1:" << endl;
17     for (int i = 0; i < N; i++)
18     {
19         cin >> temp;
20         track.push_back(temp);
21     }
22     if (reorganize(track))
23         cout << "Voz se moze reorganizovati!" << endl;
24     else
25         cout << "Voz nije moguce reorganizovati!" << endl;
26 }
27
28 bool reorganize(list <int> rightTrack)
29 {
30     list<int> leftTrack;
31     stack<int> spurTrack;
32     int length = rightTrack.size();
33     int current = 1;
34     list<int>::iterator it;
35     while (true)
36     {
37         if (spurTrack.empty() || spurTrack.top() != current)
38         {
39             it = rightTrack.begin();
40             while (it != rightTrack.end())
41             {
42                 spurTrack.push(*it);
43                 it = rightTrack.erase(it);
44                 if (spurTrack.top() == current) break;
45             }
46         }
47         if (!spurTrack.empty() && spurTrack.top() == current)
48         {
49             leftTrack.push_back(spurTrack.top());
50             spurTrack.pop();
51             current++;
52         }
53         else
54         {
55             break;
56         }
57     }
58     cout << "Na desnom kolosjeku se sada nalaze vagoni: ";
59     for (it = leftTrack.begin(); it != leftTrack.end(); ++it)
60     {
61         cout << *it << " ";
62     }
```

```
63      cout << endl;
64      return length == leftTrack.size();
65  }
```

U ovom zadatku smo slijepi kolosjek predstavili kao stek, a lijeve i desne kolosjeke kao listu. Sa listama smo se upoznali još na predmetu Programiranje I. One omogućavaju efikasno umetanje i izbacivanje elemenata i u unutrašnjosti liste, a ne samo na krajevima. Pristupanje elementima po proizvoljnom redosljedu je vrlo neefikasno. Liste mogu biti jednostruko ili dvostruko povezane odnosno dati čvor liste može pokazivati samo na sljedeći element ili i na onaj iza njega. Elementima liste može da se pristupa samo od početka ili sa kraja! Generička klasa *list* ostvaruje dvostruko povezane liste. Liste se mogu sortirati, obrnuti, spojiti, filtrirati... Kroz elemente liste je najlakše proći upotrebom iteratora. Iterator je instanca iteratorske klase pomoću kojeg pristupamo elementima struktura podataka. Primjer iteratora za listu:

```
list <typename>:: iterator it;
```

gdje *typename* može biti proizvoljni tip i zavisi od tipa liste od interesa. Iterator se obično inicijalizuje na početak liste sa:

```
it = data.begin();
```

i inkrementira se sve dok se ne dođe do kraja liste. Iteratori mogu da se zamisle kao pokazivači koji pokazuju na elemente nizova. Sve operacije nad iteratorima su ostvarene preklapanjem operatora tako da nema razlike između izraza s običnim pokazivačima i izraza s iteratorima. Vrijednosti tekućeg elementa pristupamo sa **it*. Iteratori se uvijek prefiksno inkrementiraju. Zašto? Sjetimo se preklapanja postfiksnog i prefiksnog inkrementiranja i zaključaka koje smo tada donijeli. Brisanje elementa iz liste dok vršimo iteraciju kroz nju može da dovede do problema ukoliko ne ažuriramo iterator na ispravan način. Metoda *erase* briše element na poziciji na koju ukazuje iterator, a vraća iterator koji ukazuje na element koji se nalazio nakon onog koji je obrisan. Uslov kojim se prekida iteriranje kroz listu se može pogrešno shvatiti:

```
it != leftTrack.end();
```

na način što bismo pretpostavili da ćemo se zaustaviti na pretposlednjem elementu. To se neće dogoditi zato što metoda *list::end* vraća iterator koji referencira na hipotetički element koji bi se našao nakon poslednjeg i zapravo ne pokazuje ni na jedan element.
Pa kako smo mi to riješili naš zadatak? Kako se lijevi kolosjek može puniti samo iz slijepog, prvo provjeravamo da li je slijepi kolosjek prazan (ukoliko jeste onda sigurno moramo preuzeti vagone iz desnog) ili ukoliko nije da li je vagon na vrhu onaj vagon koji očekujemo. Ukoliko je slijepi kolosjek prazan ili ukoliko nam vagon sa vrha ne odgovara, slijepi kolosjek punimo sve dok ne naiđe onaj vagon koji je po redu. Ukoliko ne uspijemo da dođemo do tog vagona, izvršenje se prekida. Ukoliko do njega dođemo on se prebacuje u lijevi kolosjek. Objasnimo reorganizaciju voza na sljedećem primjeru. Neka je N jednako 4 i neka vagoni sa desnog kolosjeka dolaze po sljedećem redu [4, 3, 1, 2]. Prvi vagon koji želimo da se nađe na lijevom kolosjeku je 1. Gledamo da li se on nalazi u slijepom kolosjeku koji je u prvoj iteraciji prazan. Kako vagona broj 1 nema, uvozimo vagone sa desnog kolosjeka sve dok ne dođemo do vagona broj 1. Dakle, stanje u steku je sada [1, 3, 4] zato što je na vrhu slijepog kolosjeka poslednji dospjeli vagon. Uvozimo vagon broj 1 u lijevi kolosjek. Sada se slijepi kolosjek sastoji od vagona [3, 4], a desni kolosjek od vagona broj 2. Sljedeći po redu za lijevi kolosjek je vagon broj 2. Posmatramo slijepi kolosjek, ali on se ne nalazi na njegovom vrhu. Uvozimo vagone iz desnog kolosjeka sve dok ne naiđemo na vagon broj 2 i naravno sve dok desni kolosjek nije prazan. Uvezli smo vagon broj 2 pa je slijepi kolosjek sada [2, 3, 4]. Vagon broj 2 prebacujemo u lijevi kolosjek koji sada očekuje vagon broj 3...