

Strukture podataka i algoritmi

Neven Soić

Institut Ruđer Bošković

Zavod za eksperimentalnu fiziku

2. krilo, soba 129A

Email: soic@lnr.irb.hr

Telefon: 4561026

<http://lnrpc2.irb.hr/soya/nastava>

Školska godina 2006/2007

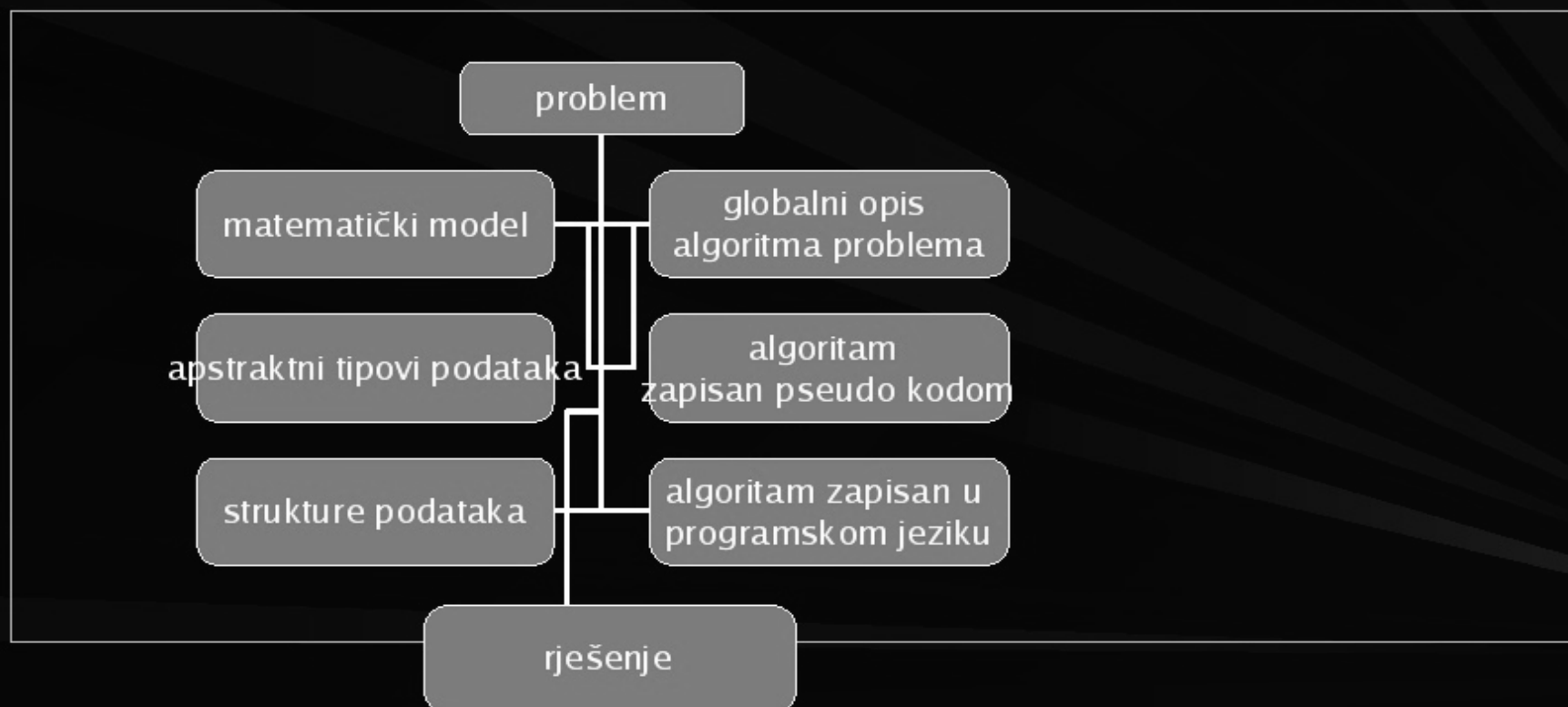
Općenito o predmetu

- Sadržaj kolegija:
- **Uvod:** Pojam tipa, apstraktnog tipa i struktura podataka. Elementi od kojih se gradi struktura: polje, zapis, pokazivač (pointer), kursor. Pojam algoritma, zapisivanje i analiziranje algoritama.
- **Pregled raznih apstraktnih tipova:** Lista, stog (stack), red, uređeno stablo, binarno stablo, skup, rječnik, prioritetni red, preslikavanje.
- **Pregled raznih struktura podataka:** Vezana lista i druge vezane strukture, hash tablica, binarno stablo traženja, gomila (heap).
- **Algoritmi za obavljanje osnovnih operacija nad strukturama:** Ubacivanje i izbacivanje podataka, pretraživanje, ispis sadržaja i sl.
- **Primjena opisanih struktura u složenijim algoritmima:** Sortiranje i sažimanje nizova podataka, izvrednjavanje aritmetičkih izraza, razni rekurzivni postupci.
- **Općenite tehnike (strategije) za konstrukciju algoritama:** "Podijeli pa vladaj", dinamičko programiranje, "pohlepni" pristup, "backtracking".

Osnovni pojmovi

- Strukture podataka – statički aspekt programa – ono sa čime se radi
- Algoritmi – dinamički aspekt programa – ono što se radi
- međusobno su vezani i utječu jedno na drugo
- Pojmovi koji se često koriste:
- tip podataka: skup vrijednosti koje neki podatak može poprimiti
- apstraktni tip podataka: zadan jedan ili više tipova podataka, te jedne ili više operacija (funkcija) – operandi i rezultati operacija su podaci danog tipa
- Struktura podataka: skupina varijabli u programu i veza među njima
- Algoritam: konačni niz instrukcija od kojih svaka ima jasno značenje i može biti izvršena u konačnom vremenu. Za bilo koje vrijednosti ulaznih podataka algoritam mora završavati nakon konačnog broja koraka.
- Implementacija apstraktnog tipa podataka: konkretna realizacija ATP u programu, sastoji se od definicije za strukturu podataka (prikaz podataka iz ATP) i od potprograma (izvedba operacija iz ATP pomoću odabranih algoritama). Za isti ATP moguće različite implementacije.

Rješavanje problema metodom postepenog profinjavanja



Algoritam

- Pojam algoritma je uveden prvo u matematici, danas se često rabi u računalnim znanostima
- Precizno opisan način rješenja nekog problema
- Jednoznačno određuje što treba napraviti, svaki korak algoritma mora biti nedvosmislena, rigorozno definirana operacija - DEFINITNOST
- Moraju biti definirani početni objekti koji pripadaju nekoj klasi objekata na kojima se obavljaju operacije - ULAZ
- Kao ishod algoritma pojave se završni objekt(i) ili rezultat(i) - IZLAZ
- Konačni broj koraka; svaki korak opisan instrukcijom, daje rezultat u konačnom vremenu i konačnom broju koraka – KONAČNOST
- Mora se moći izvesti samo uz pomoć olovke i papira u konačnom vremenu – EFEKTIVNOST
- Postupak za rješavanje nekog masovnog problema – općenito pitanje na koje je potrebno naći odgovor, a koje ima parametre koji ostaju neodređeni prilikom zadavanja problema
- Specificiranjem svih parametara masovnog problema dobiva se instanca problema
- algoritam rješava masovni problem ako rješava svaku pojedinu instancu problema

Primjer: apstraktni tip podatka Complex

- Scalar – bilo koji tip za koji su definirane operacije zbrajanja i množenja
- Complex – podaci ovog tipa su uređeni parovi podataka tipa Scalar
- ADD(z1, z2, &z3) – računa zbroj z3 za zadane z1 i z2 tipa Complex. $z1=(x1,y1)$, $z2=(x2,y2)$, $z3=(x1+x2,y1+y2)$
- MULT(z1, z2, &z3) – računa umnožak z3 za zadane z1 i z2 tipa Complex. $z1=(x1,y1)$, $z2=(x2,y2)$, $z3=(x1*x2-y1*y2,x1*y2+y1*x2)$
- Struktura podataka za prikaz kompleksnog broja:

```
struct complex {  
    scalar Re;  
    scalar Im;  
};
```
- Implementacija ATP Complex: definiranje tipa i funkcija oblika

```
void ADD(Complex z1, Complex z2, Complex *z3) {...}  
void MULT(Complex z1, Complex z2, Complex *z3) {...}
```

Razvoj algoritma: primjer množenja kompleksnih brojeva

- Za izradu efikasnog i brzog algoritma potrebno je dobro razumijevanje problema koji se rješava i metoda koje se koriste u rješavanju
- Jednostavan primjer množenja kompleksnih brojeva u kojem je potrebno znanje matematike:
 - 1) Algoritam zasnovan na znanju množenja realnih brojeva: množiti svaki element sa svakim, zatim sakupiti zajedno brojeve koji nemaju i imaju "i". Algoritam za to ima mnogo koraka i ispitivanja, spor i kompliciran, a rezultat je jednostavan izraz
 - 2) Algoritam koji direktno koristi definiciju množenja kompleksnih brojeva iz matematike: 4 množenja, 1 zbrajanje, 1 oduzimanje

$$(a+ib)*(c+id) = (ac - bd) + i(ad + bc)$$

kod većine kompjutera množenje je mnogo sporiji proces od zbrajanja, pa će se krajnji rezultat dobiti brže upotrebom izraza

$$(a+ib)*(c+id) = (ac - bd) + i [(a+b)(c+d) - ac - bd]$$

gdje se obavlja 3 množenja, 2 zbrajanja i 3 oduzimanja

Procedura

- Postupak koji ima sva svojstva kao i algoritam, ali ne mora završiti u konačnom broju koraka jest računalna procedura. U jeziku C to može biti `void` funkcija
- Primjeri za proceduru:
 - Operacijski sustav računala
 - Uređivač teksta
- Vrijeme izvođenja mora biti "razumno"
- Primjer:
 - Algoritam koji bi izabirao potez igrača šaha tako da ispita sve moguće posljedice poteza, zahtijevao bi milijarde godina na najbržem zamislivom računalu.



Algoritmi i programi

- *Program* - Opis algoritma koji u nekom programskom jeziku jednoznačno određuje što računalo treba napraviti.
- *Programiranje* - naučiti sintaksu nekog proceduralnog jezika i steći osnovna intuitivna znanja o algoritmizaciji problema opisanog riječima.
- Programiranje: razvijanje struktura podataka i razvijanje algoritama
- Algoritmi + strukture podataka = PROGRAMI
 - kako osmisliti algoritme
 - kako strukturirati podatke
 - kako formulirati algoritme
 - kako verificirati korektnost algoritama
 - kako analizirati algoritme – procjena vremena izvršavanja (broja operacija) algoritma
 - kako provjeriti (testirati) programe
- Postupci izrade algoritama nisu jednoznačni te zahtijevaju i kreativnost. Inače bi već postojali generatori algoritama. Znači da se (za sada?) gradivo ovog predmeta ne može u potpunosti algoritmizirati.
- Koristit će se programski jezik C. Za sažeti opis složenijih algoritama može se koristiti pseudokod.

Elementi od kojih se grade strukture podataka

- Manje cjeline se udružuju u veće i međusobno su vezane
- Uvode se posebni nazivi za cjeline, načine udruživanja i načine povezivanja
- Čelija (cell): varijabla koju promatramo kao zasebnu cjelinu; svaka ima svoj tip i adresu
- Polje (array): mehanizam udruživanja manjih dijelova strukture u veće. Polje čini više ćelija istog tipa (element polja) pohranjeno na uzastopnim adresama; broj ćelija zadan i nepromjenljiv
- Zapis – slog: mehanizam udruživanja manjih dijelova strukture u veće. Čelije (komponente zapisa) ne moraju biti istog tipa, ali su pohranjene na uzastopnim adresama. Broj, redoslijed i tip ćelija su zadani i nepromjenljivi.
- Pokazivač (pointer): uspostavlja vezu između dijelova struktura, ćelija koja pokazuje neku drugu ćeliju; njen sadržaj je adresa druge ćelije
- Kursor: uspostavlja vezu između dijelova struktura, ćelija koja pokazuje na element nekog polja

Statičke strukture podataka

■ Osnovni tipovi:

- `char` - znakovni tip (1 By)
- `int` - cjelobrojni tip (standardno 4 By)
- `float` - realni tip (4 By)
- `double` - realni tip u dvostrukoj preciznosti (8 By)

■ Razlika između preciznosti (*precision*) i točnosti (*accuracy*).

- Preciznost se iskazuje brojem prvih važećih točnih znamenki, a točnost je bliskost stvarnoj (nepoznatoj) vrijednosti.
- Za dovoljnu točnost potrebna je adekvatna preciznost, ali preciznost ne implicira automatski točnost jer su iskazane znamenke mogle nastati na temelju npr. pogrešnog mjerenja.

● Prefiksi ili kvalifikatori: odnose se na cijele brojeve. Duljina ovisi o procesoru.

- `short` - smanjuje raspon vrijednosti (2 By)
- `long` - eksplicitno definira duljinu od 4 By
- `signed` - dozvoljava pridruživanje pozitivnih i negativnih vrijednosti
- `unsigned` - dozvoljava pridruživanje samo pozitivnih vrijednosti

Memorijske klase

- `memorijska_klasa` utvrđuje postojanost (trajnost) i područje važenja varijabla u memoriji ovisno o mjestu deklaracije u programu.
- Postoje 4 memorijske klase:
 - `auto` automatska (vrijedi lokalno unutar funkcije)
 - `extern` vanjska (vrijedi globalno unutar programa)
 - `static` statička (vrijedi lokalno unutar funkcije ili modula)
 - `register` registarska (vrijedi lokalno unutar funkcije, ali koristi CPU registre)
- Obično se ključna riječ `auto` ne navodi, te su sve lokalne varijable i polja definirani unutar neke funkcije automatske klase. Vanjska klasa ukazuje na varijable i polja koji su globalni (zajednički) za sve funkcije unutar programa i obično se `extern` ne navodi jer položaj izvan funkcije ukazuje na to.
- Statička klasa se koristi onda kada se vrijednost varijable ili članova polja treba zadržati nakon izlaska i ponovnog povratka u neku funkciju.
- U opisu algoritama izbjegavat će se globalne varijable da bi se eksplicitno ukazalo na razmjenu informacija među funkcijama.

Niz znakova, Logička vrijednost

■ Niz znakova

Z	a	g	r	e	b	\0
---	---	---	---	---	---	----

- Numerička vrijednost 0 oznaka je kraja znakovnog niza.

```
char ime_niza[duljina_niza+1];
```

■ Logička vrijednost

- U nekim jezicima postoji poseban tip podataka LOGICAL.
- U C-u se svaki tip podatka može koristiti kao logički.

```
#define TRUE 1
```

```
#define FALSE 0
```

Polje

- Polje je *podatkovna struktura* gdje isto ime dijeli više podataka
- Svi podaci u nekom polju moraju biti istog tipa i iste *memorijske klase*
- Elementi (članovi) polja se identificiraju *indeksom*
- Indeks može biti nenegativni cijeli broj (konstanta, varijabla, cjelobrojni izraz)

`x[0]` `x[9]` `x[n]` `x[MAX]` `x[n+1]` `x[k/m+5]`

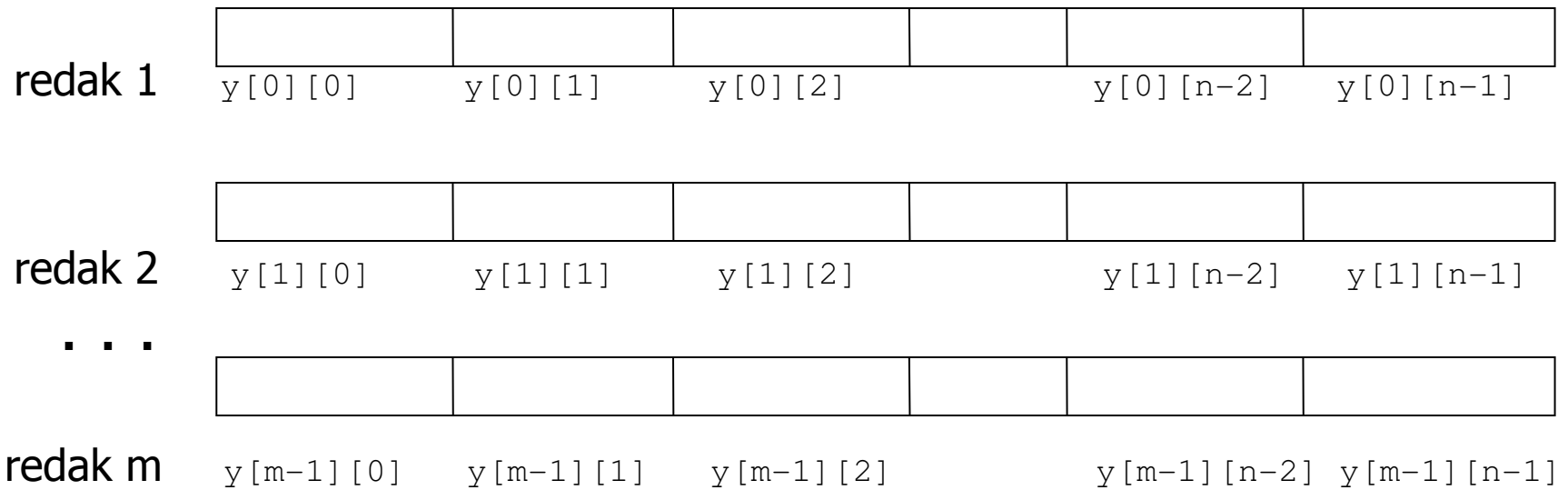
- Polje može biti
 - **jednodimenzionalno (vektor)**

```
#define N 100  
float x[N];
```

<code>x[0]</code>	<code>x[1]</code>	<code>x[2]</code>	<code>...</code>	<code>x[n-2]</code>	<code>x[n-1]</code>
-------------------	-------------------	-------------------	------------------	---------------------	---------------------

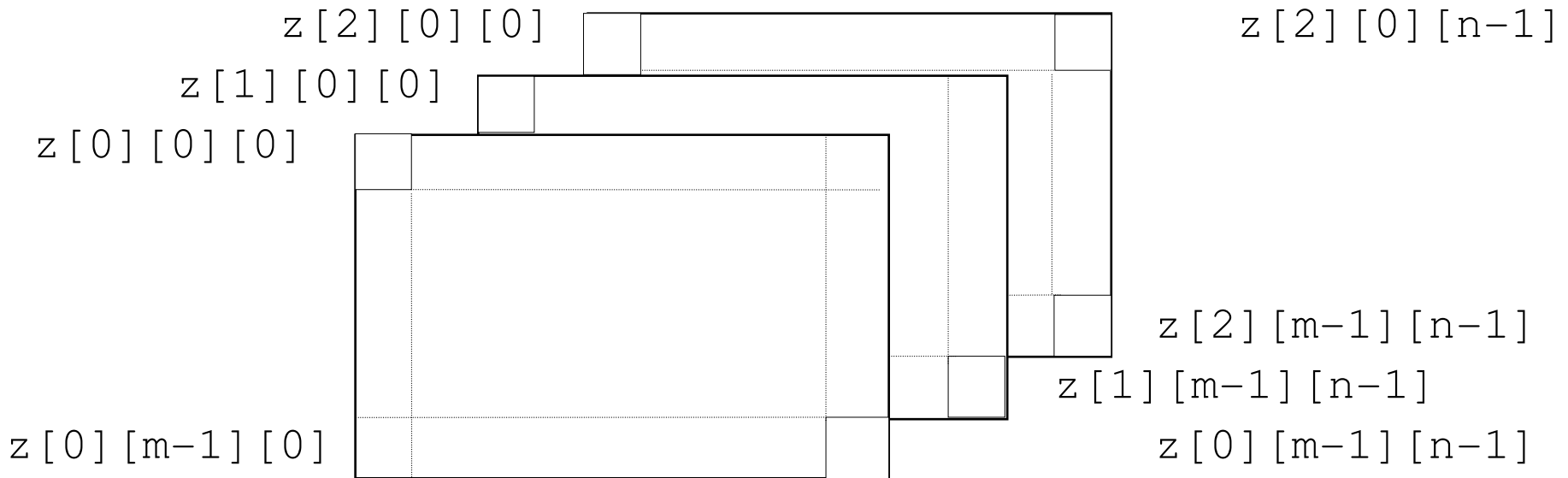
● dvodimenzionalno (matrica, tablica)

```
#define N 100  
#define M 50  
float y[M][N];
```



● trodimenzionalno i višedimenzionalno

```
# define N 100  
# define M 50  
float z[3][M][N];
```

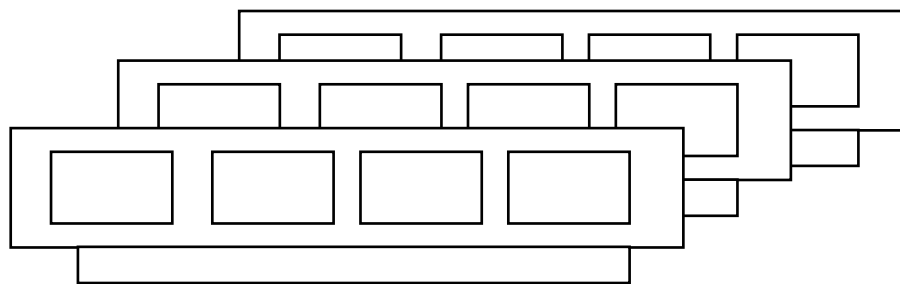


■ Opći oblik naredbe za deklaraciju polja:

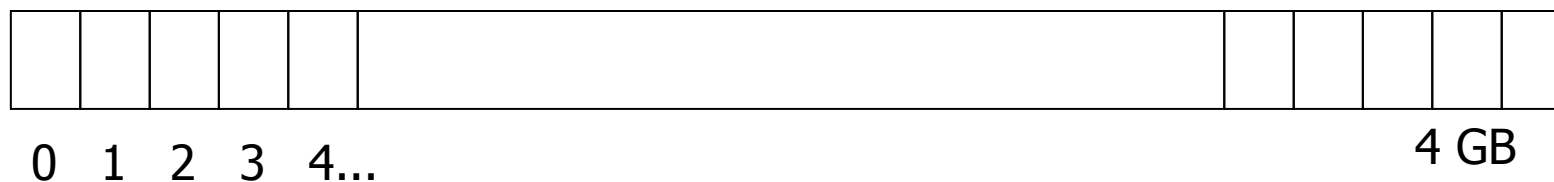
```
memorijska_klasa tip_podatka polje[izraz1][izraz2]...
```


Pokazivač (Pointer)

Memorija računala:



zapravo je kontinuirani niz bajtova:

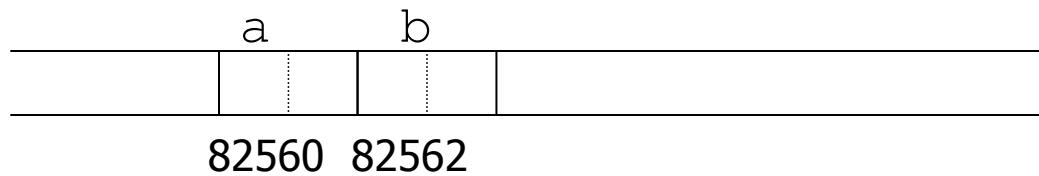


Svaki bajt ima svoj redni broj: *adresu*

Varijable i pokazivači

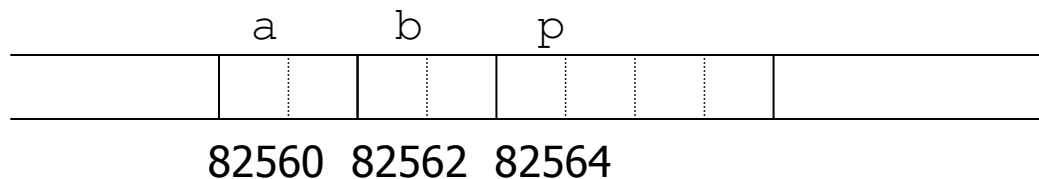
- Deklaracijom varijabli rezervira se prostor u memoriji na slobodnim adresama, npr.:

```
short a, b;
```



- Deklaracijom pokazivača rezervira se prostor u memoriji u duljini 4 bytea kako bi se pohranila bilo koja adresa u adresnom prostoru do 4GB:

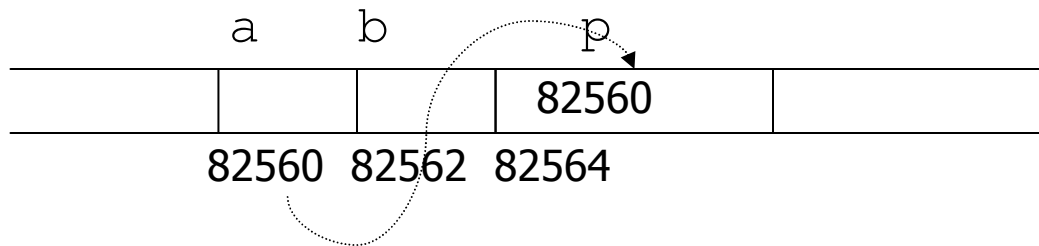
```
short *p;
```



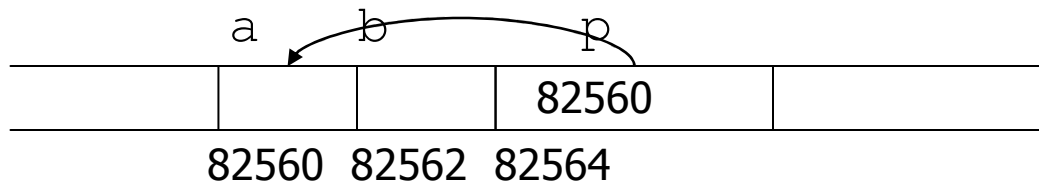
- Važno je primijetiti da deklaracijom niti jednoj od varijabli nije inicijalizirana vrijednost

- Vrijednost pokazivača svakako treba postaviti prije uporabe

`p = &a;`

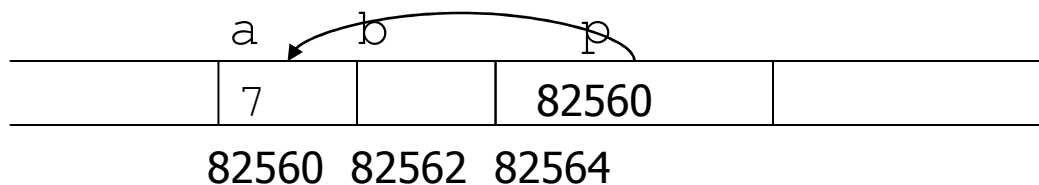


- Ovime je ostvareno pokazivanje pokazivača `p` na varijablu `a`



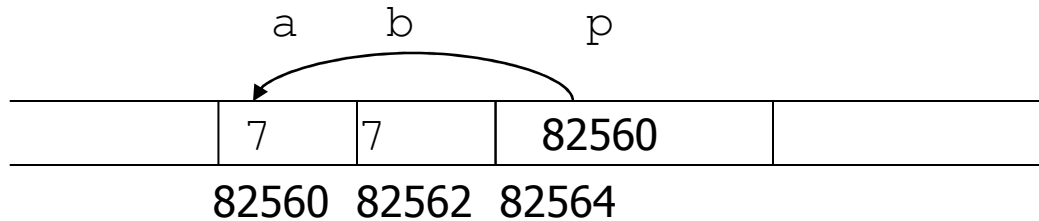
- Sada je moguće indirektno postaviti vrijednost varijable `a`

`*p = 7;`



- Indirektno se može i koristiti vrijednost varijable `a`

```
b = *p;
```



- Vrijednost pokazivača može se postaviti i rezervacijom slobodne memorije

```
main () {  
    short int *p;  
    p = (short int *) malloc (sizeof (short int));  
    *p = 7;  
}
```

- Valja razlikovati:

- `p` pokazivač veličine 4 bajta na broj tipa `short int`
- `*p` broj tipa `short int` veličine 2 bajta. Ne mora postojati!

Aritmetika s pokazivačima

- Aritmetika s pokazivačima podrazumijeva korištenje jedinica koje odgovaraju duljini (By) podatkovne strukture na koju pokazivač pokazuje.
- Uvećanje za 1 pokazivača na strukturu dugačku 4 By znači da se njegova vrijednost uvećava za 4. Ako je struktura dugačka 8 By, uvećanje pokazivača za 1 povećava vrijednost pokazivača za 8 itd.

- Primjer:

(long = 4 By, double = 8 By)

```
long dugi; double dupli;  
long *pdugi; double *pdupli;  
pdugi = &dugi;  
pdupli = &dupli;  
++pdugi;  
pdupli = pdupli + 2;
```

Vrijednosti	
<u>pdugi</u>	<u>pdupli</u>
?	?
128560	?
128560	128564
128564	128564
128564	128580

Polja i pokazivači

```
#include <stdio.h>
main () {
    int x[4] = {1,2,3,4};
    printf ("%d %d\n", *x, *(x+1));
    f (x);
}
void f (int *x) {           ili           void f (int x[]) {
    printf ("%d %d\n", *x, x[0]);
    ++x;
    printf ("%d %d %d\n", *x, x[0], *(x-1));
}
```

Ispis na zaslonu:

1 2

1 1

2 2 1

	x[0]	x[1]	x[2]	x[3]	
	1	2	3	4	

Zapisi (strukture)

■ Typedef deklaracija

```
typedef stari_tip novi_tip;
```

npr.

```
typedef unsigned size_t;
```

```
typedef int redni_broj;
```

```
typedef short logical;
```

```
redni_broj i, j;
```

```
size_t velicina;
```

```
logical da_ne;
```

■ Definiranje strukture

- Strukture podataka čiji se elementi razlikuju po tipu:

```
struct ime_strukture {  
    tip_elementa_1 ime_elementa_1;  
    tip_elementa_2 ime_elementa_2;  
    ...  
    tip_elementa_n ime_elementa_n;  
};
```

- **Primjer:**

```
struct osoba {  
    char jmbg[13+1];  
    char prezime[40+1];  
    char ime[40+1];  
    int visina;  
    float tezina;  
};
```

Ovime nije deklariran konkretan zapis, već je samo definirana struktura zapisa.

Deklaracija konkretnih zapisa:

```
struct ime_strukture zapis1, zapis2, ... , zapisN;
```

npr.

```
struct osoba o1, o2, zaposleni[500];
```


- Moguće je definiranje statičke podatkovne strukture proizvoljne složenosti jer pojedini element može također biti `struct`:

```
struct student {
    int maticni_broj;
    struct osoba osobni_podaci;
    struct adresa adresa_roditelja;
    struct adresa adresa_u_Zagrebu;
    struct osoba otac;
    struct osoba majka;
};
```

Alternativno, korištenjem naredbe `typedef`:

```
typedef struct {
    char jmbg[13+1];
    char prezime[40+1];
    char ime[40+1];
    int visina;
    float tezina;
} osoba;
osoba o1, o2, zaposleni[500];
```

```
typedef struct {
    int maticni_broj;
    osoba osobni_podaci;
    adresa adresa_roditelja;
    adresa adresa_u_Zagrebu;
    osoba otac;
    osoba majka;
} student;
```

■ Referenciranje elemenata zapisa

```
zapis.element = vrijednost;
vrijednost = zapis.element;
```

npr:

```
student pero;
pero.majka.tezina = 92.5;
```

■ Pokazivač na strukturu

```
struct osoba *p;
```

*p - zapis o osobi

p - adresa zapisa o osobi

● Referenciranje na element strukture preko pokazivača

```
p->prezime ili (*p).prezime
```

Procedure

- Programi se sastoje od procedura. Prva pozvana procedura je glavni program. Kad glavni program završi, slijedi povratak u operacijski sustav. Inače, povratak iz procedure je uvijek na onu proceduru koja ju je pozvala. Uobičajena je podjela na funkcije (*function*) koje imaju od nula do više ulaznih argumenata i vraćaju jedan rezultat, te na općenite potprograme (*subroutine*) koje rezultat predaju argumentima.
- U jeziku C sve procedure su funkcije koje daju rezultat nekog od tipova podataka, ali mogu mijenjati i vrijednost argumenata.
- Posebni slučajevi:
 - Glavni program: `main`
 - Potprogram koji u imenu ne vraća vrijednost: `void`

Razmjena podataka između funkcija

- globalne varijable
- argumenti navedeni u zagradi, prenose se vrijednosti (*call by value*)
- za prijenos vrijednosti u pozivajuću funkciju koriste se kod poziva funkcije kao argumenti adrese, a u definiciji funkcije argumenti su pokazivači (*call by reference*).
- Ako funkcija mora predati rezultat preko argumenata, nužno se koristi *call by reference*.

- Ulazno-izlazne operacije:
- Za slijedno čitanje/pisanje preko standardnih ulazno-izlaznih jedinica koristit će se odgovarajuće C funkcije ili naredbe pseudokoda:
 - ulaz (*lista adresa argumenata*)
 - izlaz (*lista argumenata*)
- Kod čitanja je dakle nužan *call by reference*, dok kod ispisa može poslužiti i *call by value*.

Operacije u programskom jeziku C

- Operacija izjednačavanja: =
- Aritmetičke operacije : +, -, *, /, %
- Operatori za skraćeno pisanje nekih aritmetičkih izjednačavanja:
 - i++; ili ++i ; odgovara i = i + 1;
 - i--; ili --i; odgovara i = i - 1;
 - x = a * b++; odgovara x = a * b ; b = b + 1;
 - x = --i * (a + b) odgovara i = i - 1 ; x = i * (a + b);
 - i += 10; odgovara i = i + 10; i -= 10; odgovara i = i - 10;
 - i *= 10; odgovara i = i * 10; i /= 10; odgovara i = i / 10;

Operator pretvorbe tipa (cast) : primjer

```
type1 i;  
type2 j;  
j = sqrt ( (type2) i);
```

- Uspoređivanje: ==, >, <, >=, <=, !=
- Logički operatori: &&, ||, !

Elementi izrade programa

- Normalan programski slijed

naredba_1
naredba_2
naredba_3

...

- Bezuvjetni skok

Pseudokod:

idi na oznaka_naredbe

C:

`goto oznaka_naredbe;`

- Grananje

- S: Oznaka za jednu ili više naredbi, odnosno programski odsječak
- Uvjetno obavljanje naredbi (jednostrana selekcija)

Pseudokod:

C:

ako je (logički_izraz) onda
| S

```
if (logički_izraz) {  
    S;  
}
```

- Grananje (dvostrana selekcija)

<u>Pseudokod:</u>	<u>C:</u>
<u>ako je</u> (<i>logički_izraz</i>) <u>onda</u>	if (<i>logički_izraz</i>) {
<i>S_1</i>	<i>S_1</i> ;
<u>inače</u>	} else {
<i>S_2</i>	<i>S_2</i> ;
	}

- Višestruko grananje (višestrana selekcija)

<u>Pseudokod:</u>	<u>C:</u>
<u>ako je</u> (<i>logički_izraz_1</i>) <u>onda</u>	if (<i>logički_izraz_1</i>) {
<i>S_1</i>	<i>S_1</i> ;
<u>inače ako je</u> (<i>logički_izraz_2</i>) <u>onda</u>	} else if (<i>logički_izraz_2</i>) {
<i>S_2</i>	<i>S_2</i> ;
<u>inače ako je</u> (<i>logički_izraz_3</i>) <u>onda</u>	} else if (<i>logički_izraz_3</i>) {
<i>S_3</i>	<i>S_3</i> ;
...	...
<u>inače</u>	} else {
<i>S_0</i>	<i>S_0</i> ;
	}

■ Skretnica

Pseudokod:

skretnica (*vrijednost*)

slučaj *C1*

| *S_1*

slučaj *C2*

| *S_2*

...

slučaj *Cn*

| *S_n*

inače

| *S_{n+1}*

C:

```
switch( cjelobrojna vrijednost) {
```

```
case C1:
```

```
    S_1;
```

```
    break;
```

```
case C2:
```

```
    S_2;
```

```
    break;
```

```
...
```

```
case Cn:
```

```
    S_n;
```

```
    break;
```

```
default:
```

```
    S_{nplus1};
```

```
}
```

- Ostvariti skretnicu naredbom `if - else`.

```
if (vr == C1) {  
    S1;  
} else if (vr == C2) {  
    S2;  
}  
  
...  
} else if (vr == Cn) {  
    Sn;  
} else {  
    S_nplus1;  
}
```

- Ako se iz skretnice izbace naredbe `break`, obavljaju se slijedno sve naredbe iza one gdje je prvi put zadovoljeno: `vr == Ci`
- Skretnica bez `break` korištenjem naredbe `if`.

```
nadjen = 0;  
if (vr == C1) {  
    S1;  
    nadjen = 1;  
}  
if (vr == C2 || nadjen) {  
    S2;  
    nadjen = 1;  
}  
...  
if (vr == Cn || nadjen) {  
    Sn;  
}  
S_nplus1;
```

■ Programska petlja

- Petlja s ispitivanjem uvjeta ponavljanja na početku

<u>Pseudokod:</u>	<u>C:</u>
<u>dok je</u> (<i>logički_izraz</i>) <u>činiti</u>	<code>while</code> (<i>logički_izraz</i>) {
<i>S</i>	<i>S</i> ;
	}

- Petlja s ispitivanjem uvjeta ponavljanja na kraju

- U nekim programskim jezicima postoji oblik: REPEAT...UNTIL

<u>Pseudokod:</u>	<u>C:</u>
<u>ponavljati</u>	<code>do</code> {
<i>S</i>	<i>S</i> ;
<u>do</u> (<i>logički_izraz</i>)	} <code>while</code> (! <i>logički_izraz</i>)

- Standardna petlja u C-u:

<u>Pseudokod:</u>	<u>C:</u>
<u>ponavljati</u>	<code>do</code> {
<i>S</i>	<i>S</i> ;
<u>dok je</u> (<i>logički_izraz</i>)	} <code>while</code> (<i>logički_izraz</i>);

- Petlja s poznatim brojem ponavljanja

Pseudokod:

C:

```
za  $i := poc$  do kraj (korak  $k$ ) činiti   for ( $i=poc; i \leq kraj; i=i+k$ ) {  
  |  $S$                                       $S;$   
                                           }
```

- Primjer: Realizacija istog odsječka petljom `while`:

```
 $i = poc;$   
while ( $i \leq kraj$ ) {  
   $S;$  // niz naredbi koje ne mijenjaju vrijednost za  $i$   
   $i += k;$   
}
```

ili općenitije:

```
 $i = poc;$   
while ( $(i - kraj) * k \leq 0$ ) {  
   $S;$  // niz naredbi koje ne mijenjaju vrijednost za  $i$   
   $i += k;$   
}
```

- Skok iz petlje

<u>Pseudokod:</u>	<u>C:</u>
<u>izađi iz petlje</u>	<code>break;</code>
<u>skoči na kraj petlje</u>	<code>continue;</code>

- Beskonačna petlja

<u>Pseudokod:</u>	<u>C:</u>
<u>ponavljaj</u>	<code>while(1) {</code>
S	<code>S;</code>
<u>zauvijek</u>	<code>}</code>

- U algoritmima ovakva petlja nije dopuštena jer je u suprotnosti sa zahtjevom da postupak bude konačan.

- U tijelu petlje mora postojati barem jednom ispitivanje uvjeta za izlazak iz petlje:

```
while(1) {  
    S1;  
    if (logički_izraz) break;  
    S2;  
    ...  
}
```

ili

```
while(1) {  
    S1;  
    if (logički_izraz) goto oznaka_naredbe;  
    S2;  
    ...  
}
```

- Naredbu `goto` treba izbjegavati ako je ikako moguće. Smanjuje se mogućnost pogreške i olakšava testiranje programa ako svaka programska cjelina (npr. petlja) ima samo jedan ulaz i jedan mogući izlaz.

Pisanje strukturiranih programa

■ Osnovni naputci

- specifikacija ulaznih i izlaznih varijabli (u C obvezatno!)
- definicija lokalnih varijabli
- tok programa prema dolje, osim petlji i neizbježne goto naredbe
- poravnati naredbe iste razine za povećanje čitkosti
- dokumentacija kratka ali smisljena
- koristiti potprograme gdje je to moguće

■ Izbor vrste petlje

- Primjer: Čitati članove nekog skupa nenegativnih brojeva sve dok njihova suma ne premaši neki predviđeni iznos n .

```
y = 0;
do {
    ulaz(&x);
    y += x;
} while (y <= n);
```

```
y = 0;
while (y <= n){
    ulaz(&x);
    y += x;
```

Programi nisu funkcionalno identični jer u prvom slučaju se uvijek obavlja barem jedno čitanje, pa program ispravno radi samo za $n \geq 0$. U drugom slučaju, ako je $n < 0$, smatra se da je bez čitanja ijednog podatka postupak završen.

- **Primjer:** Pročitati n vrijednosti i obraditi ih procedurom PROCES. Korištenjem petlje

```
while:
```

```
i = 0;
```

```
while (i < n) {
```

```
    ulaz(&x);
```

```
    PROCES(x);
```

```
    i++;
```

```
}
```

S obzirom da je unaprijed poznat broj ponavljanja, primjerenije je koristiti petlju s poznatim brojem ponavljanja:

```
for (i = 0; i < n; i++) {
```

```
    ulaz(&x);
```

```
    PROCES(x);
```

```
}
```

Ne radi se samo o manjem broju naredbi, nego se smanjuje i mogućnost pogreške u pisanju programa.

- Primjer: Treba pročitati i obraditi skup podataka sve dok ne naiđe oznaka kraja podataka (EOF). Realizacija petljom `while`:

```
ulaz(&x);  
while (x != EOF) {  
    PROCES(x);  
    ulaz(&x);  
}
```

Treba dakle pročitati prvi podatak, obrađuje ga se samo ako postoji, a nakon obrade čita se sljedeći podatak koji se obradi u idućem prolasku kroz petlju.

Bolje je rješenje skokom iz petlje:

```
while (1) {  
    ulaz(&x);  
    if (x == EOF) break;  
    PROCES(x);  
}
```

Ne narušava se željeni princip da iz jednog programskog bloka postoji samo jedan izlaz.

- Ako ima dva kriterija za završetak petlje: kraj podataka ili da rezultat procedure PROCES, pohranjen u varijabli y bude nula:

```
do {
    ulaz(&x);
    if (x == EOF) break;
    PROCES(x, &y);
} while (y != 0);
if (x != EOF) {
    ...
} else {
    ...
}
```

Na izlasku iz programskog segmenta ne zna se zbog čega je petlja završila pa se to mora ispitivati.

■ Korišćenje naredbe `case`

- Uvijek se može uporabiti elementarnija naredba `if-else`. Naredba `case` je u prednosti kad ima mnogo ravnopravnih grananja.