Osnove programiranja II
(Programski jezik C)

Borut Robič

UNIVERZA V LJUBLJANI
Fakulteta za računalništvo in informatiko

O predmetu

Vsebina

1. Uvod

1.0 Zgodovina

- Zgodovina (1)
  - 1972, C se pojavi kot orodje (splošen progr. jezik) pri razvoju Unix
  - Bell Laboratories, ZDA, Unix na PDP-11, Dennis Ritchie

  - Od 1978 dalje obstaja več prevajalnikov za C za druge rač.
  - Posledica: narečja.
  - Posledica: manjša prenosljivost programov.

  - 1983, Ameriški nacionalni institut za standarde (ANSI)
  - Prične se oblikovanje standarda za C
  - 1988 objavi standardni, ANSI C

1.0 Zgodovina

- Zgodovina (2)
  - Pisanje obsežne programske opreme terja nove načrtovalske pristope
  - Pojavi se objektno usmerjeno programiranje

  - ANSI C je potrebno dopolniti (da bo podpiral obj. usm. progr.)
  - AT&T, ZDA
  - Pojavi se C++.

1.0 Zgodovina

- Zgodovina (3)
  - Pojavljajo se nove zahteve:
    - produktivnost programerjev
    - zanesljivost progr. opreme
    - preglednost (in s tem zanesljivost) progr. opreme
    - možnost spletnega pogramiranja
    - prenosljivost progr.opreme
  - Pojavi se Java

1.1. Začetek (1)

```
// Program v C, ki izpiše neko besedilo
#include <stdio.h>
main()
{
 printf("Dober dan!\n");
}
```

– Kako program poženemo?

1.1. Začetek (2)

- Vsak C program sestavljajo *funkcije*
  - Prva se prične izvajati funkcija `main`
  - Funkcija lahko kliče druge funkcije

  - Funkcija je bodisi definirana
    - v našem programu ali pa v knjižnici.
    - Npr. `printf` je v knjižnici `stdio.h`
    - `#include <stdio.h>` ukaz prevajalniku, naj upošteva knjižnico `stdio.h`

  - Funkcija ima parametre
    - navedeni so med oklepajema, (in)
  - Klicana funkcija bo izvedla svoje stavke
    - navedeni so v njenem telesu, t.j. med zavitima oklepajema { in }

  - Klic funkcije
    - navedemo ime funkcije in v oklepajih seznam argumentov
    - `printf ("Dober dan!\n");` ima en argument
      - » `"Dober dan!"` je znakovni niz,
      - » `\n` je ubežna sekvenca

1.1. Začetek (3)

- Nekatere ubežne sekvence
  - \a … Zvočni signal
  - \b ... Pomik nazaj (backspace)
  - \f ... Nova stran (form feed)
  - \n ... Pomik v novo vrsto (newline)
  - \r ... Pomik na začetek vrstice (carriage return)
  - \t ... Vodoravni odmik (horizontal tab)
  - \v ... Navpični odmik (vertical tab)
  - \\ ... Izpis poševne črte (backslash)
  - \? ... -ii- vprašaja
  - \’ ... -ii- enojnega narekovaja
  - \” ... -ii- dvojenega narekovaja
  - \ooo ... Število ooo je v osmiškem sistemu
  - \xhh ... Število hh je v šestnajstiškem sistemu

1.1. Začetek (4)

- Nekatere (druge) funkcije iz `stdio.h`
  - `prinf()`
  - `scanf()`
  - `getchar()`
  - `putchar()`
  - `.....`
- Nekatere (druge) standardne knjižnice
  - `stdio.h`
  - `ctype.h`
  - `string.h`
  - `limits.h`
  - `float.h`
  - `.....`

1.2. Spremenljivke in aritmetika (1)

Naloga: izpiši tabelo (F,C) po formuli C = 5/9(F-32) za F = 0,20,40, ...,300.

Program.

```
#include <stdio.h>
/* izpis tabele (F,C) za F = 0,20,40,...,300  */
main()
{
 int f,c;
 int sp,zg,korak;
 sp = 0;
 zg = 300;
 korak = 20;
 f = sp;
 while (f <= zg)
  {
    c = 5*(f-32)/9;
    printf("%d\t%d\n",f,c);
    f = f + korak;
  }
}
```

1.2. Spremenljivke in aritmetika (2)

Opombe:

- Komentarji
- Deklaracije spremenljivk
- Podatkovni tipi
- Prireditveni stavki
- Zanka while
- Izracun `5*(f-32)/9` in `5/9*(f-32)`
- `printf(...)` in kontrolni nizi
- Pisanje oklepajev { in }
- Nekatere slabosti v prejšnjem programu
  - Npr. , slaba natančnost izpisanih vrednosti C
  - npr pri F = 0 se izpiše -17 namesto natančnejša -17.8

1.2. Spremenljivke in aritmetika (3)

Popravljeni program. Na primer:

```
#include <stdio.h>
/* izpis tabele (F,C) za F = 0,20,40,...,300  */
main() {
 float f,c;
 int sp,zg,korak;
 sp = 0; zg = 300; korak = 20;
 f = sp;
 while (f <= zg) {
   c = (5.0/9.0)*(f-32.0);
   printf("%3.0f %6.1f \n",f,c);
   f = f + korak;
 }
}
```

1.3. Zanka `for` (1)

Naloga: ponovno izpiši tabelo (F,C) .

Opomba. Uporabi zanko `for`.

Program.

```
#include <stdio.h>
main() {
 int f;
 for(f=0; f<=300; f=f+20)
  printf("%3d %6.1f \n",f, (5.0/9.0)*(f-32.0));
}
```

Opombe.

- Pomen stavka `for`
- Argument funkcije je lahko (aritmetični) izraz
- Dolžina programa, število spremenljivk

1.4. Simbolne konstante (1)

Opomba. Konstante raje poimenujemo s simbolnimi imeni.

Naloga. Ponovno izpiši tabelo (F,C) .
Opomba. Uporabi simbolne konstante.
Program.

```
#include <stdio.h>
#define SP 0
#define ZG 300
#define KORAK 20
main() {
 int f;
 for(f=SP; f<=ZG; f=f+KORAK)
  printf("%3d %6.1f \n",f, (5.0/9.0)*(f-32.0));
}
```

Opombe:

- Sintaksa `#define`
- Kaj se dogaja ob prevajanju programa

1.5. Primeri

Opomba. Knjižnica `stdio.h` nudi tudi funkciji `getchar` in `putchar`

- `getchar:`
  - bere naslednji znak s standardnega vhoda (tipkovnica)
  - in vrne
    - ta znak (natančneje: njegovo kodo)

      ali pa

      vrednost EOF, če na vhodu ni bilo več znakov (običajno je EOF = -1)
  - Primer:

```
int c;
c = getchar();
```

- `putchar:`
  - izpiše svoj argument kot znak na standardni izhod (zaslon)
  - in vrne
    - izpisani znak (natančneje: njegovo kodo)

      ali pa

      vrednost EOF, če je bil izpis argumenta neuspešen
  - Primer:

```
int c, izpisano;
c = ...
izpisano = putchar(c);
```

1.5.1. Kopiranje(1)

Naloga: kopiranje vhoda na izhod.

Algoritem:
preberi znak z vhoda
dokler (prebrani znak ni EOF) {
izpiši prebrani znak;
preberi nov znak z vhoda;
}

Program.

```
#include <stdio.h>
main() {
  int c;
  c = getchar();
  while (c != EOF) {
    putchar(c);
    c = getchar();
  }
}
```

Opombe:

- !=
- Zakaj `int c`
- `c = f(); Izraz(...c...)` lahko nadomestimo z `Izraz(...(c=f()) ...)`

1.5.1. Kopiranje(2)

Naloga: kopiranje vhoda na izhod.

Opomba: `c = f(); Izraz(...c...)` nadomestimo z `Izraz(...(c=f()) ...)`

Program.

```
#include <stdio.h>
main() {
  int c;
  while ( (c = getchar()) != EOF) {
    putchar(c);
  }
}
```

Opombe:

- Prednost operacij in oklepaji

1.5.2. Štetje znakov(1)

Naloga: preštej vhodne znake.

Program.

```
#include <stdio.h>
main() {
  long nc;
  nc = 0;
  while (getchar() != EOF)
    ++nc;
  printf("%ld \n", nc);
}
```

Opombe:

- `++nc, --nc, nc++, nc--`
- `long nc`

1.5.2. Štetje znakov(2)

Naloga: preštej vhodne znake.

Opomba: uporabi števec tipa double in zanko for.

Program.

```
#include <stdio.h>
main() {
  double nc;
  for(nc=0; getchar() != EOF; ++nc)
   ;
  printf("%.0f \n", nc);
}
```

Opombe:

- Telo `for` je prazno; čitljivost in jedrnatost.
- Določilo `%f, %.0f`

1.5.3. Štetje vrstic(1)

Naloga: preštej vhodne vrstice.

Opomba: vrstice so ločene z \n (pomik v novo vrstico).

Algoritem: štej, kolikokrat se na vhodu pojavi \n

Program.

```
#include <stdio.h>
main() {
  int c, n;
  while( (c=getchar()) != EOF )
    if (c == '\n')
     ++n;
  printf("%d\n", n);
}
```

Opombe:

- Relacija enakosti = =

1.5.3. Štetje vrstic(2)

Opombe:

– Znakovna konstanta (npr. ‘A’);
  - njena vrednost je koda znaka A (v računalnikovem znakovnem naboru)
  - npr. V ASCII naboru: ‘A’ ... 65,
    ‘a’ …97,
    ‘0’ … 48 itd.
  - tudi ubežne sekvence lahko vklenemo med ’ ’. ASCII kode nekaterih so:
    ‘\a’ … 7
    ‘\b’ …8
    ‘\t’ …9
    ‘\n’ … 10
    ‘\v’ … 11
    ‘\f’ … 12
    ‘\r’ … 13
  - ASCII tabela

DEC	OKT	OZNAKA	OPIS
7	007	BEL	Zvočni signal, CTRL/G, \a
65	101	A	
….	……	….	…..

1.5.4. Štetje besed(1)

Naloga: preštej vhodne vrstice, besede in znake.

Opomba: beseda je poljubno zaporedje znakov, brez presledka, odmika ali znaka za pomik v novo vrsto.

Algoritem.

- Štetje znakov: šteje po vrsti vhodne znake.
- Štetje vrstic: šteje znake `\n`
- Štetje besed: šteje, kolikokrat se je začelo branje kake besede.
  Zato mora stalno vedeti, kje je (*znotraj* ali *zunaj* besede)
  Zato šteje spremembe iz *zunaj* v *znotraj* .

1.5.4. Štetje besed(2)

Program.

```
#include <stdio.h>
#define ZNOTRAJBESEDE 1
#define ZUNAJBESEDE 0
main() {
  int c, nl, nw, nc, kje;
  kje = ZUNAJBESEDE;
  nl = nw = nc = 0;
  while( (c=getchar()) != EOF ) {
     ++nc;
     if (c == '\n')
       ++nl;
     if(c == ' ' || c == '\t' || c == '\n')
       kje = ZUNAJBESEDE;
     else if (kje == ZUNAJBESEDE) {
       kje = ZNOTRAJBESEDE;  ++nw;
     }
  }
  printf("%d %d %d \n", nl, nw, nc);
}
```

1.5.4. Štetje besed(3)

Opombe:

- Verižno prirejanje a=b=c;
- Disjunkcija ||, konjunkcija &&
- Izračun izrazov A||B||C ter A && B && C
- Stavek `if .... else`

1.6. Polja(1)

Naloga: preštej cifre, bele (presledek,odmik ali nova vrsta) in vse ostale znake .

Opomba: za štetje cifer uporabimo polje .

Program.

```
#include <stdio.h>
main() {
  int c, i, belih, ostalih;
  int cifer[10];
  belih = ostalih = 0;
  for(i=0; i<10; ++i)
    cifer[i]=0;
  while( (c=getchar()) != EOF )
    if (c >= '0'&& c <= '9')
      ++cifer[c - '0'];
    else if (c == ' '|| c == '\n' || c == '\t')
      ++belih;
    else
     ++ostalih;
    printf("cifer: ");
    for(i=0; i<10; ++i)
      printf("%d", cifer[i]);
    printf("belih: %d  ostalih: %d \n", belih, ostalih);
}
```

1.6. Polja(2)

Opombe:

- Deklaracija `int cifer[10]`
- V ASCII: če `(c >= '0' && c <= '9')`, potem je v `c` koda neke cifre; tedaj je `c-'0'` število, ki ga predstavlja ta cifra
- Večsmerna odločitev s stavkom `if ... else if ... else`

1.7. Funkcije(1)

Naloga: definiraj funkcijo `power(m,n)`, ki vrne m^n. Izpiši tabelo nekaj vrednosti.

Opomba: Doslej smo uporabljali samo že obstoječe funkcije (iz knjižnice).

Program.

```
#include <stdio.h>
int power(int m, int n);
main() {
  int i;
  for(i=0; i<10; ++i)
    printf("%d %d %d\n", i, power(2,i), power(-3,i));
  return 0;
}

int power(int b, int e)
{
  int j, rez;
  rez = 1;
  for(j=1; j<=e; ++j)
     rez = rez * b;
  return rez;
}
```

1.7. Funkcije(2)

Opombe:

- deklaracija funkcije
- definicija funkcije
- klic funkcije
- `return`
- Formalni parameter (parameter), dejanski parameter (argument)

1.8. Argumenti in prenos po vrednosti(1)

Uvod in razlaga:

- prenos po vrednosti
- polja
- uporabnost prenosa po vrednosti

Primer. Druga definicija funkcije `power`

Opomba: Parameter `e` uporabimo namesto `j`.

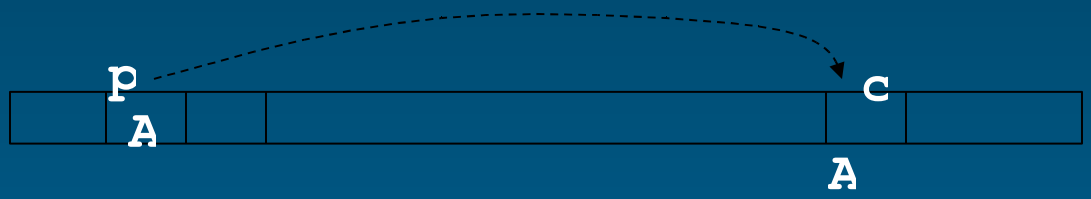
```
int power(int b, int e)
{
  int rez;
  for(rez=1; e>0; --e)
     rez = rez * b;
  return rez;
}
```

Opomba: spreminjanje argumentov (dejanskih parametrov) je možno (preko kazalcev).

1.8.a. Uvod v kazalce

Kazalec je spremenljivka, ki vsebuje naslov druge spremenljivke

- Primer. `char c;`
  `p` *kazalec na* `c;`



– Operator `&` vrne naslov dane spremenljivke (argumenta operacije `&`)
  - Primer. `p = &c;` /* v `p` se vpiše naslov spremenljivke `c` */

– operator `*` vrne spremenljivko, kamor kaže dani kazalec (argument operacije `*`)
  - Primer. `*p` /* pomeni spremenljivko, kamor kaže kazalec `p` */

– S pomočjo `*` tudi definiramo kazalce:

```
tipKazaneSpremenljivke *imeKazalca;
```

  – Primer. `char *p;` /* `p` je kazalec, ki bo kazal na spremenljivko tipa `char` */

– Primeri.

1.8.b. Kazalci in funkcijski argumenti(1)

Naloga: funkcija `menjaj(a,b)` naj zamenja vsebini svojih argumentov.

Poskus 1.

```
int a,b;
....
menjaj(a,b);
....
void menjaj(int x, int y)
{ int pomozna;
  pomozna = x;   x = y;   y = pomozna;
}
```

Opomba: Razlaga, zakaj je poskus 1 napačen.

Poskus 2.

```
int a,b;
....
menjaj(&a,&b);
....
void menjaj(int *px, int *py)
{ int pomozna;
  pomozna = *px;   *px = *py;   *py = pomozna;
}
```

Opomba: razlaga poskusa 2.

1.8.c. Primer: `scanf` in naslovi(1)

`int scanf(char *format, …)`

- Bere znake s standardnega vhoda,
- jih interpretira, kot narekuje `format`
- jih shrani na naslove, kot opredeljujejo ostali argumenti
- in vrne število uspešno prebranih argumentov (ali pa EOF, če na vhodu ni bilo ničesar).

Primer 1.

```
int dan, mesec, leto;
...
scanf("%d %d %d", &dan, &mesec, &leto);
```

Primer 2.

```
int dan, leto;
char mesec[15];
...
scanf("%d %s %d", &dan, mesec, &leto);
```

Primer 3.

```
#include <stdio>
main() {
  double vsota, novostevilo;
  vsota = 0;
  while( scanf("%lf", &novostevilo) == 1 )
   printf("\t %.2f\n", vsota += novostevilo)
}
```

Opombe.

1.9. Znakovna polja(1)

Naloga: izpiši najdaljšo vrstico vhodnega besedila.

Algoritem.

```
dokler(obstaja naslednja vrstica)
  če (je daljša od doslej najdaljše)
      shrani njo in njeno dolžino;
izpiši najdaljšo vrstico.
```

Funkciji, ki ju bomo definirali:

- `getline`
  - prebrala bo naslednjo vrstico z vhoda
  - vrnila bo dolžino prebrane vrstice, torej vsaj 1 (zaradi znaka NewLine, ki je na koncu vrstice) ali pa vrednost 0 (če na vhodu ni bilo ničesar)
- `copy` shranila bo prebrano vrstico na mesto najdaljše

1.9. Znakovna polja(2)

Program.

```
#include <stdio>
#define MAXLINE 1000
int getline (char line[],  int maxline);
void copy (char to[], char from[]);

main() {
 int len, max;
 char line[MAXLINE], longest[MAXLINE];
 max = 0;
 while(( len = getline(line, MAXLINE))>0 )
   if(len > max) {
     max = len;
     copy (longest, line);
   }
 if(max > 0)
   printf("%s", longest);
 return 0;
}
```

Razlaga in opombe.

1.9. Znakovna polja(3)

... nadaljevanje programa

```
int getline (char s[],  int lim)
{
  int c, i;
  for(i=0; i<lim-1 && (c = getchar()) != EOF && c != '\n'; ++i)
    s[i] = c;
  if (c  == '\n') {
    s[i]= c;  ++i;
  }
  s[i] = '\0';
  return i;
}
```

Razlaga in opombe.

```
void copy (char to[], char from[])
{
  int i = 0;
  while(( to[i] = from[] ) != '\0')
   ++i
}
```

Razlaga in opombe.

1.10. Zunanje spremenljivke(1)

Lokalne spremenljivke

- Deklaracija, dosegljivost, življenjska doba

Globalne (zunanje) spremenljivke

- Deklaracija, življenjska doba, dosegljivost (implicitno in z `external`)

1.10. Zunanje spremenljivke(2)

Naloga: izpiši najdaljšo vrstico vhodnega besedila.

Algoritem.

```
dokler(obstaja naslednja vrstica)
  če (je daljša od doslej najdaljše)
      shrani njo in njeno dolžino;
izpiši najdaljšo vrstico.
```

Opomba.

- Nekatere spremememenljivke bodo zunanje

1.10. Zunanje spremenljivke(3)

Program.

```
#include <stdio>
#define MAXLINE 1000
int max;
char line[MAXLINE], longest[MAXLINE];
int getline (char line[],  int maxline);
void copy (char to[], char from[]);

main() {
 int len;
 extern int max;
 extern char longest[];
 max = 0;
 while(( len = getline())>0 )
   if(len > max) {
     max = len;
     copy ();
   }
 if(max > 0)
   printf("%s", longest);
 return 0;
}
```

Razlaga in opombe.

1.10. Zunanje spremenljivke(4)

... nadaljevanje programa

```
int getline (void)
{
  int c, i;
  extern char line[];
  for(i=0; i<MAXLINE-1 && (c = getchar()) != EOF && c != '\n'; ++i)
    line[i] = c;
  if (c  == '\n') {
    line[i]= c;  ++i;
  }
  line[i] = '\0';
  return i;
}
```

Razlaga in opombe.

```
void copy (void)
{
  int i = 0;
  extern char line[], longest[];
  while(( longest[i] = line[] ) != '\0')
   ++i;
}
```

Razlaga in opombe.

2. Tipi, operatorji in izrazi

2.1. Imena spremenljivk

Omejitve pri izbiri imen spremenljivk in konstant:

- Poljubna kombinacija črk in cifer, ki s začne s črko ali podčrtajem
- Velike in male črke se razlikujejo
- Za lokalne signifikantnih 31 znakov
- Za zunanje vsaj 6
- Rezervirane besede
  - ne smejo nastopati kot imena
  - pisane morajo biti z malimi črkami
  - jih je 32

2.2. Osnovni podatkovni tipi

Osovni podatkovni tipi:

- `char ...` zlog
- `int ...` celo število
- `float ...` realno število (format v plavajoči vejici)
- `double ...` realno število z dvojno natančnostjo

- Kvantifikatorji :
  - `short` ... 16 bitov <= short <= int <= 32 bitov <= long
  - `long` ... namesto `long int` in `short int` lahko tudi `long` in `short`
  - `unsigned` ... >=0; aritmetika po modulu 2^n
  - `signed` ...

Opombe:

- `limits.h, float.h`

2.3. Konstante(1)

Konstante:

- Numerične
- Znakovne
- Konstantni izrazi
- Znakovni nizi
- Naštevne

Numerične

- int: npr. 1234
- Long: npr. 123456789L (na koncu L ali l)
- Unsigned: npr. 248U (na koncu U ali u)
- Mešano: npr. 1234567890UL (unsigned long)

- Realne: npr. 123.4 1e-2 0.1e-3
- Osmiške: npr. 037 (začnejo se z ničlo)
- Šestnajstiške: npr. 0x1F (začnejo se z 0x)

2.3. Konstante(2)

Znakovne konstante

- ‘x’ … num. vrednost znaka x v računalnikovem znakovnem naboru

  npr. `'0'` ... `48` (ASCII) `'A'` ... `65` (ASCII)

- Če x kontrolni znak, ga lahko predstavimo z ubežno sekvenco

  npr. `'\n'` ... `10` (ASCII) `'\0'` ... `0` (ASCII)

- Lahko uporabimo zapis \ooo ali \xhh

  npr. `#define VTAB '\013'` `#define BELL '\x7'`

Konstantni izraz

- je izraz, ki vsebuje samo konstante
- izračuna se v času prevajanja

2.3. Konstante(3)

Znakovni niz

- zaporedje znakov med dvojnima narekovajema, npr. `"Jaz sem niz"`
- Prazen niz: `""`
- Včasih rabimo ubežne sekvence , npr. `"\""` je niz z enim znakom: "
- `"Niz1""Niz2"` se med prevajanjem stakne v en niz.
- Notranja predstavitev
  - Polje znakov
  - ki mu prevajalnik doda znak `\0` (oznaka konca znakovnega niza)
- Dolžina znakovnega niza omejena le z razpoložljivim pomnilnikom
- `strlen(s)` ... Vrne dolžino niza s (brez znaka \0). Je v knjižnici `<string.h>`
- Lastna: npr.

```
int strlen (char s[])
{ int i=0;
  while(s[i]!='\0')
    ++i;
  return i;
 }
```

2.3. Konstante(4)

Naštevne konstante

- npr. `enum logicna {NE,DA}`
- Prva konstanta dobi vrednost 0, ostale po vrsti naslednje vrednosti
- razen če ne predpišemo drugače:

  npr. `enum meseci`
  `{JAN=1,FEB,MAR,APR,MAJ,JUN,JUL,AVG,SEP,OKT,NOV,DEC}`

  npr. `enum ubezne {BELL='\a', BACKSPACE='\b', TAB='\t'}`
- imena v različnih naštevanjih se morajo med sabo razlikovati

2.4. Deklaracije

Spremenljivke pred uporabo deklariramo.
Deklaracija spremenljivke:

- Tip Ime
- npr. `int x,y,z;`
  `char c, line[1000];`

Prednastavljanje spremenljivk

- Tip Ime = konstanta_ali_izraz
- npr. `int x = 0;`
  `char esc = '\\';`
  `int limit = MAXLINE + 1;`
- Če je avtomatska spr: prednastavi se ob vsakem vstopu v blok, kjer je dekl.; s poljubnim izrazom;
- Če ni avtomatska spr: prednastavi se le enkrat; le s konstanto; pred začetkom izvajanja; privzeto 0.

Kvalifikator `const` napove, da bo imela spr. stalno vrednost (poskus spreminjanja bo 'kaznovan')

- npr. `const double e = 2.71828182845905;`
  `const char msg[] = "opozorilo: ";`
- Lahko tudi pred parametrom funkcije, npr. `int strlen (const char s[]);`

Razlika med const-spremenljivko in konstanto, uvedeno z DEFINE

2.5. Aritmetični operatorji

Aritmetični

- Unarni: `-, +`
- Binarni: `*, / , %`
- Npr.:
  - `x % y` ... Ostanek deljenja x z y; npr. 9 % 4 je 1
  -
```
if ((leto % 4 == 0 && leto % 100 != 0) || leto % 400 == 0)
     printf("%d je prestopno", leto)
else printf("%d ni prestopno", leto);
```
  - `%` ne moremo uporabiti nad spr. tipa float, double

2.6. Relacijski in logični operatorji

Relacijski

- `<, <=, >, >=, ==, !=`
- Aritmetični imajo pred njimi prednost; npr. `i < a-1` pomeni i < (a-1)

Logični

- za sestavljanje logičnih izrazov
- logični izraz ima vrednost 0, kadar je neresničen; ter >0 (npr. 1), kadar je resničen
- ! negacija

  npr. `if (!veljavno)` pomeni isto kot `if (veljavno == 0)`
- `&&` konjunkcija (logični in)
- `||` disjunkcija (logični ali)

- Izračun logičnih izrazov se neha takoj, ko je znana končna vrednost

  Npr. `A1 &&..&& An` ima končno vrednost 0 takoj, ko se najde prvi Ai z vrednostjo 0

  Npr. `for(i=0; i<lim-1 && (c=getchar()) != '\n' && c!=EOF; ++i)`

  le če je prostor (če `i<lim-1`) se bere in obravnava nov znak `(..getchar()...)`
- Imajo prednost pred prireditvijo `=`

  Npr. `(c = getchar()) != '\n'` … beri znak v `c` in nato primerjaj `c` z `'\n'`

  `c = getchar() != '\n'` … beri znak, ga primerjaj z `'\n'`, rezultat vpiši v `c`

2.7. Pretvorbe med podatkovnimi tipi

Če v izrazu nastopajo operandi različnih tipov, se interno pretvorijo v enoten tip. Obstajajo pravila.
Primeri:

- `i + f` ... integer i se pred seštetjem 'pretvori' v float, kot je f
- Če je spr. tipa `char` v aritm.izrazu, se obravnava kot da bi bil tipa `int`

Npr. funkcija `atoi` naj pretvori niz cifer v ustrezno število. Npr. s [2 1 5 8 \0] --> n = 2158

```
int atoi (char s[])
 {
  int i, n=0;
  for(i=0; s[i]>='0' && s[i]<='9'; ++i)
    n = 10*n + (s[i]-'0');
  return n;
 }
```

Npr. funkcija lower naj pretvori velike v male črke (v ASCII)

```
int lower(int c)
 {
   if (c >='A' && c <= 'Z')
     return c + 'a' - 'A';
   else return c;
 }
```

2.8. Operatorja ++ in --

Inkrement in dekrement

- `++` … argumentu prišteje 1; `--` … argumentu odšteje 1
- Prefiksna raba: spremeni argument in ga nato uporabi v kontekstu
  Npr. `x = ++n` pomeni enako kot `n = n+1; x = n;`
- Postfiksna raba: uporabi argument v kontekstu in ga nato spremeni
  Npr. `x = n++` pomeni enako kot `x = n; n = n+1;`
- Uporaba le nad spremenljivko. Npr. `(i+j)++` ni dovoljeno.

Primer. funkcija `squeeze` naj iz niza s odstrani vse znake c.

```
void squeeze (char s[], int c)
{ int i, j;
  for(i=j=0; s[i]!='\0'; i++)
    if (s[i] != c)
      s[j++] = s[i];
  s[j] = '\0';
}
```

Razlaga in opombe.

Primer. funkcija `strcat(s,t)` naj nizu `s` pritakne niz `t`.

```
void strcat(char s[], char t[])
{ int i=0,j=0;
  while (s[i] != '\0')
    i++;
  while((s[i++] = t[j++]) != '\0')
    ;
}
```

Razlaga in opombe.

2.9. Bitni operatorji(1)

Bitni operatorji:

&	|	^	<<	>>	~
in	ali	ekskluzivni ali	pomik levo	pomik desno	eniški komplement

Primeri.

`n = n & 0177` ... v `n` briši vse razen spodnjih sedem bitov.

`n = n | POSTAVI` ... v `n` postavi vse bite, ki so postavljeni v konstanti POSTAVI.

Razlika med `x&y` in `x&&y` ter med `x|y` in `x||y` .

Pomiki: `x<<i` ... `x` pomakne `i` krat v levo; vstopijo ničle

`x>>i` ... Če `x` unsigned: `x` pomakne `i` krat v desno; vstopijo ničle

... Če `x` signed: `x` pomakne `i` krat v desno; kaj vstopi?

Odvisno: a) bit predznaka (aritmetični pomik)

b) ničla (logični pomik)

2.9. Bitni operatorji(2)

Eniški komplement

Primeri.

`n = n & ~0177` ... v `n` briši spodnjih sedem bitov.

Primer. Funkcija `getbits(x,p,n)` naj v `x` ohrani `n` bitov od vključno `p`-tega dalje

```
unsigned getbits(unsigned x, int p, int n)
{
 return( x >> (p+1-n)) & ~(~0 << n);
}
```

2.10. Prireditveni operatorji in izrazi

Naj bo `op` eden od operatorjev `+ - * / % << >> & ^ |` in `E1` ter `E2` izraza.
Tedaj pomeni `E1 op = E2` isto kot `E1 = (E1) op (E2)`.
Primeri.

`i+=2` ... `i` povečaj za 2

```
i+=j <==> i=i+j     i&=j <==> i=i&j
i-=j <==> i=i-j     i|=j <==> i=i|j
i*=j <==> i=i*j     i^=j <==> i=i^j
i/=j <==> i=i/j     i<<=j <==> i=i<<j
i%=j <==> i=i%j     i>>=j <==> i=i>>j
```

Pozor: `x*=y+1` pomeni `x=x*(y+1)`.
Primer. V celem številu preštej postavljene bite.
Algoritem: pomikaj `x` v desno in sproti štej enice na skrajno desnem mestu (LSB).

```
int bitcount (unsigned x)
{ int b;
  for(b=0; x!=0; x>>=1)
   if (x & 01)
     b++;
  return b;
}
```

2.11. Pogojni izrazi

Naj bodo `E1`, `E2` in `E3` izrazi. Tedaj pomeni `E1 ? E2 : E3` sledeče:

1) izračuna se vrednost `E1`;
2) če je ta različna od 0, se izračuna vrednost `E2`, ki postane vrednost pogojnega izraza
   sicer se izračuna vrednost `E3`, ki postane vrednost pogojnega izraza

Primeri.

```
z = (a>b)? a : b
```

```
float f,g;
int n;
... (n>0)?f:n ...
```

Izpis komponent polja `a`, po 10 komponent v vrstici, ločenih s presledkom:

```
for(i=0; i<n; i++)
  print("%6d%c", a[i], (i%10==9 || i==n-1)?'\n':' '));
```

2.12. Prioriteta in vrstni red izračuna

Uvod.

OPERATOR	ASOCIATIVNOST
`( ) [ ] -> .`	L
`! ~ ++ -- + - * & (type) sizeof`	D
`* / %`	L
`+ -`	L
`<< >>`	L
`< <= > >=`	L
`== !=`	L
`&`	L
`^`	L
`|`	L
`&&`	L
`||`	L
`?:`	D
`= += -= *= /= %= &= ^= |= <<= >>=`	D
`,`	L

Opombe in primeri.

3. Nadzor toka

3.1 Stavki in bloki

Stavek

- izraz, ki mu sledi podpičje
- npr. `X=0;`

Blok

- {stavki in deklaracije}
- npr.
```
{ x=0;
  i++;
  printf("%d", i); }
```
- V vsakem bloku lahko deklariramo spremenljivke

3.2 if-else

Sintaksa.

```
if(izraz)
  stavek1
else
  stavek2
```

Primeri.

Gnezdenje: naslednji `else` pripada najbolj notranjemu prostemu `if`

Primeri.

```
if(izraz1)
  if(izraz2)
    stavek1
  else
    stavek2
```

3.3 Večsmerne odločitve z if-else(1)

Sintaksa.

```
if(izraz1)
  stavek1
else if(izraz2)
  stavek2
else if(izraz3)
  stavek3
else                                  /* else del lahko manjka
  stavek4
```

3.3 Večsmerne odločitve z if-else(2)

Naloga: binarno iskanje elementa `x` v polju `v` z `n` komponentami.

Algoritem. Razlaga.

Program.

```
int binsearch(int x, int v[], int n)
{ int sp, zg, sr;
  sp = 0;  zg = n-1;
  while(sp <= zg)
  { sr = (sp + zg)/2;
    if(x< v[sr])
      zg = sr-1;
    else if(x > v[sr])
      sp = sr + 1;
    else
      return sr;
  }
  return -1;
 }
```

Razlaga in opombe.

3.4 Večsmerne odločitve s switch(1)

Sintaksa.

```
switch(izraz) {
  case k1: stavek1
  case k2: stavek2
  case k3: stavek3
  default stavek4
}
```

Opombe.

- `k1, k2, ...` so konstantni izrazi s celoštevilsko vrednostjo
- `default` del je lahko izpuščen

Primer.

3.4 Večsmerne odločitve s switch(2)

Naloga: preštej cifre, bele (presledek,odmik ali nova vrsta) in vse ostale znake .
Opomba: primer iz točke 1.6.

Program.

```
#include <stdio.h>
main() {
  int c, i, belih, ostalih, cifer[10];
  belih = ostalih = 0;
  for(i=0; i<10; i++)
    cifer[i]=0;
  while( (c=getchar()) != EOF ) {
    switch(c) {
      case '0': case '1': case '2': case '3': case '4': case '5': case '6':
      case '7': case '8': case '9': cifer[c - '0']++; break;
      case ' ': case '\n': case '\t': ++belih; break;
      default: ++ostalih; break;
    }
    printf("cifer: ");
    for(i=0; i<10; ++i)
      printf("%d", cifer[i]);
      printf("belih: %d  ostalih: %d \n", belih, ostalih);
}
```

Razlaga in opombe

3.5 Zanki while in for(1)

Sintaksa.

```
while(izraz)
  stavek
```

```
for(izraz1; izraz2; izraz3)
   stavek
```

Opombe.

- `izraz1, izraz2, izraz3` ... so lahko izpuščeni. Potrebna pa so podpičja!
- Če `izraz2` manjka, se šteje kot resničen.

Primeri.

3.5 Zanki while in for(2)

Naloga: funkcija `atoi` naj pretvori niz cifer v ustrezno število.

Opomba. Posplošen primer iz 2.7: upošteva predznake in presledke.

Npr. s [- 2 1 5 8 \0] --> n = -2158

Program.

```
include <ctype.h>
int atoi (char s[])
{
 int i, n, sign;
 for(i=0; isspace(s[i]); i++)
   ;
 sign = (s[i]=='-')?-1:1;
 if(s[i]=='+' || s[i]=='-')
   i++;
 for(n=0; isdigit(s[i]); i++)
   n = 10*n + (s[i]-'0');
 return sign*n;
}
```

Razlaga in opombe.

3.5 Zanki while in for(3)

Uvod: operacija 'vejica'

Naloga: funkcija `reverse(s)` naj obrne niz `s`.
Opomba. Npr. s [2 1 5 8] --> s [8 5 1 2]
Algoritem. Razlaga.
Program.

```
include <string.h>
void reverse(char s[])
{
 int c, i, j;
 for(i=0,j=strlen(s)-1; i<j; i++,j--) {
   c = s[i]; s[i] = s[j]; s[j] = c;
 }
}
```

Razlaga in opombe.

Primeri.

3.6 Zanka do_ while

Sintaksa:
```
do
   stavki
while(izraz);
```

Naloga: funkcija `itoa(n,s)` naj pretvori celo število `n` v znakovni niz `s`.
Opomba. Npr. n = 5739 ------> s [5 7 3 9 \0]. To je obratna funkcija od `atoi` (točka 3.5).
Algoritem. Razlaga.
Program.

```
void itoa(int n, char s[])
{int i, predznaceno;
 if ((predznaceno = n) < 0)
   n = -n;
 i = 0;
 do {
   s[i++] = n%10 + '0';
 while((n/=10)>0);
 if(predznaceno < 0)
   s[i++] = '-';
 s[i] = '\0';
 reverse(s);
}
```

Razlaga in opombe.

3.7 break, continue (1)

Opombe. `break` povzroči izstop iz tekoče zanke (while,for,dowhile) ali stavka switch

Naloga: funkcija `trim(s)` naj iz vhodne vrstice `s` odstrani končne presledke.

Opomba. Npr. s [a v t o \0] ---> s [a v t o \0] .

Algoritem. Razlaga.

Program.

```
int trim(char s[])
 {
  int n;
  for(n=strlen(s)-1; n>=0; n--)
   if(s[n]!=' ' && s[n]!= '\t' && s[n]!='\n')
     break;
   s[n+1] = '\0';
   return n;
 }
```

Razlaga in opombe.

3.7 break, continue (2)

Opombe. `continue` povzroči:

- v `while` in `dowhile` zanki takojšnje preverjanje pogoja
- `for(a;b;c)` zanki takojšen skok na del `c`

Naloga: v polju `a` povečaj vsako pozitivno komponento za 1.

Algoritem. Razlaga.

Izsek programa.

```
 ...
for(i=0; i<n; i++)
{
  if(a[i]<=0)
     continue;
  a[i]++;
}
...
```

3.8 goto(1)

Sintaksa.
```
    goto L;
     ...
L:  ...
```

Primer. Radikalna prekinitev izvajanja globoko gnezdenih zank.

```
        for(...)
          for(...)
            for(...)
            {
              ...
              if(katastrofalnanapaka)
                goto Potop;
            }
            ...
Potop: koda za reševanje po potopu
```

Opombe
- Prednosti programov brez goto stavkov
- Goto v resnici nepotreben; se da nadomestiti z drugimi konstrukti

3.8 goto(2)

Naloga: ugotovi, ali imata polji `a` in `b` kaki komponenti enaki.

Opomba. Iskanje prekinemo, če najdemo dve enaki komponenti.

Izsek programa:

```
      for(i=0; i<n; i++)
       for(j=0; j<m; j++)
         if(a[i]==b[j])
           goto Nasel;
      /* ce pride do tu, a in b nimata enakih komp. */
         ...
Nasel: /* enaki sta npr. a[i] in b[j]. */
```

Brez goto:

```
        nasel = 0;
        for(i=0; i<n && !nasel; i++)
          for(j=0; j<m && !nasel; j++)
            if(a[i]==b[j])
              nasel=1;
        if(nasel)
          /* enaki sta a[i-1] in b[j-1] */
        else
          /* ni enakega para */
```

4. Funkcije in zgradba programa

4.1 Osnovno o funkcijah (1)

Funkcije:

- z njimi večja opravila razbijemo na več manjših
- poznati je treba *kaj* naredi in *katere podatke* potrebuje
- so v eni ali več datotekah
- uporabimo jih lahko še v drugih programih

Naloga: izpiši vsako vhodno vrstico, ki vsebuje dani vzorec znakov.

Algoritem:

```
while(obstaja_vhodna_vrstica) ..... getline (glej 1.9)
  if(vsebuje_dani_vzorec)
    izpisi_vrstico;          ..... printf
```

- Algoritem ima 3 lepo definirane in ločene dele
- Za 2 že poznamo oz. smo že sestavili funkcijo
- Ostaja le še funkcija za iskanje vzorca `t` v dani vrstici `s`:

`strindex(s,t)` =
- indeks komponente v `s`, kjer se začenja `t`
- -1, če `s` ne vsebuje vzorca `t`

4.1 Osnovno o funkcijah (2)

Program:

```
#include <stdio.h>
#define MAXLINE 1000
int getline(char line[], int max);
int strindex(char source[], char searchfor[]);
char pattern[] = "poletje";

main()
{ char line[MAXLINE];
  int found = 0;
  while(getline(line,MAXLINE) > 0)
    if (strindex(line,pattern)>=0) {
      printf("%s",line);
      found++;
    }
  return found;
}
```

4.1 Osnovno o funkcijah (3)

Nadaljevanje programa:

```
int getline(char s[], int lim)
{ int c, i;
  for(i=0; i<lim-1 && (c = getchar()) != EOF && c != '\n'; ++i)
    s[i] = c;
  if (c  == '\n') {
    s[i]= c;  ++i;
  }
  s[i] = '\0';
  return i;
}
```

... Glej točko 1.9

```
int strindex(char s[], char t[])
{
  int i,j,k;
  for(i=0; s[i]!='\0'; i++) {
    for(j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
      ;
    if (k>0 && t[k]== '\0')
      return i;
  }
  return -1;
}
```

... razlaga algoritma za strindex

4.2 Funkcije, ki vračajo necela števila (1)

Uvod:

- Doslej uporabljene funkcije vračajo celo število ali pa ničesar
- V praksi rabimo tudi funkcije, ki vračajo rezultate tudi drugih tipov
- Npr. sin, cos, sqrt, ...

A. Kako jih definiramo? V definiciji funkcije pred njenim imenom navedemo tip rezultata.

Naloga: funkcija `atof(s)` pretvori znakovni niz `s` (ki vsebuje cifre, predznak in dec.piko) v število.

Opomba. To je razširitev funkcije `atoi` (glej 3.5) .

Program.

```
#include <ctype.h>
double atof(char s[])
{ double val, power;
  int i, sign;
  for(i=0; isspace(s[i]); i++)
   ;
  sign = (s[i] == '-') ? -1 : 1;
  if (s[i] == '+' || s[i] == '-')
    i++;
  for(val = 0.0; isdigit(s[i]); i++)
    val = 10.0 * val + s[i] - '0';
  if (s[i] == '.')
    i++;
  for(power = 1.0; isdigit(s[i]); i++) {
    val = 10.0 * val + s[i] - '0';
    power *= 10.0;
  }
  return sign * val / power;
}
```

4.2 Funkcije, ki vračajo necela števila (2)

B. V okolici moramo deklarirati, kakšnega tipa je rezultat funkcije.
Naloga:. beri vhodne vrstice, jih sproti pretvarjaj v realna števila in izpisuj delne vsote.
Program.

```
#include <stdio.h>
#define MAXLINE 100
main()
{
  double sum, atof(char []);
  char line[MAXLINE];
  int getline(char line[], int max);
  sum = 0;
  while (getline(line, MAXLINE) > 0)
    printf("\t%g\n", sum += atof(line));
  return 0;
}
```

Opombe.

Naloga: definiraj funkcijo `atoi` s pomočjo funkcije `atof`.
Program.

```
int atoi (char s[])
{
  double atof(char s[]);
  return (int) atof(s);
}
```

4.3 Kalkulator (1)

Naloga: sestavi kalkulator za operacije `+,-,*,/.` Enačaj = naj izpiše rezultat.

Uvod: postfikni zapis

- infiksni: `(1-2)*(4+5) =` postfiksni: `1 2 - 4 5 + * =`
- V postfiksnem zapisu niso potrebni oklepaji
- Izračun postfiksnega izraza poteka s pomočjo sklada:
  ko preberemo -- operand : ga porinemo na sklad
  -- operacijo: s sklada preberemo ustrezno število operandov,
  izvedemo nad njimi operacijo,
  rezultat porinemo na sklad;
- Primer. `1 2 - 4 5 + * =`

Algoritem:

```
while (naslednji znak ni znak za konec vhodnih podatkov)
  if (je število)
    porini ga na sklad
  else if (je operator)
      beri potrebne operande s sklada;
      izvedi operacijo nad njimi;
      porini rezultat na sklad;
  else
      napaka;
```

Organizacija programa. Del A (glavni del), del B (ki vsebuje, kar rabita push in pop), del C (push in pop).

4.3 Kalkulator (2)

A.
```
#include <stdio.h>
#include <stdlib.h>
#define MAXOP 100
#define NUMBER '0'
int getop(char []);
void push(double);
double pop(void);
main()
{ int type;
  double op2;
  char s[MAXOP];
  while ((type = getop(s)) != EOF) {
    switch (type) {
    case NUMBER: push(atof(s)); break;
    case '+':  push(pop() + pop()); break;
    case '*':  push(pop() * pop()); break;
    case '-':  op2 = pop(); push(pop() - op2); break;
    case '/':  op2 = pop();
               if (op2 != 0.0)
                  push(pop() / op2);
               else
                  printf("napaka: deljenje z 0!");
               break;
    case '=':  printf("\t%.8g\n", pop());  break;
    default: printf("napaka: neznana operacija %s\n", s); break;
  }
}
return 0;
}
```

Razlaga in opombe.

4.3 Kalkulator (3)

B.
```
#define MAXVAL 100
int sp = 0;
double val(MAXVAL);
```

C.
```
void push(double f)
{
  if (sp < MAXVAL)
    val[sp++] = f;
  else
    printf("napaka: premajhen sklad; ne morem vpisati %g\n", f);
}

double pop(void)
{
  if (sp > 0)
    return val[--sp];
  else {
    printf("napaka: branje iz praznega sklada!");
    return 0.0;
  }
}
```

Razlaga in opombe.

Razlaga in opombe.

4.3 Kalkulator (4)

Funkcija `getop` prebere naslednji operand (t.j. število) ali operator

Algoritem:

```
preskoči presledke in tabulatorje;
if (zadnji prebrani ni cifra ali decimalna pika)
  vrni ga kot svoj rezultat     /* prebran je bil operator */
else /* prebran je bil začetek operanda*/
  preberi cel operand in ga vrni kot svoj rezultat;
```

Lahko se zgodi, da ta algoritem ugotovi, da je prebral *dovolj* šele po tem, ko je prebral že *preveč*!!

– Primeri:

branje zaporedja `123.456 +` ter branje zaporedja `123.456+`

Pri branju drugega zaporedja mora prebrati preveč (`+`), da izve, da je bil operand (`123.456`).

4.3 Kalkulator (5)

Rešitev: uvedemo možnost vračanja prebranega znaka:

-- prebrani znak `c` se bo vrnil v *vmesni pomnilnik* `buf` s funkcijo `ungetch(c)`.

-- funkcija `getch()` bo prebrala naslednji znak;
pri tem bo najprej pogledala v vmesni pomnilnik, če pa bo ta prazen, na običajni vhod

```
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;

int getch(void)                                    Razlaga in opombe.
{
  return (bufp>0) ? buf[--bufp]: getchar();
}

void ungetch(int c)                                Razlaga in opombe.
{
  if (bufp >= BUFSIZE)
    printf("napaka: preveč jih je vrnjenih!\n");
  else
    buf[bufp++] = c;
}
```

4.3 Kalkulator (6)

Nadaljevanje `getop()` ...

```
#include <ctype.h>
int getch(void);
void ungetch(int);

int getop(char s[])
{
  int i,c;
  while ((s[0] = c = getch()) == ' ' || c == '\t')
    ;
  s[1] = '\0';
  if ( !isdigit(c) && c != '.')
    return c;
  i = 0;
  if (isdigit(c))
    while (isdigit(s[++i] = c = getch()))
      ;
  if (c == '.')
    while (isdigit(s[++i] = c = getch()))
      ;
  s[i] = '\0';
  if (c != EOF)
    ungetch(c);
  return NUMBER;
}
```

Razlaga in opombe.

Tu bi sledila definicija `buf`, `getch` in `ungetch`.

4.5 Zaglavja (1)

Uvod.

- Kako bi program za kalkulator porazdelili v *več* datotek?
- Želja: vsaka datoteka -- naj združuje *sorodne* funkcije, t.j. smiselno zaokroženo nalogo
  -- ima dostop le do informacije, ki jo nujno potrebuje
- Prednosti in slabosti.

- Npr. `main()` ..... main.c
  `sp,val,push(),pop()` ..... stack.c
  `getop()` ..... getop.c
  `getch(),ungetch(),buf,bufp` .... getch.c

- Definicije in deklaracije, ki so skupne več datotekam, shranimo v *zaglavja* .
- Koliko zaglavij?
  Eno ali več ter prednosti in slabosti.

4.5 Zaglavja (2)

calc.h

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

main.c

```
#include <stdio.h>
#include <stdlib.h>;
#include "calc.h";
#define MAXOP 100

main() {
  ...
}
```

getop.c

```
#include <stdio.h>
#include <ctype.h>;
#include "calc.h";

getop() {
  …
}
```

getch.c

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;

int getch(void) {
 …
}

void ungetch(int) {
…
}
```

stack.c

```
#include <stdio.h>
#include "calc.h";
#define MAXVAL 100
int sp = 0;
double val(MAXVAL);

void push(double) {
  …
}

double pop(void) {
  …
}
```

Opombe.

4.6 Statične spremenljivke (1)

Motivacija.

- V programu kalkulator sta spremenljivki `sp` in `val` potrebni le funkcijama `push` in `pop` (ki sta v prejšnji razdelitvi (tocka 4.5) celo znotraj iste datoteke stack.c.)
- Ker pa sta `sp` in `val` zunanji, bi lahko postali vidni tudi drugim, "tujim" funkcijam v drugih datotekah, če bi te uporabile deklaracijo `external sp, val;`
- Podobno velja tudi za spremenljivki `buf` in `bufp` v datoteki getch.c.

Kako preprečimo, da bi bile te spr. dosegljive, vidne iz drugih datotek?

- Uporabimo določilo *static*.

Primer.

getch.c

```
#include <stdio.h>
#define BUFSIZE 100
static char buf[BUFSIZE];
static int bufp = 0;

int getch(void) {
 …
}

void ungetch(int) {
…
}
```

stack.c

```
#include <stdio.h>
#include "calc.h";
#define MAXVAL 100
static int sp = 0;
static double val(MAXVAL);

void push(double) {
  …
}

double pop(void) {
  …
}
```

4.6 Statične spremenljivke (2)

Tudi funkcije lahko deklariramo kot statične.

– Npr.,

```
static int imefunkcije()
{
   ...
}
```

Tudi interna (lokalna) spremenljivka je lahko statična. Tedaj:

– je lokalna v funkciji,
– toda ostane in zadrži svojo vrednost po izteku funkcije
– Uporaba: za prenos vrednosti do naslednjega klica te funkcije
– Npr.

```
int imefunkcije()
{
  static int x;
   ...
}
```

4.7 Registrske spremenljivke

Deklaracija z določilom `register`

- Npr.,

```
register int x;
register char c;
```

Pomen:

- Prevajalnik poskuša tako spremenljivko namestiti v register procesorja
  Posledica: hitrejši dostop do take spremenljivke in zato hitrejši program
- Le *priporočilo (pomoč)* prevajalniku; včasih ne more biti upoštevano

Uporaba:

- pri avtomatskih spremenljivkah (ker nimajo dolge življenjske dobe)
- pri formalnih funkcijskih parametrih

```
int imefunkcije(register unsigned m, register int n)
{
 register int x;
   ...
}
```

Omejitve:

- Ne za vsak tip spremenljivk (odvisno od arhitekture)
- Nad tako spremenljivko ne moremo uporabiti operatorja `&` za izračun naslova

4.10 Rekurzija(1)

V C funkcija lahko pokliče samo sebe

- neposredno ali posredno:

Neposredno:

```
int f(...)
{
 ...
 ...f(...)...
 ...
}
```

Posredno (preko druge funkcije):

```
int f(...)
{
 ...
 ...g(...)...
 ...
}
```

```
int g(...)
{
 ...
 ...f(...)...
 ...
}
```

Opombe:

- primerni problemi
- učinkovitost izvajanja

4.10 Rekurzija(2)

Naloga: rekurzivno izpiši celo število.

Rekurzivni algoritem:

pozitivno število, npr. `n = 78613`

rekurzivno izpišemo tako, da

a) najprej *rekurzivno izpišemo* število `n/10` (če to ni 0)

b) in nato običajno izpišemo znak z ASCII kodo `n%10 + '0'`

Program.

```
#include <stdio.h>
void printd(int n)
{
  if (n<0) {                    ... Če je n negativno,
    putchar('-');               ... izpišemo predznak
    n = -n;                     ... in n pretvorimo v pozitivno
  }
  if (n/10)
    printd(n/10);               ... a)
  putchar(n%10 + '0');          ... b)
}
```

4.10 Rekurzija(3)

Naloga: rekurzivno uredi polje `v[levi…desni]` v naraščajočem vrstnem redu.

Rekurzivni algoritem Quicksort:

polje *rekurzivno uredimo* tako, da

(1) izberemo neko komponento v polju, npr. srednjo

(2) preostale komponente premečemo v dve (še neurejeni) podpolji: Slika.

podpolje (a), kjer so vse komponente < od srednje

podpolje (b), kjer so vse komponente >= od srednje

(3) *rekurzivno uredimo* podpolje (a)

(4) *rekurzivno uredimo* še podpolje (b)

4.10 Rekurzija(4)

Program.

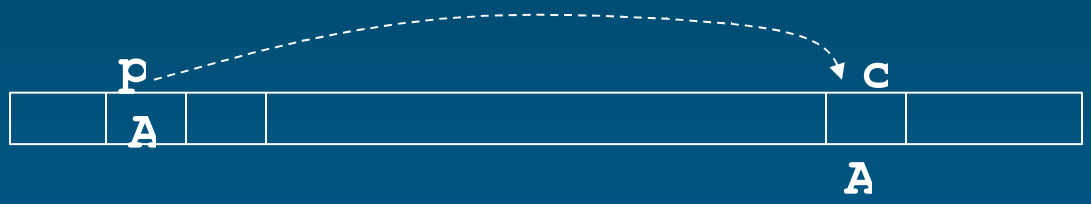
```
void qsort(int v[], int levi, int desni)
{ int i, zadnji;
  void zamenjaj(int v[], int i, int j);

  if(levi >= desni)
     return;
  zamenjaj(v, levi, (levi+desni)/2);          ...(1)     Razlaga in opombe.
  zadnji = levi;                              ...(2)
  for(i = levi+1; i <= desni; i++)
    if (v[i] < v[levi])
       zamenjaj(v, ++zadnji, i);
  zamenjaj(v, levi, zadnji);
  qsort(v, levi, zadnji-1);                   ...(3)
  qsort(v, zadnji+1, desni);                  ...(4)
}

int zamenjaj(int v[], int i, int j)
{ int pom;
  pom = v[i]; v[i] = v[j];  v[j] = pom;
}
```

5. Kazalci in polja

5.1 Kazalci in naslovi(1)

Kazalec je spremenljivka, ki vsebuje naslov druge spremenljivke.

- Primer. `char c;`
  `p` *kazalec na* `c;`



Operator `&` vrne naslov dane spremenljivke (argumenta operacije `&`)

- Primer. `p = &c;` ..... v `p` se vpiše naslov spremenljivke `c`
  (Pravimo: "`p` pokaža na `c`"; "`p` usmerimo na `c`")

- `&` lahko uporabimo le nad objekti, ki so v pomnilniku
  - npr. spremenljivke, komponente polj, komponente struktur, ...

- `&` ne smemo (ne moremo) uporabiti nad
  - Izrazi; npr. `&(x+y)`
  - Konstantami
  - Spremenljivkami tipa `register`

5.1 Kazalci in naslovi(2)

Operator `*` vrne spremenljivko, kamor kaže dani kazalec (argument operacije `*`)

- Primer: `*p` ..... pomeni spremenljivko, kamor kaže kazalec `p`
  Drugače: `*` razume svoj argument kot naslov nekega objekta in vrne ta objekt

Definicija kazalca. Kazalec definiramo s pomočjo operatorja `*` :

- Primer: `tip *kaz;` .... pomeni: tisto, na kar bo kazal `kaz`, bo tipa `tip`
  Drugače: `kaz` naj bo kazalec, ki bo kazal na nekaj, kar je (ali bo) tipa `tip`

5.1 Kazalci in naslovi(3)

Primer.

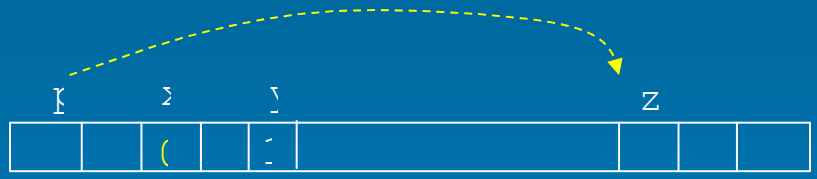
```
int x = 1, y=2, z[10];
int *p;
```

```
p = &x;
```

```
y = *p;
```

```
*p = 0;
```

```
p = &z[0];
(enako tudi: p = z;)
```



5.1 Kazalci in naslovi(4)

Primer.

```
int x = 1;
int *p, *q;
```

```
p = &x;
```

```
q = p;
```

5.1 Kazalci in naslovi(5)

Operaciji `*` in `&` imata prednost pred aritm. operacijami (`*, /, %, +, -`) (tabela na 2.12)

Primeri.

- `y = *p + 1;` .... prišej 1 k vrednosti, na katero kaže p, nato rezultat shrani v y
- `*p +=1;` .... prištej 1 k vrednosti, na katero kaže p.
- ...

5.2. Kazalci in funkcijski argumenti(1)

Naloga: funkcija `menjaj(a,b)` naj zamenja vsebini svojih argumentov.

Poskus 1.

```
int a,b;
....
menjaj(a,b);
....
void menjaj(int x, int y)
{ int pomozna;
  pomozna = x;   x = y;   y = pomozna;
}
```

Opomba: Razlaga, zakaj je poskus 1 napačen.

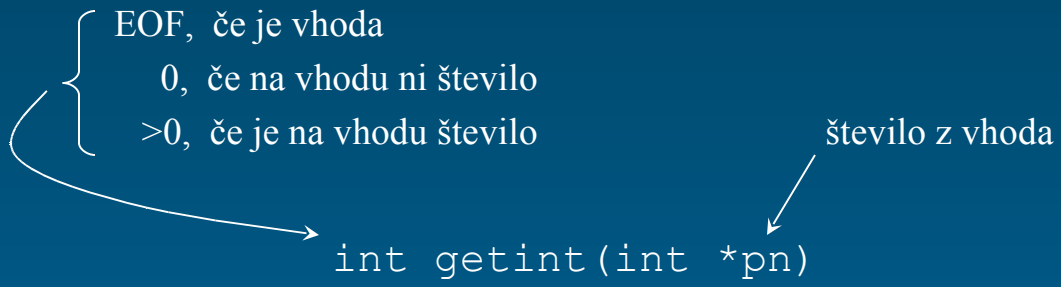
Poskus 2.

```
int a,b;
....
menjaj(&a,&b);
....
void menjaj(int *px, int *py)
{ int pomozna;
  pomozna = *px;   *px = *py;   *py = pomozna;
}
```

Opomba: razlaga poskusa 2.

5.2. Kazalci in funkcijski argumenti(2)

Naloga: funkcija `getint` naj bere cela števila z znakovnega vhoda in vrne:



5.2. Kazalci in funkcijski argumenti(3)

Program

```
#include <ctype.h>
int getch(void);
void ungetch(int);

int getint(int *pn)
{ int c, predz;
  while(isspace(c = getch()))
    ;
  if(!isdigit(c) && c != EOF && c != '+' && c != '-') {
    ungetch(c);
    return 0;
  }
  predz = (c == '-')? -1: 1;                          Razlaga in opombe.
  if(c == '+' || c == '-')
    c = getch();
  for(*pn = 0; isdigit(c); c = getch())
    *pn = 10 * *pn + (c - '0');
  *pn *= predz;
  if (c != EOF)
    ungetch(c);
  return c;
}
```

5.2. Kazalci in funkcijski argumenti(4)

Uporaba `getint`: beri števila z vhoda in jih vpisuj v komponente polja `a`.

Progam.

```
...
int i, a[STKOMP], getint(int *);
...
for(i=0; i<STKOMP && getint(&a[i]) != EOF; i++)
  ;
...
```

5.3 Kazalci in polja(1)

Zveza med kazalci in polji je tesna.

Naj bosta dani definiciji

```
int a[10];
int *p;
```

S prireditvijo

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

kazalec `p` pokaže na prvo komponento polja `a`. Slike, razlaga.

Če *po tem* zapišemo

```
x = *p;
```

je to enakovredno

```
x = a[0];
```

5.3 Kazalci in polja(2)

Če `p` kaže na neko komponento polja `a`,
potem `p + i` kaže na komponento, ki je za i komponent dalje,
`p - i` -ii- nazaj
ne glede na tip komponent!

Primer.
Naj bo

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

Po tem

`*(p+1)` pomeni isto kot `a[1]`, Slike, razlaga.

`*(p + i)` pa isto kot `a[i]`.

5.3 Kazalci in polja(3)

Sinonim za *naslov prve komponente polja* je kar *ime tega polja.*
Torej:

`&a[0]` isto kot `a`

zato `p = &a[0];` -ii- `p = a;`

in `a[i]` -ii- `*(a + i)`

ter `&a[i]` -ii- `a + i`

Primer.

`x = a[4];` isto kot `x = *(a + 4);`

Primer.

```
for (i=0; i<10; i++)          for (i=0; i<10; i++)
 x+=a[i];                isto kot     x+=*(a+i);
```

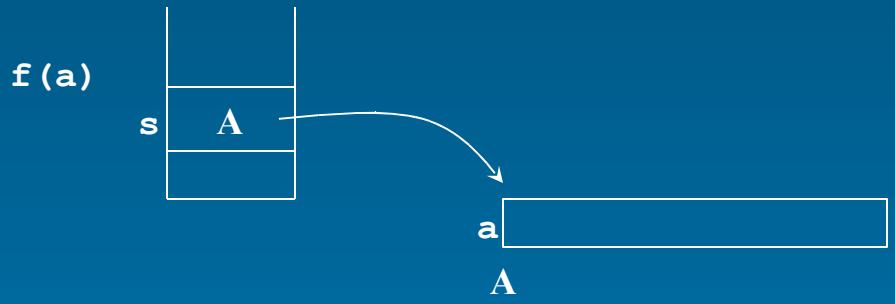
Med kazalcem `p` na polje `a` in imenom tega polja *so razlike*!

Primeri. Pravilno:
```
p = a;      /* usmeritev kazalca na polje */
p++;        /* preusmeritev kazalca na naslednjo komponento polja */
```
Nepravilno:
```
a = p;
a++;
p = &a;
```
Razlaga

5.3 Kazalci in polja(4)

Če je funkcijski argument polje `a`,

- se argument `a` ne prenese po vrednosti
  - ob klicu funkcije se ne ustvari kopija argumenta (polja `a`) na skladu,
  - razlog: učinkovitost klicev
- temveč po referenci
  - ob klicu se ustvari lokalna spremenljivka `s`, ki vsebuje naslov argumenta (polja).
  - preko nje dosežemo argument (polje `a`)

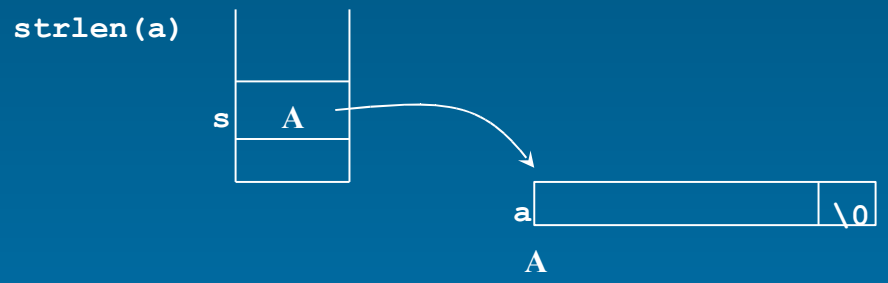


5.3 Kazalci in polja(5)

Naloga: funkcija `strlen` naj izmeri polje (t.j. vrne število komponent brez stražarja \0).
Algoritem: s pomočjo kazalcev (brez kazalcev pa glej 2.3).
Program.

```
int strlen (char *s)          /* klic bo npr. strlen(a); */
{ int i;
  for(i=0; *s[i]!='\0'; s++)
    i++;
  return i;
}
```

Ob klicu `strlen(a)`, kjer je `a` neko polje znakov, se bo ustvarila naslednja situacija:



Primeri klicev.

```
strlen("Vroce poletje.");  /* argument je nek znakovni niz - konstanta */
strlen(polje);             /* argument je neko polje, npr. char polje[99] */
strlen(kazalec);           /* argument nek kazalec na polje, npr. char *kazalec */
```

5.4 Kazalčna aritmetika(1)

Naloga: sestavi funkciji `alloc` in `afree` za delo s pomnilnikom:

- `alloc(n)` naj rezervira blok `n` zlogov pomnilnika in vrne kazalec na alocirani blok
- `afree(p)` naj sprosti blok, kamor kaže kazalec `p`

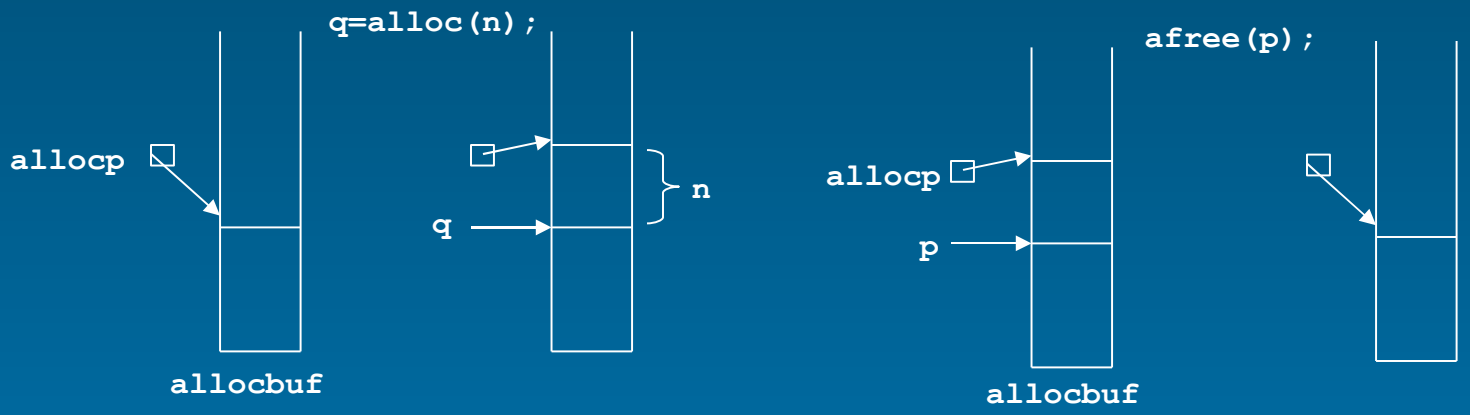
Opombe:

- Pomnilnik, s katerim bosta upravljali funkciji, naj bo *sklad*
- Zato: sproščanje blokov bo v nasprotnem redu kot dodeljevanje (=> `alloc, afree` enostavnejši)
- Obstajata standardni funkciji `malloc, free` brez teh omejitev

5.4 Kazalčna aritmetika(2)

Priprava:

- `allocbuf` ... znakovno polje; predstavlja pomnilnik; dosegljivo le `alloc` in `afree;` statično.
- `allocp` ... kazalec na naslednji prosti zlog v `allocbuf`
- `alloc(n)` ... preveri, če je dovolj prostora v `allocbuf`; preusmeri `allocp` za `n` zlogov; vrne kazalec na začetek pravkar dodeljenega bloka `n` zlogov
- `afree(p)` ... preusmeri `allocp` na začetek vrhnjega bloka (ki naj se sprosti)



5.4 Kazalčna aritmetika(3)

Program.

```
#define ALLOCSIZE 10000
static char allocbuf[ALLOCSIZE];
static char *allocp = allocbuf;

char *alloc(int n)
{
  if(allocbuf + ALLOCSIZE - allocp >= n) {
    allocp +=n;
    return allocp - n;
  } else return 0;
}

void afree(char *p)
{
  if(p >= allocbuf && p < allocbuf + ALLOCSIZE)
    allocp = p;
}
```

Razlaga in opombe

5.4 Kazalčna aritmetika(4)

V prejšnjem primeru smo:

```
#define ALLOCSIZE 10000
static char allocbuf[ALLOCSIZE];
static char *allocp = allocbuf;

char *alloc(int n)
{
  if(allocbuf + ALLOCSIZE - allocp >= n) {
    allocp +=n;
    return allocp - n;

  } else return 0;
}

void afree(char *p)
{
  if(p >= allocbuf && p < allocbuf + ALLOCSIZE)
    allocp = p;
}
```

prednastavili vrednost kazalcu

odšteli kazalec (od drugega naslova)

kazalcu prišteli/odšteli celoštevilsko konstanto

kazalcu priredili vrednost 0

primerjali kazalce z drugimi kazalci oz. naslovi

kazalcu priredili drug kazalec

5.4 Kazalčna aritmetika(5)

Velja tudi v splošnem:

- Kazalcu lahko priredimo vrednost 0 (NULL) in ga z njo primerjamo
- Kazalcu lahko prištejemo ali odštejemo celo število
- Če kazalca kažeta na elementa istega polja, so dovoljene/smiselne tudi relacije <, <=, >, >=, ==, !=

  Npr., `p<q` velja, če `p` kaže na komponento polja, ki je pred komponento, kamor kaže `q`

Izjema: naslov prve lokacije po koncu polja še lahko uporabimo za računanje/primerjanje.

- Odštevanje kazalcev je dovoljeno (kadar kažeta na isto polje tudi smiselno).

  Če `p` in `q` kažeta na isto polje in velja `p<q`, je v *tem* področju `q-p+1` komponent

- Prepovedane (oz. brez pomena) so operacije:

  množenje, deljenje, seštevanje, maskiranje, prištevanje/odštevanje necelih števil

5.4 Kazalčna aritmetika(6)

Naloga: funkcija `strlen` naj vrne število komponent danega polja (brez stražarja \0).

Opomba. Uporabimo dva kazalca. (glej tudi 2.3 in 5.3)

Algoritem.

Kazalca `p,q` postavimo na začetek polja;
kazalec `p` pomaknemo do konca polja;
kazalca odštejemo.

Program.

```
int strlen(char *s)
{
  char *p = s;
  while (*p != '\0')
    p++;
  return p-s;
}
```

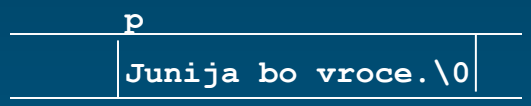
Razlaga in opombe.

5.5 Kazalci, znakovni nizi in funkcije(1)

Znakovni niz lahko deklariramo s pomočjo polja.

Npr.

```
char p[] = "Junija bo vroce.";
```



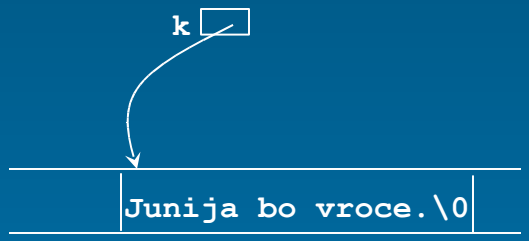
Opombe.

- `p` je polje, ki se nanaša na vedno isti del pomnilnika
- Komponente znakovnega niza so dosegljive preko `p[…]`

Znakovni niz lahko deklariramo tudi s pomočjo kazalca nanj.

Npr.

```
char *k = "Junija bo vroce.";
```



Opombe.

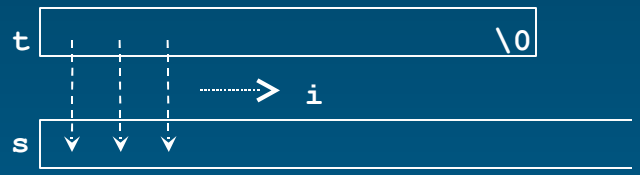
- `k` ob definiciji pokaže na začetek znakovnega niza
- `k` lahko preusmerimo na kako drugo komponento niza
- Komponenta znakovnega niza dosegljiva preko `*k`

5.5 Kazalci, znakovni nizi in funkcije(2)

Naloga: funkcija `strcpy(s,t)` naj prekopira niz `t` v niz `s`.

Opomba. Obstaja že v `string.h`.

Algoritem 1 (z uporabo polj).



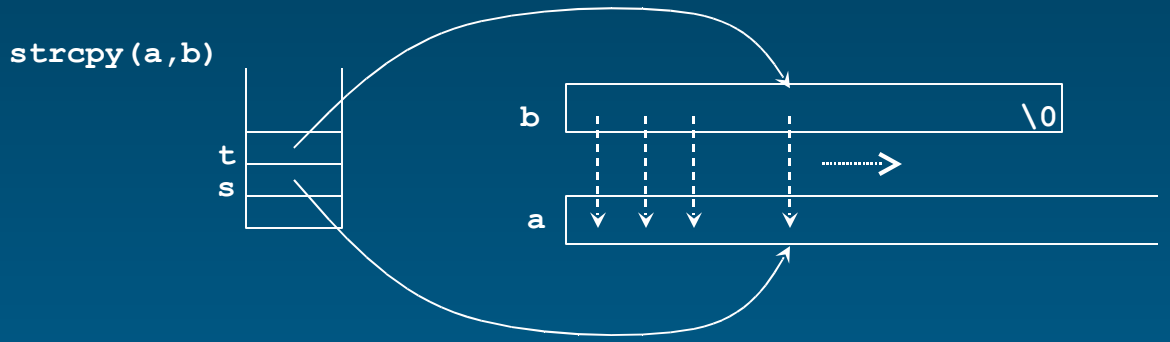
Program.

```
void strcpy(char s[], char t[])
{
  int i = 0;
  while ((s[i] = t[i]) != '\0')
    i++;
 }
```

Razlaga in opombe.

5.5 Kazalci, znakovni nizi in funkcije(3)

Algoritem 2 (z uporabo kazalcev).



Program.

```
void strcpy(char *s, char *t)
{
  while ((*s = *t) != '\0') {
    s++;  t++;
  }
}
```

Razlaga in opombe.

Bolj zgoščeno:

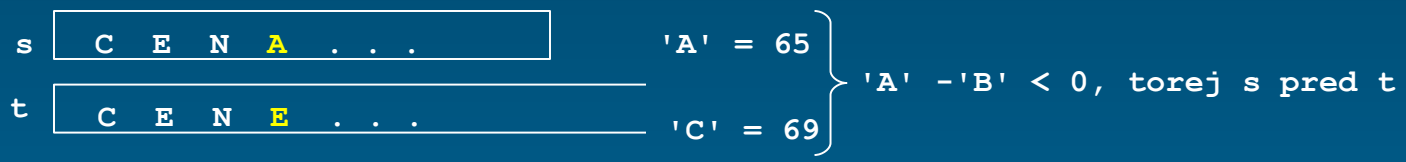
```
void strcpy(char *s, char *t)
{
  while ((*s++ = *t++) != '\0')     /* lahko celo samo ((*s++ = *t++)) */
    ;
}
```

5.5 Kazalci, znakovni nizi in funkcije(4)

Naloga: funkcija `strcmp(s,t)` naj vrne rezultat, ki je $\begin{cases} <0 \\ 0 \\ >0 \end{cases}$, če je `s` leksikografsko $\begin{cases} \text{pred } t \\ \text{enak } t. \\ \text{za } t \end{cases}$

Opomba. Obstaja že v `string.h`.

Algoritem. Odštejemo ASCII kodi prvih istoležnih toda različnih komponent.



Program (s polji).

```
int strcmp(char s[],char t[])
{
  int i;
  for(i=0; s[i] == t[i]; i++)
    if(s[i] == '\0')
      return 0;
  return s[i]-t[i];
}
```

Program (s kazalci).

```
int strcmp(char *s, char *t)
{
 for( ; *s == *t; s++,t++)
    if(*s == '\0')
      return 0;
  return *s - *t;
}
```

5.5 Kazalci, znakovni nizi in funkcije(5)

Opombe.

`*p++` ... 1) uporabi `*p`
2) preusmeri (povečaj) `p`

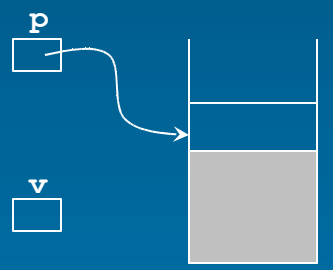
`*++p` ... 1) preusmeri (povečaj) `p`
2) uporabi `*p`

Primer (sklad).

Naj `p` kaže na prvo prosto mesto na skladu.
Če pozabimo na preverjanje, ali je sklad poln oz. prazen,
sta operaciji

push(v): `*p++ = v;` <==> `*p = v;`
`p++;`

v = pop(); `v = *--p;` <==> `--p;`
`v = *p;`



5.6 Polja kazalcev(1)

Kazalci lahko nastopajo tudi kot komponente polja.

Naloga: vhodne vrstice uredi v abecedno (leksikografsko) ureditev.

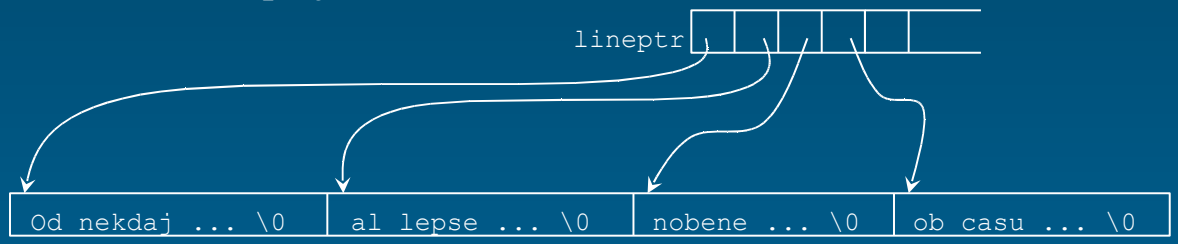
Primer.

Od nekdaj lepe so Ljubljanke slovele,		*al lepše od Urške bilo ni nobene,*
al lepše od Urške bilo ni nobene,	==>	*nobene očem bilo bolj zaželene*
nobene očem bilo bolj zaželene		*ob času nje cvetja dekleta ne žene*
ob času nje cvetja dekleta ne žene.		*Od nekdaj lepe so Ljubljanke slovele,*

5.6 Polja kazalcev(2)

Podatkovna struktura:

- Vrstice shranimo eno za drugo v eno znakovno polje
- Vrstico dosežemo s kazalcem na njen prvi znak
- Kazalce združimo v polje kazalcev



Zamisel:

- Vrstici primerjamo s funkcijo `strcmp` (glej 5.5), ki ji posredujemo kazalca na vrstici
- Če bi bilo treba med urejanjem dve vrstici zamenjati, raje zamenjamo kazalca nanju

Algoritem:	Funkcija:
beri vrstice z vhoda (pri tem zgradi zgornjo pod. strukturo.....	`readlines`
uredi vrstice (tako, da premešaš kazalce nanje)	`qsort`
izpiši vrstice v zahtevani ureditvi	`writelines`

5.6 Polja kazalcev(3)

Program.

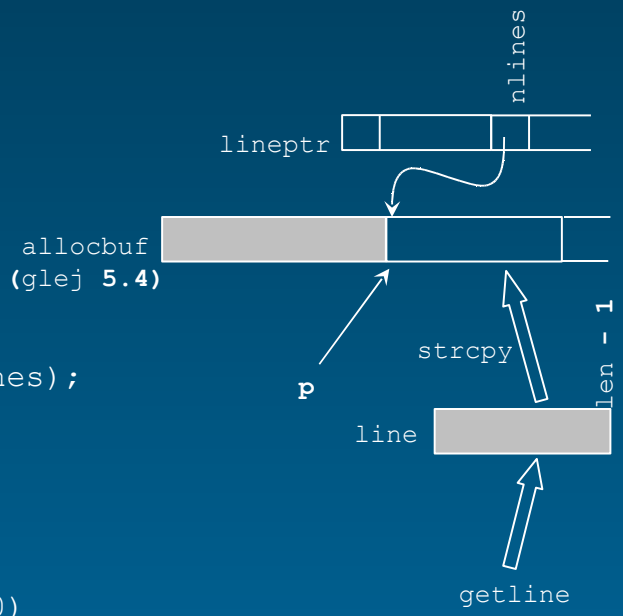
```
#include <stdio.h>
#include <string.h>
#define MAXLINES 5000
char *lineptr[MAXLINES];
int readlines(char *lineptr[], int nlines);
void qsort(char *lineptr[], int left, int right);
void writelines(char *lineptr[], int nlines);

main()
{
  int nlines;
  if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
      qsort(lineptr, 0, nlines-1);
      writelines(lineptr, nlines);
      return 0;
  } else {
      printf("napaka: na vhodu preveč vrstic\n");
      return 1;
  }
}
```

Razlaga in opombe.

5.6 Polja kazalcev(4)

Program (nadaljevanje).

```
#define MAXLEN 1000
int getline(char *, int);
char *alloc(int);  /* glej 5.4 */

int readlines(char *lineptr[], int maxlines);
{
  int len, nlines;
  char *p, line[MAXLEN];
  nlines = 0;
  while((len = getline(line, MAXLEN)) > 0)
    if(nlines >= maxlines || (p = alloc(len)) == NULL)
      return -1;
    else {
      line[len-1] = '\0';
      strcpy(p, line);
      lineptr[nlines++] = p;
    }
  return nlines;
}
```



Razlaga in opombe

5.6 Polja kazalcev(5)

Program (nadaljevanje; to je prirejena funkcija `qsort` iz točke 4.10)

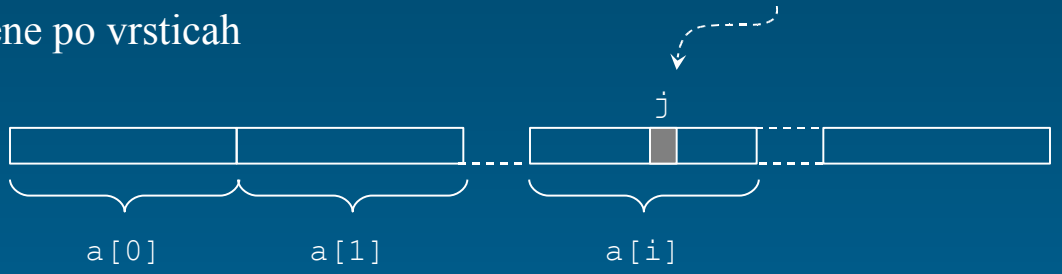
```
void qsort(char *v[], int left, int right)
{ int i, last;
  void swap(char *v[], int i, int j);

  if (left >= right)
     return;
  swap(v, left, (left + right)/2);
  last = left;
  for(i= left+1; i <= right; i++)
    if(strcmp(v[i], v[left]) <0)
      swap(v, ++last, i);
  swap(v, left, last);
  qsort(v, left, last-1);
  qsort(v, last+1, right);
}

void swap(char *v[], int i, int j)
{ char *temp;
  temp = v[i]; v[i] = v[j]; v[j] = temp;
}
```

Razlaga in opombe (glej točko 4.10)

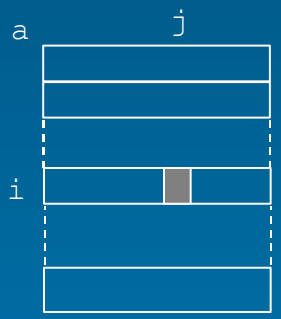
5.7 Večdimenzionalna polja(1)

Dvodimenzionalno polje:

- enodimenzionalno polje `a`, kjer je vsaka komponenta `a[i]` spet enodimenz. polje.
- indekse pišemo ločeno, `a[i][j]`
- `[]` je levo asociativna operacija, zato `a[i][j]` pomeni `(a[i])[j]`
- komponente so shranjene po vrsticah



Drug pogled:



5.7 Večdimenzionalna polja(2)

Deklaracija in inicializacija:

- Število vrstic in stolpcev
- Seznam začetnih vrednosti; med zavitimi oklepaji; po vrsticah

Primer. `int a[2][3] = {{9,8,7},{6,5,4}};`

a	0	1	2
0	9	8	7
1	6	5	4

Če je polje formalni parameter funkcije, je prva dimenzija prosta, ostale moramo navesti.
Obstaja več možnosti:

```
tipf f(tipkomponent p[2][3]){…};
tipf f(tipkomponent p[][3]){…};
tipf f(tipkomponent (*p)[3]){…};
```

Opombe. (*p)[3] in *p[3]

Posplošitve.

5.7 Večdimenzionalna polja(3)

Naloga: funkcija `zap_dan` naj izračuna, kateri dan v letu je datum `d,m,l`.

Npr. `zap_dan(3,2,1999)` je 34. dan v letu 1999.

Algoritem. Začetni `d` postopno povečujemo z dolžinami predhodnih mesecev v tem letu.

Sestavimo še obratno funkcijo `dan_mes;`

Npr. `dan_mes(34,1999)` vrne dve števili, 3 (dan) in 2 (mesec) v danem letu.

Algoritem. Zap.številko dneva postopno zmanjšujemo z dolžinami predhodnih mesecev.

5.7 Večdimenzionalna polja(4)

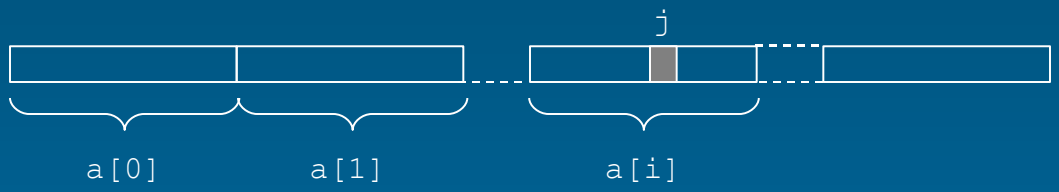
Program.

```
static char tab[2][13]={{0,31,28,31,30,31,30,31,31,30,31,30,31},
                        {0,31,29,31,30,31,30,31,31,30,31,30,31}};

int zap_dan(int d, int m, int l)
{ int i, prest;
  prest = l%4 == 0 && l%100 != 0 || l%400 == 0;
  for(i=0; i<m; i++)
    d+= tab[prest][i];
  return d;
 }

void dan_mes(int zd, int l, int *pd, int*pm)
{ int i,prest;
  prest = l%4 == 0 && l%100 != 0 || l%400 == 0;
  for(i=1; zd>tab[prest][i]; i++)
    zd-=tab[prest][i];
  *pm = i;
  *pd = zd;
}
```

5.9 Polja kazalcev in večdimenzionalna polja(1)

```
int a[10][20];
```

- Polje sestavlja 10 vrstic, vsako vrstico pa 20 števil
- Definicija rezervira prostor za 200 komponent (števil)
- Pri dostopanju do komponente `a[i][j]` se izračuna njena lega (indeks)
  `a[i][j]` je *10*i+j* -ta komponenta polja `a`
- Množenje in seštevanje; čas!



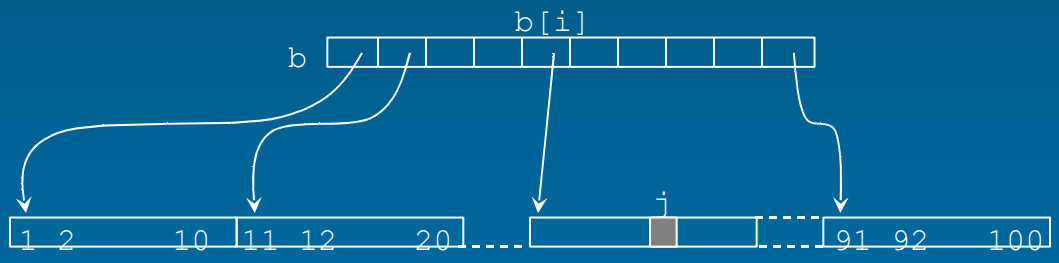
5.9 Polja kazalcev in večdimenzionalna polja(2)

```
int *b[10];
```

- Polje sestavlja 10 komponent; vsaka je kazalec na število
- Definicija rezervira prostor za 10 kazalcev;
- Usmerimo jih ob definiciji, npr.

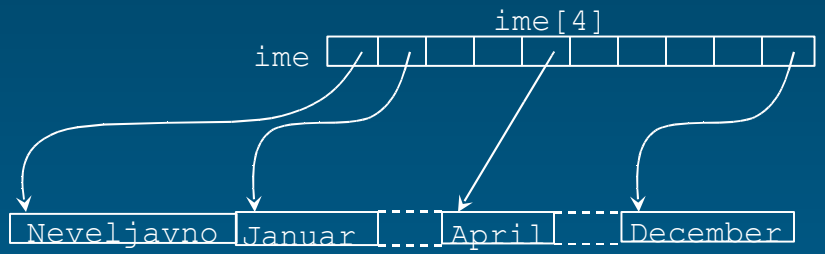
  `int *b[10] = {{1,2,3,…,10},{11,12,…20},…,{91,…,100}}`

  ali kasneje, npr. `b[0]={1,2,3,…,10};`
- Dolžina vrstic lahko spremenljiva
- Pri dostopanju do komponente `b[i][j]` se izračuna njena lega takole: *b[i] + j;* hitrost



5.8 Prednastavitev polja kazalcev

Naloga: funkcija `mesec(n)` naj vrne ime `n`-tega meseca v letu; npr. `mesec(4)` vrne `"april"`
Opomba. Funkcija uporabi statično lokalno polje kazalcev na znakovne nize - imena mesecev.



Program.

```
char *mesec(int n)
{
  static char *ime[] = {"Neveljavno",
                        "Januar",
                        "Februar",
                            …,
                        "December"};
  return (n<1 || n>12)? ime[0] : ime[n];
}
```

5.10 Prenos argumentov z ukazne vrstice(1)

Ukazna vrstica, kjer poženemo program `progr` z vhodnimi podatki `a,b,c` je

```
> progr a b c...
```

Ko ukazno vrstico zaključimo (s pritiskom na Enter),
operacijski sistem OS (lupina, monitor, CLI, ...) naredi tudi sledeče:

7. Ugotovi, da želimo pognati program, ki je v datoteki z imenom progr, in poišče datoteko
8. Ugotovi, da so v ukazni vrstici navedeni še trije vhodni argumenti: a, b, c
9. Požene program, ki je opisan v datoteki progr
10. Temu programu preda argumente a, b, c

Kako?

5.10 Prenos argumentov z ukazne vrstice(2)

Kako?

- Program opisuje funkcija `main` (v datoteki progr), ki se požene prva
- `main` je zapisana z dvema parametroma:

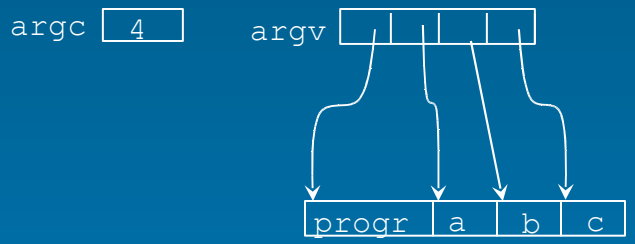
```
main(int argc, char *argv[])
```

- Ob klicu funkcije `main` OS nastavi parametra takole:

  `argc` ... število vseh argumentov v ukazni vrstici (vštevši ime programa! Zgoraj npr. 4.)

  `argv` ... polje kazalcev na znakovne nize -- argumente ukazne vrstice

Primer.



5.10 Prenos argumentov z ukazne vrstice(3)

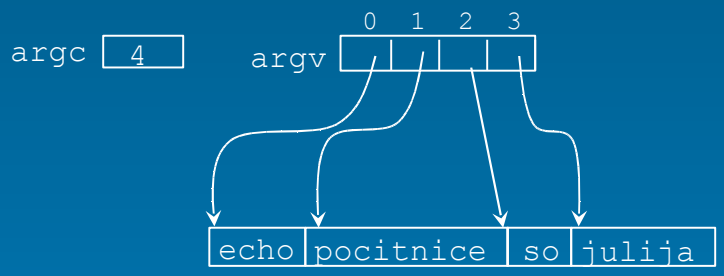
Naloga: program *echo* naj izpiše svoje vhodne argumente.

Npr. > `echo pocitnice so julija` [Enter]

`echo pocitnice so julija`

Program.

```
#include <stdio.h>
main(int argc, char *argv[])
{
 int i;
 for(i=1; i<argc; i++)
   printf("%s%s",argv[i],(i<argc-1)?" ":"");
 printf("\n");
 return 0;
}
```

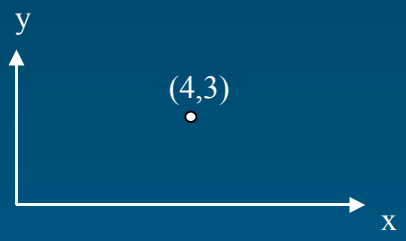


6. Strukture

6.1 Osnove(1)

Uvod: struktura je skupina podatkov, lahko različnih tipov, zbranih pod enim imenom.

Primer. Točko v 2D prostoru določata dve koordinati, npr. števili x in y:



Deklaracija *tipa* strukture:

- rezervirana beseda `struct` ,
- ime tipa,
- opisi komponent.

Primer. Deklaracija tipa `tocka`:

```
struct tocka{
        float x;
        float y;
       };
```

Deklaracija tipa še ne rezervira prostora za strukturo.

6.1 Osnove(2)

Prostor se rezervira, če navedemo tudi imena spremenljivk, ki bodo take strukture.

Primer.
```
struct tocka{
        float x;
        float y;
      } b,c;
```
Deklaracija dveh struktur `b,c` ob deklaraciji njunega tipa

Primer.
```
struct tocka a = {4,3};
```
Še ena struktura, ki jo takoj prednastavimo; uporabimo že prej deklarirano ime `tocka`

Ime tipa struktue lahko opustimo, če se v nadaljevanju nanj ne bomo sklicali.

Primer.
```
struct {
        float teza;
        int starost;
      } w;
```
Struktura `w` (brez eksplicitnega poimenovanja njenjega tipa)

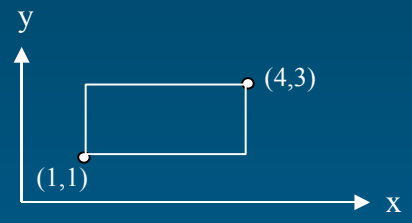
Dostop do komponent strukture: `imestrukture.imenjenekomponente`

Primer.
```
printf("Komponenti tocke sta %f %f", a.x, a.y);
```

6.1 Osnove(3)

Gnezdenje struktur: strukture lahko uporabimo v drugih strukturah.

Primer. Pravokotnik lahko predstavimo z dvema nasprotnima ogliščema:




```
struct pravokotnik{
          struct tocka ls;    /* levo spodnje */
          struct tocka dz;    /* desno zgornje */
        } p = {{1,1},{4,3}}; /* prednastavitev strukture p */
```

Dostop do komponent gnezdene strukture z uporabo operacije "."

Primer: Levo spodnje oglišče premakni v točko (2,0):

```
p.ls.x = 2;
p.ls.y = 0;
```

6.2 Strukture in funkcije(1)

Struktureo lahko funkciji posredujemo na različne načine:

- po komponentah
- kot celoto
- s pomočjo kazalca na strukturo

Prenos po komponentah

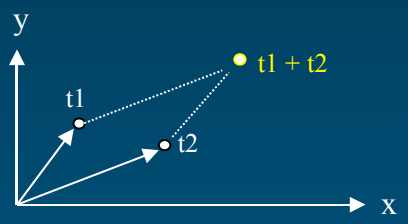
Primer.

```
struct tocka nareditocko(int u, int v) /* iz u in v sestavi in vrni strukturo*/
{                                      /* u in v lahko komponenti druge tocke
  struct tocka zacasna;
  zacasna.x = u; zacasna.y = v;
  return zacasna;
};

struct pravokotnik p;
struct tocka tezisce;
struct tocka nareditocko(int, int);

p.ls = nareditocko(0,0);
p.dz = nareditocko(XMAX, YMAX);
tezisce = nareditocko((p.ls.x + p.dz.x)/2,(p.ls.y + p.dz.y)/2));
```

6.2 Strukture in funkcije(2)

Prenos celotne strukture

Primer. Seštej dva vektorja (točki).




```
struct tocka sestejtocki(struct tocka t1, struct tocka t2)
{
  t1.x += t2.x;
  t1.y += t2.y;
  return t1;
};
```

Razlaga in opombe.

Primer. Ali je točka t v pravokotniku p?

```
int znotrajpravokotnika(struct tocka t, struct pravokotnik p)
{
  return t.x > p.ls.x && t.x < p.dz.x
      && t.y > p.ls.y && t.y < p.dz.y;
};
```

Razlaga in opombe.

6.2 Strukture in funkcije(3)

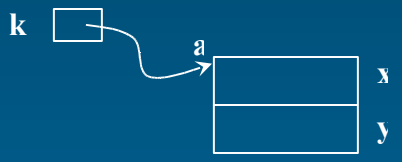
Prenos strukture s pomočjo kazalca nanjo

– Časovno in prostorsko bolj učinkovito.

Primer.

```
struct tocka *k;        /* k bo kazalec na strukturo tipa tocka */
struct tocka a;

k = &a;
```

Sedaj imamo:

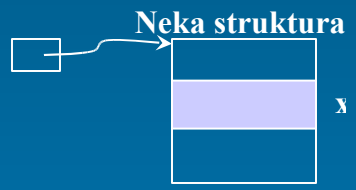


in lahko to uporabimo kot argument funkcije, npr.

```
printf("Tocka je (%f,%f).",(*k).x,(*k).y);
```

Opombe.

Alternativni zapis:

če k 

potem `k->x` pomeni isto kot `(*k).x`

Npr.

```
printf("Tocka je (%f,%f).", k->x, k->y);
```

6.2 Strukture in funkcije(4)

Opombe.

– Operaciji . in -> sta levo asociativni.

Npr., če imamo

```
struct pravokotnik p, *kp = &p;
```

so naslednji izrazi enakovredni:

```
p.ls.x
kp->ls.x
(p.ls).x
(kp->ls).x
```

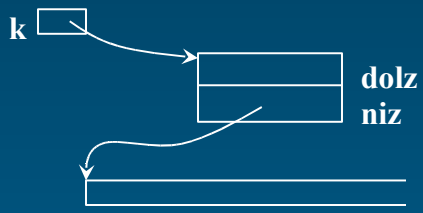
6.2 Strukture in funkcije(4)

Opombe.

– Operacije `.` `->` `()` in `[]` imajo najvišjo prioriteto (glej 2.12)

Npr., če deklariramo

```
struct {
  int dolz;
  char *niz;
} *k;
```



potem:

– `++k -> dolz;` ... `++(k->dolz)` ... poveča se `dolz`

– `(++k) -> dolz;` ... Preusmeri `k` (na naslednjo tako strukturo)

– in od tam vrni komponento `dolz`.

– `k++ -> dolz;` ... Vrni komponento `dolz` kazane strukture,

nato preusmeri `k` (na naslednjo takšno strukturo).

Isto kot `(k++)->dolz`.

– `*k -> niz` ... `*(k->niz)` ... Vrne tisto, na kar kaže `niz`.

– `*k -> niz++` ... `*(k->niz)++` ... Vrne tisto, na kar kaže `niz`

in preusmeri `niz` na naslednji znak.

– `(*k -> niz)++` ... Poveča tisto, na kar kaže `niz`.

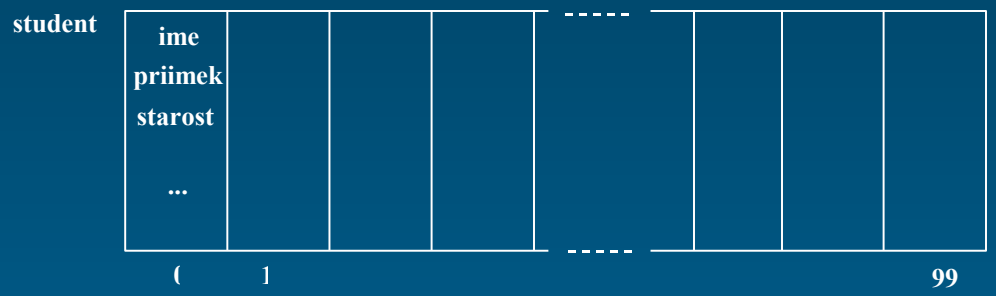
– `*k++ -> niz` ... Vrne tisto, na kar kaže `niz`

in preusmeri k

Varnost in čitljivost: uporabi oklepaje in/ali razbij v več korakov.

6.3 Polja struktur(1)

Primer.



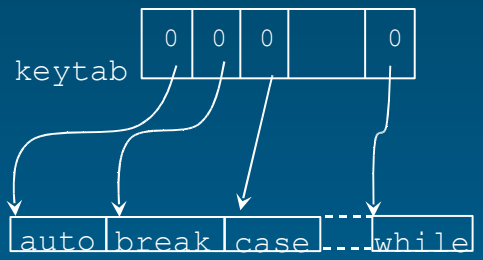

```
struct oseba {
        char ime[15];
        char priimek[20];
        int starost;
        ...
    };
    struct oseba student[100];
```

6.3 Polja struktur(2)

Primer.

Naloga: določi pogostost rezerviranih besed v vhodnem besedilu.

Podatkovna struktura, ki jo bomo uporabljali:




```
struct key {
   char *word;
   int count;
} keytab[] = {{"auto",0},
              {"break",0},
              {"case",0},
               ...
              {"while",0}};
```

Algoritem: ponavljaj, dokler lahko:

preberi naslednjo rezervirano besedo z vhoda;

z binarnim iskanjem jo poišči v tabeli in povečaj števec njenih pojavov

6.3 Polja struktur(3)

Program.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100
int getword(char *, int);
int binsearch (char *, struct key *, int);
main()
{ int n;
  char word[MAXWORD];
  while (getword(word,MAXWORD) != EOF)
    if (isalpha(word[0]))
     if ((n = binsearch(word, keytab, NKEYS)) >= 0)
       keytab[n].count++;
  for (n=0; n<NKEYS; n++)
    if (keytab[n].count > 0)
      printf("%4d %s\n", keytab[n].count, keytab[n].word);
  return 0;
}
```

Razlaga in opombe.

Nadaljevanje ...

6.3 Polja struktur(4)

Nadaljevanje programa ...

```
int getword(char *word, int lim)
{ int c, getch(void);
  void ungetch(int);
  char *w = word;
  while (isspace(c = getch()))
    ;
  if (c != EOF)
    *w++ = c;
  if (!isalpha(c)) {
    *w = '\0';
    return c;
  }
  for ( ; --lim>0; w++)
    if (!isalnum(*w=getch())) {
      ungetch(*w);
      break;
    }
  *w = '\0';
  return word[0];
}
```

Razlaga in opombe.

Nadaljevanje ...

6.3 Polja struktur(5)

Nadaljevanje programa ...

```
int binsearch (char * word, struct key tab[], int n)
{ int prim, sp, zg, sr;
  sp = 0;
  zg = n-1;
  while (sp <= zg) {
    sr = (sp + zg)/2;
    if ((prim = strcmp(word,tab[sr].word))<0)
      zg = sr - 1;
    else if (prim>0)
      sp = sr + 1;
    else return sr;
  }
  return -1;
}
```

Razlaga in opombe.

6.3 Polja struktur(6)

Motivacija

- V prejšnjem primeru je `NKEYS` pomenil število ključnih besed v tabeli.
- Koliko je `NKEYS`? Kdo ga bo nastavil ali izračunal? Kaj če bi bila tabela velika?

Možnosti:

- Komponente v tabeli preštejemo sami, ročno
- Njih število izračunamo s funkcijo `sizeof`

`sizeof`

- vrne velikost svojega argumenta (v enotah, definiranih kot tipa `size_t` v `stddef.h`)
- Uporaba: `sizeof imespr;` ... velikost dane spremenljivke
  `sizeof (tip imetipa);` ... velikost spremenljivk danega tipa

Primer.

```
#define NKEYS (sizeof keytab / sizeof(struct key))
```

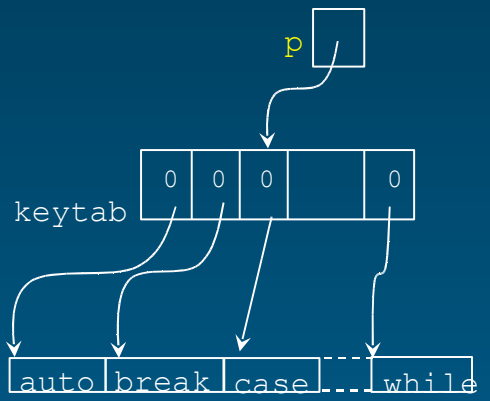
ali pa

```
#define NKEYS (sizeof keytab / sizeof keytab[0])
```

Opombe: velikost strukture ni nujno enaka vsoti velikosti njenih komponent. Razlog: poravnavanje

6.4 Kazalci na strukture(1)

Naloga: določi pogostost rezerviranih besed v vhodnem besedilu.
Opomba: Uporabili bomo še dodatni kazalec.



Program.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100
int getword(char *, int);
struct key *binsearch(char *, struct key *, int);
main()
{char word[MAXWORD];
 struct key *p;
 while (getword(word,MAXWORD) != EOF)
   if (isalpha(word[0]))
     if ((p=binsearch(word,keytab,NKEYS)) != NULL)
       p -> count++;
 for (p=keytab; p<keytab+NKEYS; p++)
   if (p -> count > 0)
     printf("%4d %s\n", p->count, p->word);
 return 0;
}
```

Razlaga in opombe.

Nadaljevanje ...

6.4 Kazalci na strukture(2)

Nadaljevanje programa ...

```
struct key *binsearch(char *word, struct key *tab, int n)
{int prim;
 struct key *sp = &tab[0];
 struct key *zg = &tab[n];
 struct key *sr;
 while (sp < zg) {
   sr = sp + (zg-sp)/2;
   if (prim = strcmp(word, sr->word)) < 0)
     zg = sr;
   else if (prim > 0)
     sp = sr + 1;
   else
     return sr;
 }
 return NULL;
}
```

Razlaga in opombe.

6.5 Rekurzivne strukture(1)

Uvod. Dvojiško iskalno drevo:

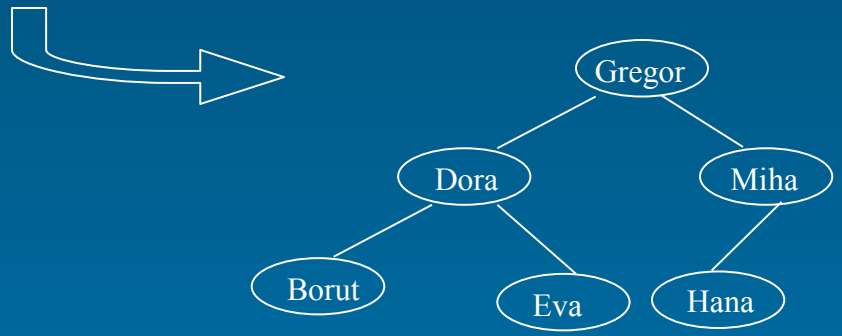
– vsaka beseda je v svojem vozlišču drevesa

– za vsako vozlišče drevesa velja:

besede v levem (oz. desnem) poddrevesu so *pred* (oz. *za*) tisto v vozlišču.

Primer. Imena naj se pojavljajo v naslednjem vrstnem redu

```
{Gregor,Dora,Eva,Borut,Miha,Hana}
```



Rekurzivno iskanje besede v drevesu:

če besede ni v korenu, nadaljuj *rekurzivno iskanje besede* v nepraznem poddrevesu, kamo sodi.

6.5 Rekurzivne strukture(2)

Naloga: določi pogostost rezerviranih besed v vhodnem besedilu.

Opomba: uporabi dvojiško iskalno drevo.

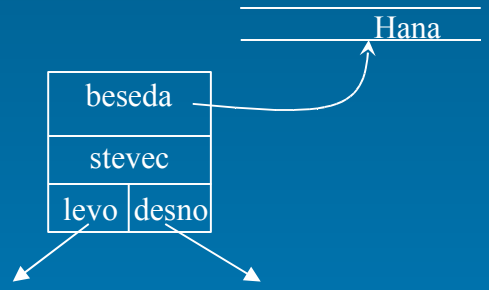
Algoritem:

ponavljaj, dokler je še kaj na vhodu:
- preberi naslednjo vhodno besedo;
- če besede še ni v drevesu, jo vstavi na ustrezno mesto;
- povečaj števec njenih pojavov.

Podatkovna struktura: vozlišča bodo predstavljena s strukturami (rekurzivno deklariranega) tipa

```
struct vozl {
    char *beseda;
    int stevec;
    struct vozl *levo;
    struct vozl *desno;
};
```

V vozliščih bomo hranili kazalce na besede:



6.5 Rekurzivne strukture(3)

Program.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100
struct vozl *azuriraj_drevo(struct vozl *,char *);
void izpisidrevo(struct vozl *);
int getword(char *, int);

main()
{struct vozl *koren;
 char word[MAXWORD];
 koren = NULL;
 while (getword(word,MAXWORD) != EOF)
   if (isalpha(word[0]))
     koren = azuriraj_drevo(koren,word);
 izpisi_drevo(koren);
 return 0;
}
```

Razlaga in opombe.

Nadaljevanje ...

6.5 Rekurzivne strukture(4)

Nadaljevanje programa ...

```
void izpisi_drevo(struct vozl *p)
 {
   if (p != NULL) {
     izpisi_drevo(p->levo);
     printf("%4d %s\n", p->stevec, p->beseda);
     izpisi_drevo(p->desno);
   }
 }
```

Razlaga in opombe.

Nadaljevanje ...

6.5 Rekurzivne strukture(5)

Nadaljevanje programa ...

```
struct vozl *novo_vozl(void);
char *shrani_bes(char *);

struct vozl *azuriraj_drevo(struct vozl *p, char *w)
{
  int prim;
  if(p == NULL) {
    p = novo_vozl();
    p -> beseda = shrani_bes(w);
    p -> stevec = 1;
    p -> levo = p -> desno = NULL;
  } else if ((prim = strcmp(w,p->beseda)) == 0)
      p -> stevec++;
    else if (prim < 0)
      p -> levo = azuriraj_drevo(p->levo,w);
    else
      p -> desno = azuriraj_drevo(p->desno,w);
   return p;
}
```

Razlaga in opombe.

Nadaljevanje ...

6.5 Rekurzivne strukture(6)

Nadaljevanje programa ...

Dinamično dodeljevanje(zaseganje) prostora

- Ko prvič preberemo kako vhodno besedo, moramo ustvariti novo vozlišče tipa `vozl`. Kako?
- Funkcija `malloc` iz knjižnice `stdlib.h`:
  - Argument je velikost zahtevanega prostora
  - Poišče tako velik del pomn.prostora za novo spremenljivko;
  - Če prostora ne najde, vrne NULL, sicer vrne kazalec na `void`.
  - Slednjega preoblikujemo z operacijo `cast` v kazalec na spremenljivko, ki je na dodeljenem prostoru.

```
#include <stdlib.h>
struct vozl *novo_vozl(void)
{
 return (struct vozl *) malloc(sizeof(struct vozl));
}

char *shrani_bes(char *s)
{char *p;
 p = (char *)malloc(strlen(s)+1);
 if (p!=NULL)
   strcpy(p,s);
 return p;
}
```

Prostor, ki smo ga zasegli z `malloc`, lahko sprostimo s `free`.

6.7 Deklaracije lastnih imen tipov(1)

`typedef` omogoča deklaracijo lastega imena podatkovnega tipa

Primer.

```
typedef int Dolzina;                          /* Dolzina ... int
Dolzina dol, maxdol, *d[3];
```

Primer.

```
typedef char *Niz;                           /* Niz ... kaz.na char
Niz p, lineptr[MAXLINES], alloc(int);
```

Primer.

```
typedef struct vozl *Kaz_drevo;
typedef struct vozl {
  char *beseda;
  int stevec;
  Kaz_drevo levo;
  Kaz_drevo desno;
} Vozlisce_drevesa;

Kaz_drevo novo_vozl(void)
{
 return (Kaz_drevo) malloc(sizeof(Vozlisce_drevesa));
}
```

6.8 Unije(1)

Unija:

- Omogoča hranjenje različnih spremenljivk na istem mestu (v različnih časih).
- Spominja na strukturo, vendar ...

Deklaracija.

```
union uoznaka {
    int ival;
    float fval;
    char *sval;
} u;
```

komponente si delijo isto mesto!

`u` je dovolj prostorna za največjo od komponent

Dostop .

`imeunije.imekomponente`

ali `kazalecnaunijo->imekomponente`

Npr. `u.ival++;`

`k->ival++;`

Včasih je potrebna evidenca (npr. v spremenljivki `utype`) o tem, kaj trenutno gosti unija `u`:

`INT, FLOAT, STRING` ... definirane konstante

```
int utype;
...
if (utype == INT)
  ... unija u gosti celo število, zato uporabimo u.ival
else if (utype == FLOAT)
  ... unija u gosti realno število, zato uporabimo u.fval
else if (utype == STRING)
  ... unija u gosti kazalec na znak, zato uporabimo u.sval
```

6.8 Unije(2)

Unije lahko gnezdijo v drugih podatkovnih strukturah.

Primer.

```
struct {
  char *name;
  int flags;
  int utype;
  union {
    int ival;
    float fval;
    char *sval;
  } u;
} symtab[NSYM];
```

Dostop.

npr. `symtab[i].u.ival`

npr. `symtab[i].u.sval[0]` ali pa npr. `*symtab[i].u.sval`

6.9 Bitna polja(1)

Uvod:

– Včasih želimo shraniti več spremenljivk v eno besedo. Vsaka taka spremenljivka je strnjeno zaporedje enega ali več bitov (bitno polje).

Deklaracija.

```
struct {
 unsigned int a : 1;
 unsigned int b : 1;
 unsigned int c : 1;
} zastavice;
```


```
struct {
  unsigned int d : 3;
} trije;
```

Dostop.

```
zastavice.a = zastavice.b = zastavie.c = 0;  ...
if (zastavice.a == 0 && zastavice.b == 0) ...
if (trije.d = 5) ...
```

Opombe:

– samo dvopičje in velikost za bitno polje brez imena (da z njim preskočimo nekaj bitov)

– `:0` povzroči premik naslednjega bitnega polja v naslednjo besedo (na rob naslednje besede)

– bitno polje ni indeksirano kakor polje

– nima lastnega naslova, zato nad njim ne deluje operacija `&`

Zadnja prosojnica