8. NIZOVI

Niz je indeksirani skup podataka - elemenata niza. Niz se deklarira imenom iza kojeg se u uglatim zagradama zapisuje broj elemenata niza, a ispred imena se zapisuje tip elemenata.

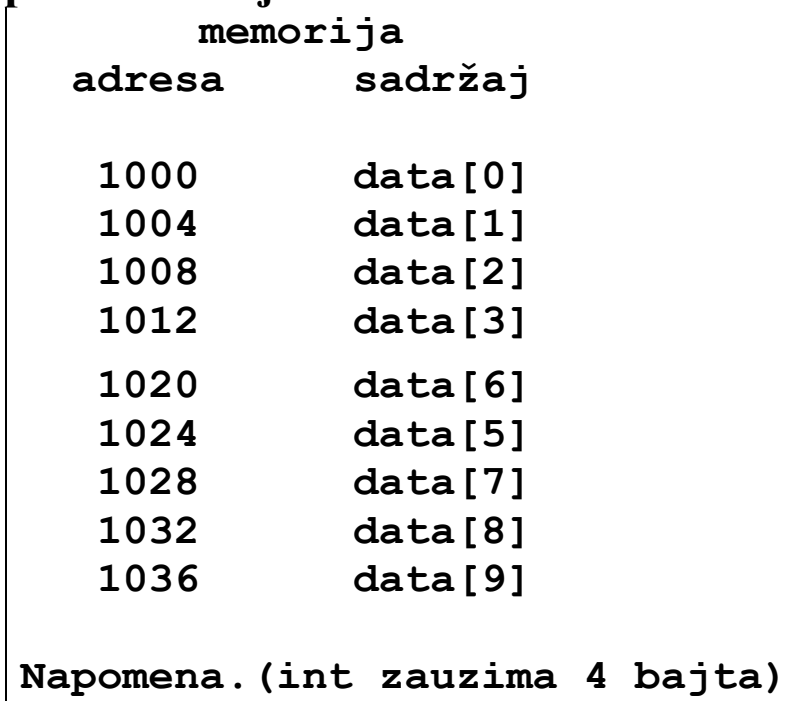
```
// deklaracija niza od 10 elemenata: data[0], data[1],..data[9]
int data[10] ;
```

S elementima niza se operira kao s prostim varijablama

Elementi niza zauzimaju sukscesivne lokacije u memoriji.

Vrijedi pravilo:

***adresa*(data[n]) = *adresa*(data[0]) + n*sizeof(data[0]))**



Niz se može inicijalizirati i s deklaracijom sljedećeg tipa:

```
int data[9]= {1,2,23,4,32,5,7,9,6};
char str[6]= {'H', 'e', 'e', 'l', 'l', '\0'};
```

Unutar vitičastih zagrada navodi se lista konstanti.

Ako se inicijaliziraju svi potrebni elementi niza, tada nije nužno u deklaraciji navesti dimenziju niza:

```
int data[]= {1,2,23,4,32,5,7,9,6};
```

Broj elemenata ovakvog niza uvijek možemo dobiti naredbom:

```
int numelements = sizeof(data)/sizeof(int);
```

Niz se može i parcijalno inicijalizirati. U deklaraciji

```
int data[10]= {1,2,23};
```

prva tri elementa imaju vrijednost 1, 2, 23, a ostale elemente kompilator postavlja na vrijednost nula.

Kako se niz prenosi u funkciju

Kada se jednodimenzionalni niz pojavljuje u deklaraciji ili definiciji argumenata funkcije, tada se ne navodi dimenzija niza.

```
int suma_niza(int A[], int numelements)
{
    int sum =0;
    for(int i=0; i<numelements; i++)
        sum += A[i];
    return sum;
}
```

Uočimo de se vrijednost elemenata niza može mijenjati unutar funkcije. Očito je da se niz ne prenosi po vrijednosti (by value), jer tada to ne bi bilo moguće.

Pravilo je:
Nizovi se u funkciju prenose kao memorijske reference (by reference).

Kompilator memorijsku referencu niza (tj. adresu) pamti "u njegovom imenu" stoga se pri pozivu funkcije navodi samo ime, bez uglatih zagrada.

```
int suma_niza(int A[], int numelements)
{
    int sum =0;
    for(int i=0; i<numelements; i++)
        sum += A[i];
    return sum;
}

int main()
{
    int X[]={2,6,7,7,7,8,9,3,56,854};

    cout << "suma="
         << suma_niza( X,  sizeof(X)/sizeof(int))<< endl;
    cout << "adresa niza je " << A << endl;

    return 0;
}
```

Dobije se ispis:

```
suma=959
adresa niza je 0x22ff30
```

Zadatak: U datoteci "ocjene.dat" u tekstualnom obliku zapisane su ocjene u rasponu od 1 do 10. Sadržaj datoteke "ocjene.dat" neka čini slijedeći niz ocjena:

```
2 3 4 5 6 7 2 4 7 8 9 1 3 4 6 9 8 7 2 5 6 7 8 9
3 4 5 1 7 3 4 10 10 9 10 5 7 6 3 8 9 4 5 7 3 4 6 1
2 9 6 7 8 5 4 6 3 2 4 5 6 1 3 4 6 9 10 7 2 10 7 8 1
```

Treba izraditi program imena hist.cpp pomoću kojeg će se prikazati histogram ocjena u 10 grupa (1, 2,.. 10).

```
c:>hist < ocjene.dat
```

```
10      *****
 9      *******
 8      ******
 7      **********
 6      *********
 5      *******
 4      **********
 3      ********
 2      ******
 1      *****
```

Pri pozivu programa u komandnoj liniji je sa

```
c:> hist < ocjene.dat
```

izvršeno "preusmjerenje" standardnog ulaza na sadržaj datoteke.

Efekt preusmjeravanja je da se sadržaj datoteke `ocjene.dat` tretira kao da je unesen sa standardnog ulaza.

```
/* kostur programa */

int main(void)
{
    int ocjena, counts[10] = {0};

     while (cin >> ocjena) {
         if(ocjena <1 || ocjena >10)
             break;
         record(ocjena, counts);
     }

     cout << "Histogram" << '\n';
     printhistogram(counts);
     return 0;
}

void record( int ocjena, int counts[])
{
    counts[ocjena-1]++;
}
```

```
void printhistogram(int counts[])
{
    for (int i = 10; i > 0; i--)
    {
        cout <<  i << '\t';
        printstars(counts[i-1]);
    }
}


void printstars(int n)
{
    while (n-- > 0)
       cout << '*'; cout << '\n';
}
```

Višedimenzionalnim nizovi

Višedimenzionalnim nizovima se pristupa preko dva ili više indeksa. Deklaracijom:

```
int x[3][4];
```

definira se dvodimenzionalni niz koji ima 3 x 4 = 12 elemenata. Deklaraciju se može čitati i ovako: definiran je niz kojem su elementi 3 niza od 4 elementa tipa int.

Elementima se pristupa preko dva indeksa. U slijedećem primjeru pokazano je kako se dobije suma svih elemenata matrice x;

```
int i, j, num_rows=3, num_cols=4;
int sum=0;

for (i = 0; i < num_rows; i++)
   for (j = 0; i < num_cols; j++)
       sum += x[i][j];

cout << "suma elemenata matrice = " << sum;
```

U C jeziku dvodimenzionalni nizovi, koji opisuju matrice, u memoriji su složeni po redovima matrice.

Najprije prvi redak, zatim drugi, itd..

matrični prikaz

```
x[0][0]  x[0][1]  x[0][2]
x[1][0]  x[1][1]  x[1][2]
x[2][0]  x[2][1]  x[2][2]
```

memorija	
adresa	sadržaj
1000	x[0][0]
1004	x[0][1]
1008	x[0][2]
1012	x[1][0]
1016	x[1][1]
1020	x[1][2]
1024	x[2][0]
1030	x[2][1]

Višedimenzionalni se niz može inicijalizirati već u samoj deklaraciji, npr.

```
int x[3][4] = { { 1, 21, 14, 8},
                {12,  7, 41, 2},
                { 1,  2,  4, 3}};
ili

int x[3][4] = {1,21,14,8,12,7, 41, 2, 1, 2, 4, 3};
```

Prijenos dvodimenzionalnih nizova u funkciju

Pri deklariranju argumenta funkcije koji su višedimenzionalnih nizovi ne navodi se prva dimenzija, ali ostale dimenzije treba deklarirati, kako bi kompilator znao kojim su redom elementi složeni u memoriji.

```
int suma_el_matrice(int x[][4],
                    int nredaka, int nstupaca)
{
int i, j, sum=0;
  for (i = 0; i < nredaka; i++)
    for (j = 0; i < nstupaca; j++)
       sum += x[i][j];
  return sum;
}
```

Parametri `nredaka` i nstupaca koji opisuju broj redaka i stupaca matrice.

Očito je da ovo nije baš najbolji način za rad s matricama, jer je korištenje ove funkcije moguće samo za matrice koje imaju 4 stupca.

Nizovi objekata

Elementi niza mogu biti bilo kojeg tipa. Primjerice, deklaracijom

```
Counter counts[10];
```

deklarira se niz od 10 objekata tipa `Counter`. Ovakovi oblik deklaracije podrazumijeva da kompilator uz rezerviranje potrebne memorije izvrši i poziv predodređenog konstruktora za svaki element niza.

Članskim funkcijama objekata pristupa se pomoću točka operatora, primjerice

```
counts[3].get_count()
```

Primjer: Histogram realiziram pomoću niza objekata tipa Counter

```
void main(void) {
int ocjena;
Counter counts[10];  // jedina razlika od prethodnog pr.

    while (cin >> ocjena) {
        if(ocjena <1 || ocjena >10)
            break;
        record(ocjena, counts);
    }
    cout << "Histogram" << '\n';
    printhistogram(counts);
}
void record( int ocjena, Counter counts[]) {
     counts[ocjena-1].incr_count();
}

void printhistogram(Counter counts[]) {
   for (int i = 10; i > 0; i--)  {
       cout <<  i << '\t';
       printstars(counts[i-1].get_count());
   }
}
```

Pokazivači i nizovi

```
int a[10], *p;  // a je niz, p je pokazivač
```

1. Ime niza ima se tretira se kao "pokazivačka konstanta" koja predstavlja adresu prvog elementa niza. Naredba

```
p = a;
```

vrijednost pokazivača `p` postavlja na adresu elementa `a[0]`. Ovo je ekvivalentno naredbi:

```
p = &a[0];
```

2. Pravilo pokazivačke aritmetike:

Ako pokazivač `pi` pokazuje na `a[i]`, tada `pi + k` pokazuje na `a[i+k]`, odnosno, ako je

```
pi = &a[i];
```

tada vrijedi odnos;

```
*(pi+k) ⇔ *(p+i+k) ⇔ a[i+ k]
```

3. Ekvivalentnost indeksne i pokazivačke notacije niza.

Ime niza napisano bez zagrada predstavlja pokazivač na prvi element. Vrijede slijedeći odnosi:

```
*(a)      ⇔ a[0]
*(a + 1)  ⇔ a[1]
*(a + 2)  ⇔ a[2]
...
*(a + n)  ⇔ a[n]
```

S pokazivačima se također može koristiti indeksna notacija. Tako, ako je inicijaliziran pokazivač p:

```
p = a;
```

tada vrijedi:

```
p[0] ⇔ *p ⇔ a[0]
p[1] ⇔ *(p + 1) ⇔ a[1]
p[2] ⇔ *(p + 2) ⇔ a[2]
...
p[n] ⇔ *(p + n) ⇔ a[n]
```

Nije dozvoljena pokazivačka aritmetika s memorijskim referencama niza:

```
array++;
```
// **nedozvoljen izraz, (ne može se mijenjati konstanta)**

Pokazivači i argumenti funkcije tipa niza

- **Pri pozivu funkcije stvarni argumenti se kopiraju u formalne argumente (parametre) funkcije.**

- **Kada je argument funkcije ime niza, kopira se adresa prvog elementa, dakle stvarni argument koji se prenosi u funkciju je pokazivač na prvi element niza. Kažemo da se niz u funkciju prenosi po adresi - pomoću pokazivača.**

- **Funkcija ne raspolaže s podatkom o broju elemenata niza. Zadatak je programera da tu vrijednost, ukoliko je potrebna, predvidi kao dodatni parametar funkcije.**

Primjer: obje funkcije `print` imaju isto djelovanje i mogu se pozvati s istim argumentima:

```
void print(int x[], int size)
{
for (int i = 0; i < size; i++)
     cout << x[i] << endl;
}
```

```
void print( int *x, int size)
{
  while (size-- > 0)
    cout << *x++ << endl;
}
```

```
/* poziv funkcije print je isti za obje definicije */

            int niz[10], size=10;
               . . . . .
            print(niz, size);
               . . . .
```

8.3 Dinamičko alociranje nizova

Najčešća je primjena pokazivača u dinamičkom alociranju nizova. Ono se se provodi pomoću operatora `new` i `delete`.

Sintaksa za alociranje niza je:

***pokazivačka_varijabla* = `new` *oznaka_tipa*[*izraz*];**

Veličina alocirane memorije iznosi:

sizeof(*oznaka_tipa*) * *izraz*

Podrazumijeva sa da izraz mora rezultirati prirodnim brojem.

Ako se alokacija izvrši uspješno, adresa alocirane memorije se pridjeljuje *pokazivačkoj_varijabli.*

Ako se ne može izvršiti alokacija vrši poziv funkcije za prihvat iznimki (exception handler - taj slučaj će biti pojašnjen kasnije).

Kada nije implementiran mehanizam prihvata iznimki *pokazivačkoj_varijabli* se pridjeljuje vrijednost 0 (NULL) .

Danas, svi kompilatori imaju ugrađen mehanizam prihvata iznimki.

Primjerice, s

```
int n = 10;
float * A= new float[n];
```

alocira se memorija za n elemenata tipa float. Adresa te memorije se pridjeljuje pokazivaču A.

Pomoću tog pokazivača pristupamo elementima niza na isti način kako bi to radili sa statički alociranim nizovima, primjerice petljom

```
for (int i = 0; i < n; i++)
{
    A[i] = i * i;
}
```

elementi niza dobivaju vrijednost $A_i=i^2$.

Nakon završenog rada s nizom, odnosno kada se izlazi iz bloka u kojem je niz alociran, niz treba dealocirati.

Sintaksa iskaza za dealokaciranje niza je:

`delete []` ***pokazivač_niza*;**

```
void main()
{
    int n;                          // Broj elemenata niza
    float *A;                       // Pokazivač niza

    cout << "Unesite broj elemenata niza? "; cin >> n;

    A = new float [n];   // niz A od n float elemenata.

    for (int i = 0; i < n; i++)
        A[i] = i * i;                       //  A[i] = i^2.

    // Ispis vrijednosti elemenata niza.
     for (int i = 0; i < n; i++)
         cout << "A[" << i << "] == " << A[i] << endl;

     delete [] A;  // Dealokacija niza.
}
```

Primjer ispisa:

```
    Unesite broj elemenata niza? 5
    A[0] == 0
    A[1] == 1
```

Pokazivači na funkcije

Funkcije su također memorijski objekti pa se može deklarirati i inicijalizirati pokazivače na funkcije.

Pravilo je da se za funkciju imena F, koja je deklarirana (ili definirana) u obliku:

oznaka_tipa **F (*list_ parametara*);** **// int F(int)**

pokazivač na tu funkciju, imena pF, deklarira u obliku:

oznaka_tipa **(*pF) (*list_ parametara*);** **// int (*pF) (int)**

Ime funkcije, napisano bez zagrada predstavlja adresu funkcije, pa izraz:

pF = F;

inicira pokazivač pF na adresu funkcije F. Kada je pokazivač iniciran, indirekcijom pokazivača može se izvršiti poziv funkcije u obliku:

(*pF)(*lista_argumenata*);

ili još jednostavnije, sa

pF(*lista_argumenata*);

jer, oble zagrade predstavljaju operator poziva funkcije (kompilator sam vrši indirekciju pokazivača, ako se iza njega napišu oble zagrade).

Pokazivač na funkciju može biti i argument funkcije, primjerice funkcija

```
void Print( double (*pMatFun)(double), double  x)
{
    cout << pMatFun (x);
}
```

se može koristiti za ispis vrijednosti matematičkih funkcija.

```
Print(sin, 3.1);  /* ispisuje se sinusne funkcije
                     za vrijednost argumenta 3.1*/
```

Često se koriste nizovi pokazivača na funkciju. Primjerice, deklaracijom

```
#include math.h
double (*pF[4])(double) = {sin, cos, tan, exp};
```

inicijaliziran je niz od 4 elementa koji sadrže pokazivač na funkciju kojoj je parametar tipa double i koja vraća vrijednost tipa double.

`x = (*pF[1])(3.14)` **ekvivalentna naredbi** `x= cos(3.14)`**.**

```
/* program: pfun.c - korištenje niza pokazivača na funkciju */
 #include <iostream>
 #include <cmath>
 using namespace std;

 double Kvadrat (double x) {return x*x;}

 int main()
 {
    double val=1;
    int izbor;
    double (*pF[3])(double)={Kvadrat, sin, cos};

    cout <<  "Upisi broj:";
    cin >> val;

    cout <<  "\n(1)Kvadrat  \n(2)Sinus  \n(3)Kosinus \n";
    cout <<  "\n   Odaberi 1, 2 li 3\n";
    cin >> izbor;

    if (izbor >=1 && izbor <=3)
       cout << "\nRezultat je "
            << (*pF[izbor-1])(val) << endl;
    return 0;
 }
```

Prihvat greške pri alokaciji memorije

Promotrimo program:

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

int main()
{
    char* p = new char[2000000000];

    cout << "kraj, p = " << long(p) << endl;
    return 0;
}
```

U ovom programu se pokušava alocirati 2GB memorije. To nije moguće kod većine današnjih računala. Ovisno o implementaciji kompilatora, rezultat izvršenja programa će biti različit:

1. Kada se primjeni Microsoft VC kompilator ver. 6 program će završiti s porukom:

```
Kraj, p=0
```

dakle, taj kompilator u slučaju da se ne može izvršiti alokacija memorije za vrijednost pokazivača p postavlja vrijednost 0 (NULL).

2. Kada se primjeni Microsoft VC kompilator ver. 7 program će završiti s porukom:
This application has requested the Runtime to terminate it in an unusual way. Please contact the application's support team for more information.

Nedostatak slobodne memorije, je iznimna situacija, pa je ostavljeno da se taj slučaj obradi posebnim mehanizmom koji se naziva "exception handler". Njega ćemo upoznati kasnije.

Prema novom ISO standardu iz 1988. godine, koji je implementiran u Visual C ne pokreće se mehanizam iznimki (ne prekida se program) i vraća NULL pokazivač, samo ako se alokacija izvrši s argumentom (nothrow), primjerice:

```
char* p = new (std::nothrow) char[1000000];
```

Ukoliko se ne koristi argument (nothrow) poželjno je da korisnik definira i registrira funkciju za prihvat pogreške pri alokaciji memorije.

Prihvat greške pri alokaciji memorije

C++ run-time sustav osigurava da se aktivira funkcija za prihvat greške pri alokaciji memorije. Tu funkciju se može redefinirati na sljedeći način:

1. **Ta funkcija mora imati prototip bez argumenata i bez povrata vrijednosti, primjerice funkcija:**

```
void no_memory()
{
    cerr << "operator new: nema slobodne memorije\n";
    exit(1);
}
```

će dojaviti da nema slobodne memorije i prekinuti izvršenje programa.

2. **Funkcija za prihvat greške se mora registrirati pomoću funkcije `set_new_handler` koja je deklarirana u `<new.h>`:**

```
set_new_handler(no_memory);
```

Nakon toga, pri bilo kojoj alokaciji memorije, ukoliko se pojavi greška, bit će pozvana funkcija `no_memory()`.

Kompletni primjer je dan u programu new1.cpp.

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

void no_memory()
{
    cerr << "operator new: nema slobodne memorije\n";
    exit(1);
}


int main()
{
    set_new_handler(no_memory);
    char* p = new char[1000000000];
    cout << "Kraj, p = " << long(p) << endl;
    return 0;
}
```

Napomena: preporučuje se definirati prihvat greške alokacije memorije u svim programima koji koriste veće memorijske objekte.

ASCIIZ string

Cilj je da nizove znakova tretiramo kao varijable. Na taj način moći ćemo vršiti obradu tekstualnih zapisa. U C++ jeziku se koriste dva načina rada sa stringovima.

- **Prvi način je da se string tretira kao niz znakova kojem posljednji znak mora biti jednak nuli. Ovakovi stringovi se nazivaji i ASCIIZ stringovi (ASCII +zero). U standardnoj biblioteci C jezika postoji više funkcija za rad s ovakovim stringovima, a deklarirane su u datoteci <string.h> odnosno <cstring>.**

- **Drugi je način, standardiziran u C++ jeziku, da se koristi klasa string koja je deklarirana u datoteci <string>.**

ASCIIZ string je svaki niz koji se deklarira kao:

- **niz znakova (npr. `char str[10];`) , ili kao**
- **pokazivač na znak (`char *str;`),**

pod uvjetom da se pri inicijalizaciji niza, i kasnije u radu s nizom uvijek vodi računa o tome da poljednji element niza mora biti jednak nuli.

```
#include <iostream>
#include <cstring>     // deklaracija funkcije strlen
using namespace std;

void main()
{
char hello[14] = { 'H', 'e', 'l', 'l', 'o',
',', ' ', 'W', 'o', 'r', 'l', 'd', '!', '\0' };
cout << hello << endl;

cout << "Duljina stringa je "  << strlen(hello) << endl;
}
```

`size_t strlen(char *s)` **funkcija koja daje duljinu stringa (bez 0-znaka)**

String se može inicijalizirati i pomoću literalnog stringa:

```
char hello[] = "Hello, World!";  // kompilator rezervira
memoriju
```

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

unsigned strlen(char *s)
{
   unsigned len = 0;
   while (*s++ != '\0') //petlja se prekida za*s==0, inače
         len++;         //inkrementira se brojač znakova
    return len;         //len sadrži duljinu stringa
}

int main()
{
char hello[] = "Hello, world!";
cout << hello << endl;
cout << "Duljina stringa je " << strlen(hello) << endl;
cout << "Zauzece memorije je" << sizeof(hello)*sizeof(char)
     << " bajta" << endl;
}
```

Nakon izvršenja programa, dobije se poruka:

```
Hello, World!
Duljina stringa je 13
Zauzece memorije je 14 bajta
```

U standardnoj biblioteci postoji niz funkcija za manipuliranje sa stringovima . One su deklarirane u datoteci `<cstring> ili <string.h>.`

```
size_t strlen(const char *s)

char *strcpy(char *s, const char *t)
char *strncpy(char *s, const char *t, size_t n)

char *strcat(char *s, const char *t)
char *strncat(char *s, const char *t, size_t n)

int strcmp(const char *s, const char *t)

char *strchr(const char *s, int c)
char *strrchr(const char *s, int c)
char *strstr(const char *s, const char *t)


char *strpbrk(const char *s, const char *t)
char *strerror(int n)
char *strtok(char *s, const char *sep)
```

Primjer: `strcmp` služi usporedbi dva stringa `s1` i `s2`. Deklarirana je s:

```
int strcmp(const char *s1, const char *s2)
```

Funkcija vraća vrijednost 0 ako je sadržaj oba stringa isti, negativnu vrijednost ako je `s1` leksički manji od `s2`, ili pozitivnu vrijednost ako je `s1` leksički veći od `s2`. Leksička usporedba se izvršava znak po znak prema kodnoj vrijednosti ASCII standarda.

```
int strcmp( char s1[], char s2[])
{
   int i = 0;
   while(s1[i] == s2[i] && s1[i] != '\0')
      i++;
   if (s1[i] < s2[i])      return -1;
   else if (s1[i] > s2[i])return +1;
   else                   return 0;
}
```

U for petlji se uspoređuje da li su znakovi s istim indeksom isti i da li je dostignut kraj niza (znak '\0'). Kad petlja završi, ako su oba znaka jednaka, to znači da su i svi prethodni znakovi jednaki. U tom slučaju funkcija vraća vrijednost 0. U suprotnom funkcija vraća 1 ili –1 ovisno o numeričkom kodu znakova koji se razlikuju.

Verzija s inkrementiranjem pokazivača:

```
int strcmp(char *s1, char *s2)
{
        for(; *s1 == *s2; s1++, s2++)
        {
                if(*s1 == '\0')  return 0;
        }
        return *s1 - *s2;
}
```

Verzija s pokazivačima predstavlja optimiranu verziju prethodne funkcije. Uočite da se u u for **petlji ne koristi izraz inicijalizacije petlje. Najprije se uspoređuju znakovi na koje pokazuju s1 i s2. Ako su znakovi isti inkrementira se vrijednost oba pokazivača. Tijelo petlje će se izvršiti samo u slučaju ako su stringovi isti (tada da s1 i s2 konačno pokazuju na znak '\0'). U suprotnom petlja se prekida, a funkcija vraća numeričku razliku ASCII koda znakova koji se razlikuju.**

Koja se verzija brže izvršava?