12. Generičke funkcije i klase – template (predlošci)

Pod pojmom generičkog programiranja podrazumijeva se izrada programskog koda koji se u nepromijenjenom obliku može primijeniti na različite tipove podataka.

11.1. Generičke funkcije

U C jeziku se za generičko programiranje koriste predprocesorske makro naredbe.

```
#define Kvadrat(x)  x*x
```

je makro naredba kojom se dio koda koji je u zagradama označen s x zamjenjuje s x*x. Makro naredbe nisu funkcije. One se realiziraju u toku kompajliranja leksičkom zamjenom teksta. Primjerice

makro naredba	izvršava se kao
int i; Kvadrat(i) * 3;	int i; i*i*3
float x; x*Kvadrat(6.7)	float x; x*6.7*6.7
float x; x = Kvadrat(x + 6.7)	float x; x = x + 6.7 * x + 6.7 //greška

U prva dva primjera pokazano je kako se ista makro naredba koristi za tipove `int i float`.
Greška u trećem primjeru, se može izbjeće tako da se "argumenti" makro naredbe napišu u zagradama, tj.

```
#define Kvadrat(x)  (x)*(x)
```

U C++ se isti efekt može postići korištenjem definiranjem `inline` funkcija:

```
inline int    Kvadrat(int x)    {return  x*x;}
inline float  Kvadrat(float x)  {return  x*x;}
inline double Kvadrat(double x) {return  x*x;}
```

Preporuka: radije koristite inline funkcije nego makro naredbe, jer se time osigurava bolja kontrola tipova i izbjegava mogućnost pojave prethodno opisane pogreške.

Predložak generičkih funkcija

Primjetimo da sva tri tipa definicije funkcije `Kvadrat` imaju isti oblik:

```
T Kvadrat(T x) {return  x*x;}
```

gdje `T` može biti `int, float ili double`. Ova definicija ima generički oblik. U C++ jeziku se može vršiti generičko definiranje funkcija, na način da se ispred definicije ili deklaracije funkcije, pomoću ključne riječi `template`, označi da T predstavlja "generički tip".

```
template <class T>  T Kvadrat(T x) {return  x*x;}
ili
template <typename T> T Kvadrat(T x) {return  x*x;}
```

Razlika u ova dva zapisa je u upotrebi ključne riječi class ili typename, ali značenje je potpuno isto.
Izvršenje poziva funkcije se mora obaviti sa stvarnim tipovima:

```
int x,y;
y = Kvadrat<int>(x);
```

Kompajler tek u trenutku poziva funkcije može odrediti način kako se prevodi generička funkcija, stoga su generičke funkcije po upotrebi ekvivalentne makro naredbama.

> Zapamti: **da bi se moglo izvršiti kompajliranje programa u kojem se koristi generička funkcija, kompajleru mora biti na raspolaganju potpuna definicija funkcije, a ne samo deklaracija funkcije kao što je slučaj kod upotrebe regularnih funkcija.**

Generički tip T se može koristiti unutar funkcije za oznaku tipa lokalne varijable

```
template <class T>  T Kvadrat(T x)
{
  T result = x*x;
  return result;
}

int main ()
{
  int i=5, k;          float a=10.9, b;
  k=Kvadrat<int>(i);   b=Kvadrat<float>(a);
  cout << k << endl;   cout << b << endl;
  return 0;
}
```

Ispis je:

```
25
118.81
```

Ne mora se uvijek, pri pozivu funkcije, specificirati generički tip ako se iz tipova argumenata nedvosmisleno može zaključiti koji se parametrički tip koristi.

```
int i=5, k;
float a=10.9, b;
k=Kvadrat(i);
b=Kvadrat(a);
cout << k << endl;
cout << b << endl;
```

Preporuka je ipak, da se uvijek eksplicitno specificira parametrički tip.

Mogu se definirati generičke funkcije s više generičkih tipova. Primjerice.

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

definira funkciju s dva argumenta kojima se generički tipovi T i U. Moguće je izvršiti poziv funkcije na slijedeći način:

```
int i,j;
long l;
i = GetMin<int,long> (j,l);
```

ili još jednostavnije

```
i = GetMin (j,l);
```

Iako su j i l različitog tipa, kompajler sam vrši pretvorbe integralnih tipova.

Predložak generičke klase

Na isti način, kao što se definiraju generičke funkcije, mogu se definirati i generičke klase.

Definicija članova klase se provodi pomoću generičkog tipa.

Primjerice, u datoteci arrtempl1.cpp je definirana generička klasa array:

```
template <class T>  class array
{.... }
```

pomoću koje se može realizirati nizove proizvoljnog tipa. Primjerice, deklaracijom

```
array<int> myints(5);
```

se formira niz od 5 cjelobrojnih elemenata, a s

```
array<string> mystrings(5);
```

se formira niz od 5 stringova

```
template <class T> class array
{
    T *m_pbuff;
    int  m_size;

public:
    array(int N = 10): m_size(N) {m_pbuff=new T[N];}
    ~array() {delete [] m_pbuff;}
    int size() {return m_size;}

    void set(int x, T value);
    T get(int x);

    T & operator [] (int i)
              {return m_pbuff[i];}
    const T & operator [] (int i) const
              {return m_pbuff[i];}
};

template <class T>  void array<T>::set(int x, T value)
{  m_pbuff[x]=value; }

template <class T>  T array<T>::get(int x)
{  return m_pbuff[x]; }
```

`Get()` i `set()` funkcije smo mogli definirati inline unutar definicije klase. One su definirane izvan definicije klase kako bi se pokazalo da se tada uvijek ispred definicije funkcije mora napisati generički prefiks: `template <class T>` .

```
int main () {
  array<int> myints(5);
  array<float> myfloats(5);
  myints.set (0, 100);
  myfloats.set(3, 3.1416);

  cout << "myints ima: "
       << myints.size() <<" elemenata"<< '\n';

  cout << myints[0] << '\n';
  cout << myfloats[3] << '\n';

  return 0;
}
```

Rezultat je:

```
myints ima: 5 elemenata
100
3.1416
```

Elementima niza se pristupa pomoću set/get funkcija ili pomoću indeksne notacije.

PRIMJER:
Klasa `pair` služi kao kontenjer dva elementa koji mogu biti različitih tipova. Formirat ćemo niz takovih parova pomoću generičke klase `array`. U taj niz ćemo upisati i iz njega ispisati niz parova kojima prvi element predstavlja redni broj (tip int), a drugi element je kvadratni korijen prvoga (tip float).

```
template <class T1, class T2>
class pair
{
    T1 value1;
    T2 value2;

  public:
    pair (T1 first=0, T2 second=0)
    {value1=first; value2=second;}

    T1 GetFirst () {return value1;}
    void SetFirst (T1 val) {value1 = val;}

    T2 GetSecond () {return value2;}
    void SetSecond (T2 val) {value2 = val;}
};
```

```
int main ()
{
  // niz od 5 parova int,float

  array <pair<int,float> > arr(5);

  for(int i=0; i<arr.size(); i++)
  {
    arr[i].SetFirst(i); // prvi sadrži redni broj
    arr[i].SetSecond(sqrt(i));// kvadratni korijen prvog
  }

  for(int i=0; i<arr.size(); i++)
  cout << arr[i].GetFirst() <<':'
       <<arr[i].GetSecond()<< endl;
  cout << endl;
  return 0;
}
```

Rezultat je:

```
0:0
1:1
2:1.41421
3:1.73205
4:2
```

Važno je upozoriti na oblik deklaracije niza parova:

```
array <pair<int,float> > arr(5);
```

1. Vidimo da se konkretna realizacije generičkog tipa može izvršiti i pomoću drugih generičkih tipova.

2. U deklaraciji se pojavljuju dva znaka > >. Između njih obvezno treba napisati razmak, jer ako bi napisali

```
array <pair<int,float>> arr(5); // greška : >>
```

kompajler bi dojavio grešku zbog toga jer se znak >> tretira kao operator.

Da bi se izbjegle ovakove greške, preporučuje se koristiti typedef deklaracije oblika:

```
typedef pair<int,float> if_pair;
array <if_pair> arr(5);
```

Specijalizacija predloška

Ponekad se s jednim predloškom ne mogu obuhvatiti svi slučajevi realizacije s različitim tipovima. U tom se slučaju, za tipove kojima realizacija odstupa od predloška, može definirati poseban slučaj realizacije koji nazivamo specijalizacija predloška.
Za klasu koju definiramo predloškom:

template <class ***gen_tip***> ***identifikator*** **{...}**

specijalizacija za konkretni tip se zapisuje u obliku:

template <> class ***identifikator*** <***konkretni_tip***> **{...}**

Specijalizacija se smatra dijelom predloška, stoga njena deklaracija započima sa template <>**. U njoj se moraju definirati svi članovi predloška, ali isključivo s konkretnim tipovima za koje se vrši specijalizacija predloška. Podrazumijeva se da moraju biti definirani konstruktor i destruktor, ukoliko su definirani u općem predlošku. To ćemo demonstrirati banalnim primjerom u programu pair-spec.cpp. U klasi `pair` definirana je člansku funkciju `modul()` koja daje ostatak dijeljenja prvog s drugim elementom kada su oba elementa tipa `int`, a kada su elementi drugih tipova funkcija `modul()` vraća vrijednost 0.**

```
template <class T1, class T2>  //opći predložak
class pair {
    T1 value1;
    T2 value2;
  public:
    ....
    int modulo () {return 0;}
};


template <>                    // specijalizacija predloška
class pair <int,int> {         // za tip int,int
    int value1, value2;
  public:
    ....
    int modulo () {return value1 % value2;}
};
```

Predlošci s konstantnim parametrima

Parametri predloška mogu biti i integralne konstante(int,char,long, unsigned). Primjerice,

```
template <class T, int N> class array {...}
```

je predložak za definiranje klase array pomoću generičkog tipa T i integralne konstante N. Vrijednost integralne konstante se mora specificirati pri deklaraciji objekta. Primjerice, s

```
array <float,5> myfloats;
```

deklarira se myfloat kao niz realnih brojeva kojem se dodatna svojstva zadaju s konstantom N=5. U slijedećem primjeru koristit ćemo ovaj konstantni parametar za postavljanje veličine niza.

```
template <class T, int N>
class array
{
    T m_buff[N];   // niz od N elemenata
    int m_size;
public:
    array() : m_size(N) {}
    int size() {return m_size;}
    T & operator [] (int i) {return m_buff[i];}
    const T & operator [] (int i) const
             {return m_buff[i];}

};

int main () {
  array <int,5> myints;
  array <float,5> myfloats;
  myints[0]= 100;
  myfloats[3]= 3.1416;
```

```
    cout << "myints ima: "<< myints.size()
         <<" elemenata"<< '\n';
    cout << myints[0] << '\n';
    cout << myfloats[3] << '\n';
    return 0;
}
```

Dobije se ispis:

```
100
3.1416
```

Predodređeni i funkcijski parametri predloška

U predlošcima se mogu koristiti predodređeni parametri. Primjerice, predloškom

```
template <class T=int, int N=10> class array {...}
```

se omogućuju deklaracije oblika:

```
array<> a_int_10;               // niz od 10 int
array<float> a_float_10;        // niz od 10 float
array<char,100> a_char_100;     // niz od 100 char
```

Napomena: predodređeni parametri se ne smiju koristiti u funkcijskim predlošcima.

Parametri predloška mogu biti funkcije

```
template <int Tfunc (int)>
class X
{...
   y = Tfunc(x); !!!
 ...
}
```

Usporedna predložaka i makro naredbi

Predloške možemo smatrati inteligentnim makro naredbama. Primjena predložaka ima nekoliko prednosti u odnosu na maro naredbe:

- **Lakše ih je shvatiti jer predlošci izgledaju kao regularne klase (ili funkcije).**
- **U razvoju predloška lakše se provodi testiranje s različitim tipovima.**
- **Kompajler osigurava znatno veću kontrolu pogreški nego je to slučaj s makro naredbama.**
- **Funkcijski predlošci imaju definiran doseg, jer se mogu definirati kao dio klase (friend) ili namespace.**
- **Funkcijski predlošci mogu biti rekurzivni.**
- **Funkcijski predlošci se mogu preopteretiti.**

Pri kompajliranju neke datoteke, kompajler mora raspolagati s potpunom implementacijom predloška. Stoga je uobičajeno da se specifikacija i implementacija funkcija zapisuje u istoj ".h" datoteci.

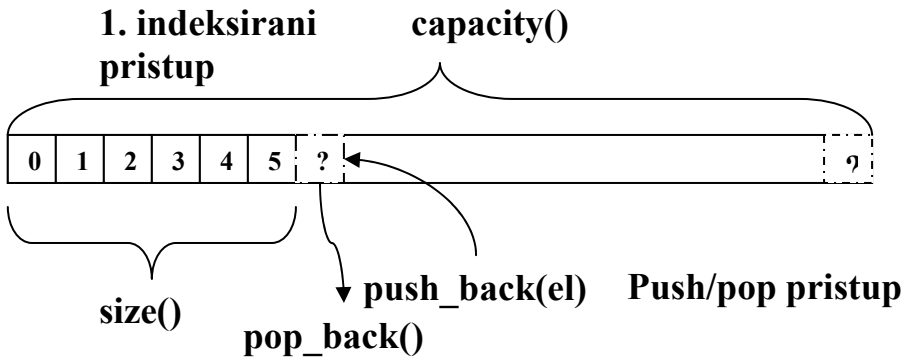
Makro karakter predložaka se posebno očituje kod klasa koje imaju statičke članove. U tom slučaju se za svaku realizaciju predloška inicira posebna statička varijabla (kod regularnih se klasa za sve objekte jedne klase generira samo jedna statička varijabla).

9.3 Definicija i upotreba klase tvector<class T>

Generička klasa tvector predstavlja podskup standardne klase vector. Namjena joj je manipuliranje s homogenim kolekcijama elemenata prozvoljnog tipa. Izvršit ćemo specifikaciju i implementa klase tvector temeljem znanja koja smo stekli razmatrajući generičku klasu array i klasu TString.

Objekti tvector klase imaju slijedeće karakteristike:

- **capacity() - kapacitet vektora je broj ćelija koje se automatski alociraju za spremanje elemenata vektora,**
- **size() -veličina vektora je broj elemenata koje stvarno sadrži vektor.**
- **operator [] - Pojedinom elementu vektora se pristupa pomoću indeksnog operatora, ili se koriste dvije funkcije:**
  - **push_back(el) dodaje element na indeksu iza posljednje upisanog elementa, a**
  - **pop_back(el) briše posljednji element iz vektora.**
  - **Pri pozivu push_back() funkcije vodi se računa o kapacitetu vektora i vrši se automatsko alociranje potrebne memorije (memorija se udvostručuje ako potrebna veličina vektora premašuje trenutni kapacitet). Namjena ove dvije funkcije je da se vektor može koristiti kao dinamička kolekcija elemenata (kontenjer).**
- **resize(n) - ako je potreban veći kapacitet vektora, on se može dobiti pozivom funkcije resize(n).**



Specifikacija ADT tvector

```
//*************************************************
//  template <class T> tvector {....}
//  Generički parametar tipa T mora zadovoljiti:
//  (1) T ima predodređeni konstructor
//  (2) za tip T definiran je operator =
//
// tvector( )
//     POST: Kapacitet vektora == 0
//
// tvector( int size )
//     PRED: size >= 0
//     POST: kapacitet/veličina vektora == size
//
// tvector( int size, const T & Value )
//     PRED:: size >= 0
//     POST: Kapacitet/veličina == size, svi elementi se iniciraju na Value
//
// tvector( const tvector & vec )
//     Opis: kopirni konstruktor
//     POST: vector je  kopija od vec
//
// ~tvector( )
//     Opis: destruktor
//
//  operator pridjele vrijednosti
//
// const tvector & operator = ( const tvector & rhs )
//     POST: pridjela vrijednosti od rhs;
//           ako se razlikuju veličine ova dva vektora
//           vrši se promjena veličine na rhs.size()
```

```
//
//  pristupnici
//
// int  length( ) const
// int  capacity( ) const
//     POST: vraća broj ćelija alociranih za vektor
// int  size( ) const
//     POST: vraća stvarni broj elemenata u vektoru
//           koristi se za push_back i pop_back funkcije
//
// indeksirani pristup i mutatori
//
//  void push_back(const T& t);
//      POST: umeće element t na poziciji size()
//            ako je size>= capacity dvostruko
//            povećava kapacitet
//
// void pop_back();
//     POST: odstranjuje element s pozicije size()
//           i smanuje size za 1
//
// T & operator [ ] ( int k );
// const T & operator [ ] ( int k );
//     Opis: Dobava k-tog elementa, uz kontrolu indeksa
//     PRED: 0 <= k < capacity()
//     POST: vraća k-ti element
//
// void resize( int newSize );
//     Opis:  promjena veličine vektora u newSize
//     PRED:  newSize >= 0,
//            kapacitet je capacity, a veličina je size()
//     POST:  1. size() == newSize;
```

```
//                Ako je newSize > size() tada i
//                capacity=newSize
//                inače se kapacitet ne mijenja
//             2. za 0 <= k <= min(size(), newSize),
//                vector[k]je kopija originala
//                ostali elementi se iniciraju
//                pomoću predodređenog T konstruktora
//    Nota: ako je newSize < size(), gube se neki elementi
//
// void clear();
//    POST: size = 0,  capacitet nepromijenjen
//
// void reserve(int size);
//     Opis: rezervira memoriju
//     PRED: size >0
//     POST: resize(size) ako je size > capacity()
//
```

Primjer deklaracije:

```
   tvector<int> v1;          // vektor s 0-elementa
   tvector<int> v2(4);       // vektor s 4-elementa
   tvector<int> v3(4, 22);   // vektor s 4-elementa -  svi  vrijednosti 22.
```

<u>Zadatak:</u> Napišite program u kojem korisnik unosi proizvoljan niz brojeva. Kao kontenjer brojeva koristite `tvector` objekt. Nakon završenog unosa izračunajte sumu i srednju vrijednost unesenog niza.

```
#include <iostream>
#include "tvector.h"
using namespace std;
int main()
{
    tvector<double> vec;
    double val;

    // unos proizvoljnog niza brojeva u vektor vec
    while (cin >> val)
        vec.push_back(val);

    // unos završava kada se otkuca neko slovo!!
    //zatim računamo sumu i srednju vrijednost

    double sum = 0;
    for (int i=0; i<vec.size(); i++)
        sum += vec[i];
    double avg =sum /vec.size();
    cout <<"Suma od "<<vec.size()
         <<" elemenata: "<< sum
        <<". Srednja vrijednost:"<< avg << endl;
    return 0;
}
```

<u>Uočite:</u> pri unosu podataka vektor koristimo kao kontenjer – u njega odlažemo elemente pomoću funkcije push_back. Kasnije vektor koristimo kao običan niz (elementima pristupamo pomoću indeksa).

<u>Korištenje vektora je pogodnije od rada s običnim nizovima je ne moramo voditi računa o alokaciji memorije.</u>

<u>Zadatak:</u> Formirajte vektor koji će sadržavati kolekciju stringova. Otvorite neku .cpp datoteku, očitavajte liniju po liniju teksta. Svaku liniju spremite u kolekciju stringova, ali na način da se odstrane svi komentari koji započimaju s //.

```
#include <iostream>
#include <fstream>        // ifstream
#include <cstdlib>        // exit()
#include <string>
using namespace std;
#include "tvector.h"

void Decomment(const string& in, string& out)
{
//PRED: in sadrži liniju teksta
//POST: out sadrži liniju bez komentara  koji počima s //

  out = "";
  int idx = in.find("//");  //da li počima komentar
       if(idx==0)
            return;    // komentar počinje u prvom stupcu
        else if(idx>0)  // dobavi dio stringa do komentara
            out = in.substr(0, idx);
      else //linija bez komentara
            out = in;
}

int main()
{
    string infilename, outfilename;
```

```
cout << "Ime ulazne .cpp datoteke: ";
cin >> infilename;
cout << "Ime izlazne datoteke: ";
cin >> outfilename;
if(infilename == outfilename)
{  cout << "Imena datoteka se moraju razlikovati";
   exit(1);
}
ifstream input(infilename.c_str()); // otvori ulaz
if (input.fail() )
{   cout << "Ne moze se otvoriti: "
         << infilename << endl;
    exit(1);
}

string str;              // radni string
tvector<string> txt;     // vektor za kolekciju stringova

// rezerviraj memoriju za 1000 stringova.

txt.reserve(1000);

// Time se smanjuje broj potrebnih alokacija
// rezerviranjem se ne mijenja trenutna veličina vektora


while (getline(input, str))   // čitaj iz datoteke
{
  string out;
  Decomment(str, out);
  txt.push_back(out);
}
```

```
    input.close();

    // otvori izlaznu datoteku
    ofstream output(outfilename.c_str());
    if (output.fail() )
    {   cout << "Ne moze kreirati: " << outfilename << endl;
        exit(1);
    }
    // ispiši u izlaznu datoteku
    for(int i=0; i<txt.size(); i++)
    {
        output << txt[i] << endl;
    }
    output.close();
    return 0;
}
```

U predloženom rješenju biti će odstranjen dio teksta koji započima s // pa sve do kraja linije. To znači da će biti odstranjen i tekst koji se nalazi unutar literalnog stringa, primjerice linija

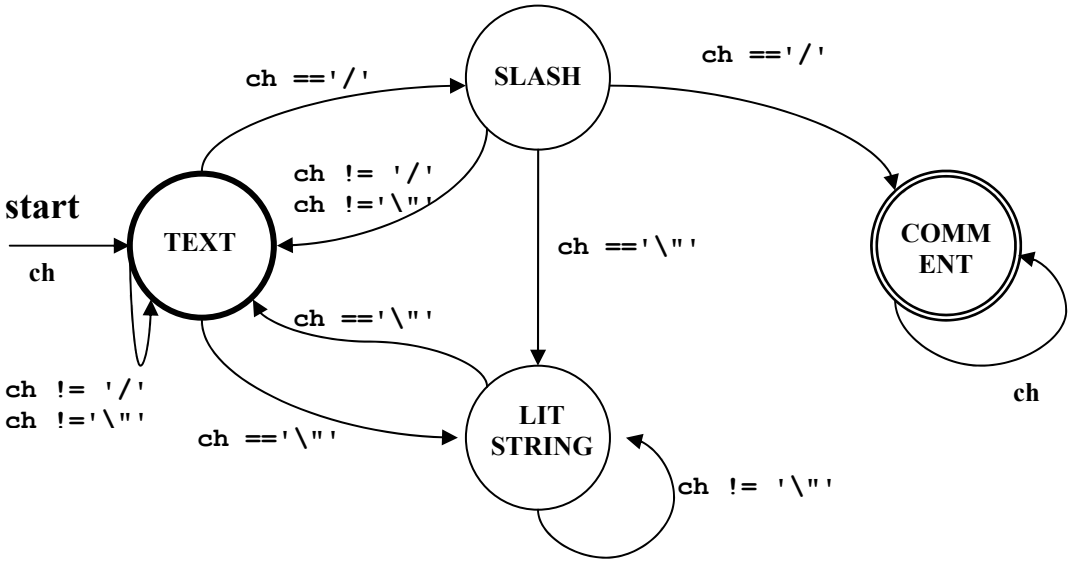
```
int idx = str.find("//");  //da li počima komentar
```

će biti ispisana kao

```
int idx = str.find("
```

Da bi izbjegli ovo pogrešku mora se uzeti u obzir i mogućnost da unutar literalnog stringa mogu biti zapisani znakovi //.

Proces analize linije teksta najbolje je izvršiti znak po znak. U tu svrhu poslužit ćemo se prikazom jednog automata. Automat analizira trenutni znak (ch) i na temelju tog znaka i stanja u kojem se nalazi odlučuje o prijelazu u neko drugo stanje ili zadržava trenutno stanje. Na slici su krugovima označena 4 stanja, a strelicama su označeni prijelazi iz stanja.



1. **TEXT – je početno stanje. Označeno je podebljanom kružnicom. U tom stanju se analizira slijedeći znak i odlučuje o prijelazu u novo stanje.**
2. **SLASH je stanje koje prelazi automat ako u stanju TEXT slijedeći znak je ' /' (slash).**
3. **COMMENT je stanje koje automat prelazi ako se u SLASH stanju ponovo pojavi znak ' /' . U ovom stanju automat ostaje sve do kraja linije. Ovo stanje je završno stanje (označeno je dvostrukom kružnicom). Kada je automat u završnom stanju može se prekinuti rad automata, bez obzira što još nisu analizirani svi znakovi.**
4. **LIT_STRING je stanje koje nastaje ako u TEXT stanju ili u SLASH stanju slijedeći znak je navodnik. Automat ostaje u ovom stanju dok se ponovo ne pojavi znak navodnika. Tada se automat vraća u stanje TEXT.**

Automat se jednostavno realizira pomoću sljedeće funkcije:

```
enum State{TEXT, FIRST_SLASH, COMMENT, LIT_STRING};

void Decomment(const string& input, string& output)
{
    State state = TEXT;
    output = "";
    for(int i=0; i<input.length(); i++)
    {   char ch = input[i];
        switch(state){
        case TEXT:
            if (ch == '/') // potencijalno početak komentara
               state = FIRST_SLASH;
             else {
                if (ch == '\"') state = LIT_STRING;
               output += ch;
             }
            break;
        case LIT_STRING:
             if (ch == '\"')
                 state = TEXT; //literalni string završava
             output += ch;
             break;
        case FIRST_SLASH:
            if (ch == '/')        // dvostruki slash
              state = COMMENT;    // započima komentar
            else {                 // samo jedan SLASH
              output += '/';  // ispiši slash prošlog stanja
              output += ch;   // i trenutni znak
              if (ch == '\"') state = LIT_STRING
```

```
                else  state = TEXT;      // stanje početno
            }
            break;
        case COMMENT: // završno stanje
            return;    // komentar je do kraja linije
                       // output sadrži liniju bez komentara
        }
    }
}
```

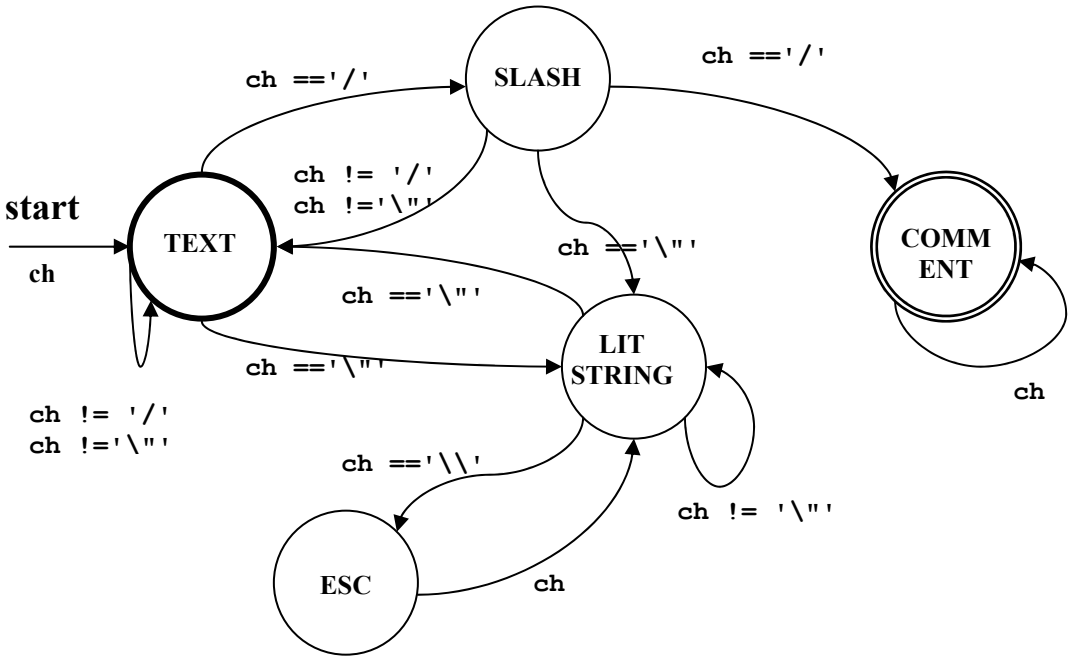
Prikazani automat se često naziva konačni deterministički automat. Njegova je primjena česta u leksičkoj analizi teksta.

I ovo rješenje ima manu. Što ako se unutar literalnog stringa pojavi escape sekvenca, primjerice

```
"string s \" escape sekcencom\" "
```

Tada bi se prekinulo stanje LIT_STRING prije kraja stringa.

Problem bi mogli riješiti tako da u automat dodamo peto stanje – ESC, prikazano na slici:



Napišite funkciju Decomment() kojom se realizira ovaj automat.

12.4 STOG ili STACK

Stog (eng. stack) je naziv za kolekciju podataka kojoj se pristupa po principu LIFO – last in, first out. Primjerice, kada slažemo tanjure tada stavljamo jedan tanjur poviše drugog – tu operaciju zovemo *push*, a kada uzimamo tanjur onda uvijek uzimamo onaj kojeg smo posljednjeg stavili u stog – tu operaciju nazivamo *pop*.

12.4.1 Specifikacija apstraktnog tipa Stack

Specifikacija ADT Stack je:

- **empty() - vraća true ako je stack prazan**
- **size() – vraća broj elemenata stoga**
- **push(el) – unosi se element el na stog**
- **pop() – skida se element s vrha stoga (operacija je uspješna ako na stogu postoji bar jedan element)**
- **el = top() – vraca vrijednost elementa s vrha stoga (operacija je uspješna ako na stogu postoji bar jedan element)**

12.4.2 Specifikacija i implementacija klase Stack

Ovaj ADT ćemo realizirati pomoću klase `vector`.

Funkcije `push` i `pop` ćemo realizirati pomoću članskih funkcija klase `vector`:
`push_back()` i `pop_back()`,

Funkciju `top()` pomoću pomoću članske funkcija `back()`, koja vraća posljednji element vektora.

Nema ograničenja na veličinu stoga, jer vektor može rasti do proizvoljne veličine.

Funkcije `empty()` i `size()` su iste kao kod klase `vector`.

```
#include <vector>
template <class Type> class Stack
{
public:
   Stack(){}
   ~Stack(){}

   bool empty() const {return m_v.empty();}
   // Vraća true ako je stog prazan

   int size() {return m_v.size();}
   // Vraća broj elemenata na stogu

   Type& top()              {return (m_v.back()); }
   const Type& top() const {return (m_v.back());}
   // PRE: na stogu je bar jedan elemant
   // Vraća vrijednost elementa na vrhu stoga

   void pop() {v.pop_back(); }
   // PRE: na stogu je bar jedan elemant
   // odstranjuje element s vrha stoga

   void push(const Type &elem)    {m_v.push_back(elem); }
   // postavlja element na vrh stoga
   // POST: na stogu je jedan element više

private:
   std::vector<Type> m_v;
};
```

Napomena:

Klasa `Stack` sadrži samo jednu člansku varijablu `vector<T> m_v`; dakle `Stack` klasa sadrži člana koji je objekt tipa `vector`. Često se ovakova povezanost, kada jedna klasa ima članove drugih klasa, naziva kompozicija (composition) ili "has-a" veza klasa.

Prividno izgleda da konstruktor i destruktor klase Stack ne obavljaju nikakovu radnju, jer su definirani kao "prazne" funkcije. U slučaju kompozicije klasa kompajler uvijek pri pozivu konstruktora klase poziva i konstruktor članskih objekata, u ovom slučaju se poziva konstruktor vektora v. Isto vrijedi i za destruktor. Ako članski objekt ima više konstruktora, onda se mora u okviru konstruktora eksplicitno navesti koji se oblik konstruktora članskih objekata poziva. Ako se to ne navede izvršava se predodređeni konstruktor.

12.4.3 Primjena stoga za proračun izraza postfiksne notacije

Korištenjem programski simuliranog stoga jednostavno se provodi računanje matematičkih izraza u postfiksnoj notaciji. Postfiksna notacija izraza se piše tako da se prije pišu operandi, a tek onda operator koji na njih djeluje, npr.

infiksna notacija izraza	postfiksna notacija izraza
A + B * C	A B C * +
(A + B) * C	A B + C *
(a + b)/(c – d)	a b + cd - /
a * b / c	a b * C /

Ovaj tip notacije se naziva i obrnuta poljska notacije, prema autoru Lukasiewiczu.

Svojstva postfiksne notacije su:

- **Svaka formula se može napisati bez zagrada.**
- **Infiksni operatori moraju uvažavati pravila pioriteta što nije potrebno kod postfiksne notacije.**
- **Za proračun postfiksne notacije prikladna je upotreba stoga.**

Pretvorba infiksne u postfiksnu notaciju se izvodi slijedećim algoritmom:

1. **Kompletno ispiši zagrade između svih operanada, tako da zagrade potpuno odrede redoslijed izvršenja operacija.**
2. **Pomakni svaki operator na mjesto desne zagrade**
3. **Odstrani zagrade**

Primjerice, pretvorba izraza (8+2*5) / (1+3*2-4), prema gornjem pravilu, tj.

```
prema 1.        ( ( 8 + ( 2 * 5 ) ) / ( 1 + ( ( 3 * 2 ) - 4 ) ) )
prema 2. i 3.   ( ( 8 + ( 2 * 5 ) )   ( 1 + ( ( 3 * 2 ) - 4 ) ) /
                ( 8   ( 2 * 5 ) +     1   ( ( 3 * 2 ) - 4 ) + /
                  8     2   5 * +     1   ( ( 3 * 2 )   4 - + /
                  8     2   5 * +     1       3   2 *   4 - + /
```

daje postfiksnu notaciju: 8 2 5 * + 1 3 2 * + 4 - /

Za izračun postfiksnog izraza, koji ima n simbola, vrijedi algoritam:

1. **Neka je k =1 indeks prvog simbola**
2. **Dok je k <= n ponavljaj**
   Ako je k-ti simbol operand tada ga gurni na stog.
   Ako je k-ti simbol operator tada dobavi dvije vrijednosti sa stoga (najprije drugi, pa prvi operand), izvrši naznačenu operaciju i rezultat vrati na stog.
   Uvećaj k za 1
3. **Algoritam završava s rezultatom na stogu.**

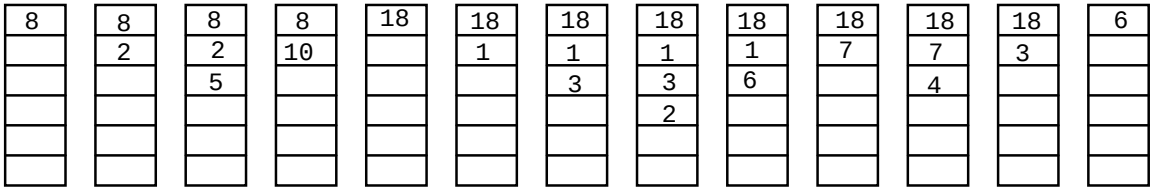
Primjer:

Izraz zapisan infiksnoj notaciji: (8+2*5) / (1+3*2-4)

ima postfiks notaciju: 8 2 5 * + 1 3 2 * + 4 - /

1	8 2 5 * + 1 3 2 * + 4 - /	push 8			
2	2 5 * + 1 3 2 * + 4 - /	push 2			
3	5 * + 1 3 2 * + 4 - /	push 5			
4	* + 1 3 2 * + 4 - /	b=top(), pop(), a=top(), pop(), push (10)	a=2	*	b=5
5	+ 1 3 2 * + 4 - /	b=top(), pop(), a=top(), pop(), push(18)	a=8	+	b=10
6	1 3 2 * + 4 - /	push 1			
7	3 2 * + 4 - /	push 3			
8	2 * + 4 - /	push 2			

9	* + 4 - /	b=top(), pop(), a=top(), pop(), push (6)	a=3	*	b=2
10	+ 4 - /	b=top(), pop(), a=top(), pop(), push(7)	a=1	+	b=6
11	4 - /	push 4			
12	- /	b=top(), pop(), a=top(), pop(), push(3)	a=7	-	b=4
13	/	b=top(), pop(), a=top(), pop(), push(6)	a=18	/	b=3



Raspored stanja na stogu nakon pojedinog koraka proračuna.

Izračun izraza pomoću postfiksne notacije je jedan od uobičajenih načina kako neki intepreteri izračunavaju izraze zapisane u višim programskim jezicima - najprije se vrši pretvorba infiksnog zapisa izraza u postfiksni zapis, a zatim se izračun izraza vrši pomoću stoga. U ovom slučaju ne koristi se stog kojim upravlja procesor već se rad stog simulira programski.

```
void main(int argc, char *argv[])
{
    int i, len, arg1, arg2;
    Stack<int> S;
    char *str = argv[1]; len = strlen(str);

    for (i = 0; i < len; i++) {

      if (str[i] == '+')
      {
          arg2 = S.top(); S.pop(); arg1 = S.top(); S.pop();  S.push(arg1+arg2);
      }
      else if (str[i] == '*')
      {
           arg2 = S.top(); S.pop(); arg1 = S.top(); S.pop(); S.push(arg1*arg2);
      }
      else if (str[i] == '-')
      {
          arg2 = S.top(); S.pop(); arg1 = S.top(); S.pop(); S.push(arg1-arg2);
      }
      else if (str[i] == '/')
      {
          arg2=S.top(); S.pop(); arg1= S.top();S.pop();
          if (arg2==0)  cout << "Djeljenje s nulom\n"; exit(1);
          S.push(arg1/arg2);
      }
```

```
      else if (isdigit(str[i])) // konverzija znamenki u broj
      {   S.push(0);
          do { arg1 = S.top(); S.pop();
               S.push(10*arg1 + ((int)str[i]-'0'));
               i++;
          } while (isdigit(str[i]));
          i--;
      }
    }
    arg1 = S.top();
    cout << "Rezultat: "<< str <<"="<<arg1<<" \n";
}
```

Kada se u komandnoj liniji pozove program polish s argumentom:

```
c:> polish  "30 5 2 - / 10 *"

Rezultat: 30 5 2 - / 10 *=100
```

12.4.5 Simulacija izvršnog stoga (run-time stack)

Do sada smo više puta spomenuli da se za prijenos argumenata i rezultata funkcije koristi dio memorije koji se naziva izvršni stog (eng. run-time stack). Razlog tome je činjenica da u strojnom jeziku procesora ne postoje naredbe za prijenos parametara pri pozivu funkcije. Poziv funkcije zapravo znači da se izvršenje programa nastavlja od adrese funkcije. Nakon izvršene posljednje naredbe neke funkcije ponovo se vrši skok na naredbu koja slijedi nakon naredbe poziva funkcije. Dakle, strojnim jezikom se pristupa funkcijama koje se u C jeziku tretiraju kao void funkcije i koje ne koriste argumente. U višim programskim jezicima se pak definiraju funkcije koje koriste argumente i koje vraćaju vrijednosti. Pokazat ćemo kako je to realizirano simulirajući izvršni stog s objektom Stog:

```
vector<int> Stog;     // simulator izvrsnog stoga
int Reg;              // registar u kojeg se upisuje
                      // rezultat koji vraća funkcija

void push(int n)    {Stog.push_back(n);}
int & top(int n=0) {return Stog [Stog.size()-1+n];}
void pop()          {Stog.pop_back();}
```

Uobičajeno je da se pri prevođenju programa u strojni kod, prijenos argumenata u funkciju obavlja na sljedeći način:

Pozvana funkcija se realizira tako da se očekuje da se argumenti funkcije nalaze na stogu, na način da se posljednji argument nalazi na vrhu stoga. Vrijednost koju funkcije vraća se sprema u jedan od registara procesora kojeg ćemo nazvati Reg.

Primjerice funkciju:

```
int minimum(int x, int y)
{
   if (x > y)
       return y;
   else
       return x;
}
```

se na razini strojnog koda tretira kao void funkciju:

```
void minimum()   // funkcija definirana bez parametara
{

      //pozivna funkcija postavlja argumente na vrh stoga

      int &y = top();    // y argument je na vrhu stoga
      int &x = top(-1);  // x argument je na stogu ispod y,
                         // jer je prvi gurnut na stog
      if(x > y)
        Reg = y;          // rezultat vraćamo u Reg
      else
        Reg = x;
}
```

ii) U pozivnoj se funkciji prije poziva neke funkcije svi argumenti stavljaju na izvršni stog, jedan za drugim, pomoću naredbe push(arg1); push(arg2)...a zatim se vrši poziv funkcije. Nakon povrata iz funkcije vrijednost koju funkcija vraća se nalazi u registru Reg. Također, nakon povrata iz funkcije u pozivnoj funkciji treba očistiti stog pozivom funkcije pop(), onoliko puta koliko je prethodno bilo stavljeno argumenbata na stog.
To je pokazano u programu params1.cpp

```
//Datoteka: params1.cpp
//program koji racuna kvadrat broja 5

int main()
{
   int x = 5, y = 7, rezultat;
   push(x);                  // argument stavljamo na stog
   push(y);                  // argument stavljamo na stog
   minimum();                // pozivamo funkciju minimum
   rezultat = Reg;           // rezultat je u registru Reg
   pop();                    // ocistimo stog, sadrži (y)
   pop();                    // ocistimo stog, sadrži (x)
   cout << "minimum od "<< x <<" i "<<y
        <<" je " << rezultat << endl;
    return 0;
}
```

Dobijer se ispis:

```
Minimum od 5 i 7 je 5
```

Uočite da su reference x i y sinonimi za varijable top() i top(-1). Mogli smo pisati i

```
void minimum()
{
  if(top(-1) > top())
    Reg = top();          // rezultat vraćamo u Reg
  else
    Reg = top(-1);
}
```

ali se uvođenjem referenci dobija bolje rješenje jer se smanjuje broj poziva funkcije top().

Pogledajmo još primjer u kojem se realizira kvazi-asemblerski kod funkcije kojom se računa suma elemenata nekog niza.

```
int suma_niza(int A[], int n)
{
  int sum = 0;
  while(n--)
  {
     sum += A[n]
  }
  return sum;
}
```

Cilj nam je pokazati kako se reference prenose u funkciju.

Kompletni primjer kvazi-asemblerske realizacije ove funkcije dan je u programu

```
//Program racuna sumu elemenata niza od 5 elemenata
#include <iostream>
#include <vector>
using namespace std;

vector<int> Stog;   // simulator izvrsnog stoga
int Reg;            // registar rezultata funkcija

void push(int n)   {Stog.push_back(n);}
int & top(int n=0) {return Stog[Stog.size()-1+n];}
void pop()         { Stog.pop_back();}

void suma_niza();

int main()
{
   int A[5] = {1,2,3,4,5};
   int len = 5, rezultat;
   push((int)A);    // prvo na stog adresu niza
   push(len);       // zatim na stog duljina niza
   suma_niza();     // pozivamo funkciju
   rezultat = Reg;  // rezultat je u registru R
   pop();           // konacno ocistimo stog
   pop();
   cout << "suma niza je "<< rezultat << endl;
   return 0;
}
```

```
void suma_niza()
{
  int &n =  top();           // duljina na vrhu stoga
  int* &A =(int*&) top(-1); // adresa niza je ispod
  Reg = 0;
petlja:
  if(n == 0) goto end_while;
  n--;
  Reg += A[n];
  goto petlja;
end_while:
  ;  // rezultat se nalazi  u registaru Reg
}
```