PROGRAMIRANJE 2

Matej Mihelčić

Zagreb, 29.04.2024.

Ova je skripta nastala kao objedinjeni dokument predavanja iz kolegija Programiranje 2 od pokojnog prof. dr. sc. Saše Singera. Skripta sadrži dodatke, modifikacije i adaptacije koje je kroz godine unio doc. dr. sc. Matej Mihelčić.

Uvod

Kolegij `Programiranje 2` je jednosemestralni kolegij sljedbenik kolegija `Programiranje 1`. Osnovni ciljevi kolegija su upoznati studente s tehnikama programiranja koje omogućuju realizaciju (osnovnih) algoritama, te učenje konkretnog proceduralnog programskog jezika `C` kao sredstva za realizaciju tih algoritama.

Programski jezik `C` je čisti **proceduralni jezik** srednje razine. Njegova sintaksa visoke razine je prilagođena i razumljiva čovjeku, za razliku od jezika niske razine, kao npr. `Asembler`, dok je po dizajnu još uvijek blisko vezan uz građu računala – i omogućava razumijevanje iste (dinamička alokacija memorije, lokalne/statičke varijable, naredbe za spremanje varijabli u `cache` memoriju računala itd). Građa programskog jezika `C` je refleksija njegove prvobitne namjene, biti programski jezik za stvaranje jezgre operacijskog sustava `UNIX`. `C` je strogo tipizirani jezik (uz svaku varijablu treba stajati odgovarajući tip). Iako navedeno svojstvo jezika, često zahtjeva pisanje nešto dužih izvornih kodova, istovremeno osigurava stvaranje transparentnih, relativno razumljivih programa. Glavni nedostatak jezika je manjak kontrole i provjera pri memorijskim manipulacijama što može dovesti do neočekivanog, nedefiniranog ponašanje, te u nekim slučajevima omogućiti maliciozne radnje. Korišteni podskup programskog jezika `C`, tzv. `C99` bez korištenja polja varijabilne duljine i `GOTO` naredbi, omogućava podučavanje o radu i građi računala, ali i o osnovama algoritamskog razmišljanja, dizajnu i izvršavanju algoritama te o standardima o pisanju izvornog koda. Kao iznimno brz jezik, `C` i njegovo objektno orijentirano proširenje `C++` (često nazvani jednim imenom `C/C++`) se koristi pri implementaciji algoritama u raznim područjima programiranja (npr. za izradu brzih `SAT` solvera, razvoj algoritama strojnog učenja i dubokog učenja, razvoju paralelnih i distribuiranih algoritama) u znanstvenom računanju, financijama i bankarstvu (softver za algoritamsko trgovanje visokih performansi). `C/C++` je glavni izbor pri programiranju uređaja s malom količinom memorije (eng. `embedded devices`), npr. računala

i čipova u automobilima, perilicama posuđa, pečnicama i sličnim uređajima (`Arduino`, `Raspberry Pi`). `C/C++` je također jako popularan programski jezik u robotici. Korišten je za stvaranje većine biblioteka korištenih od strane znantno sporijeg, interpretiranog, dinamički tipiziranog, zbog svoje jednostavne sintakse i orijentiranosti *korisnicima računala* trenutno popularnog jezika visoke razine `Python`. Programski jezik `C` je korišten pri izgradnji popularnih matematičkih alata kao što su `Wolfram Mathematica`, `Maple`, `Matlab`, `GNU Octave` itd. i za razvoj jezgre većine današnjih operacijskih sustava `Windows`, `UNIX`, `Linux`, `macOS`, djelomično i `Android`.

Studenti će na kolegiju utvrditi i osnažiti sposobnost razumijevanja problema, konceptualizacije rješenja – uvažavajući ograničenja današnjih računala, formuliranja rješenja u algoritamskom obliku te zapisa, prevođenja i izvođenja razvijenog algoritma u programskom jeziku `C`. Tehnike programiranja kao što su kreiranje i upotreba rekurzivnih funkcija, višedimenzionalnih polja, stringova, upotreba dinamičke alokacije, kreiranje korisničkih tipova i korištenje datoteka će studentima omogućiti stvaranje potpunih programa, konceptualno bliskih programima koji se stvaraju kao alati pri znanstvenom istraživanju ili u raznim industrijskim primjenama. Dodaci vezani uz korištenje `pretprocesora` i biblioteka će produbiti razumijevanje procesa stvaranja izvršnog programa iz izvornog koda te dati jako blagi uvod u postpuke potrebne pri razvoju jednog programskog jezika.

Gradivo je podijeljeno u četiri poglavlja. U prvom poglavlju ponavljamo bitnije koncepte usvojene na kolegiju `Programiranje 1` kao što su osnovna struktura programa u jeziku `C`, izvrednjavanje izraza, korištenje operatora i njihovi prioriteti, konverzije, osnove korištenja pokazivača, osnove korištenja funkcija i jednodimenzionalnih polja.

U drugom poglavlju uvodimo naprednije koncepte proceduralnog programiranja. Korištenje rekurzija za rješavanje teških problema i konstrukciju brzih algoritama za sortiranje. Detaljnije ćemo opisati strukturu programa, specifičnosti korištenja atributa varijabli te utjecaja atributa na dio radne memorije u koji se spremaju vrijednosti varijabli, trajanje varijabli i njihov doseg. Objasnit ćemo kako razdvojiti `C` program na više datoteka.

U trećem poglavlju ćemo naučiti koristiti višedimenzionalna polja, kojima možemo reprezentirati složenije matematičke objekte kao što su matrice, tenzori i slični. Naučit ćemo kako upotrebljavati pokazivače pri radu s višedimenzionalnim poljima, koristiti pokazivače za upravljanje hrpom - dijelom radne memorije računala dodijeljene `C` programu u koji se spremaju dinamički alocirani objekti. Definirat ćemo stringove, posebne nizove

znakova konstruirane s ciljem učinkovite memorijske reprezentacije teksta u radnoj memoriji. Objasnit ćemo kako prosljeđivati argumente `C` programu prilikom pokretanja programa. Završni dio poglavlja će opisati korištenje mehanizma definiranja korisničkih tipova za kreiranje jednostavnih i složenih objekata. Definiranje jednostavnih tipova nam uglavnom olakšava razumijevanje napisanog koda i omogućuje pisanje kraćeg koda. Konstrukcija složenih tipova nam omogućava jednostavniju reprezentaciju i manipulaciju složenim objektima u `C` programu, te stvaranje objekata koji omogućuju brže i efikasnije rješavanje određenih problema.

U četvrtom poglavlju ćemo naučiti kako upravljati trajnom memorijom računala (učitavati, upisivati, pregledavati, mijenjati itd.). Objasnit ćemo mogućnosti, specifičnosti i primjene `pretprocesora` te spomenuti i opisati neke korištenije funkcije iz standardne `C` biblioteke.

Sadržaj

Poglavlje 1

Ponavljanje

Programski jezik `C` je konstruirao **Dennis Ritchie**, zaposlenik Bell Telephone Laboratories između 1972. i 1973. godine. Jezik je opisan u knjizi Brian W. Kernighan i Dennis M. Ritchie, *The C Programming Language* [1].

1.1 Struktura programa u `C`-u

`C` je **proceduralan** programski jezik (pogrami se sastoje od jedne ili više procedura). Procedura ili funkcija je slična matematičkoj funkciji. Ima domenu (ulazne podatke) i kodomenu (povratne vrijednosti). Procedure sadrže naredbe koje koristeći zadane ulazne podatke računaju odgovarajuću povratnu vrijednost. U svakom `C` programu postoji posebna procedura (funkcija) koja se zove `main`. Pri pokretanju izvršne verzije `C` programa, prvo se slijedno (od prve do posljednje) počinju izvršavati naredbe funkcije `main`. Funkcija `main` vraća cjelobrojan rezultat (tip `int`), uobičajeno vrijednost 0 ukoliko su sve naredbe uspješno izvršene, a u jednostavnijim primjerima ne prima nikakve ulazne podatke (`void`). U sljedećim poglavljima ćemo vidjeti da funkcija `main` može primati parametre preko kojih ćemo moći zadati neke ulazne vrijednosti pri pokretanju programa.

```
1 int main(void){
2
3
4    return 0;
5 }
```

Uobičajeno pri pisanju funkcije `main` koristimo funkcije za unos podataka iz komandne linije i ispis rezultata u komandnu liniju. Navedene funkcije su, uz niz drugih funkcija i varijabli, definirane u biblioteci sa zaglavljem `stdio.h` (eng. *standard input output*) [1]. Funkcije i konstante navedene biblioteke uključujemo u program korištenjem `pretprocesorske` naredbe `#include`.

```c
#include <stdio.h>

int main(void){
   int a;

   scanf("%d", &a);
   printf("%d\n",a);

   return 0;
}
```

`Pretprocesor` prije izvođenja programa, kopira definicije iz traženog zaglavlja u naš program, stoga `prevodioc` (eng. *compiler*) koji se pokreće nakon `pretprocesor`-a u datoteci pronalazi sve potrebne deklaracije funkcija koje su nam potrebne za uspješno izvođenje programa. Naredbe funkcije `main` se pišu unutar bloka {}. Kod pokretanja izvršnog programa, koji odgovara kodu iznad, prvo će se izvršiti naredba `int a;` koja deklarira jednu varijablu cjelobrojnog tipa i daje joj ime *a*. Istovremeno se zauzima dio radne memorije (RAM) potreban za spremanje jednog cijelog broja (4 byte-a).

```
4
4

Process returned 0 (0x0)   execution time : 1.921 s
Press any key to continue.
```

Nakon toga se izvodi naredba za učitavanje vrijednosti na adresu koja je pridružena varijabli *a* u naredbi `scanf("%d", &a)`, te se vrijednost iste memorijske lokacije ispisuje u komandni prozor `printf("%d\n",a);` Funkcija `main` kao povratnu vrijednost vraća 0.

[1] Sve funkcije i konstante navedene biblioteke možete vidjeti preko poveznice: `https://en.cppreference.com/w/c/io`

1.2 Operatori, prioriteti i izrazi

Vrijednosti nekog tipa (konstante, varijable, vrijednosti funkcija, podizrazi) čine **operande**, dok su **operatori** posebne funkcije koje djeluju na operande (različitih tipova) i vraćaju vrijednost nekog tipa kao rezultat. Operatori imaju **prioritete** i **asocijativnost**. Prioritet operatora definira koji operator ima prvenstvo izvršavanja ukoliko više operatora djeluje na operande. Asocijativnost definira kojim redoslijedom (u kojem smjeru) će se operatori izvršavati ukoliko postoji više uzastopnih pojavljivanja operatora istog prioriteta. **Izrazi** su posebne jezične konstrukcije programskog jezika koje se evaluiraju da bi im se utvrdila vrijednost. Javljaju se s desne strane naredbe pridruživanja, kao argumenti funkcija, uvjeti u uvjetnim naredbama i petljama, kao granice u petljama. Zagradama () možemo mijenjati prioritete izvođenja tako da se uvijek prvo izračunava podizraz u zagradama.

```
1 #include <stdio.h>
2
3 int main(void){
4    int a = 5, b = 10, c, d;
5
6    d = c = 4*a+b;
7
8    return 0;
9 }
```

Kod izvršavanja naredbe: `d = c = 4*a+b`, između operatora $=$, $+$ i $*$, najveći prioritet izvršavanja ima operator $*$, stoga će se prvo izračunati $4 * a$. Sljedeći po prioritetu je operator $+$, stoga će se izračunati $4 * a + b$. Slijedi operator $=$. Pošto operator $=$ ima asocijativnost s desna na lijevo, prvo će se izvršiti najdesniji operator $=$, dakle $c = 4 * a + b$, a nakon toga i $d = c = 4 * a + b$.

Prioritete svih bitnijih operatora možete vidjeti u Tablici 1.1. Neki operatori kao $+$ i $-$ mogu biti i unarni (primaju samo jedan argument) i binarni (primaju dva argumenta).

Oprez: kod binarnih operatora, osim logičko I (`&&`) i logičko ILI (`||`) nema pravila koje definira kojim redoslijedom se računaju (izvrednjavaju) operandi.

Analizirajmo nekoliko primjera i promotrimo ispise odsječaka programa.

Table 1.1: Prioriteti i asocijativnosti operatora u C-u

Kategorija	Operatori	Asocijativnost
primarni	() [] -> .	$L \to D$
unarni	! ~ ++ -- + - * & (type) sizeof	$D \to L$
aritm. mult.	* / %	$L \to D$
aritm. adit.	+ -	$L \to D$
op. pomaka	« »	$L \to D$
relacijski	< <= > >=	$L \to D$
rel. jednakost	== !=	$L \to D$
bit-po-bit I	&	$L \to D$
bit-po-bit eks. ILI	^	$L \to D$
bit-po-bit ILI	|	$L \to D$
logičko I	&&	$L \to D$
logičko ILI	||	$L \to D$
uvjetni	? :	$D \to L$
pridruživanje	= += -= *= /= %= &= ^= |= «= »=	$D \to L$
operator zarez	,	$L \to D$

Primjer 1.2.1

```
1 int a = 40, b = 40, c = 4;
2
3 a/=c*5;
4 b=b/c*5;
5 printf("a␣-␣b␣=␣%d\n", a-b); //a - b = -48
```

Izraz `a/=c*5` se prema definiciji operatora / = računa kao `a = a/(c*5)`, što daje $a = 40/20$, odnosno $a = 2$. Izraz `b = b/c*5` se računa prema prioritetima operatora, dakle $b = 40/4 * 5 = 10 * 5 = 50$, stoga $a - b = 2 - 50 = -48$.

Primjer 1.2.2

```
1 int b, a = b = 50, c = 5;
2
3 a = c *b;
4 printf("a␣=␣%d,␣b␣=␣%d\n", a, b); //a = 250, b = 50
```

U gornjem isječku se prvo deklarira varijabla b (bez inicijalizacije), zatim se izvrši pridruživanje $b = 50$, a zatim deklaracija i inicijalizacija $a = b$. Konačno, izvrši se deklaracija i inicijalizacija $c = 5$.

Primjer 1.2.3

```
1 int i, j, k;
2 k = 0; i = 60; j = 2;
3
4 if(i-i && j++) k = 100000;
5 printf("j␣=␣%d,␣k␣=␣%d\n",j, k); //j = 2, k = 0
```

`i-i` uvijek iznosi 0, za cjelobrojnu varijablu i. Po pravilima računanja operatora `&&` **uvijek** se prvo izvrijedni lijevi operand, a zatim desni operand. Lijevi operand operatora `&&` unutar naredbe `if` ima vrijednost 0 (logička laž u `C`-u). Zbog lijenog izvrednjavanja logičkog operatora `&&` (poznato je da je vrijednost cijelog izraza logička laž već nakon izvrednjavanja lijevog operatora), desni operand se neće izvrijediti. Vrijednosti od j i k se ne mijenjaju.

Primjer 1.2.4

```
1 int i,j,k;
2 k = 0; i = 15; j = 0;
3
4 if(i+i && j++) k = 1;
5 printf("k␣=␣%d\n", k); //j = 1, k = 0
```

$i+i$ je > 0 za $i = 15 > 0$, stoga lijevi operand operatora `&&` ima logičku vrijednost istina. Nakon izvrednjavanja desnog operanda dobijemo 0 (logička laž), pošto `j++` inkrementira vrijednost od j za jedan, ali vraća **staru vrijednost** (0) kao povratnu. Ne dolazi do promjene vrijednosti varijable k.

Primjer 1.2.5

```
1 int z = 5; double y = 5.8;
2
3 printf("%d\n", (int) y/2);//2
4 printf("%d\n", (int) (double) z/2);//2
5 printf("%f\n", (float)'1');//49.000000
```

U liniji 3 gornjeg primjera, zbog prioriteta izvođenja operatora, prvo se izvede cast operator `(int)`, stoga se cjelobrojno računa $5/2 = 2$. U liniji 4, se prvo provede eksplicitna konverzija varijable z u tip `double` pa zatim eksplicitna pretvorba iste varijable u tip `int`. Stoga je konačni rezultat $5/2 = 2$. U trećem primjeru pretvorimo znak u float i dobijemo realnu reprezentaciju njegovog ASCII koda.

Primjer 1.2.6

```
1 double y = 1.8e+20;
2
3 printf("%d\n", (int) y);//-2147483648
4 printf("%d\n", (int) y/2);//-1073741824
```

U liniji 3 gornjeg primjera se događa eksplicitna konverzija varijable y iz tipa `double` u tip `int`. Dolazi do gubitaka podataka (pretvorba iz šireg u uži tip), te zbog raspona prikazivosti u tipu `int` do prikaza broja $(int)\ y\ mod\ 2^n$ u 4 byte-a memorije. U liniji 4 se dobiveni broj cjelobrojno dijeli s 2.

Primjer 1.2.7

```
1 double x = 5.1;
2
3 printf("%d\n", (int) (1000*x));//5100
```

```
1 double x = 64.1;
2
3 printf("%d\n", (int) (1000*x));//64099
```

U gornjim primjerima se izvršava množenje u aritmetici pomične točke, koja po dizajnu (zbog ograničenog prikaza uzrokovanog konačnom memorijom računala) može dovesti do pogrešaka pri računanju. Do greške ne dolazi u prvom primjeru, međutim moguće je da će za isti primjer na nekom drugom prevodiocu doći do pogreške pri računanju. Greška je vidljiva u drugom primjeru (dolazi do dvije greške pri zaokruživanju), stoga nakon pretvaranja broja u cijeli broj s predznakom dobijemo netočan rezultat za 1 (64099 umjesto 64100). Prva greška pri zaokruživanju nastaje zbog prikaza broja 64.1 u tipu `double`.

```
1. rijec: 0110 0110 0110 0110 0110 0110 0110 0110
2. rijec: 0100 0000 0101 0000 0000 0110 0110 0110
```

Vidimo periodičnost zapisa (zadnji dio 2. riječi i 1. riječ). Druga greška nastaje pri računanju $1000 * x$. Pošto prikaz broja x nije egzaktan, $1000 * x$ će imati prikaz:

```
1. rijec: 1111 1111 1111 1111 1111 1111 1111 1111
2. rijec: 0100 0000 1110 1111 0100 1100 0111 1111
```

To je reprezentacija broja 64099.999999999993. Pretvaranjem navedenog broja u tip `int` dobijemo konačni rezultat 64099. Moramo biti svjesni ograničenja računala i specifičnosti reprezentacije realnih brojeva za uspješno i točno računanje kompleksnih numeričkih izraza.

Primjer 1.2.8

```
1 int c = 1, i;
2
3 for( i = 0; i < 20; ++i){
4    double x = c * 0.1;
5    printf("%d\n", (int)(1000*x));
6    c*=2;}
```

U gornjem programu računamo vrijednost izraza $\lfloor 1000 * 2^i * 0.1 \rfloor$. To je matematički ekvivalentno izrazu $100 * 2^i$. Računanjem možemo uočiti da dobivamo točne rezultate:

```
100
```

```
200
400
800
1600
3200
6400
12800
25600
51200
102400
204800
409600
819200
1638400
3276800
6553600
13107200
26214400
52428800
```

Uočimo da prikaz broja $2^i * 0.1$ u tipu `double` daje reprezentaciju koja je neznatno veća od stvarne vrijednosti $2^i * 0.1$ (mantisa je identična, mijenja se eksponent), stoga su svi rezultati korektno prikazani nakon konverzije u tip `int`.

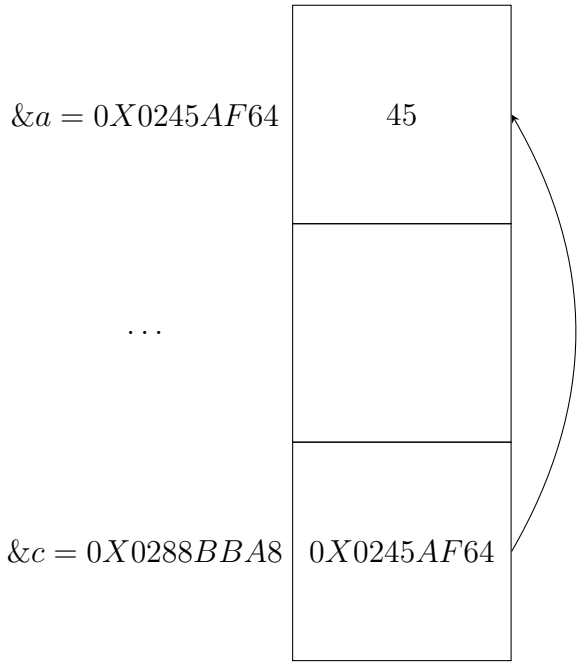
Zadatak: ispitajte hoće li se korektno izračunati brojevi u programskom isječku:

```
1 int c = 1, i;
2
3 for( i = 0; i < 20; ++i){
4    double x = c + 0.1;
5    printf("%d\n", (int)(1000*x));
6    c*=2;}
```

Pripomoć: neće, otkrijte zašto.

1.3 Pokazivači, funkcije i polja

1.3.1 Pokazivači



Prisjetimo se, pokazivači (eng. pointers) su posebni tipovi koji kao vrijednosti sadrže adrese neke memorijske lokacije. U gornjem prikazu, varijabla a ima adresu $\&a = 0X0245AF64$ (prisjetimo se da se adrese najčešće zapisuju u heksadekadskoj bazi zbog kraćeg prikaza) i cjelobrojnu vrijednost 45. Pokazivač c ima adresu $\&c = 0X0288BBA8$ i vrijednost $0X0245AF64$ (uočimo da je vrijednost pokazivača c jednaka adresi varijable a).

Primjer 1.3.1

```
1 int a = 1;
2 int *b;
3
4 b = &a;
5 *b = 8;
6 printf("%d␣%d\n", a, *b);//8 8
```

U gornjem primjeru deklariramo i inicijaliziramo varijablu a cjelobrojnog tipa, te pokazivač na `int` b. Pokazivač deklariramo pisanjem unarnog opera-

tora $*$ prije imena varijable u deklaraciji. Operator $*$ se primjenjuje samo na sljedeću varijablu desno od njega. Npr. `int* a, b;` će deklarirati pokazivač na `int` a i varijablu b tipa `int`. U liniji 4, kao vrijednost pokazivača b postavljamo adresu varijable a. U liniji 5 preko pokazivača b promijenimo vrijednost varijable a u 8. Primjenom unarnog operatora $*$ nad pokazivačem, dohvaćamo vrijednost memorijske lokacije na koju taj pokazivač pokazuje. Konačno, isipsujemo vrijednost varijable a na dva način, koristeći varijablu a i dohvaćanjem vrijednosti preko pokazivača b.

1.3.2 Funkcije i pokazivači

Funkcije su programske cjeline koje: a) uzimaju neke ulazne podatke (jako rijetko se događa da funkcija ne prima ulazne podatke - primjer funkcije `main`), b) izvršavaju određeni niz naredbi, c) vraćaju rezultat svog izvršavanja na mjesto poziva (moguće je da funkcija i ne vraća povratnu vrijednost). Slične su matematičkoj funkciji, imaju domenu, kodomenu i pravilo.

Primjer 1.3.2

Pretpostavimo da želimo izračunati najveću zajedničku mjeru dva nenegativna cijela broja. Kao što znamo, problem rješavamo proizvoljnom varijantom Euklidovog algoritma[2]. Funkcije su u `C`-programu vidljive od linije koda u kojoj završava njezina definicije do kraja izvorne datoteke. Međutim, dosta često želimo funkcije deklarirati na kraju izvorne datoteke (npr. zbog preglednosti). U tom slučaju funkciju moramo **deklarirati** prije linije u izvornom kodu u kojoj planiramo napraviti poziv te funkcije. Deklaracija funkcije `nzm` se može vidjeti u liniji 5 u doljnjem primjeru. U navedenoj liniji prvo deklariramo varijablu m tipa `unsigned int`, a zatim funkciju `nzm` koja vraća povratnu vrijednost tipa `unsigned int`, a kao formalne argumente prihvaća dvije varijable tipa `unsigned int`. Pri deklaraciji funkcije imena ulaznih argumenata ne moramo pisati, ali moramo navesti njihove tipove. Nakon poziva funkcije `nzm` u liniji 7 nad argumentima 12 i 48, te spremanja najveće zajedničke mjere u varijablu m, u liniji 8 ispisujemo vrijednost varijable m (najveće zajedničke mjere).

[2] `https://enciklopedija.hr/clanak/euklidov-algoritam`

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5   unsigned int m, nzm(unsigned int, unsigned int);
6
7    m = nzm(12, 48);
8    printf("mjera␣=␣%u\n", m);//mjera = 12
9
10   return 0;
11 }
12
13 unsigned int nzm(unsigned int a, unsigned int b)
14 {
15
16    while (a != b)
17          if(a > b)
18             a -= b;
19          else
20             b -= a;
21   return a;
22 }
```

Funkcija `nzm` je definirana nakon funkcije `main`. Korištena je verzija Euklidovog algoritma koja od aritmetičkih operacija koristi samo oduzimanje. Modificirajte kod tako da umjesto oduzimanja koristite operaciju računanja ostatka pri cjelobrojnom dijeljenju. Funkcija vraća najveću zajedničku mjeru u liniji 21 (pozivom naredbe `return`).

Argumenti se funkcijama u `C`-u prosljeđuju po vrijednosti. Pri pozivu funkcije `nzm` u gornjem primjeru, konstante 12 i 48 se kopiraju na memorijske lokacije pridijeljene lokalnim varijablama a i b. Varijable a i b se zovu lokalne jer su one vidljive samo unutar tijela funkcije `nzm`, a njihove vrijednosti se čuvaju samo do završetka izvođenja te funkcije.

Ukoliko želimo omogućiti mijenjanje vrijednosti varijabli unutar funkcije, moramo koristiti **varijabilne argumente** . Umjesto da prosljeđujemo varijable po vrijednosti, funkciji proslijedimo adrese memorijskih lokacija pridjeljenih potrebnim varijablama. Dakle, funkciji prosljeđujemo pokazivače koji

sadrže adrese potrebnih varijabli po vrijednosti (dolazi do kopiranja adresa u memorijske lokacije pridružene pokazivačima, argumentima funkcije). Osim što omogućuje mijenjanje vrijednosti varijabli unutar funkcije, korištenje varijabilnih argumenata omogućava da se umjesto kopiranja cijelog ulaznog elementa (koji ponekad može biti velik), kopira samo adresa koja je ograničene veličine (npr. 8 byte-ova na `GNU GCC` kod 64-bitnog `Windowsa 11`).

Primjer 1.3.3

```
1  #include <stdio.h>
2
3  void kvadrat(int *x, int *y)
4  {
5     *y = *x**x; /* = (*x) * (*x). */
6     printf("Unutar funkcije: x = %d, y = %d.\n",
7             *x, *y);
8   return; }
9
10 int main(void){
11     int x = 3, y = 5;
12
13     printf("Prije: x = %d, y = %d.\n", x, y);
14     kvadrat(&x, &y);
15     printf("Nakon: x = %d, y = %d.\n", x,y);
16     return 0; }
```

U gornjem primjeru funkcija `kvadrat` prima dva varijabilna argumenta x i y. Funkcija u memorijsku lokaciju na koju pokazuje pokazivač y sprema vrijednost x^2 (linija 5) i ispisuje vrijednosti memorijskih lokacija na koje pokazuju pokazivači x i y (linija 6). U glavnom programu deklariramo dvije varijable x i y (pošto su varijable lokalne i vrijede samo unutar funkcije, možemo imati ista imena varijabli u funkciji `main` kao imena ulaznih argumenata, npr. funkcije `kvadrat`). Poziv funkcije (linija 14) određuje što se zapravo prosljeđuje funkciji `kvadrat`, ovdje adrese varijabli x i y iz funkcije `main`. Funkcija `kvadrat` mijenja vrijednost varijable y iz funkcije `main`. Promjena vrijednosti varijable y je vidljiva u ispisu definiranom u liniji 15.
Potpuni ispis programa je:

```
Prije: x = 3, y = 5.
Unutar funkcije: x = 3, y = 9.
```

```
Nakon: x = 3, y = 9.
```

Jako važno svojstvo funkcija u C-u je što mogu pozivati same sebe (u tijelu funkcije, imamo poziv te iste funkcije s nekim parametrima). Takve funkcije se zovu **rekurzivne funkcije** , a poziv funkcije u svom tijelu se zove **rekurzivni poziv**. Rekurzivne funkcije se **uvijek** sastoje od dva glavna dijela: a) rekurzivnog poziva, b) ne rekurzivnog uvjeta prekida rekurzije. Rekurzivne funkcije se mogu koristiti za brže rješavanje nekih poznatih problema (npr. sortiranje), ali i za rješavanje mnogih teških problema za koje nije jednostavno konstruirati učinkovito ne rekurzivno rješenje.

Primjer 1.3.4

Rekurzivna funkcija `f` ima ne-rekurzivni uvjet prekida rekurzije u linijama 4 i 5. Uvjet vraća vrijednost 2 kao povratnu vrijednost funkcije, ukoliko je vrijednost parametra n jednaka 0 (u tom slučaju se više ne poziva funkcija `f`). Za $n > 0$ imamo rekurzivni poziv funkcije f u liniji 6.

```
1  #include <stdio.h>
2
3  int f(int n){
4     if( n == 0)
5           return 2;
6      else return f(--n);}
7
8  int main(void){
9     printf("%d\n", f(4));//2
10    return 0;
11 }
```

Funkciju `f` u glavnom programu pozivamo s konstantom 4.

```
f(4)                        ->2
  f(3)                   ->2
    f(2)              ->2
      f(1)         ->2
         f(0) -> 2
```

Što bi se dogodilo kada bi unutar funkcije imali poziv `f(n--)`?

```
f(4)
 f(4)
   f(4)
        .
          .
            .
             f(4)
                .
                  .
                    .
```

Dobijemo **beskonačnu** rekurziju zato što $n--$ smanji vrijednost argumenta za 1, ali vrati staru vrijednost, stoga se u rekuzrivnom pozivu stalno događa poziv `f(4)`.

Pošto računalo može raditi samo s konačnim elementima (ograničeno je količinom memorije), ono ne može izvoditi niti beskonačne rekurzije. Kod svake beskonačne rekurzije, nakon određenog broja poziva, doći će do rušenja programa. Kod svakog poziva funkcije, računalo napravi poseban okvir u koji sprema određene informacije o funkciji, koje će omogućiti nastavak izvođenja funkcije nakon što se izračuna neka druga funkcija pozvana unutar njenog tijela. Te informacije uključuju adresu instrukcije koja se mora izvoditi nakon prestanka izvođenja funkcije pozvane u tijelu, argumenti funkcije, lokalne varijable, potencijalno registri i poseban pokazivač koji služi pretraživanju informacija unutar okvira. Takvi okviri se spremaju na **sistemski stog**, posebnu strukturu određene veličine, ovisne o operacijskom sustavu i arhitekturi računala. Stoga, kod beskonačnih rekurzija, u nekom trenutku će doći do prepunjenja stoga i program će se srušiti s porukom `stack overflow`[3]. Zbog spremanja informacija o pozivu funkcija na sistemski stog, te zbog potencijalnog višestrukog računanja dijelova koji dovode do rješenja u rekurzivnim pozivima, rekurzivne funkcije su **često sporije** od iterativnih rješenja (ukoliko postoje).

Primjer 1.3.5

Fibonaccijevi brojevi su definirani rekurzijom: $F_n = F_{n-1} + F_{n-2},\ n \geq 2$, uz $F_0 = 0,\ F_1 = 1$. Rekurzivno pravilo za računanje n-tog Fibonaccijevog

[3]Poznati forum istog imena `https://stackoverflow.com/` sadrži brojna pitanja i odgovore na temu programiranja u raznim programskim jezicima.

broja jednostavno zapišemo u obliku rekurzivne funkcije u C-u.

```
1 long int fib(int n)
2 {
3   if( n == 0) return 0;
4   if( n == 1) return 1;
5
6   return fib(n-1) + fib(n-2);
7 }
```

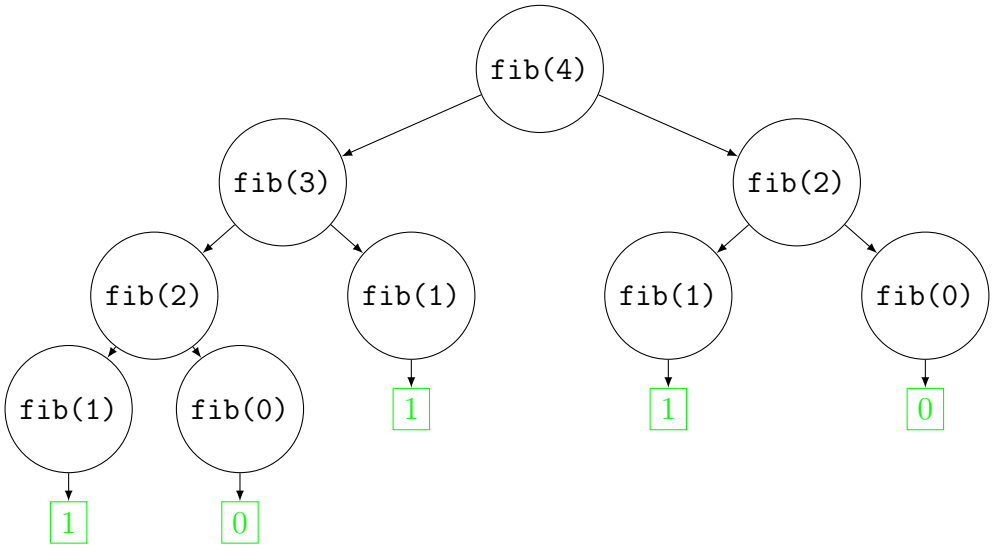
Broj poziva funkcije `fib` možemo izračunati dodvanjem globalne varijable `broj_poziva` (treba je deklarirati izvan svih funkcija u izvornom kodu). Pri svakom pozivu funkcije `fib`, inkrementiramo `broj_poziva` za jedan.

Broj poziva funkcije fib

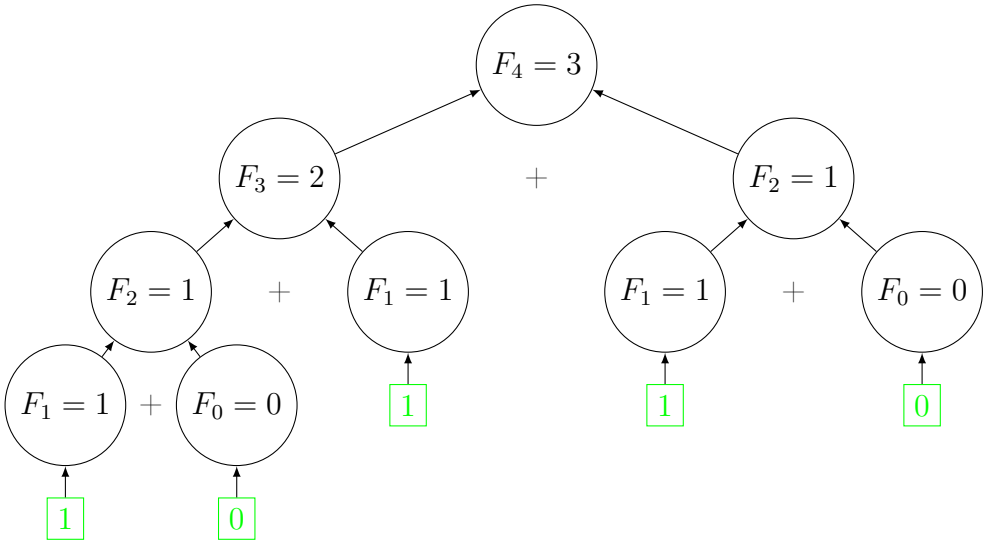
```
1 long int fib(int n)
2 {
3   ++broj_poziva; /*Globalni brojac*/
4   if( n == 0) return 0;
5   if( n == 1) return 1;
6
7   return fib(n-1) + fib(n-2);
8 }
```

Pri računanju n-tog Fibonaccijevog broja, rekurzivna funkcija više puta računa vrijednosti nekih Fibonaccijevih brojeva koji se u listi nalaze prije n-tog.

Npr. pri rekurzivnom računanju vrijednosti četvrtog Fibonaccijevog broja, `fib(4)`, dobijemo sljedeće stablo rekurzivnih poziva (vrijednosti djece su potrebne za računanje vrijednosti roditelja):



Kao što vidimo sa slike, poziv `fib(2)` se računa 2 puta, `fib(1)` 3 puta, a `fib(0)` 2 puta. Povratna vrijednost funkcije `fib(n)` za svaki $n > 1$ se dobije kao zbroj rezultata poziva `fib(n-1)` i `fib(n-2)`.



n-ti Fibonaccijev broj je puno bolje izračunati iterativno. Time uz izbjegavanje spremanja niza vrijednosti u sistemski stog zbog rekurzivnih poziva, izbjegavamo višestruko računanje vrijednosti F_k za neke $k < n$. Pri iterativnom računanju koristimo 3 varijable (pomični prozor). Varijable `f_pp`, `f_p` i `f_n` redom označavaju F_{k-2}, F_{k-1} i F_k. Za računanje F_{k+1} nam trebaju samo vrijednosti F_{k-1} i F_k. Redom pamtimo: `f_pp`= `f_p` $= F_{k-1}$, `f_p`= `f_n` $= F_k$

i računamo `f_n`= `f_pp` + `f_p` = F_{k+1}. Uz inicijalizaciju `f_p`= 0, `f_n`= 1, slijedeći opisani postupak, možemo izračunati F_n za proizvoljan $n > 1$.

```
1  long int fibonacci(int n){
2    long int f_n, f_p, f_pp;
3    int i;
4
5    if( n == 0) return 0;
6    if (n == 1) return 1;
7
8    f_p = 0; /*Prosli F[0]*/
9    f_n = 1; /*Ovaj F[1]*/
10
11   for (i = 2; i<=n; ++i){
12      f_pp = f_p; /*F[i-2]*/
13      f_p = f_n /*F[i-1]*/;
14      f_n = f_p + f_pp /*F[i]*/; }
15
16   return f_n;    }
```

Primjer 1.3.6

Dokažite da je broj rekurzivnih poziva funkcije `fib` za računanje F_n, uz $n \geq 2$ jednak $(F_1 + F_2 + \cdots + F_n) + F_{n-1}$.

Razmislimo koliko puta se događa poziv $fib(k)$ za $k = 1, \ldots, n$, a koliko puta $fib(0)$. Označimo s $g(k)$ broj poziva funkcije `fib(k)` pri računanju `fib(n)` za neki $n \in \mathbb{N}$, $n > 1$. Tada vrijedi, $g(n) = 1 = F_1$, `fib(n)` se uvijek pozove samo jednom za proizvoljni n. $g(n-1) = g(n) = 1 = F_2$, svaki poziv `fib(n)` pozove točno jednom u tijelu `fib(n-1)`. $g(n-2) = g(n) + g(n-1) = F_1 + F_2 = F_3$, `fib(n-2)` se pozove točno jednom pri svakom pozivu `fib(n)` i `fib(n-1)`. $g(n-3) = g(n-2) + g(n-1) = F_2 + F_3 = F_4$. Stoga, za svaki $k > 0$, vrijedi da je $g(k) = F_{n-k+1}$. Još preostaje izračunati koliki je $g(0)$. Primjetimo da se $g(0)$ računa kod svakog poziva `fib(2)`, stoga je $g(0) = g(2) = F_{n-1}$. Sada imamo da je broj rekurzivnih poziva funkcije `fib` jednak $\sum_{k=0}^{n} g(k) = \sum_{i=1}^{n} F_i + F_{n-1}$.

Primjer 1.3.7

Dokažite da je broj rekurzivnih poziva funkcije `fib` za računanje F_n, uz $n \geq 2$

zapravo jednak $2F_{n+1} - 1$.

Uputa: pokažite, $\sum_{i=0}^{n} F_i = F_{n+2} - 1$.

1.3.3 Polja, funkcije i pokazivači

Polja su nizovi varijabli nekoga tipa. U memoriji se radi o uzastopnim memorijskim lokacijama istog (zadanog) tipa. Varijabla kojom deklariramo polje je **konstantan pokazivač** koji sadrži adresu prve memorijske lokacije polja. Konstantan pokazivač je pokazivač čiju vrijednost ne možemo mijenjati (on uvijek pokazuje na istu adresu).

Primjer 1.3.8

U donjem kodu deklariramo polje cijelih brojeva s predznakom, te inicijaliziramo deklarirano polje **inicijalizacijskom listom**. Inicijalizacijska lista započinje i završava vitičastim zagradama, a sadrži niz vrijednosti koje se redom zapisuju u memorijske lokacije deklariranog polja. Pri inicijalizaciji polja inicijalizacijskom listom ne trebamo navoditi maksimalnu duljinu polja, pošto prevodioc može zaključiti potrebnu duljinu iz same inicijalizacijske liste. Duljinu polja, u tom slučaju uglavnom navodimo ako želimo imati polje koje može sadržavati više elemenata od broja elemenata navedenih u inicijalizacijskoj listi. Preostali elementi polja se tada inicijaliziraju na 0.

```
1  #include <stdio.h>
2
3  int main(void) {
4    int polje[] = {-6, 2, 1, 7, 3, 8, 0, 5, 9, 4};
5    int indeks;
6    indeks = 0;
7    while (polje[indeks] != 0)
8        ++indeks;
9    printf("Broj␣el.␣prije␣nule:␣%d\n", indeks);
10   //Broj el. prije nule: 6
11   return 0;}
```

Program računa broj elemenata u polju prije prvog pojavljivanja elementa s vrijednosti 0, te ispisuje navedeni broj u komandni prozor. Primijetimo da koristimo varijablu `indeks` i za iteriranje po polju i kao brojač.

Primjer 1.3.9

U ovom primjeru pokazujemo ekvivalenciju aritmetike pokazivača i indeksiranja pri radu s poljem. U liniji 1 deklariramo polje `a` u koje možemo spremiti 10 elemenata tipa `int` i pokazivač na `int` `pa`. U liniji 2 spremimo adresu prvog elemenata polja u `pa`. Koristeći konstantni pokazivač `a` i aritmetiku pokazivača, u liniji 3 dohvaćamo vrijednost drugog element polja koristeći izraz `*(a+1)`. Prisjetimo se, interno prevodioc računa adresu elementa kao $\&a + 1 \cdot sizeof(int)$. U liniji 4 u pokazivač `pa` spremamo adresu trećeg elementa polja. Pozivom `pa++` u liniji 5 u `pa` spremamo adresu četvrtog elementa polja. Konačno u liniji 6 mijenjamo vrijednost šestog elementa polja u 20.

```
1 int a[10], *pa;
2 pa = a;
3 *(a+1) = 10; /*a[1] = 10;*/
4 pa = pa+2; /*&a[2]*/
5 pa++; /*&a[3]*/
6 *(pa + 3) = 20; /* a[6] = 20;*/
```

Primjer 1.3.10

Problem brojanja elemenata prije prvog pojavljivanja 0 u polju cijelih brojeva možemo riješiti i koristeći pokazivače.

```
1  #include <stdio.h>
2
3  int main(void) {
4    int polje[] = {-6, 2, 1, 7, 3, 8, 0, 5, 9, 4};
5    int *pokazivac;
6    pokazivac = polje;
7    while ((*pokazivac) != 0)
8        ++pokazivac;
9    printf("Broj␣el.␣prije␣nule:␣%d\n",  pokazivac - polje);
10  //Broj el. prije nule: 6
11   return 0;}
```

Koristimo dopušteni izraz aritmetike pokazivača: `pokazivač - polje` da izračunamo broj pozicija od početka polja do pozicije na kojoj se nalazi vrijednost 0 (računa se razlika adresa). **Oprez:** primjena binarnog operatora

oduzimanja na dva argumenta koji su pokazivači uglavnom ima smisla raditi kada pokazivači pokazuju na memorijske lokacije koje pripadaju **istom** polju.

Primjer 1.3.11

U ovom primjeru demonstriramo statičku alokaciju memorije za polje fiksne duljine koristeći preprocesorsku konstantu `MAX`. Korištenje konstante za definiranje duljine polja je jedini način deklaracije statičkog polja u verziji `C`-a pod nazivom `C90`. Kasnije verzije `C99` i novije, dozvoljavaju deklaraciju tzv. **polja varijabilne duljine** , umjesto konstante možemo koristiti proizvoljnu varijablu, međutim mi iz pedagoških razloga takva polja nećemo koristiti. Umjesto korištenja polja varijabilne duljine, mi ćemo detaljnije proučiti mehanizme kojima možemo zatražiti memoriju određene veličine za korištenje od strane operacijskog sustava, te postupke dojavljivanja da neke dijelove memorije više ne želimo koristiti već ih operacijski sustav može dodijeliti drugim procesima. Poznavanje navedenih mehanizama daje puno bolji uvid u stvarni rad računala.

```
1  #include <stdio.h>
2  #define MAX 10
3
4  int main(void) {
5    int polje[MAX];
6    int i, *pokazivac;
7    pokazivac = polje;
8    for (i=0; i < MAX; ++i)
9          polje[i] = i;
10   printf("%d\n", *pokazivac);//0
11
12   return 0;}
```

Program redom sprema vrijednost i na i-ti indeks polja ($i+1$ poziciju), te ispisuje vrijednost prvog elementa koristeći `pokazivac`.

Primjer 1.3.12

Želimo konstruirati funkcije `unos` i `ispis` te glavni program koji upisuje i ispisuje polje s maksimalno 50 elemenata. Primijetimo da obična polja elemenata ne možemo učitavati i ispisivati koristeći jedan poziv naredbi tipa

`scanf` i `printf`, već moramo učitati (ispisati) svaki element polja posebno.

```
1  #include <stdio.h>
2  #define MAX 50
3
4  void unos(int a[], int n){
5     int i;
6     for(i = 0; i < n; ++i)
7         scanf("%d", &a[i]);}
8
9  void ispis(int *a, int n){
10     int i;
11     for (i = 0; i < n; ++i)
12         printf("%d\n", *a++); }
13
14 int main(void) {
15    int n, polje[MAX];
16        /*Koliko ce se rezervirati bajtova?*/
17    scanf("%d", &n);
18
19    unos(polje, n);
20    ispis(polje, n);
21
22    return 0;
23 }
```

Jednodimenzionalno polje (kao u primjerima koje promatramo) može proslijediti funkciji kao u liniji 4 (navodeći ili ne navodeći maksimalnu duljinu polja) ili kao u liniji 9 koristeći pokazivač na prvi element polja. Maksimalna duljina polja nije bitna za indeksiranje unutar funkcije (već samo adresa prvog elementa polja). Također, granice polja se u `C`-u nikada ne provjeravaju već se za korektnost pristupa memoriji mora pobrinuti programer.

Nakon što učitamo vrijednost cijelog broja n (u liniji 17), unosimo (linija 19) i ispisujemo (linija 20) vrijednosti prvih n elemenata polja `polje`. **Oprez:** u programu se pretpostavlja da je broj n korektno unesen: $0 \leq n \leq 50$.

Poglavlje 2

Napredniji koncepti proceduralnog programiranja

U ovom poglavlju ćemo se detaljnije upoznati s rekurzivnim funkcijama i njihovim primjenama, te ćemo proučiti strukturu programa u programskom jeziku `C` (trajanje, doseg i tip memorije u koje se spremaju varijable, razdvajanje programa na više datoteka).

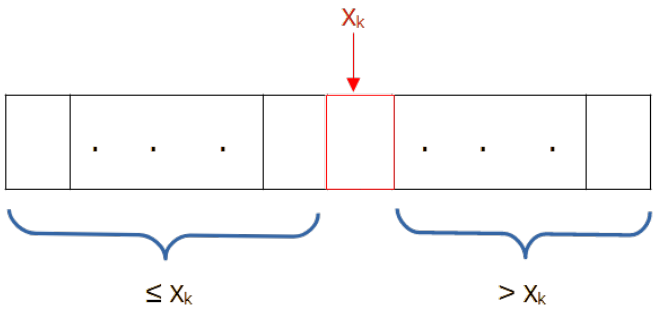
2.1 Rekurzivne funkcije

Rekurzivne funkcije su funkcije koje pozivaju same sebe unutar svoga tijela. Sastoje se od dva bitna dijela, rekurzivnog poziva i ne rekurzivnog uvjeta prekida izvođenja rekurzivnog poziva (terminalni uvjet). Često se koriste za rješavanje teških kombinatornih problema, definiranje i rad s prirodno rekurzivnim strukturama (npr. stabla) te za učinkovito rješavanje određenih problema (npr. sortiranje). Rekurzije se često mogu učinkovito primijeniti za rješavanje problema ukoliko problem možemo podijeliti na pod probleme istih karakteristika, na način da pod problemi ne ovise jedni od drugome. Postupak podjele većeg problema na manje pod probleme istih svojstava se zove *podijeli pa vladaj*.

2.1.1 QuickSort algoritam

Kao glavni primjer korištenja rekurzija za izradu učinkovitog algoritma sortiranja, navodimo **QuickSort** algoritam. Konstruirao ga je C. A. R. Hoare[1], 1962. godine. QuickSort algoritam se temelji na ranije objašnjenom principu *podijeli pa vladaj*. Ugrubo, algoritam se sastoji od sljedećih koraka:

- Uzmemo element x_k niza (nazivamo ga **ključni** element) i dovedemo ga na njegovo **pravo** mjesto u nizu.
- **Lijevo** od x_k stavimo elemente koji su **manji** ili **jednaki** x_k (u proizvoljnom poretku).
- **Desno** od x_k stavimo elemente koji su **veci** od x_k (u proizvoljnom poretku).
- **Rekurzivno** ponovimo postupak na lijevi dio niza (lijevo od x_k) i desni dio niza (desno od x_k)



Primijetimo da je sortiranje lijevog i desnog dijela niza problem istog tipa kao i sortiranje cijelog niza, ali manji (svaki pod niz sadrži strogo manje elemenata od cijelog niza). Također, sortiranje lijevog pod niza niti na koji način ne utječe na ili ovisi o sortiranju desnog pod niza. Izbor elementa x_k je **jako bitan** dio algoritma i uobičajeno se naziva *pivotiranje*. Postoji više načina na koji se to može napraviti, pravilan izbor može znatno ubrzati izvođenje algoritma.

Ovisno o izboru x_k:

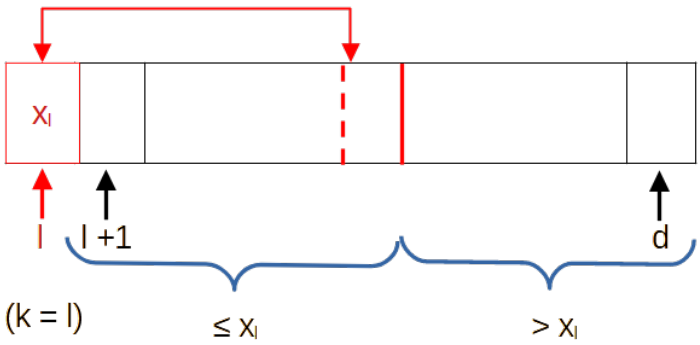
[1] `https://www.cs.ox.ac.uk/people/tony.hoare/`

- Ukoliko se x_k nalazi blizu sredine sortiranog niza, odnosno **prava pozicija** mu je blizu polovice niza, moramo rekurzivno sortirati dva podniza **polovične duljine**.
- U **najgorem** slučaju, x_k se nalazi na rubu sortiranog polja, trebamo sortirati jedan niz duljine $n - 1$.

Nesortirani dio niza ćemo omeđiti s dva indeksa: lijevim (l) i desnim (d). Ta dva indeksa i polje će biti argumenti funkcije koja će implementirati algoritam. Sortiranje se izvršava ako odabrani dio niza ima barem 2 elementa ($l < d$). To je ne rekurzivni uvjet u algoritmu. Kao **ključni element** se najčešće uzima $k = l$, tj. x_l želimo dovesti na poziciju na kojoj treba biti u sortiranom nizu. Kod takvog izabira nam x_l služi kao granica na lijevom rubu niza. Niz ćemo sortirati **uzlazno** stoga ćemo:

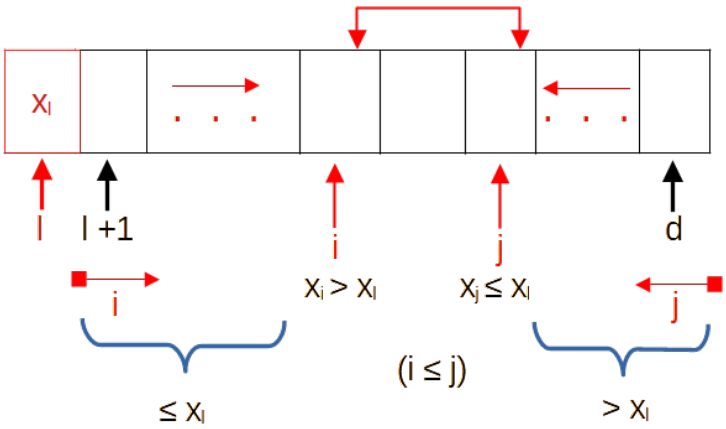
- **lijevo**, ispred prave pozicije od x_l stavljati elemente koji su manji ili jednaki x_l.
- **desno**, iza prave pozicije od x_l stavljati elemente koji su strogo veći od x_l.

Tada će prava pozicija elementa x_l u sortiranom nizu biti zadnja u lijevom dijelu niza.



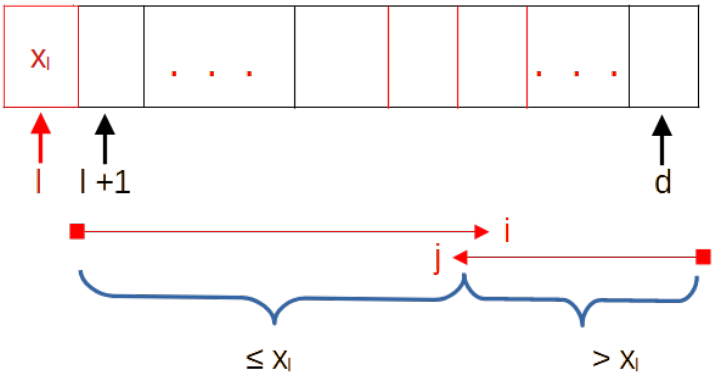
Prava pozicija elementa x_l se određuje **dvostranim** pretraživanjem po **ostatku** niza. Taj postupak se sastoji od dva glavna koraka:

- Sa **svake strane** (prvo lijeve, zatim desne) tražimo prvi sljedeći element koji ne pripada toj strani niza.
- Ako nađemo par takvih elemenata, **zamijenimo im mjesta**.



Dvostranu pretragu prekidamo kada:

- Indeksi i i j moraju biti u **obrnutom** poretku $j < i$.
- **Prava** pozicija elementa x_l je na **indeksu** j (napravimo zamjenu ukoliko je potrebno).



Algoritam za **dvostrano** pretraživanje možemo vidjeti u donjem programskom odsječku. Algoritam izvodimo samo ako je $l < d$. Primijetimo da u liniji 6 moramo provjeriti i slučaj $i == j$. Navedena provjera omogućava križanje indeksa i i j, što omogućava pronalaženje ispravne pozicije elementa x_l.

```
1 if (l < d) {
2     i = l + 1;
3     j = d;
```

```
4
5     /* moramo provjeriti i za i == j */
6      while (i <= j) {
7                 while (i <= d && x[i] <= x[l]) ++i;
8                 while (x[j] > x[l]) --j;
9                 if (i < j) swap(&x[i], &x[j]);
10                   }
```

Nakon što dvostranim pretraživanjem lociramo pravu poziciju elementa x_l trebamo:

- **dovesti** element x_l na njegovu **pravu poziciju** - indeks te pozicije je j.
- **rekurzivno** sortirati **lijevi** i **desni** podniz, bez x_j.

Donji programski odsječak demonstrira navedeni postupak. Funkcija `swap` radi zamjenu vrijednosti ulaznih argumenata, dok funkcija `quick_sort` provodi sortiranje dijela polja x određenih indeksima l i d (u primjeru l, $j-1$ i $j+1$, d).

```
1 if (l < j) swap(&x[j], &x[l]);
2 quick_sort(x, l, j - 1);
3 quick_sort(x, j + 1, d);
4  }
```

Cijeli program uz sve potrebne funkcije za izvođenje QuickSort algoritma navodimo u donjem programskom kodu.

```
1  #include <stdio.h>
2  /*QuickSort algoritam, gdje je x[l] kljucni element.*/
3
4  void swap(int *a, int *b)
5  {
6    int temp;
7    temp = *a;
8    *a = *b;
9    *b = temp;
10   return;
11 }
12
```

```
13 void quick_sort(int x[], int l, int d)
14 {
15    int i, j;
16    if (l < d) {
17        i = l + 1; j = d;
18
19       while (i <= j) {
20           while (i <= d && x[i] <= x[l]) ++i;
21           while (x[j] > x[l]) --j;
22           if (i < j) swap(&x[i], &x[j]); }
23
24      if (l < j) swap(&x[j], &x[l]);
25      quick_sort(x, l, j - 1);
26      quick_sort(x, j + 1, d);
27      }
28 return; }
29
30 int main(void) {
31     int i, n;
32     int x[] = {42, 12, 55, 94, 18, 44, 67};
33     n = 7;
34     quick_sort(x, 0, n - 1);
35     printf("\n␣Sortirano␣polje␣x:\n");
36     for (i = 0; i < n; ++i) {
37       printf("␣x[%d]␣=␣%d\n", i, x[i]);
38      }
39 return 0;
40 }
```

Izlaz glavnog programa je:

```
Sortirano polje x:
 x[0] = 12
 x[1] = 18
 x[2] = 42
 x[3] = 44
 x[4] = 55
 x[5] = 67
 x[6] = 94
```

Primjer 2.1.1

1) QuickSort algoritmom sortiramo ulazno polje.

2) 42 je ključni element. Počinjemo dvostranu pretragu. 12 je na dobroj strani ≤ 42.

3) 55 je na krivoj strani (> 42), zaustavljamo pretragu s lijeve strane. Pokrećemo pretragu s **desne** strane. 67 je na **dobroj** strani (> 42).

4) 44 je na **dobroj** strani (> 42).

5) 18 je na **krivoj** strani (≤ 42) stoga stajemo pretragu s desne strane. Pošto je $i < j$ vršimo zamjenu para elemenata na krivim stranama.

6) Nastavljamo dvostranu pretragu s **lijeve** strane. 94 je na krivoj strani (> 42) - zaustavljamo pretragu s lijeve strane.

7) Nastavljamo dvostranu pretragu s **desne** strane. 94 je na **dobroj** strani (> 42).

8) 18 je na krivoj strani (≤ 42) - zaustavljamo pretragu s desne strane. $j < i$ - nema zamjene, kraj dvostrane pretrage.

9) Pošto je $l < j$, vršimo zamjenu x_l i x_j.

10) Prava pozicija za 42 je x_2.

11) Istim postupkom rekurzivno sortiramo podnizove $[x_0,\ x_1]$ i $[x_3,\ x_4,\ x_5,\ x_6]$. Prvo rekurzivno primijenimo algoritam na $[x_0,\ x_1]$.

12) Pretraga s lijeve strane pokazuje da je 12 na pravoj strani. $12 < 18$ pa se ne pomičemo s desne strane.

13) $i > j$ pa radimo zamjenu $x[l]$ i $x[j]$.

14) 18 je sada na svojoj pravoj poziciji.

15) $[x_0]$ je polje duljine 1 ($l == d$) pa je i 12 na svojoj pravoj poziciji.

16) Primijenjujemo QuickSort na desno polje. Započinjemo pretragu s lijeva. 55 je na pravoj strani ($55 < 94$).

17) $44 < 94$ pa je i 44 na dobroj strani.

18) $67 < 94$ pa je i 67 na dobroj strani.

19) Pretragom s desna ne pomičemo granicu jer su svi elementi < 94. Pošto $i > j$ radimo zamjenu $x[l]$ i $x[j]$.

20) 94 je sada na svojoj pravoj poziciji.

21) Započinjemo QuickSort na polju $[x_3, x_4, x_5]$, drugi poziv QuickSort algoritma neće raditi promjene nad poljem pošto je 94 na ispravnoj poziciji. Započinjemo dvostranu pretragu s lijeve strane i zaključujemo da je $55 < 67$ na dobroj strani.

22) $44 < 67$ je također na pravoj strani.

23) $i > j$ pa radimo zamjenu $x[l]$ i $x[j]$.

24) Sada je 67 na svojoj pravoj poziciji.

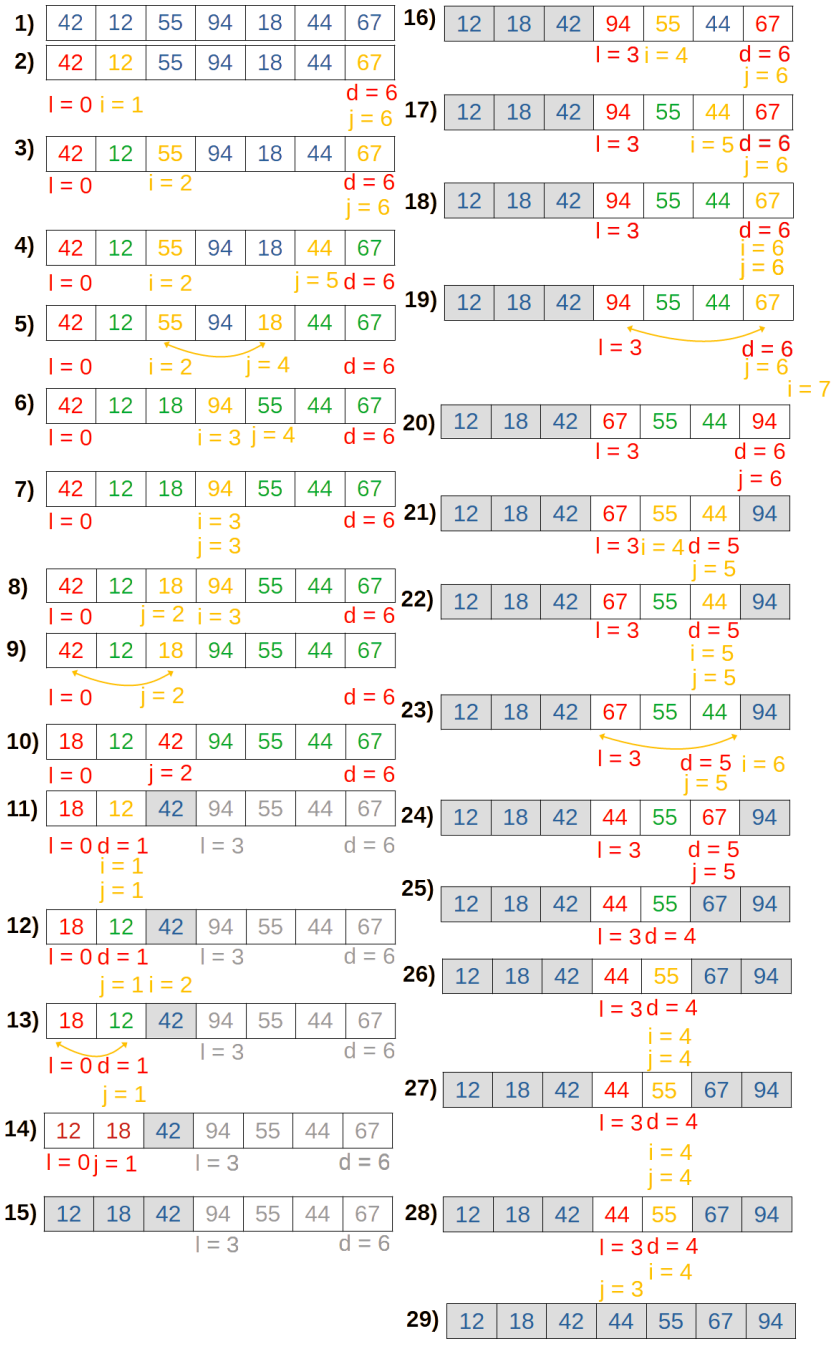
25) Primijenjujemo QuickSort na polje $[x_3, x_4]$, drugi poziv QuickSort algoritma neće mijenjati polje pošto je 67 na svojoj pravoj poziciji.

26) Započinjemo pretragu s lijeve strane i zaključujmo da je $55 > 44$, stoga on nije na pravoj strani. Završavamo s pretragom s lijeve strane i započinjemo pretragu s desne strane.

27) $x[j] > x[l]$ pa smanjujemo j na 3. Prekidamo pretragu s desna pošto $x[l] == x[j]$.

28) $i > j$ i $l == j$ pa ne radimo nikakve zamjene. Pozivamo QuickSort algoritam s indeksima $l = 3$, $j = 2$ što odmah završava s izračunavanjem jer $l > d$ i s indeksima $l = 4$, $d = 4$ što odmah završava s izračunavanjem jer $l == d$.

29) Dobili smo sortirano polje.



Prosječna složenost algoritma je $\mathcal{O}(n \cdot log_2(n))$ za slučajne **dobro ran-**

domizirane nizove. U najgorem slučaju je složenost $\mathcal{O}(n^2)$ za **već sortirani** i **naopako sortirani** niz. Pošto se randomizirani nizovi uglavnom javljaju u praksi, QuickSort algoritam se jako puno koristi. Na randomiziranim nizovima, uz pravilno pivotiranje, QuickSort algoritam može biti brži i od algoritama sortiranja koji uvijek imaju složenost $\mathcal{O}(n \cdot log_2(n))$, kao što je npr. **MergeSort**.
Predstavljenu implementaciju QuickSort algoritma možemo dodatno poboljšati:

- Za $n = 2, 3$ sortiramo klasičnim algoritmom (provjere zamjena).
- Za $n > 3$ kao ključni element izaberemo *srednji* od neka 3 (ubrzanje oko 30%).
- Kontrola dubine rekurzije (prvo obradi **kraće** od dva preostala polja). Dulje polje smjesti na programski stog.

Uz navedeno, postoji niz drugih optimizacija QuickSort algoritma.
U standardnoj `C` biblioteci - datoteka zaglavlja `<stdlib.h>`, postoje funkcije:

- `qsort` - QuickSort algoritam za sortiranje niza podataka.
- `bsearch` - Binarno traženje zadanog podatka u sortiranom nizu.

Pri korištenju navedenih funkcija moramo sami zadati funkciju za uspoređivanje podataka u nizu.

```
1 void qsort(void *base, size_t n, size_t size,
2 int (*comp) (const void *, const void *));
3
4 void *bsearch(const void *key, const void *base,
5 size_t n, size_t size,
6 int (*comp) (const void *, const void *));
```

Funkcije su definirane tako da podržavaju sortiranje (`qsort`) odnosno pretraživanje (`bsearch`) polja proizvoljnih elemenata. Kod funkcije `bsearch` je prvi argument void pokazivač na traženi element, a drugi argument (prvi kod `qsort`) je `void` pokazivač na prvi element ulaznog polja. `void` pokazivač može uz jaka ograničenja pokazivati na element proizvoljnog tipa. Sljedeća dva argumenta definiraju broj elemenata u polju i veličinu svakog elementa u bajtovima. Zadnji argument moramo definirati sami i predstavlja pokazivač na funkciju `comp` (dolazi od komparator - funkcija za usporedbu elemenata),

koja prima dva `const void` pokazivača na dva elementa u polju, uspoređuje ih i ovisno o tome vraća cjelobrojnu vrijednost (> 0 ako je prvi argument veći od drugoa, 0 ako imaju jednaku vrijednost, inače < 0).

Zadatak: isprobajte funkcije `qsort` i `bsearch` koristeći polja s elementima različitih tipova. Napišite odgovarajuće komparatore za usporedbu elemenata.

2.1.2 Brojanje particija

Particija dolazi od latinske riječi *partitio*, a znači dijeljenje, dioba, razdioba[2]. Particije u matematici nam omogućuju analizu brojeva i uvid u njihovu strukturu. Npr. particije se koriste pri analizi distribucije i generiranju prostih brojeva, iznimno interesantne teme u teoriji brojeva, relevantne i za kriptografiju te računalnu sigurnost. Interesantan je i broj particija brojeva gdje su svi faktori prosti brojevi. U statističkoj mehanici, particije se koriste za poboljšanje razumijevanja distribucije energetskih stanja između čestica. Particioniranje resursa se javlja u prirodi, prvenstveno da se izbjegne natjecanje više vrsta za ograničene resurse. Particioniranje je također važan alat pri analizi raznih bioloških objekata (proteini, DNA, RNA). Kod raznih problema iz svakodnevnog života je često potrebno pronaći i na koliko (različitih) načina se neki prirodan broj n može rastaviti kao suma ne padajućeg niza prirodnih brojeva, odnosno izračunati broj particija broja n. Broj particija možemo koristiti za računanje broja načina da n jednakih objekata smjestimo u identične kutije. Međutim, u stvarnom životu često imamo uvjete na broj dostupnih kutija ili maksimalan broj predmeta koji stane u kutiju, stoga ćemo u algoritme koji računaju broj particija morati dodavati uvjete koji će onemogućiti brojanje nekih, neodgovarajućih, particija.

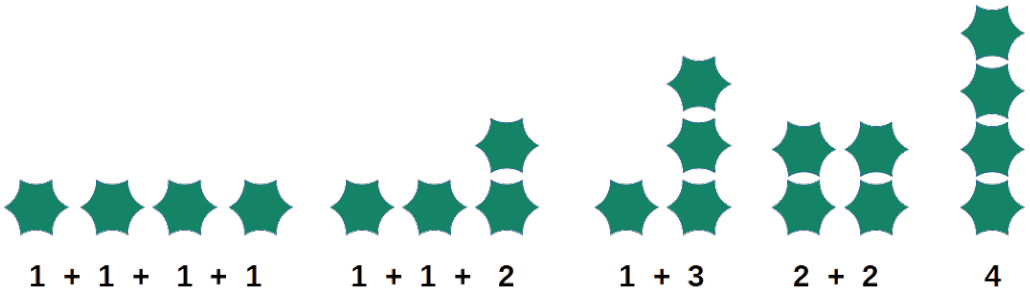
Definicija 2.1.1. Particija (rastav) prirodnog broja $n \in \mathbb{N}$ je bilo koji rastav zadanog broja u **zbroj pribrojnika** koji su također **prirodni brojevi**, pri čemu **poredak** pribrojnika **nije bitan**. Particija od n ima oblik: $n = a_1 + a_2 + \cdots + a_m$, gdje je $m \in \mathbb{N}$ (broj pribrojnika u rastavu), $a_1, \ldots, a_m$ su **pribrojnici**. **Broj particija** od n označavamo s $p(n)$, gdje broj pribrojnika m može biti proizvoljan.

S obzirom na to da poredak pribrojnika nije bitan, možemo smatrati da su pribrojnici poredani (nepadajuće): $n = a_1 + a_2 + \cdots + a_m,\ a_1 \leq a_2 \leq \cdots \leq a_m$.

[2] `https://www.enciklopedija.hr/clanak/particija`

Primjer 2.1.2

Promotrimo broj particija za $n = 4$:[3]



Broj particija je: $p(4) = 5$.

Problem s kojim se susrećemo pri izradi algoritma za brojanje particija prirodnog broja je što broj pribrojnika m može varirati od 1 do n. Zbog toga **ne možemo** koristiti niz petlji za rješavanje problema!

Particioniranje cijelog broja na jedan faktor se može napraviti točno na jedan način, stoga takav algoritam ne trebamo niti pisati. Broj rastava na 2 faktora je $\lfloor \frac{n}{2} \rfloor$. Funkciju za ispis svih rastava (particija) broja n na točno 2 faktora konstruiramo jednostavno.

```
1 int particionirajNaDva(int n){
2 int broj = 0;
3 for(int i=1;i<=n/2;i++){
4             printf("%d␣%d\n", i, n-i);
5             broj++;
6 }
7    return broj;
8 }
```

Funkcija koja broji i ispisuje rastav prirodnog broja na tri faktora koji su u nepadajućem poretku zahtijeva korištenje minimalno dvije petlje.

```
1 int particionirajNaTri(int n){
2 int broj = 0;
```

[3]Particije se uobičajeno vizualiziraju koristeći Ferrer-ove `https://mathworld.wolfram.com/FerrersDiagram.html` i Young-ove `https://encyclopediaofmath.org/wiki/Young_diagram` dijagrame.

```
3  for(int i=1;i<=n/2;i++){
4      for(int j=i;n-i-j>=j;j++){
5              printf("%d␣%d␣%d\n", i, j, b-i-j);
6              broj++;
7      }
8  }
9     return broj;
10 }
```

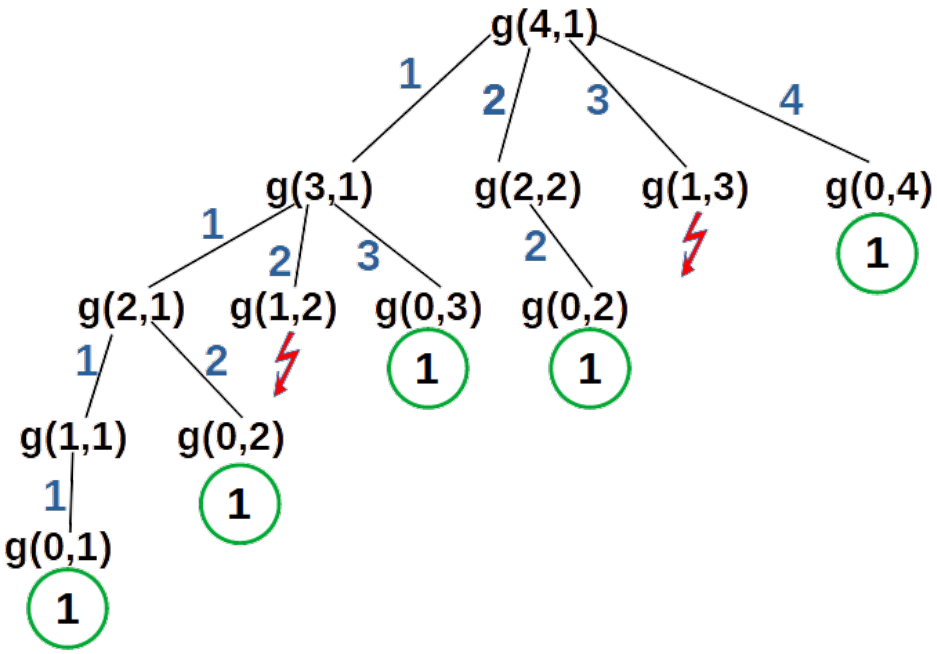
Računanje broja rastava prirodnog broja na k faktora bi tako zahtijevalo korištenje $k-1$ petlje. Očito na ovaj način ne možemo konstruirati algoritam kojim možemo izračunati broj particija prirodnog broja. U navedenim primjerima smo istovremeno generirali i brojali particije. Generiranje particija od n (u nekom redoslijedu) svodimo na nekoliko koraka visoke razine:

- generiramo pribrojnik po pribrojnik
- pazimo da pribrojnici **ne padaju** ($a_l \geq a_{l-1}$ za $l \geq 2$).
- pojedini pribrojnik moguće dodati i više puta

Problem možemo riješiti rekurzivno.[4] U svakom pozivu funkcije izvršavamo petlju za izbor pribrojnika a_l. **Rekurzivni pozivi** realiziraju petlje unutar petlji (broj pojavljivanja pribrojnika a_l u sumi). Označimo s g funkciju koja kao parametre prima trenutni broj n za koji tražimo particiju i broj k koji označava minimalni pribrojnik a_l koji možemo dodati u sljedećem koraku. Funkcija ispituje, na koliko načina možemo particionirati broj n pribrojnicima $a_k \geq a_l$. Nakon dodavanja broja $a_l = k$ u particiju, treba pronaći particiju broja $n-k$ (to je rekurzivni korak). Broj k možemo dodavati maksimalno n puta i brojeve (prema prethodnom razmatranju) dodajemo u uzlaznom poretku. Postavlja se pitanje što je terminalni uvjet ove rekurzije? Poziv funkcije kod kojeg je $n = 0$, tada brojač povećamo za 1. Pronašli smo niz pribrojnika $a_1, a_2, \ldots a_k$ takvih da $n - a_1 - a_2 - \cdots - a_k = 0$, dakle uspješno smo rastavili n. U slučaju $n < a_l$ ne povećavamo brojač. U tom slučaju, izabrani pribrojnici daju sumu $> n$, stoga taj izbor zanemarujemo.

[4]Problem se može riješiti i nerekurzivno korištenjem tehnike koja se zove *Dinamičko programiranje* , međutim njome se nećemo trentno baviti. Više informacija možete vidjeti na web stranicama kolegija Strukture podataka i algoritmi `https://web.math.pmf.unizg.hr/nastava/spa/index.php`.

Funkciju g inicijalno pozovemo: $g(n, 1)$ (na koliko načina možemo particionirati n ukoliko koristimo pribrojnike ≥ 1).



Slika gore prikazuje stablo rekurzije funkcije g koju koristimo za računanje $p(4)$. Plavi brojevi prikazuju koji broj smo dodali rastavu u nekom koraku. Primijetimo da na dubljoj razini uvijek moramo koristiti broj koji je veći ili jednak prethodno korištenom (da bi imali nepadajuće particije). Kao što vidimo, pronašli smo svih 5 slučajeva uspješnog rastava broja 4.

Sada možemo zapisati funkciju koja računa broj particija prirodnog broja i odgovarajuću funkciju `main`.

```
1  #include <stdio.h>
2
3  int particije(int suma, int prvi){
4       int i, broj = 0;
5
6       if (suma == 0) return 1;
7
8       for (i = prvi; i <= suma; ++i)
9            broj += particije(suma - i, i);
10 return broj;
```

```
11 }
12
13 int main(void){
14
15 int n;
16 printf("␣Upisi␣prirodni␣broj␣n:␣");
17 scanf("%d", &n);
18 printf("\n␣Broj␣particija␣p(%d)␣=␣%d\n",n,
19 particije(n, 1) );
20
21 return 0;
22 }
```

Trag poziva možemo dobiti tako da na vrh funkcije dodamo ispis ulaznih vrijednosti.

```
1  int particije(int suma, int prvi){
2
3  int i, broj = 0;
4  printf("␣suma␣=␣%d,␣prvi␣=␣%d\n", suma, prvi);
5
6  if (suma == 0) return 1;
7
8  for (i = prvi; i <= suma; ++i)
9       broj += particije(suma - i, i);
10
11 return broj;
12 }
```

Promotrimo dva primjera u kojima uvodima neka realna ograničenja na broj faktora ili njihovu veličinu.

Primjer 2.1.3

Imamo n identičnih proizvoda koje želimo rasporediti u identične kutije. Svaka kutija može sadržavati proizvoljnu količinu proizvoda, međutim imamo na raspolaganju maksimalno k kutija. Želimo izračunati na koliko načina možemo rasporediti proizvode u kutije.

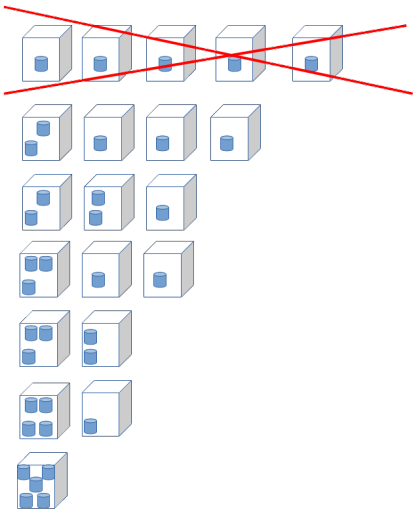
Problem rješavamo modifikacijom algoritma za računanje svih particija. Glavno ograničenje u odnosu na računanje svih particija je što imamo na raspolaganju maksimalno k kutija. Kao što smo vidjeli kod računanja particije, svaki

faktor je reprezentiran jednim rekurzivnim pozivom, stoga moram uvesti ograničenje na dubinu rekurzije.

```
1  int KutijeOgraniceno(int n, int prvi, int k, int kMax){
2      int broj = 0;
3
4      if(n == 0) return 1;
5      if(k == kMax) return 0;
6
7      for(int i=prvi;i<=n;i++){
8          broj+=KutijeOgraniceno(n-i,i,k+1, kMax);
9      }
10
11     return broj;
12 }
```

U gornjem kodu nam n označava broj proizvoda, prvi označava koliko minimalno proizvoda ćemo staviti u kutiju (osigurava nepadajući niz faktora), k broj trenutno iskorištenih kutija i $kMax$ maksimalan broj dostupnih kutija. Vidimo da smo u funkciju dodali terminalni uvjet `if(k == kMax) return 0` koji prekida rekurzivni poziv ukoliko nismo uspjeli smjestiti proizvode u maksimalno $kMax$ kutija.

Za $n = 5$, $k = 4$ postoji 6 rasporeda koji zadovoljavaju uvjete zadatka.



Primjer 2.1.4

Imamo n identičnih proizvoda koje želimo rasporediti u identične kutije. Svaka kutija može sadržavati maksimalno s proizvoda i imamo na raspolaganju maksimalno k kutija. Želimo izračunati na koliko načina možemo rasporediti proizvode u kutije.

Ovdje, uz uvjet of maksimalnom broju kutija, imamo dodatni uvjet, svaka kutija može sadržavati maksimalno s proizvoda. Uvjet realiziramo tako da u petlji koja stavlja određeni broj proizvoda u kutiju (linija 8), ograničimo broj na s ($maxProstor$ u liniji 8).

```
1  int KutijeOgraniceno1(int n, int prvi, int k, int kMax,
2                                              int maxProstor){
3      int broj = 0;
4
5      if(n == 0) return 1;
6      if(k == kMax) return 0;
7
8      for(int i=prvi;i<=maxProstor;i++){
9          broj+=KutijeOgraniceno1(n-i,i,k+1, kMax,
10                                              maxProstor);
11     }
12
13     return broj;
14 }
```

Ograničavanje petlje u liniji 8, zahtjeva dodavanje argumenta $maxProstor$ koji sadrži informaciju o broju proizvoda koje možemo smjestiti u kutije. Isti argument se prosljeđuje funkciji u svakom rekurzivnom pozivu.

Sada za $n = 5$, $k = 4$, $s = 2$ postoje samo 2 rasporeda koja zadovoljavaju sve uvjete zadatka. Kao i u prethodnom primjeru, prvi raspored je odbačen zato što koristi više od k kutija, dok su posljednja 4 rasporeda odbačena jer prva kutija sadrži više od s proizvoda, što nije dozvoljeno.

Svaku rekurzivnu funkciju možemo svesti na ne-rekurzivnu korištenjem dodatne strukture podataka koja se zove `stog` (eng. `stack`). Pristup zahtijeva kodiranje informacija o stanju varijabli, koje se umjesto rekurzivnog poziva stavlja na stog. Umjesto povrata iz rekurzivnog poziva, skidamo informacije sa stoga i nastavljamo računanje.

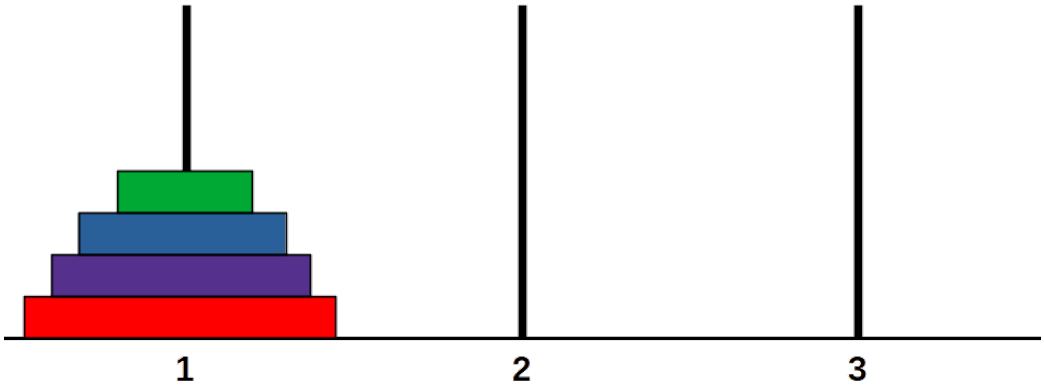
2.1.3 Problem Hanojskih tornjeva

Problem Hanojskih tornjeva je 1883. godine predstavio francuski matematičar Éduard Lucas. Problem je navodno inspiriran legendom[5].

Najčešća formulacija problema je:

Imamo zadana 3 štapa, označimo ih s $1, 2, 3$. Na štapu 1 se na početku nalazi n **diskova** međusobno različitih veličina, poslaganih **sortirano** od **najvećeg** prema **najmanjem** (odozdo prema gore) dok su štapovi 2 i 3 prazni. Zadatak je preseliti **svih** n diskova sa štapa 1 na štap 3, u **minimalnom** broju **poteza**, korištenjem pomoćnog štapa 2. Pri pomicanju štapova, moramo se pridržavati sljedećih pravila:

[5]Za više detalja vidi: `https://repozitorij.pmf.unizg.hr/islandora/object/pmf:9549` i `https://www.fer.unizg.hr/_download/repository/diskont1-01.pdf`



- u svakom potezu se može prebaciti samo jedan disk (najgornji na nekom štapu, na vrh nekog drugog)
- veći disk nikada se ne smije staviti iznad manjeg diska
- manji (najgornji) disk se smije preseliti iznad bilo kojeg većeg diska na nekom drugom štapu, ili na prazan štap.

Osnovni korak u ovom problemu je prebacivanje najgornjeg (najmanjeg) diska s jednog štapa (**odakle**) na drugi (**kamo**) poštujući pravila igre.

Primijetimo da ukoliko možemo određenim nizom poteza prebaciti gornjih $n-1$ diskova sa štapa **odakle** na štap **pomoćni** (pritom nam najdonji i najveći disk ne smeta pri prebacivanju jer su preostali manji od njega), tada bi osnovnim korakom mogli prebaciti najdoljnji (najveći) disk sa štapa **odakle** na štap **kamo**. U tom slučaju bi imali problem prebacivanja $n-1$ diskova sa štapa **pomoćni** na štap **kamo**. Dakle, sveli bi problem prebacivanja n diskova na istovjetni problem prebacivanja $n-1$ diskova. Navedeni postupak se naziva *smanji pa vladaj* i posebna je varijanta postupka *podijeli pa vladaj*.

Sljedeći gore spomenutu ideju, želimo rekurzivno smanjivati n dok ne bude $n = 1$. Taj slučaj riješimo korištenjem **osnovnog poteza** (prebacimo najmanji disk na vrh štapa 3) i time riješimo početni problem.

Funkcija za rješavanje problema će morati biti parametrizirana brojem diskova koje još moramo prebaciti n, te štapovima (**odakle**, **pomoćni**, **kamo**). Na početku će biti **odakle** $= 1$, **pomoćni** $= 2$ i **kamo** $= 3$, no tijekom rješavanja problema, štapovi će mijenjati uloge. Kao što smo naveli, nakon što najveći disk prebacimo na kraj, trebamo $n-1$ preostalih diskova prebaciti sa štapa **pomoćni** (štapa 2) na štap **kamo** (štap 3). To ćemo opet ostvariti tako da

gornjih $n-2$ diskova pomaknemo na štap 1 (u ulozi **kamo**) koristeći štap 3 (u ulozi **pomoćni**). Nakon čega možemo najdoljnji disk na štapu 2 pomaknuti na štap 3, čime reduciramo problem na prebacivanje $n-2$ diskova sa štapa 1 na štap 3. Prebacivanje jednog diska sa štapa **odakle** na štap **kamo** ćemo ostvariti pozivom funkcije `void prebaci_jednog(int odakle, int kamo)`. Funkcija će ispisati potez na komandnu liniju[6]. Funkcija `void Hanojski_tornjevi(int n, int odakle, int kamo, int pomocni)` realizira **rekurzivno** prebacivanje **gornjih** n diskova u obliku:

- prebaci **gornjih** $n-1$ diskova s pomoćnog na odredišni štap.
- prebaci **jedan** disk (najdonji) na odredišni štap.
- prebaci gornjih $n-1$ disk s pomoćnog štapa na odredišni štap.

Realizacija funkcije može varirati ovisno o tome želimo li kao **rješenje problema** računati redoslijed koraka ili broj osnovnih poteza. Također, varijablu **pomocni** možemo izračunati iz **odakle** i **kamo**, ali ju navodimo zbog jednostavnosti[7]

U nastavku navodimo implementaciju funkcija potrebnih za realizaciju računalnog rješenja problema Hanojskih tornjeva u kojem ispisujemo niz koraka koje trebamo napraviti da bi prebacili sve diskove na odredište (štap 3). Prilažmo i implementaciju odgovarajuće glavne funkcije.

```
1  #include <stdio.h>
2  void prebaci_jednog(int odakle, int kamo){
3     printf("␣prebaci␣s␣%d␣na␣%d\n", odakle, kamo);
4  return;
5  }//ispisuje osnovni potez
6
7  void Hanoj(int n, int o, int k, int p){
8  if (n <= 1) // Uz n > 0, to znaci n == 1.
9      prebaci_jednog(o, k);
10 else {
11    Hanoj(n - 1, o, p, k);
12    prebaci_jednog(o, k);
13    Hanoj(n - 1, p, k, o); }
```

[6]Kod aplikacije s grafičkim sučeljem bi ostvarila crtanje pomaka diska na novi štap.

[7]**Animaciju** računanja rješenja problema Hanojskih tornjeva možete naći na web stranici: `https://www.mathsisfun.com/games/towerofhanoi.html`.

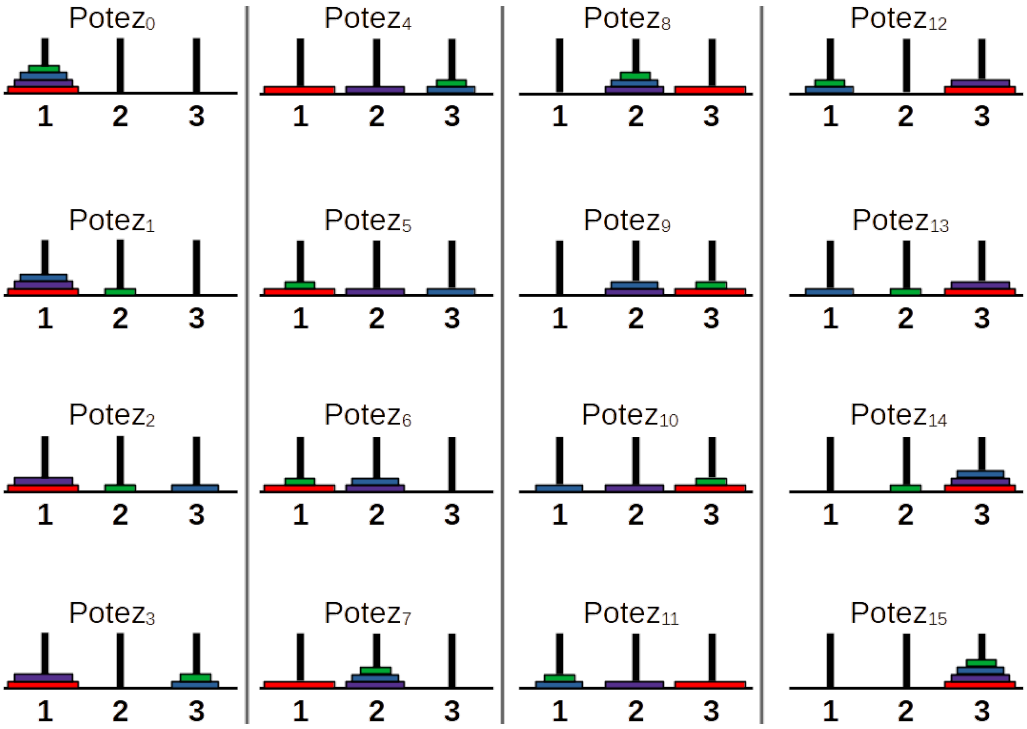
```
14 return; }
15 //o-odakle, k-kamo, p-pomocni
16
17 int main(void) {
18 int n;
19
20 for (n = 1; n <= 3; ++n) {
21   printf("\n Prebaci %d diskova s 1 na 3:\n", n);
22   Hanoj(n, 1, 3, 2);
23 }//Broj diskova iteriramo od 1 do 3
24 return 0;
25 }
```

Izlaz programa je:

```
Prebaci 1 diskova s 1 na 3:
prebaci s 1 na 3

Prebaci 2 diskova s 1 na 3:
prebaci s 1 na 2
prebaci s 1 na 3
prebaci s 2 na 3

Prebaci 3 diskova s 1 na 3:
prebaci s 1 na 3
prebaci s 1 na 2
prebaci s 3 na 2
prebaci s 1 na 3
prebaci s 2 na 1
prebaci s 2 na 3
prebaci s 1 na 3
```

Primjer 2.1.5

Ilustrirat ćemo korake algoritma za rješavanje problema Hanojskih tornjeva u uz 3 štapa i 4 diska.



Pretpostavimo da imamo n diskova i neka je h_n **broj osnovnih** poteza prebacivanja po jednog diska dok ne dođemo do rješenja. Tada, zbog prebacivanja $n-1$ diska na **pomocni**, prebacivanja najvećeg diska na **kamo** i prebacivanja $n-1$ diska s **pomocnog** na **kamo** imamo:

$$h_n = h_{n-1} + 1 + h_{n-1} = 2h_{n-1} + 1, \ n \geq 1$$

Rješenje gornje nehomogene rekurzije je: $h_n = 2^n - 1, \ n \geq 0$ (način rješavanja na višim godinama studija)[8].
Iz algoritma i analize možemo zaključiti:

- polazni problem za n diskova uvijek ima rješenje za svaki n,
- minimalni broj poteza je jednak $2^n - 1$

optimalno rješenje je **jedinstveno**.

Razlog brzog rasta broja poteza je činjenica da koristimo samo 3 štapa.

[8] Vidi `https://web.math.pmf.unizg.hr/nastava/oaa/`

Poopćenje problema dobijemo korištenjem $k \geq 3$ štapova. Tada problem ima više rješenja a traži se minimalan broj poteza. Generalno je problem lakše riješiti uz povećanje broja štapova. Npr. problem uz n diskova i n štapova možemo riješiti u $2n + 1$ poteza, dok problem uz n diskova i $n + 1$ štapova možemo riješti u $2n-1$ poteza. Frame-Stewartov algoritam[9] predlaže moguće rješenje generaliziranog problema Hanojskih tornjeva. Optimalnost je dokazana za $k = 4$[10], međutim, koliko je autoru poznato, do sada nije dokazana optimalnost algoritma u punoj generalnosti.

Ograničeni problem Hanojskih tornjeva dobijemo uvođenjem dodatnih ograničenja na poteze. Npr., možemo imati 3 štapa uz dodatno ograničenje:

- u svakom potezu se disk smije prebaciti s nekog štapa samo na **susjedni** štap.
- **zabranjeno** je izravno prebacivanje s **prvog** na **treći** štap ili obratno.

Takva ograničena verzija problema zahtjeva minimalno $3^n - 1$ korak da bi došli do rješenja problema.

2.2 Struktura programa

`C` program je skup definicija **objekata** (varijabli) i **funkcija**. Varijablama programer dodjeljuje ime (identifikator), koje omogućava identifikaciju i rad s nekim objektom, a operacijski sustav adresu, koja omogućava komunikaciju s radnom **memorijom** gdje su podaci koji definiraju objekt fizički smješteni. Ukratko, varijable **zauzimaju memoriju** a funkcije **sadrže instrukcije**. Funkcije komuniciraju preko **argumenata** i **povratnih vrijednosti**, **vanjskih ili globalnih varijabli** (varijable definirane izvan bilo koje funkcije).

Funkcije mogu biti u **proizvoljnom poretku** u izvornom programu, a program može biti smješten (rastavljen) u **više datoteka** izvornog koda, koje su spremljene u trajnoj memoriji. Dijeljenje neke funkcije na više datoteka izvornog koda **nije dozvoljeno**.

Objekti u `C` programu mogu biti:

[9] `https://www.jstor.org/stable/2304268`

[10] Vidi `https://doi.org/10.36045/bbms/1420071861` i `https://arxiv.org/pdf/1601.04298`

- Globalni ili vanjski (eng. external) su definirani izvan bilo koje funkcije. Imena vanjskih varijabli su **vanjski** simboli. Svaka referenca (ime, identifikator) na takav objekt s istim imenom se odnosi na taj konkretan objekt. Pripadni objekti su univerzalno dohvatljivi (čak i kada su funkcije koje ih dohvaćaju smještene u raznim datotekama izvornog koda)[11]. Univerzalna dohvatljivost se može ograničiti korištenjem ključne riječi `static`. Tada je varijabla globalna, ali dohvatljiva samo iz datoteke izvornog koda u kojoj je definirana.

- Lokalni ili unutarnji (eng. internal) su definirani lokalno unutar neke funkcije. **Unutarnji objekti** nisu univerzalno dohvatljivi, već su argumenti funkcija i varijable definirane unutar funkcija.

Funkcije u `C`-u su **uvijek globalne ili vanjske**. Nije dozvoljeno definirati funkcije unutar drugih funkcija [12].

Blok naredbi je **svaki niz** naredbi koji se nalazi unutar vitičastih zagrada (npr. tijelo funkcije). C dozvoljava da se u svakom bloku naredbi deklariraju varijable (lokalne varijable). Deklaracija varijabli unutar bloka mora prethoditi prvoj izvršnoj naredbi u bloku (standard `C90`). U standardu `C99` je taj uvjet ukinut.

```
1  if (n > 0) {
2  int i; /* deklaracija varijable */
3  for (i = 0; i < n; ++i)
4  ...
5  } //C90
6
7  if (n > 0) {
8  for (int i = 0; i < n; ++i)
9  ...
10 } //C99 i noviji
```

Vidljivost varijabli ovisi o mjestu deklaracije te varijable:

- Varijabla definirana unutar bloka vidljiva je samo unutar tog bloka.

- Izvan bloka, u kojem je definirana lokalna varijabla, se toj varijabli ne može pristupiti (nije definirana). Iznad ili ispod tog bloka može

[11] Postoje iznimna stanja u kojima je globalna varijabla *pokrivena* lokalnom varijablom istog imena. O tome ćemo više reći u nastavku.

[12] U nekim programskim jezicima, npr. Pascal-u je to moguće.

biti deklarirana varijabla istog imena kao varijabla deklarirana unutar bloka. Varijabla istog imena deklarirana iznad bloka će biti nedostupna unutar bloka jer je prekrivena lokalnom varijablom.

- Varijabla deklarirana iznad bloka vidljiva je u tom bloku ako nije pokrivena lokalnom varijablom istog imena. Dostupna je najlokalnija varijabla istog imena.

```
1 int main(void) {
2 int x, y;
3 ...
4 if (x > 0){
5    double y; /* int y NIJE vidljiv u bloku */
6 /* int x JE vidljiv u bloku */
7 ...
8   }
9 }
```

```
1 int main(void) {
2 int x;
3 ...
4 if (x > 0){
5    double y;
6 ...
7   }
8 int y; //OK! double y nije vidljiv van bloka
9 }
```

Formalni argument funkcije vidljiv je unutar funkcije i nije dohvatljiv (niti definiran) izvan nje. Doseg (vidljivost) formalnog argumenta je isti kao i doseg lokalne varijable definirane na početku funkcije.

```
1 int x, y;
2 ...
3 void f(double x) {
4      double y; /* int x i int y NISU */
5 ... /* vidljivi unutar funkcije */
6 }
7
8 int main(void) { ... }
```

Varijable imaju tri atributa: a) tip, b) doseg (engl. *scope*) i c) vijek trajanja (eng. *lifetime*).

- Tip - način interpretacije sadržaja bloka[13], uključuje i veličinu bloka.
- Vijek trajanja - određuje u kojem dijelu memorije programa se rezervira taj blok. Postoje tri dijela memorije: statički, programski stog (eng. *run-time stack*) i programska hrpa (eng. *heap*).
- Doseg - u kojem dijelu programa je dio memorije dohvatljiv ili vidljiv za korištenje (čitanje, mijenjanje vrijednosti).

Vrste varijabli:

- Varijable prema tipu mogu biti: `int`, `char`, `double`, `float` itd.
- Prema dosegu, varijable mogu biti: **lokalne** (unutarnje) i **globalne** (vanjske).
- Prema vijeku trajanja, varijable mogu biti: **automatske**, **statičke** i **dinamičke**.

Doseg i vijek trajanja određeni su mjestom deklaracije/definicije objekta (varijable) unutar ili izvan neke funkcije. Upravljanje vijekom trajanja (a ponekad i dosegom), te dijelom memorije u kojem se nalazi varijabla vrši se **identifikatorima memorijske klase**. Ključne riječi `auto`, `extern`, `register` i `static` kod deklaracije objekta označavaju njegovu memorijsku klasu[14].

Identifikatori memorijske klase se pišu pri deklaraciji varijable ili funkcije ispred identifikatora tipa varijable ili funkcije.

```
1 identif_mem_klase tip_var ime_var;
2 identif_mem_klase tip_fun ime_fun ... ;
```

Način korištenja identifikatora memorijske klase možemo vidjeti u donjem odsječku koda.

```
1 auto int *pi;
2 extern double l;
3 register int z;
4 static char polje[10];
```

[13] Blok se piše i čita u bitovima.

[14] `typedef` se također može uvrstiti u identifikatore memorijske klase iako se on primarno koristi za definiranje novih tipova

2.2.1 Automatske varijable

Automatska varijabla je svaka varijabla kreirana **unutar nekog bloka** (unutar svake funkcije) koja nema ključnu riječ `static` u deklaraciji. Takve varijable su lokalne po dosegu. Automatske varijable kreiraju se na ulasku u blok u kome su deklarirane i uništavaju se na izlasku iz tog bloka. Nakon uništavanja se memorija koju su zauzimale oslobađa za druge varijable (vrijednosti se gube). Ovaj postupak se odvija na **programskom (izvršnom) stogu**.

Identifikator `auto` deklarira automatsku varijablu. Međutim, sve varijable deklarirane unutar nekog bloka implicitno su automatske (ako nisu deklarirane `static`). Ključna riječ `auto` se obično ne koristi u programima iako je dopušteno.

Ekvivalentno je pisati:

```
{                               {
   char c;                        auto char c;
   int i,j,k;                     auto int i,j,k;
   ...                            ...
}                               }
```

```
...
void f(double x) {
     double y = 2.71;
     static double z;
...
}
```

U gornjem odsječku koda su automatske varijable x (formalni argument) i y. Varijabla z je **statička** varijabla.

Inicijalizacija automatske varijable, ukoliko postoji, se vrši prilikom svakog novog ulaza u blok u kojem je varijabla definirana. Tada se rezervira memorija i izvrši inicijalizacija. Automatska varijabla koja nije inicijalizirana na ulasku u blok u kojem je definirana dobiva nepredvidivu vrijednost. Tada se rezervira memorija bez promjene sadržaja memorijske lokacije. Inicijalizaciju automatske varijable moguće je izvršiti konstantnim ili nekonstantnim izrazom.

```
void f(double x) {//inicijalizacija konstantnim izrazom
```

```
2       double y = 2.71; }
3
4  void f(double x) {//inicijalizacija nekonstantnim
5       double y = 2*x; } //izrazom
```

2.2.2 Vrijednosti u registru procesora

Identifikator memorijske klase `register` sugerira prevoditelju da varijablu smjesti u registar procesora. Navedenu preporuku prevoditelj ne mora izvršiti i uglavnom je većina implementacija prevodioca ovu instrukciju ignorirala. Moderni prevodioci koriste razne optimizacije i poznavanje strukture programa za ubrzavanje izvođenja, zato su uglavnom optimizacije prevodioca bolje od prijedloga programera. Zbog toga je funkcionalnost ukinuta u verziji `C11`, ali je ključna riječ zadržana (i dalje je rezervirana ključna riječ) za potencijalnu buduću uporabu.

Ideja korištenja je bila skratiti vrijeme izvođenja programa koristeći brži pristup memoriji (registar procesora). Uglavnom se u registar stavljaju često korištene varijable (kontrolne varijable petlji, brojači). Identifikator `register` je primijenjiv samo na automatske varijable (unutar nekog bloka).

```
1  int f (register int m, register long n) {
2        register int i;
3        ...
4  }
```

Zabranjeno je primijeniti **adresni operator** i **pokazivač** na `register` varijable.

2.2.3 Statičke varijable

Statička varijabla je varijabla definirana **izvan svih funkcija** ili varijabla deklarirana u nekom bloku (npr. tijelu funkcije) identifikatorom memorijske klase `static`. Statičke varijable se spremaju u statički dio memorije.

Statičke varijable su aktivne tijekom cijelog izvršavanja programa. Kreiraju se na početku izvršavanja programa i uništavaju tek na završetku programa. Statičke varijable je moguće eksplicitno inicijalizirati konstantnim izrazima. Neinicijalizirane statičke varijable prevoditelj inicijalizira na nulu (svi bitovi su jednaki 0).

```
int f(int j){
    static int i = j; /* greska */
     ...
}
```

Gornji kod nije ispravan jer statička varijabla nije inicijalizirana konstantnim izrazom.

Statička varijabla deklarirana unutar nekog bloka inicijalizira se **samo** pri prvom ulazu u blok, međutim zadržava svoju vijednost pri izlasku iz bloka. Statičke varijable deklarirane unutar bloka su dohvatljive samo unutar tog bloka.

```
void foo() {
     int a = 10; static int sa = 10;
     a += 5; sa += 5;
     printf("a = %d, sa = %d\n", a, sa);
return;
}

int main(void) {
    int i;

    for (i = 0; i < 3; ++i)
          foo();
return 0;
}
```

Izlaz je:

```
a = 15, sa = 15
a = 15, sa = 20
a = 15, sa = 25
```

U funkciji `foo`, varijabla *a* je automatska i inicijalizira se prilikom svakog ulaska u funkciju. *sa* je statička varijabla i inicijalizira se prilikom prvog ulaska u funkciju. *sa* zadržava vrijednost nakon izlaska iz funkcije iako nije dohvatljiva izvan funkcije `foo`.

2.2.4 Globalne varijable

Globalne varijable su varijable definirane izvan bilo koje funkcije (ili bloka). Uobičajeno se globalne varijable deklariraju na početku datoteke, prije svih funkcija. U tom slučaju svaka funkcija može koristiti takvu globalnu varijablu i promijeniti joj vrijednost. Više funkcija može komunicirati koristeći globalne varijable bez upotrebe formalnih argumenata. Globalne varijable se spremaju u statičkoj memoriji i njihova vrijednost se čuva do kraja izvođenja programa. Globalne varijable su implicitno eksterne. Kod globalnih varijabli razlikujemo **definiciju** varijable i **deklaraciju** varijable (slično i kod funkcija). U **definiciji** varijable deklarira se njezino ime i tip te se rezervira memorijska lokacija za varijablu. Kod **deklaracije** (navođenjem identifikatora memorijske klase `extern`[15]) se samo deklarira ime i tip bez rezervacije memorije. Za deklarirane varijable se podrazumijeva da je varijabla definirana negdje drugdje i da joj je tamo pridružena memorijska lokacija. Definicija varijable je uvijek i njezina deklaracija. Globalne varijable mogu imati više deklaracija, ali samo jednu definiciju.

```
1  int a;                          extern int a;
2  //definicija                    //deklaracija
3  int main(void){                 int main(void){
4
5    retun 0;                          return 0;
6  }                               }
```

- Globalne (vanjske) varijable dobivaju prostor u **statičkom** dijelu memorije programa kao i **statičke** varijable.
- Pravila inicijalizacije **su ista** kao i kod statičkih varijabli. Globalna varijabla može biti inicijalizirana konstantnim izrazom (kod definicije). Globalne varijable koje nisu eksplicitno inicijalizirane, inicijaliziraju se automatski nulom.
- Kod deklaracije globalne varijable mora se koristiti ključna riječ `extern`, a inicijalizacija nije moguća[16].

[15] Ključna riječ ima dodatnu ulogu kod programa čiji izvorni kod je zapisan u više datoteka.

[16] Ukoliko se program sastoji od samo jedne datoteke izvornog koda, inicijalizacija je moguća ali uzrokuje upozorenje od strane prevodioca.

- Kod definicije globalnog polja, mora biti definirana njegova dimenzija (zbog rezervacije memorije). Kod deklaracije se ne mora navoditi dimenzija.

Jedna od primjena globalnih varijabli je za brojanje kod rekurzivnih funkcija. Npr. brojanje particija možemo ostvariti upotrebom globalne varijable.

```
#include <stdio.h>
int broj = 0; /* globalni brojac */

void particije(int suma, int prvi){
int i;

if (suma == 0)
     ++broj;
else
     for (i = prvi; i <= suma; ++i)
           particije(suma - i, i);
return;
}

int main(void){

    int n;

    printf("␣Upisi␣prirodni␣broj␣n:␣");
    scanf("%d", &n);
    particije(n, 1);
    printf("\n␣Broj␣part.␣p(%d)␣=␣%d\n", n, broj);
    return 0;
}
```

Ovisno o položaju u programu, globalne varijable mogu imati različiti doseg.

```
int a; /* staticka memorija */

void f(int);

int main(void) {
    int c, d; /* auto, programski stog */
    f(); f(); f();
```

```
8
9       return 0;
10  }
11
12  int b = 0; /* staticka memorija */
13
14  void f(int i) {//auto, programski stog
15       int x, y; /* auto, programski stog */
16       printf("b␣=␣%d\n",++b);
17  }
```

Varijabla a vidljiva je i u funkciji `main` i u funkciji f, dok je varijabla b vidljiva u funkciji f, ali ne i u funkciji `main`. **Izlaz gornjeg programa je:**

```
b = 1
b = 2
b = 3
```

2.2.5 Izvorni kod smješten u više datoteka

C program može biti smješten u više datoteka. Npr. svaka funkcija definirana u programu može biti smještena u zasebnu `.c` datoteku. Globalne varijable i funkcije definirane u jednoj datoteci mogu se koristiti i u bilo kojoj drugoj datoteci (uz korektnu deklaraciju u drugoj datoteci).

Globalna varijabla ili funkcija koja je definirana u nekoj drugoj datoteci se deklarira u trenutnoj koristeći ključnu riječ `extern` (vanjski). `extern` ispred deklaracije objekta (varijable ili funkcije) informira prevoditelj da se radi o objektu koji je definiran u nekoj drugoj datoteci. Nakon deklaracije, vanjsku varijablu ili funkciju možemo koristiti unutar trenutne datoteke izvornog koda. Povezivanje deklaracija vanjskih simbola s njihovim definicijama radi `linker` (povezivač).

```
1  //Datoteka 1
2  #include <stdio.h>
3
4  int g(int);
5
6  void f(int i) {//definicija funkcije f
7       printf("i␣=␣%d\n", g(i));
8  }
```

```
9
10 int g(int i) {//definicija funkcije g
11     return 2 * i - 1;
12 }
```

```
1 //Datoteka 2
2 extern void f(int); /* deklaracija funkcije f */
3
4 int main(void) {
5     f(3);
6 return 0;
7 }
```

Nakon prevođenja i povezivanja obje datoteke, rezultat je `i = 5`.

Deklariranjem funkcije ili globalne varijable kao `static` **ograničimo doseg** funkcije ili varijable (ona se može dohvatiti samo iz datoteke u kojoj je definirana, postaje privatan simbol).

```
1 #include <stdio.h>
2 static int g(int); /* static ogranicava doseg. */
3
4 void f(int i) {
5     printf("i␣=␣%d\n", g(i));
6 }
7
8 int g(int i) {
9      return 2 * i - 1;
10 }
```

```
1 #include <stdio.h>
2
3 extern void f(int); /* deklaracija funkcije f */
4 extern int g(int); /* Nije vanjski simbol! */
5
6 int main(void) {
7 f(3); /* Ispravno. */
8 printf("g(2)␣=␣%d\n", g(2)); /* Neispravno. */
9 return 0;
10 }
```

Korištenje globalnih varijabli i funkcija, definiranih u jednoj datoteci izvornog koda, u drugoj datoteci izvornog koda možemo vidjeti u donjem programskom odsječku.

```
#include <stdio.h> //datoteka 1
int z = 3; /* Definicija varijable z. */
void f(int i) /* Definicija funkcije f. */
{
   printf("i␣=␣%d\n", i); }

//druga datoteka
extern void f(int); /* Deklaracija funkcije f.*/
extern int z; /* Deklaracija varijable z.*/

int main(void) {
   f(z);
return 0; }
```

Oznaka memorijske klase `static` može se primijeniti i na globalne varijable, s istim djelovanjem kao i za funkcije. `static` sužava doseg (područje vidljivosti) varijable na datoteku u kojoj je definirana. Ime takve varijable više nije dohvatljivo kao vanjski simbol (postaje privatna).

Oznaka memorijske klase `static` ispred globalne i lokalne varijable ima različito značenje:

```
static int z = 3; /*z nije vidljiv izvan datoteke.*/

void f(int i) {
      static double x;
      /* x je staticka varijabla. */
... }
```

Kad se program sastoji od više datoteka, grupe deklaracija vanjskih simbola (varijabli i funkcija) smještaju se u posebnu datoteku zaglavlja (`*.h`), koja se uključuje s `#include "*.h"` u svaku `*.c` datoteku u kojoj su te deklaracije potrebne. Na taj se način osigurava konzistentnost svih deklaracija. U datotekama zaglavlja `*.h` implicitno se sve deklaracije tretiraju kao `extern`. Podrazumijeva se da su svi objekti definirani negdje drugdje.

U donjem odsječku koda možemo vidjeti primjer deklaracije vanjskih simbola u datoteci zaglavlja `dekl.h`:

```c
extern void f(int); /* extern NE treba pisati! */
extern int g(int);
extern int z;
```

```c
//datoteka 1
#include <stdio.h>
#include "dekl.h"

int z = 3; /* Definicija varijable z. */
void f(int i) /* Definicija funkcije f. */
{
    printf("i = %d\n", g(i));
}

int g(int i) /* Definicija funkcije g. */
{
    return 2 * i - 1;
}
```

```c
//datoteka 2
#include <stdio.h>
#include "dekl.h"
int main(void)
{
   f(z);
   printf("g(2) = %d\n", g(2));
return 0;
}
```

Programe s više datoteka možemo iz komandne linije[17] prevesti naredbom `gcc prva.c druga.c -o imeIzvrsnog`. U okruženju `CodeBlocks`, programe s više datoteka prevodimo koristeći **projekte**[18].

Razdvajanje izvornog koda `C` programa u više datoteka nudi brojne prednosti. Osim očitih, organizacijskih gdje srodne funkcije i varijable možemo

[17] Naredbeni redak (eng. *Command Prompt*) u operacijskom sustavu Windows, terminal u operacijskim sustavima Unix i MacOS.

[18] `File` → `New` → `Project` → `Console application`. Nakon stvaranja, u projekt dodamo datoteke `dekl.h`, `prva.c` i `druga.c`, te pokrenemo prevođenje i pokretanje `Build` → `Build and run`

grupirati u za to određene datoteke, razdvajanjem koda u više izvornih kodova možemo stvarati cjeline dobro testiranih programskih rješenja. Ta rješenja možemo koristiti s raznim korisničkim programima ili kao sastavne djelove raznih aplikacija, što znatno ubrzava proces razvoja softvera. Razdvajanje programskog koda u više datoteka omogućava i stvaranje biblioteka (eng. *library*). Biblioteke omogućuju korištenje koda sadržanog u biblioteci bez ponovnog prevođenja odgovarajućeg izvornog koda iz kojeg je biblioteka stvorena. Kod velikih aplikacija, korištenje biblioteka može znatno ubrzati proces prevođenja i konstrukcije izvršne datoteke[19].

[19]Postoje dvije vrste biblioteka **statičke** i **dinamičke**. Kod aplikacija koje koriste statičke biblioteke, linker kopira odgovarajući prevedeni kod biblioteke u izvršnu datoteku programa. Korištenjem dinamičkih biblioteka izbjegavamo kopiranje koda u izvršnu datoteku, međutim biblioteka mora biti prisutna na specificiranoj putanji. Pri pokretanju programa se program i dinamička biblioteka učitaju u radnu memoriju te se tada povezuju.

Poglavlje 3

Složeni objekti i upravljanje radnom memorijom

U ovom poglavlju ćemo se upoznati sa složenijim objektima u C-u. Krećemo od višedimenzionalnih polja, prirodne generalizacije jednodimenzionalnih polja i stringova, polja znakova koji sadrže posebnu oznaku za kraj. Nakon toga, uvodimo nešto složeniji objekt rječnik koji omogućava efikasnije korištenje memorije i stvaranje tabularnih objekata nepravilnih dimenzija (npr. retci mogu imati različite brojeve elemenata). Dalje objašnjavamo postupak stvaranja proizvoljnih korisničkih tipova koji se mogu sastojati od elemenata raznih jednostavnih ili složenih tipova. Npr. naučit ćemo kako reprezentirati objekte kao što su automobil, student, osoba i slično. Konačno navodimo dva primjera kompleksnijih samoreferencirajućih struktura (stabla i vezanih lista) uz fokus na vezane liste. Analiziramo neke prednosti i mane vezanih lista. Za efikasan rad s navedenim složenim objektima, nužno je da se upoznamo s načinom upravljanja radnom memorijom, konkretno memorijom na programskoj hrpi. Upravljanje memorijom na programskoj hrpi će nam omogućiti stvaranje objekta točno željene veličine ili povećanje/smanjenje objekta tijekom izvođenja programa. Upravljanje memorijom na programskoj hrpi je nužno za korištenje rječnika i samoreferencirajućih struktura.

3.1 Višedimenzionalna polja

Jednodimenzionalno polje reprezentira niz podataka nekog tipa u programskom jeziku C. Do sada smo koristili nizove podataka jednostavnog,

standardnog tipa. Matematički analogon tih polja je vektor, uređeni niz skalara.

U programskom jeziku `C` možemo reprezentirati i kompliciranije objekte. Npr. možemo imati jednodimenzionalno polje čiji elementi su ponovo jednodimenzionalna polja određene fiksne duljine. Takvo polje možemo smatrati **dvodimenzionalnim** pošto nam indeks elementa u polju koje sadrži složene elemente predstavlja prvu dimenziju, a indeks elementa u polju koje čini jedan element polja koje se sastoji od složenih elemenata nam predstavlja drugu dimenziju. Matematički analogon takvog polja je matrica. Prema strukturi dvodimenzionalnog polja u `C`-u, matematički analogon bi bio da matricu promatramo kao vektor čiji elementi su vektori. Trodimenzionalno polje je jednodimenzionalno polje čiji elementi su dvodimenzionalna polja istog tipa. Matematički analogon je vektor čiji elementi su matrice. Analogno gradimo polja i za više dimenzije.

Primjer deklaracije višedimenzionalnog polja:

```
1 mem_klasa tip ime[izraz_1]...[izraz_n];
2
3 /*staticko dvodimenzionalno polje -
4 matrica s 2 retka i 3 stupca*/
5 static double m[2][3];
```

Elemente matrice `m` možemo prostorno prikazati:

```
m[0][0] m[0][1] m[0][2]
m[1][0] m[1][1] m[1][2]
```

Višedimenzionalno polje s deklaracijom:

```
1 mem_kl tip ime[izraz_1][izraz_2]...[izraz_n];
```

je jednodimenzionalno polje duljine `izraz_1`, a elementi tog polja su polja dimenzije $n-1$, oblika:

```
1 tip[izraz_2]...[izraz_n]
```

Isto vrijedi za svaku dimenziju slijeva nadesno. Operator [] ima asocijativnost $L \to D$.

Navedeni elementi se u memoriji računala pamte **jedan za drugim - kao jednodimenzionalno polje**. Kod spremanja kao jednodimenzionalno polje

najsporije se mijenja prvi indeks a najbrže zadnji indeks. Razlog tome je što je prvi indeks zapravo indeks složenog elementa polja ($n-1$ dimenzionalnog polja). Zadnji index označava element nekog osnovnog ili korisničkog tipa, stoga svaka promjena pozicije takvog elementa zahtjeva promjenu zadnjeg indeksa. n dimenzionalno polje sadrži $\prod_{k=1}^{n} n_k$ takvih elemenata.

Dvodimenzionalno polje `m[2][3]` iz prethodnog primjera se sastoji od dva retka `m[0]` i `m[1]` koja su tipa `double[3]`. Matrica `m` se u memoriju sprema na način da se slijedno zapišu svi elementi prvog retka, zatim svi elementi drugog retka:

```
m[0][0], m[0][1], m[0][2], m[1][0], m[1][1], m[1][2]
```

prvo elementi retka `m[0]`, zatim elementi retka `m[1]`.

Ukoliko u slučaju dvodimenzionalnog polja m iz primjera, indekse označimo s i i j, dvodimenzionalni indeks (i, j) možemo pretvoriti u jednodimenzionalni indeks (linearno indeksiranje) elementa računanjem `i*MAX_j+j`, gdje `MAX_j = 3` označava broj stupaca matrice iz deklaracije polja m. Dimenzija `MAX_i`, odnosno broj redaka matrice m nije potrebna za indeksiranje već samo za **rezervaciju memorije**. Primijetimo da linearno indeksiranje možemo napraviti na više načina. Konkretan način indeksiranja ovisi o načinu spremanja višedimenzionalnih polja u memoriju[1]. **Trodimenzionalno polje** `float MM[2][3][4]` je jednodimenzionalno polje s dva elementa `MM[0]` i `MM[1]`. Oba elementa su dvodimenzionalna polje tipa `float[3][4]`, odnosno matrice s 3 retka i 4 stupca. Element `MM[i][j][k]` smješten je na poziciju `(i*MAX_j+j)*MAX_k+k`, gdje `MAX_i = 2, MAX_j = 3, MAX_k = 4` su dimenzije polja. `MAX_i` je bitan jedino za rezervaciju memorije.

Elementima polja možemo pristupiti na dva načina:

- Navođenjem pripadnog indeksa za svaku dimenziju polja.
- Preko pokazivača, koristeći ekvivalenciju između polja i pokazivača

Ukoliko koristimo pristup elementima preko indeksa, svaki indeks mora biti u svojim uglatim zagradama []. Nije dozvoljeno odvajanje indeksa zarezom[2].

[1] U `Fortranu` se indeksiranje vrši po *stupcima*. Izvedite formulu za linearno indeksiranje dvodimenzionalnog polja u tom slučaju.

[2] Sintaksa ovisi o programskom jeziku, odvajanje indeksa zarezom je dozvoljeno npr. u `Fortranu`, `Pascalu` i Pythonu.

Dvodimenzionalno polje m iz prethodnog primjera možemo inicijalizirati na sljedeći način:

```
static double m[2][3] =
        {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};
```

Inicijalne vrijednosti pridružuju se elementima matrice redom kojim su elementi smješteni u memoriji (po recima):

`m[0][0] = 1.0, m[0][1] = 2.0, m[0][2] = 3.0,`
`m[1][0] = 4.0, m[1][1] = 5.0, m[1][2] = 6.0.`

Ovakav način inicijalizacije ćemo uglavnom upotrebljavati za inicijalizaciju polja manjih dimenzija.
Inicijalne vrijednosti se mogu grupirati **vitičastim zagradama**. Rezultantne grupe se pridružuju pojedinim recima.

```
static double m[2][3] = { {1.0, 2.0, 3.0},
                          {4.0, 5.0, 6.0}};
```

Ukoliko je neka grupa kraća od odgovarajuće dimenzije, preostali elementi se inicijaliziraju nulama (bez obzira na `static`). **Ukoliko postoji inicijalizacija, uvijek se inicijalizira cijelo polje**.

Prvu dimenziju polja (ako nije navedena) prevoditelj može izračunati iz inicijalizacijske liste:

```
char A[][2][2] = { {{'a','b'}, {'c','d'}},
                   {{'e','f'}, {'g','h'}}};
```

Kao rezultat dobijemo dvije matrice znakova, `A[0]` i `A[1]` tipa `char[2][2]`.

$$A[0] = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \; A[1] = \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

Pri definiciji funkcije koja kao argument prima višedimenzionalno polje, moramo navesti sve dimenzija polja osim prve[3].
Pošto dimenzije statičkih polja ne možemo mijenjati, a ne želimo mijenjati izvorni kod programa kod svakog novog pokretanja, često za potrebe programa definiramo polja fiksnih, velikih dimenzija, a korisnik ovisno o potrebama koristi cijelu ili samo dio dostupne memorije. Npr. funkciju koja čita

[3]Umjesto prve dimenzije možemo staviti prazne zagrade []. Npr. funkciju f koja prima trodimenzionalno polje dimenzija $5 \times 4 \times 10$ i vraća cjelobrojnu vrijednost možemo deklarirati kao `int f(int [][4][10])`.

matricu s m redaka i n stupaca ćemo definirati tako da kao argument prima matricu s **maksimalno** `MAX_m`$\geq m$ redaka i `MAX_n`$\geq n$ stupaca.

```
1 int mat[MAX_m][MAX_n];
2 ...
3 void readmat(int mat[MAX_m][MAX_n], int m, int n)
```

Argumenti m i n su **stvarni** broj redaka i stupaca matrice, koje **učitavamo**, memorija zbilja potrebna korisniku pri pokretanju programa.

Alternativne deklaracije.

```
1 void readmat(int mat[][MAX_n], int m, int n)
2 void readmat(int (*mat)[MAX_n], int m, int n)
```

U donjem primjeru je `mat` **pokazivač** na **redak**, **polje** od `MAX_n` elemenata tipa `int`. Matrica se reprezentira kao **polje redaka** matrice.

Postoji mogućnost reprezentacije i kao **niz nizova**. Međutim, taj generalniji način **ne prikazuje matricu**. Reci mogu biti različitih duljina. Također, reci nisu spremljeni u bloku već su razasuti po memoriji.

```
1 void readmat(int **mat, int m, int n)
```

`mat` je pokazivač na pokazivač na `int`. Odnosno, možemo proslijediti polje pokazivača na `int`.

Primjer 3.1.1

Računamo Euklidsku (Frobeniusovu) normu matrice.

Neka je A matrica tipa $m \times n$ s elementima $a_{i,j},\ i = 1, \ldots, m,\ j = 1, \ldots, n$. **Euklidska** ili **Frobeniusova** norma matrice A definira se kao:

$$||A||_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} a_{i,j}^2}$$

Ekvivalentan rezultat bi dobili računanjem Euklidske norme vektora koji sadrži sve elemente matrice.

Navodimo dvije varijante funkcije za računanje Frobeniusove norme matrice.

```
1 double E_normaR(double a[1000][1000], int m, int n) {
2 int i, j;
```

```
3 double suma = 0.0;
4 for (i = 0; i < m; ++i)
5    for (j = 0; j < n; ++j)
6        suma += a[i][j] * a[i][j];
7 return sqrt(suma);
8 }
```

Matricu reprezentiramo dvodimenzionalnim poljem maksimalnih dimenzija 1000×1000[4]. Prave dimenzije matrice zadaje korisnik definiranjem vrijednosti argumenata m i n. Povratnu vrijednost gornje funkcije možemo ispisati u jednoj liniji koda:

```
1 printf("%g\n", E_norma(a, 3, 5));
```

Isti rezultat možemo dobiti i tako da u funkciji okrenemo poredak petlji:

```
1 double E_normaC(double a[1000][1000], int m, int n) {
2 int i, j;
3 double suma = 0.0;
4 for (j = 0; j < n; ++j)
5     for (i = 0; i < m; ++i)
6         suma += a[i][j] * a[i][j];
7 return sqrt(suma);
8 }
```

Ovdje je bitno uočiti da "okretanje" (zamjena poretka) petlji ima jedino smisla ukoliko je poredak indeksa pri pristupu elementima matrice nepromijenjen. Ukoliko promijenimo poredak petlji i poredak indeksa pri pristupu elementima matrice, tada dobijemo kod ekvivalentan prvom primjeru uz supstituciju $i \leftrightarrow j$.

Zanima nas koja od dvije prethodno definirane funkcije je brža?
Brži je prvi kod u kojem iteriramo sljedno po dvodimenzionalnom polju. Prisjetimo se da se višedimenzionalna polja spremaju u memoriju kao jedno veliko jednodimenzionalno polje čiji elementi su složeni (polja manjih dimenzija). Poredak spremanja u memoriju, na primjeru dvodimenzionalnog polja

[4]Pošto se radi o statičkom polju, ono se sprema na programski stog. Programski stog standardno ima relativno malu memoriju 1-8MiB, stoga da bi koristili polje dimenzije 1000×1000 elemenata tipa `double` trebamo povećati veličinu sistemskog stoga. To u okruženju `CodeBlocks` možemo napraviti tako da u `Settings`→`Compiler`→`Linker Settings` → `Other linker options:` upišemo tekst: `-Wl,-stack=16777216`. To će povečati veličinu programskog stoga na 16MiB.

je da su elementi koji pripadaju redcima matrice spremljeni na uzastopne memorijske lokacije. Zbog optimizacija procesora, koje omogućavaju učitavanje većeg broja susjednih elemenata u `cache` memoriju procesora, puno je bolje iterirati po redcima matrice. Kod takvog iteriranja, procesor će imati u brzoj `cache` memoriji nekoliko uzastopnih elemenata koji će mu biti potrebni za računanje norme. U drugom kodu, koji elementima matrice pristupa po stupcima, procesor učita puno elemenata koje neće koristiti, stoga njih treba eliminirati iz `cache` memorije da bi mogao učitati sljedeći element u stupcu, koji je u memoriju spremljen na većoj prostornoj udaljenosti od prethodno učitanog elementa (između njih su u memoriji spremljeni svi elementi retka u kojem se nalazi prethodno učitani element). Dodatna prednost računanja s elementima koji su sljedno spremljeni u memoriju, kod modernih procesora, je što mogu koristiti vektorizirane operacije. Učinkovite instrukcije koje mogu izvršiti operaciju nad nizom elemenata.

Na računalu s `Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz` je sekvencijalnim izračunavanjem, izračunata Frobeniusova norma matrice A, dimenzija 1000×1000 sa slučajno generiranim realnim elementima. Računanje norme je ponovljeno 10000 puta da bi mogli dovoljno precizno izmjeriti vrijeme izvršavanja. Vrijeme izvršavanja koristeći funkciju `E_normaR` iznosi $21s$, dok isto računanje koristeći funkciju `E_normaC` iznosi $29s$. Računanje norme matrice A, s realnim elementima, dimenzija 1000×1000 u prosjeku traje $2.1ms$ koristeći funkciju `E_normaR`, a $2.9ms$ koristeći funkciju `E_normaC`.

Primjer 3.1.2

Zadana je pravokutna matrica A tipa $m \times n$, i vektor x duljine n. Treba izračunati vektor $y = Ax$.

Primijetimo da će vektor y biti duljine m. Formula za izračun elemenata vektora y je:

$$y_i = \sum_{j=1}^{n} a_{i,j} \cdot x_j, \ \forall i = 1, \ldots, m$$

Za računanje moramo koristiti dvije petlje, jednu za iteriranje po indeksu i i jednu za iteriranje po indeksu j. Treba imati na umu da petlje u C-u počinju od nule. `MAX_m` i `MAX_n` su konstante koje definiramo koristeći `#define`.

```
1 #include <stdio.h>
2 #define MAX_m 10
```

```
3  #define MAX_n 10
4  int main(void) {//dio glavnog programa
5     int A[MAX_m][MAX_n], x[MAX_n], y[MAX_m];
6     int m, n; /* Stvarne dimenzije matrice A. */
7     //deklaracija funkcije umnozak
8     void umnozak(int, int, int mat1[][MAX_n],
9                  int mat2[MAX_n], int mat3[MAX_m]);
10 ...
11 umnozak(m, n, A, x, y);
12 ... }
13
14 void umnozak(int m, int n, int mat1[][MAX_n],
15 int mat2[MAX_n], int mat3[MAX_m])
16 {
17 int i, j;
18 /* Mnozenje matrice i vektora. */
19 for (i = 0; i < m; ++i) {
20       mat3[i] = 0;
21       for (j = 0; j < n; ++j)
22          mat3[i] += mat1[i][j] * mat2[j];
23      }
24 return;
25 }
```

Primjer 3.1.3

Zadane su 3 pravokutne matrice, A dimenzija $m \times l$, B dimenzija $l \times n$ i C dimenzija $m \times n$.

Želimo izračunati izraz:

$$C = C + A \cdot B$$

Navedena operacija (*nazbrajavanje*) produkta $A \cdot B$ matrici C je standardni oblik `BLAS-3` funkcije `xGEMM` za množenje matrica. Operacija se često koristi u praksi. Matematički, operaciju po elementima realiziramo:

$$c_{i,j} = c_{i,j} + \sum_{k=1}^{l} a_{i,k} \cdot b_{k,j}, \ \forall i = 1, \ldots, m, \ j = 1, \ldots n$$

Potrebne su nam **tri** petlje, uz uvažavanje činjenice da u C-u indeksi kreću od nule.

```
1  void matmul(int m, int n, int l, double A[][lda],
2  double B[][ldb], double C[][ldc] ) {
3  int i, j, k;
4
5  for (i = 0; i < m; ++i)
6     for (j = 0; j < n; ++j)
7         for (k = 0; k < l; ++k)
8              C[i][j] += A[i][k] * B[k][j];
9
10 return;
11 }
```

Pošto se množenje matrica vrši u trostrukoj petlji, imamo $3! = 6$ verzija algoritma. Računala imaju hijerarhijski organiziranu memoriju, u kojoj se bliske memorijske lokacije mogu dohvatiti brže od udaljenih (blok-transfer u *cache* memoriju). U prethodnom kodu, u unutarnjoj petlji (po k), računa se skalarni umnožak i-tog retka matrice A i j-tog stupca matrice B. Elementi retka od A su na susjednim lokacijama pa je dohvat brz. Međutim, elementi stupca od B se nalaze na memorijskim lokacijama udaljenim za duljinu retka (`ldb`). Kod velikih matrica ta udaljenost može biti velika što dovodi do sporijeg izvršavanja.

Efikasnija verzija algoritma koristi petlju po j kao unutarnju.

```
1  ...
2  for (i = 0; i < m; ++i)
3       for (k = 0; k < l; ++k)
4            for (j = 0; j < n; ++j)
5                 C[i][j] += A[i][k] * B[k][j];
6  ...
```

U unutarnjoj petlji (po j), dohvaćaju se reci matrica C i B, a nema dohvata stupaca. Element `A[i][k]` može se čuvati u cache memoriji. Elemente od A isto dohvaćamo po recima.

Ovo je najbrža od svih 6 varijanti prezentiranog algoritma pri množenju velikih matrica.

Ukoliko želimo računati samo umnožak $C = A \cdot B$, inicijaliziramo $C = 0$.

```
1  for (i = 0; i < m; ++i)
2       for (j = 0; j < n; ++j)
```

```
3           C[i][j] = 0.0;
4
5  for (i = 0; i < m; ++i)
6       for (k = 0; k < l; ++k)
7           for (j = 0; j < n; ++j)
8                 C[i][j] += A[i][k] * B[k][j];
```

Problem u C90 standardu je što dimenzije polja u deklaraciji argumenata funkcije moraju biti konstantni izrazi. Promjena dimenzije višedimenzionalnog polja čini nužnom promjenu deklaracije svih funkcija.

C99 uvodi polja *varijabilne duljine*. Polje varijabilne duljine je **automatsko polje** čije dimenzije mogu biti zadane vrijednostima varijabli.

```
1  int m = 3;
2  int n = 3;
3  double a[m][n]; /* polje varijabilne duljine. */
```

Korištenjem polja varijabilne duljine možemo napisati generalnije funkcije. **Treba paziti da dimenzije, osim prve predstavljaju dimenziju polja iz definicije**. U suprotnom će se memorija polja, unutar funkcije, interpretirati na krivi način.

```
1  double A[m][dim_n]; //deklaracija - main
2
3  //funkcija
4  double E_norma(int m, int n, double A[m][n]){
5  double suma = 0.0;
6  int i, j;
7  //A je niz redaka duljine n, ukoliko n!=dim_n
8  //pristupamo memoriji na krivi nacin
9  for (i = 0; i < m; ++i)
10      for (j = 0; j < n; ++j)
11          suma += A[i][j] * A[i][j];
12 return sqrt(suma); }
```

U gornjem primjeru, ukoliko $n \neq dim\_n$, prevodioc smatra da redci matrice A imaju maksimalno n elemenata. Ukoliko $n < dim\_n$ prevodioc će za element $A[1][0]$ zapravo vratiti $n+1$-prvi element prvog redka matrice A. Ukoliko $n > dim\_n$, interpretiramo neki element drugog redka kao dio prvog.

Ukoliko želimo koristiti funkciju nad poljem fiksne duljine, ali koristiti samo dio alocirane memorije za računanje unutar funkcije, dodajemo još jedan parametar (stvarni broj stupaca iz definicije).

```
1 double E_norma(int m, int n, int lda,
2                          double A[m][lda])
```

Tijelo funkcije je kao u prethodnom primjeru. Funkcija sada interpretira memoriju na pravi način. Argument `lda` se u listi argumenata funkcije mora nalaziti ispred `double A[m][lda]`.

Zadatak: preuredite funkciju `matmul` za **množenje matrica** tako da koristi polja **varijabilne** duljine.

3.1.1 Svojstva pokazivača i njihova povezanost s višedimenzionalnim poljima

Pokazivač na `tip` je varijabla koja sadrži adresu varijable tipa `tip`. Pokazivač se deklarira kao:

```
1 mem_klasa tip *p_var;
```

Tip pokazivača je vezan uz tip objekta zapisanog na memorijsku lokaciju na koju pokazivač pokazuje. Vrijedi za sve pokazivače osim **generičke pokazivače** `void *p` koji pokazuju na bilo što. Znak * uvijek djeluje na prvi sljedeći simbol.

```
1 static int *pi;           double *px;
2 char* pc;                 int a, *b;
3 float* pf, f;             void *p;
```

U gornjem primjeru, pf je tipa `float *`, a f tipa `float`.

Adresni operator & se koristi za dohvaćanje adrese varijable. &x vraća adresu varijable x. Operator dereferenciranja * dohvaća sadržaj na zadanoj adresi. *p dohvaća vrijednost sadržanu na memorijskoj lokaciji na koju pokazuje p.

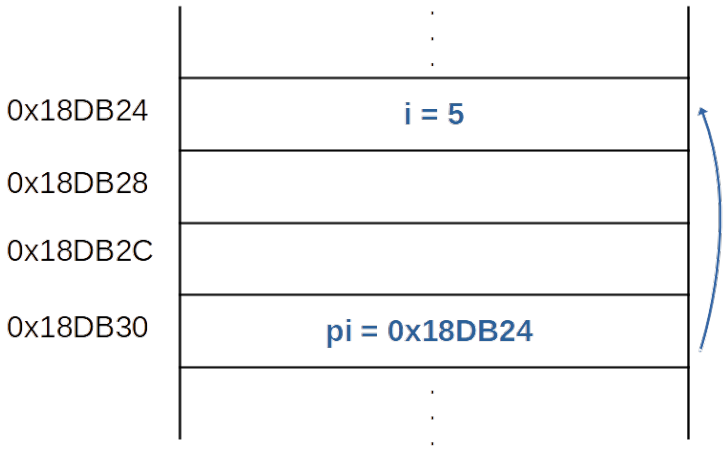
Operatori & i * su unarni operatori, asocijativnosti $D \rightarrow L$ ukoliko su zapisani ispred operanda. Ukoliko su zapisani između dva operanda, radi se o binarnim operatorima bit-po-bit i, te množenju.

Varijablu tipa pokazivač možemo inicijalizirati pri definiciji adresom neke druge varijable. Ta druga varijabla mora biti definirana prije no što se na

nju primijeni adresni operator (mora imati adresu).

```
1 int i = 5;
2 int *pi = &i; /* Inicijalizacija adresom */
3 i = 2 * (*pi + 6); /* Indirektni pristup */
4 printf("i = %d, adresa od i = %p\n", i, pi);
```

U gornjem primjeru je `&i` **adresa** varijable i, a `*pi` **vrijednost** spremljena u memorijsku lokaciju na koju pokazuje pi.



Pokazivači mogu biti argumenti funkcije. U tom slučaju, funkcija može promijeniti vrijednost varijable na koju pokazivač pokazuje. Takve argumente zovemo **varijabilnim argumentima**.

```
1 void zamjena(int *px, int *py) {
2 int temp = *px;
3 *px = *py;
4 *py = temp; }
```

Poziv funkcije treba biti:

```
1 zamjena(&a, &b); /* Moramo proslijediti adrese! */
```

Nad pokazivačima je definiran ograničen skup operacija koje imaju smisla pri radu s memorijskim adresama. Množenje pokazivača nema smisla i **nije dozvoljeno**, iako su pokazivači vrsta cijelih brojeva bez predznaka.

Dozvoljene operacije nad pokazivačima su:

- Dodavanje ili oduzimanje cijelog broja pokazivaču.

- Oduzimanje pokazivača istog tipa (jedina dozvoljena aritmetička operacija za dva pokazivača).
- Uspoređivanje pokazivača istog tipa relacijskim operatorima.

Ime polja je **konstantni pokazivač** na prvi element polja. Sve aritmetičke operacije nad pokazivačima **ekvivalentne su** aritmetici indeksa u polju odgovarajućeg tipa, a ne stvarnoj aritmetici adresa.

Za polje a, `a = &a[0]` ili `*a = a[0]`. Također, `a + i = &a[i], *(a + i) = a[i]`, $\forall$ i.

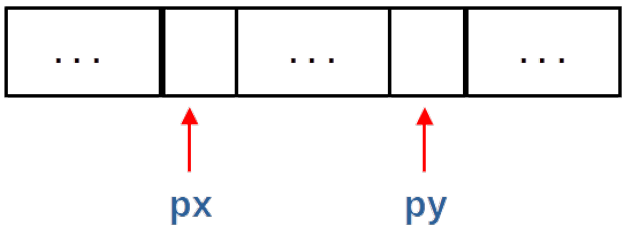
Stvarne adrese ovise o **tipu** (veličini elemenata u polju).
`a+i` $\Leftrightarrow$ `adresa jednka a+i*sizeof(tip elemenata u polju a)`. Prevodioc automatski računa stvarne adrese iz aritmetičkih operacija nad pokazivačima.

Svaki pokazivač na neki objet možemo interpretirati i kao pokazivač na **prvi element** u polju objekata tog tipa. Svakom pokazivaču možemo **dodati i oduzeti** cijeli broj. Ako je p pokazivač (osim generičkog) i n varijabla cjelobrojnog tipa, dozvoljene su operacije:

```
1 p + n   p - n   ++p   --p   p++   p--
```

Pokazivač `p+n` pokazuje na n-ti objekt nakon onog na kojeg pokazuje p, odnosno vrijedi: `p+n` $\Leftrightarrow$ `adresa jednaka p + n*sizeof(tip objekta na koji pokazuje p)`.

Pokazivače istog tipa smijemo oduzimati. Operacija ima smisla ako oba pokazivača pokazuju na **isto polje**. Za dva pokazivača `px` i `py`, koji pokazuju na isto polje), `py-px+1` je broj elemenata između `px` i `py` (uključujući rubove).



Razlika pokazivača je vrijednost cjelobrojnog tipa (tipa `ptrdiff_t`) definiranog u `<stddef.h>`.

Pokazivače istog tipa smijemo međusobno **uspoređivati** relacijskim operatorima. Nad dva pokazivača `px` i `py` istog tipa možemo koristiti izraze:

```
px <= py   px >= py   px < py   px > py
px == py   px != py
```

Uspoređivanje ima smisla samo ako oba pokazivača pokazuju na **isto** polje[5].

Pokazivaču nije moguće pridružiti vrijednost cjelobrojnog tipa, osim nule. Nula nije legalna adresa već označava da pokazivač nije inicijaliziran, ne pokazuje na ništa (`double *p = 0`). Bolje je naglasiti da se radi o pokazivaču i koristiti simboličku konstantu `NULL` definiranu u zaglavlju `<stdio.h>` (`double *p = NULL`).

```
double *px;
if (px != 0) ... /* Dozvoljeno */
if (px != NULL) ... /* Standardno */
if (px == 0x3451) ... /* Greska, usporedjivanje
s cijelim brojem. */
```

Pokazivači na različite tipove podataka općenito se ne mogu međusobno pridruživati (osim pridruživanja pokazivača proizvoljnog tipa generičkom pokazivaču). Pridruživanje pokazivača različitih tipova se može izvršiti jedino korištenjem eksplicitnog pretvaranja tipa (cast operator).

```
char *pc;
int *pi;
pi = pc; /* Greska */
pi = (int *) pc; /* Ispravno */
```

Generički pokazivač deklarira se kao pokazivač na `void`, `void *p;` Vrijednost pokazivača na bilo koji tip može se dodijeliti pokazivaču na `void` i obratno, bez promjene tipa pokazivača (ne treba cast).

```
double *pd0, *pd1;
void *p;
...
p = pd0; /* Ispravno */
pd1 = p; /* Ispravno */
```

[5] Usporedba pokazivača koji pokazuju na različita polja nam samo može reći koje polje je spremljno na memorijske lokacije s većim adresama. Ta informacija uglavnom nije korisna, pošto se pri svakom pokretanju programa objektima pridružuje memorija na različitim adresama. Može se dogoditi da polje koje je u prethodnom izvršavanju imalo veće adrese u sljedećem ima manje.

Osnovna uloga generičkog pokazivača je omogućavanje definiranja funkcija koje mogu primati pokazivač na proizvoljni tip podataka.

```
1 double *pd0;
2 void f(void *);
3 ...
4 f(pd0); /* OK. */
```

Generički pokazivač se **ne smije** dereferencirati, povećati ili smanjiti zato što te operacije ovise o tipu pokazivača.

U `<stdlib.h>` postoje funkcije `qsort` i `bsearch` za općenito sortiranje niza podataka i binarno traženje koje sadrže generičke pokazivače da omoguće sortiranje, odnosno pretraživanje polja koja sadrže elemente proizvoljnog tipa. Više detalja možete vidjeti u Odjeljku 2.1.1.

Modifikator (ključnu riječ) `const` koristimo za definiciju konstanti. Npr. `const double g = 9.81;` je deklaracija konstante za ubrzanje gravitacije. Nakon gornje deklaracije varijabli g ne smijemo promijeniti vrijednost. Modifikator `const` smijemo primijeniti i na pokazivače. Trebamo razlikovati konstantnost pokazivača od konstantnosti sadržaja (memorijske lokacije na koju pokazuje). **Konstantni pokazivač** uvijek pokazuje na istu lokaciju. Moguće je definirati obični i konstantni pokazivač na nekonstantni ili konstantni tip.

```
1  double x[] = {0.1, 0.2, 0.3}; //polje realnih brojeva
2  const double y[] = {0.1, 0.2, 0.3};
3  //konstantno polje realnih brojeva
4  const double *p1; // Pokazivac na konstantan double
5  double * const p2 = x;
6  // Konstantan pokazivac na double
7  const double * const p3=y;
8  //Konstantan pokazivac na konstantan double
9  p1 = x; // ispravno, ali x ne mozemo mijenjati kroz p1
10 p1[1] = 4.0; // Greska, x ne mozemo mijenjati kroz p1
11 p2 = &x[2]; // Greska, ne mozemo mijenjati p2
12 p3 = &y[2]; /* Greska, ne mozemo mijenjati p3 */
13 *p3 = 4.0; /* Greska, y ne mozemo mijenjati kroz p3 */
```

Pokazivač p na proizvoljni tip, koji nije generički, može se interpretirati kao pokazivač na prvi element u polju odgovarajućeg tipa (`p = &p[0]`). Za pokazivač p smijemo koristiti i aritmetiku pokazivača i indekse (mogu se i miješati).

Veza između aritmetike pokazivača i indeksiranja je `p+i = &p[i], *(p+i) = p[i]`, gdje cijeli broj i može biti i negativan. Pokazivaču p koji nije konstantan smijemo mijenjati vrijednost.

```
1 char *px, x[128];
2 px = &x[0]; // Isto kao px = x;
3 *(px + 3) = 'z'; // Isto kao px[3] = 'z';
4 //ekvivalentno s x[3] = 'z'
5 ++x; // Greska, x je konstantan pokazivac
6 ++px; // Isto kao px = &x[1];
7 *(px + 1) = 'h'; // Isto kao px[1] = 'h';
8 //ekvivalentno s x[2] = 'h';
9 *(px + 130)='d';//Izvan polja, ne javlja gresku
```

U gornjem primjeru u pokazivač px spremimo adresu prvog elementa polja x. To možemo napraviti ili dohvaćanjem adrese prvog elementa (`&x[0]`) ili kopiranjem vrijednosti konstantnog pokazivača x (`y = x`). Polje x nakon pridruživanja vrijednosti pokazivaču px, možemo mijenjati i preko pokazivača. `*(px + 3)` se pomiče za 3 pozicije od pozicije na koju pokazuje px (početak polja), dohvaća vrijednost memorijske lokacije i mijenja ju u slovo z. U liniji 6 gornjeg primjera u px spremamo poziciju drugog elementa u polju x. Operator indeksiranja, koji možemo primjenjivati i na pokazivače, uvijek dohvaća indeks u odnosu na pokazivač s kojim je pozvan. Zato je u liniji 7 gornjeg primjera `*(px+1)` $\leftrightarrow$ `px[1]` $\leftrightarrow$ `x[2]`.

Generalno vrijede sljedeće ekvivalencije:

- Za jednodimenzionalno polje `double x[10];` vrijedi `x[i]` $\Leftrightarrow$ `*(x+i)`.
- Za višedimenzionalna polja, specifično dvodimenzionalno polje `double x[10][20];`, vrijedi `x[i][j]` $\Leftrightarrow$ `*(x[i]+j)` $\Leftrightarrow$ `*(*(x+i)+j)`.

```
1 #include <stdio.h>
2 int main(void) {
3 int a[2][3] = {{1,2,3}, {4,5,6}}, *pa;
4 pa = a[0]; /* Ekvivalentno s pa = (int *) a; */
5 pa = pa + 3; /* Pomak za 3 int-a desno u polju. */
6 printf("%d\n", *pa); /* 4 */
7 printf("%d\n", **(a + 1)); /* 4 */
8 printf("%d\n", *(a[1] + 1)); /* 5 */
9 return 0; }
```

U liniji 4 gornjeg primjera, pokazivaču *pa* pridružujemo adresu prvog retka dvodimenzionalnog polja *a* (`pa = a[0]`). Ekvivalntno možemo postići i pretvorbom pokazivača *a* iz tipa `int (*)[3]` u tip `int*` i pridruživanjem adrese prvog elementa polja *a* pokazivaču *pa*. Inkrementiranje pokazivača *pa* pomiče *pa* na sljedeći element u polju u redoslijedu kakvim su spremljeni u memoriji (odgovara iteriranju po redcima matrice *a*). Stoga se u liniji 6 ispiše vrijednost prvog elementa drugog redka. U liniji 7 izraz `a+1` vraća pokazivač na drugi redak matrice *a*. Izraz *(`a+1`) dohvaća adresu drugog redka, a izraz **(`a+1`) dohvaća vrijednost elementa na adresi koja je na početku drugog redka (element 4). U liniji 8, `a[1]` dohvaća adresu drugog retka, izraz `a[1]+1` vraća adresu drugog elementa u drugom redku, a izraz `*(a[1]+1)` vraća vrijednost drugog elementa u drugom redku (element 5).

Polje smije biti argument funkcije. Pri prijenosu kao argument funkcije, polje se ne kopira već samo pokazivač na prvi element polja. Kod poziva funkcije smijemo navesti polje (bez zagrada) ili pokazivač na bilo koji element polja (objekt odgovarajućeg tipa). Unutar funkcije elementi se mogu dohvatiti/mijenjati korištenjem indeksa ili aritmetike pokazivača.

Kod deklaracije jednodimenzionalnog polja kao formalnog argumenta funkcije, može se koristiti:

```
1 tip_pod ime[izraz_1]
2 tip_pod ime[]
3 tip_pod *ime
4 tip_pod (*ime)
```

Okrugle zagrade su nužne kod višedimenzionalnih polja. Kod deklaracije višedimenzionalnog polja kao formalnog argumenta funkcije, može se koristiti:

```
1 tip_pod ime[izraz_1][izraz_2]...[izraz_n]
2 tip_pod ime[][izraz_2]...[izraz_n]
3 tip_pod (*ime)[izraz_2]...[izraz_n]
```

Ako ne želimo dozvoliti mijenjanje polja unutar funkcije, ispred tipa dodamo ključnu riječ `const`.

3.2 Dinamička alokacija memorije

Do sada smo objekte stvarali ili na programskom stogu (**automatski objekti**) ili u statičkom dijelu memorije (**statički objekti**). U ovom poglavlju ćemo naučiti kako stvarati objekte na hrpi (**dinamički objekti**). Za razliku od programskog stoga čiju veličinu mjerimo u MB (ili MiB). Veličina hrpe (eng. *runtime heap*) je uglavnom određena količinom radne memorije u računalu, mjeri se u GB. Glavne prednosti korištenja hrpe su: a) mogućnost dinamičke rezervacije memorije (alokacije) za objekte proizvoljne veličine, ograničene dostupnom količinom radne memorije, b) mogućnost izmjene veličine objekta po potrebi tijekom izvođenja programa, c) objekt možemo uništiti na proizvoljnom mjestu u programu i osloboditi zauzetu memoriju za ponovno korištenje.

Glavna primjena **dinamičke alokacije** je stvaranje polja čiju dimenziju ne znamo prije pokretanja programa, npr. ovisi o korisničkom ulazu ili o međurezultatu izvršavanja programa, te stvaranje dinamičkih struktura podataka (npr. vezane liste, stabla).

Funkcije za alokaciju i dealokaciju memorije deklarirane su u datoteci zaglavlja `<stdlib.h>` (standardna biblioteka).

- alokacija: funkcije `malloc`, `calloc`, `realloc`.
- dealokacija: funkcija `free`.

Funkcija `malloc` kao jedini argument prima varijablu tipa `size_t`. `size_t` je cjelobrojni tip bez predznaka (za spremanje veličina objekata) definiran u `<stddef.h>`, a n je jednak **ukupnom broju bajtova** koji treba alocirati.

```
1 void *malloc(size_t n);
```

Funkcija `malloc` rezervira blok memorije od n bajtova. Vraća pokazivač na rezervirani blok memorije, ili `NULL`, ako se zahtjev ne može ispuniti. Vraćeni pokazivač je generički, tipa `void*`. Prije upotrebe ga treba eksplicitno konvertirati u potrebni tip pokazivača (cast operatorom).

Funkcija `calloc` rezervira blok memorije za spremanje `nobj` objekata, od kojih svaki pojedini objek ima veličinu `size`, tj. ukupan broj rezerviranih bajtova je `nobj *size`. Dodatno, inicijalizira cijeli rezervirani prostor na nule, preciznije na nul-znakove `'\0'`. Kao i `malloc`, vraća pokazivač na rezervirani blok ili `NULL`.

```
void *calloc(size_t nobj, size_t size);
```

Alokaciju memorije za 150 elemenata tipa `double` koristeći funkciju `malloc` možemo napraviti kao u donjem odsječku koda.

```
double *p;
...
p = (double *) malloc(150 * sizeof(double));
if (p == NULL) {
     printf("Greska: alokacija nije uspjela!\n");
     exit(EXIT_FAILURE); /* exit(1); */
}
```

Možemo koristiti i `calloc` koji inicijalizira vrijednosti na nulu.

```
p = (double *) calloc(128, sizeof(double));
```

Kod dinamičke alokacije memorije uvijek treba provjeriti je li alokacija uspjela: `if (p == NULL) ...`. U slučaju neuspjele alokacije treba **prekinuti** izvršavanje programa. Program prekidamo korištenjem funkcije `exit` deklarirane u biblioteci `<stdlib.h>`. Poziv `exit(status);` zaustavlja izvršavanje programa (bez obrzira je li pozvan unutar funkcije `main` ili neke druge funkcije) i vrijednost `status` predaje operacijskom sustavu. Radi isto što i `return status;` u funkciji `int main`. `status` $\neq 0$ signalizira grešku.

```
void exit(int status);
```

Alociranu memoriju, nakon upotrebe, možemo osloboditi funkcijom `free`.

```
void free(void *p);
```

Funkcija `free` uzima pokazivač p na početak alociranog bloka memorije i oslobađa taj blok memorije. Ako je `p == NULL`, onda ne radi ništa. Funkcija `free` **ne mijenja** pokazivač p. Nakon poziva `free(p);` pokazivač p i dalje pokazuje na isti (oslobođeni) dio memorije i taj dio (tj. *p) se ne smije koristiti. Dobra praksa je iza poziva `free(p);` staviti `p = NULL;`.

Primjer 3.2.1

Program učitava broj elemenata polja n, dinamički kreira polje a brojeva tipa `int` i ispisuje zbroj svih elemenata u polju.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void){

int *a; /* Pokazivac na dinamicko polje. */
int i, n, zbroj;
printf("Upisi broj elemenata polja a:");
scanf("%d", &n);

if ((a = (int*) calloc(n, sizeof(int)))== NULL) {
        printf("Alokacija nije uspjela.\n");
        exit(EXIT_FAILURE);} /* exit(1); */

for (i = 0; i < n; ++i) {
    printf("Upisi element polja a[%d]: ", i);
    scanf("%d", &a[i]); }

zbroj = 0;

for (i = 0; i < n; ++i)
      zbroj = zbroj + a[i];

printf("Zbroj svih elemenata = %d\n", zbroj);
free(a); /* Ne treba a = NULL; kraj programa. */
return 0; }
```

Treća mogućnost za dinamičku alokaciju memorije je funkcija `realloc`. Služi za promjenu veličine već alociranog bloka.

```c
void *realloc(void *p, size_t size);
```

Funkcija `realloc` mijenja veličinu objekta na kojeg pokazuje p na zadanu veličinu `size` (tj. realocira memoriju). Sadržaj objekta (`*p`) ostaje isti do minimuma stare i nove veličine (kopira se po potrebi). Ako je nova veličina objekta veća od stare, dodatni prostor se ne inicijalizira. Vraća pokazivač na novorezervirani prostor, ili `NULL`, ako zahtjev nije ispunjen (tada `*p` ostaje nepromijenjen). `realloc` uglavnom koristimo za produljivanje dinamičkih objekata (polja) nepoznate duljine. Npr. učitavamo rečenicu proizvoljne

duljine bez ograničenja na maksimalnu duljinu riječi, spremamo elemente učitane od strane korisnika u polje bez ograničenja na maksimalan mogući broj ulaznih elemenata.

Dinamičku alokaciju ćemo upotrebljavati u dva česta scenarija:

- Duljinu niza (broj n) učitamo i rezerviramo memoriju za svih n elemenata niza.
- Broj članova ne znamo unaprijed, čitamo nepoznati broj članova element po element do oznake za kraj niza. Pri dodavanju novih članova niza produljujemo postojeći niz za određeni broj članova, počevši od praznog niza[6].

Isti princip možemo iskoristiti kod kreiranja rječnika (ili niza stringova). Više detalja možete vidjeti u Odjeljku 3.4.

```
1 int *p = NULL, i;
2
3 p = (int*) realloc(p,10*sizeof(int));
4
5 for(i=0;i<10;i++)
6     p[i] = 2*i;
```

U gornjem odsječku koda alociramo memoriju za 10 elemenata tipa `int` i na poziciju i zapisujemo vrijednost $2 \cdot i$. Ukoliko kreiramo novo polje koristeći `realloc` moramo obavezno inicijalizirati odgovarajući pokazivač na `NULL`. To je u gornjem primjeru ostvareno naredbom `int *p = NULL;`.

```
1  int *p = NULL, n, i, br;
2  scanf("%d",&n);
3
4  if(n<0){ exit(0); }
5  p = (int*) realloc(p,sizeof(int));
6  br = 1; p[0] = n;
7
```

[6]Najjednostavniji način proširenja niza je proširenje po jednu poziciju pri učitavanju svakog novog elementa. Međutim, takav način proširenja zahtjeva konstanto proširivanje memorije i potencijalno kopiranje elemenata kraćeg polja na memorijske lokacije dodijeljene proširenom polju. Puno bolja strategija je proširiti polje za veći broj pozicija, te raditi naknadna proširenja tek kada se slobodna memorija popuni. Jedna od strategija je proširiti polje duljine n na veličinu $k \cdot n$, gdje je $k > 1$ faktor rasta. Često se koristi $k = 2$.

```
8   while(n>=0){
9       scanf("%d",&n);
10      if(n<0){
11          for(i=0;i<br;i++)
12            printf("%d␣",p[i]);
13          printf("\n");
14          free(p); exit(0); }
15      p = (int*) realloc(p,++br*sizeof(int));
16      p[br-1] = n; }
```

U gornjem odsječku koda učitamo jedan cijeli broj, dinamički kreiramo polje duljine 1 i spremimo učitani element u polje. U nastavku iterativno učitavamo elemente dok korisnik ne učita negativan broj, proširimo polje za jednu poziciju i spremimo novo učitani element u polje. Nakon učitavanja negativnog broja, ispišemo sve elemente učitanog polja, oslobodimo memoriju alociranu za spremanje polja i prekinemo izvođenje programa.

3.3 Stringovi

Kod rada s vektorima (poljima), matricama (višedimenzionalnim poljima), čak i kada u programu koristimo više objekta, uglavnom dimenzije ne variraju. Međutim, kod obrade i reprezentacije teksta imamo riječi i rečenice čije veličine jako variraju. Rad s nizovima riječi spremljenim u regularna polja bi zahtjevalo memoriranje duljina svih riječi. To nije praktično zbog povećane memorijske potrošnje. Već jedna stranica teksta može sadržavati nekoliko stotina riječi, dok knjige mogu imati i nekoliko desetaka ili stotina tisuća riječi različitih duljina. **Rješenje** se sastoji od uvođenja posebnog znaka koji će označavati kraj radnog dijela niza. Time umjesto memoriranja veličine polja, što zauzima 4 byte-a memorije, koristimo jedan znak koji zauzima 1 byte memorije. Realizacija niza znakova koji sadrži poseban znak za kraj u C-u se zove **string**. Ideja posebne oznake za kraj se koristi i kod posebne samoreferencirajuće strukture koja se zove **vezana lista**[7].

```
1   "Jedan␣primjer␣stringa."
```

Stringovi se u C-u spremaju u obična polja znakova (oni jesu polja znakova), međutim na kraju sadrže bar jedan nul-znak `'\0'`. Prvi nul-znak (onaj s

[7] Vidi odjeljak 3.6

najmanjim indeksom) ima ulogu oznake za kraj niza (radni sadržaj se nalazi ispred tog znaka). Korištenjem formata `%s`, sadržaj se interpretira kao string, međutim tada mora sadržavati nul-znak.

```
1 char poruka[128]; // Polje od 128 znakova.
2 char niz[4] = {'a', 'b', 'b', 'a'};//Polje od 4 znaka
3 char s[5] = {'a', 'b', 'b', 'a', '\0'};//String duljine 4
4 char s[5] = "abba";//Ekvivalentni string duljine 4
```

Duljina stringa je **broj znakova** ispred nul-znaka, tj. duljina **radnog sadržaja** stringa. Osnovna funkcija za rad sa stringovima je funkcija `strlen` koja vraća duljinu zadanog stringa. Deklarirana je u datoteci zaglavlja `<string.h>`. Prototip (zaglavlje) funkcije `strlen` je:

```
1 size_t strlen(const char *s)
```

`strlen` vraća duljinu stringa s bez oznake za kraj niza, znaka `'\0'`. String je zadan **pokazivačem** na (konstantni) **prvi znak**, tj. funkcija **ne smije** promijeniti sadržaj stringa. Generalno, funkcije za rad sa stringovima imaju nedefinirano ponašanje kada se primjene na obične nizove znakova (bez `'\0'`).

Postoje dva različita načina za inicijalizaciju konstantnim stringom:

```
1 char a[] = "Poruka"
2 char *p = "Poruka";
```

- Pridruživanje konstantnog stringa polju je moguće **samo prilikom inicijalizacije**. Smijemo mijenjati **vrijednosti** unutar polja, ali ne i vrijednost pokazivača a.
- Pridruživanje konstantnog stringa pokazivaču smijemo vršiti pri inicijalizaciji i korištenjem operatora pridruživanja. **Ne smijemo** mijenjati sadržaj stringa (string je konstantan, rezultat izmjene je nedefiniran). Međutim, možemo mijenjati vrijednost pokazivača (smije pokazivati na neki drugi string).

Na lijevoj strani naredbe pridruživanja smije biti bilo koji objekt koji nije konstantan (*lvalue* izraz). To znači da je dozvoljeno promijeniti vrijednost tog objekta. Isto vrijedi i za pokazivač na `char`. Ako nije konstantan, smijemo mu pridružiti i konstantni string.

```
1  char *p;
2  ...
3  p = "Poruka";//OK, p pokazuje na konstantan string
4
5  char polje[10], s[5];
6  ...
7  polje = "Poruka"; /* Pogresno, polje je konstantan
8                                          pokazivac */
9  s = "tri"; /*Pogresno */
10 /*mozemo koristiti strcpy*/
11 strcpy(polje, "Poruka");
12 strcpy(s, "tri");
```

Stringovi (kao i polja) su zadani pokazivačem na prvi element (znak).

```
1  char s1[] = "Dobar dan", s2[] = "Dobar dan";
2  ...
3  if (s1 == s2) printf("jednaki");
4  else printf("razliciti");
```

Rezultat gornje usporedbe je **različiti** jer se uspoređuju **adrese** stringova (pokazivači na prve elemente) koji su različiti. Usporedbu stringova **znak po znak** radi funkcija `strcmp`.
Formatirano čitanje stringova se vrši funkcijom `scanf` (uvijek treba zadati maksimalnu duljinu polja) koristeći format:

- `%s` - string omeđen bjelinama
- `%[...]` i `%[^...]` - string koji sadrži ili ne sadrži (drugi slučaj) navedene znakove.

Cijeli redak znakova, uz pretvorbu `'\n'` → `'\0'`, možemo učitati funkcijama `fgets` (ili `gets_s`)[8].

Pisanje stringova se vrši funkcijom `printf` korištenjem formata `%s`. Funkcija ispiše string bez završnog `'\0'`. Funkcija `puts` ispisuje cijeli red uz pretvorbu `'\0'` → `'\n'`.

[8]Funkcija `gets` je izbačena iz jezika od verzije `C11` zbog nemogućnosti definiranja maksimalnog broja znakova koji se trebaju učitati. To je dovodilo do prenapunjenja polja i potencijalno omogućavalo korištenje funkcije u maliciozne svrhe.

```c
#include <stdio.h>
#include <string.h>

int main(void) {

char string[80];
scanf("%79s", string);
/* scanf("%79[^\n]", string); */

puts(string);
printf("␣Duljina␣str.␣=␣%u\n", strlen(string));
return 0; }
```

Ulaz: `Dobar dan\n`
Izlaz: za `scanf("%79s", string)`:
Dobar
Duljina str. = 5

Izlaz: za `scanf("%79[^\n]", string)`:
Dobar dan
Duljina str. = 9

Pri radu sa stringovima koristit ćemo `C` funkcije iz biblioteka sa zaglavljem `<string.h>` (za rad sa stringovima) i `<ctype.h>` (za rad sa znakovima).

Primjer 3.3.1

Želimo napisati funkciju `invertiraj` koja invertira ulazni string. String je zadan pokazivačem na prvi element.

Za razliku od prosljeđivanja polja funkciji, gdje kao dodatni parametar trebamo proslijediti i duljinu polja, kod stringa ne proslijeđujemo duljinu polja već sami tražimo kraj ili koristimo funkciju `strlen`.

Pokazat ćemo tri varijante funkcije: preko polja koristeći indekse (dva načina) i preko pokazivača (koristeći aritmetiku pokazivača).

```c
void invertiraj(char s[]){
int p, k; /* p = pocetak, k = kraj. */
char temp;

for (p = 0, k = strlen(s)-1; p < k; ++p, --k) {
```

```
6         temp = s[p];
7         s[p] = s[k];
8         s[k] = temp; }
9
10 return;
11 }
```

U `for` petlji gornje funkcije, unutar svake iteracije inkrementiramo indeks p i dekrementiramo indeks k.

```
1  void invertiraj(char s[]){
2  int i, n = strlen(s);
3  char temp;
4  /* Ne koristiti strlen u petlji. */
5  for (i = 0; i < n/2; ++i) {
6        temp = s[i];
7        s[i] = s[n - 1 - i];
8        s[n - 1 - i] = temp;
9     }
10 return;
11 }
```

Istu stvar možemo napraviti koristeći samo jedan indeks, računanjem $k = n-1-i$. Primijetimo da je dovoljno izračunati duljinu stringa samo jednom (prije naredbe `for`). Poziv funkcije `strlen` unutar funkcije `for` bi znatno usporio izvođenje programa, pošto bi u svakoj iteraciji iterirali po cijelom stringu i ponovo računali njegovu duljinu. Pošto se duljina stringa ne mijenja, takvo računanje je suvišno.

```
1  void invertiraj(char *s){
2  char temp, *p, *k; /* p = pocetak, k = kraj. */
3  p = s; k = p + (strlen(s) - 1);
4
5  while (p < k) {
6       temp = *p;
7       *p++ = *k;
8       *k-- = temp;
9     }
10 return;
11 }
```

Gornji kod invertira string koristeći pokazivače. Nad pokazivačima se prvo izvršavaju operatori ++/−−, zatim operator *. Postfix oblik vraća pokazivač na staru lokaciju (prije inkrementiranja pokazivača).

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void invertiraj(char *s); /* ili (char s[]) */
5
6  int main(void) {
7
8      char str[80]; /* Nije inicijaliziran! */
9      /* Citanje i provjera za prazan string. */
10     if (scanf("%79[^\n]", str) < 1) str[0] = '\0';
11     printf("␣String␣(duljine␣%u):\n", strlen(str));
12     puts(str);
13     invertiraj(string);
14     printf("␣Invertirani␣string:\n");
15     puts(string);
16 return 0; }
```

U nastavku ističemo neke značajnije funkcije za obradu stringova, deklarirane u datoteci zaglavlja `<string.h>`:

Funkcija `strlen` vraća duljinu stringa s bez završnog `'\0'` znaka. String je zadan pokazivačem s na konstantni prvi znak `(*s)`, tj. funkcija ne smije promijeniti sadržaj stringa preko pokazivača s.

```
1  size_t strlen(const char *s)
```

Funkcija `strcpy` kopira string t u string s (uključujući i završni `'\0'`) i vraća string pokazivač na prvi znak iz s.

```
1  char *strcpy(char *s, const char *t)
```

Funkcija `strcat` nadovezuje (konkatenira) string t na kraj stringa s, te vraća s. Prvi znak iz t kopira se na mjesto završnog nul-znaka `'\0'` u stringu s sve do kraja stringa t (uključujući `'\0'`).

```
1  char *strcat(char *s, const char *t)
```

Funkcija `strcmp` leksikografski uspoređuje stringove s i t.

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

Funkcija `strcmp` vraća broj:

- < 0, ako je $s < t$
- $= 0$, ako je $s = t$
- > 0, ako je $s > t$

Ponekad se `strcmp` implementira tako da je izlazna vrijednost razlika znakova na prvoj poziciji na kojoj se stringovi razlikuju, ukoliko takva pozicija postoji. Npr. `strcmp("BAB", "BCB") = -2` jer je 'A' - 'C' = -2.

Funkcije `strchr` i `strstr` vraćaju pokazivač na znak koji je početak prvog pojavljivanja znaka c (za `strchr`), odnosno, stringa t (za `strstr`), u stringu s, ako takvo mjesto postoji. Ako ne postoji takvo mjesto vraćaju `NULL`.

```
1 char *strchr(const char *s, const int c)
2 char *strstr(const char *s, const char *t)
```

Funkcije za obradu znakova deklarirane su u `<ctype.h>` i često se koriste pri radu sa stringovima. Funkcije imaju jedan argument tipa `int`, koji je:

- znak `EOF`, standardno `EOF` = -1 (zato se koristi `int`).
- znak prikaziv kao `unsigned char` (standardni znak).

Izlazna vrijednost je tipa `int`. Možemo ih podijeliti u dvije grupe:

- funkcije za provjeru znakova, vraćaju `int` različit od nule (istina), ako ulazni znak pripada određenoj grupi znakova. U protivnom vraćaju nulu (laž).
- funkcije za pretvaranje znakova, vraćaju pretvoreni ulazni znak.

Funkcije za provjeru znakova su:

```
1 int isalpha(int c); /* Malo ili veliko slovo */
2 int isdigit(int c); /* Decimalna znamenka */
3 int isalnum(int c); /* Alfanumericki znak */
4 int isxdigit(int c); /* Heksadecimalna znamenka */
```

```
int islower(int c); /* Malo slovo */
int isupper(int c); /* Veliko slovo */
int iscntrl(int c); /* Kontrolni znak */
int isgraph(int c); /* Ispisiv znak, bez blanka */
int isprint(int c); /* Ispisiv znak, ukljucujuci blank */
int ispunct(int c); /* Ispisiv znak, bez blanka, slova i
                       decimalnih znamenki */
int isspace(int c); /* Bl, \n, \t, \v, \f, \r */
```

U 7-bitnom ASCII kodu (0 do `0x7F`, ili 0 do 127), ispisivi znakovi su: `0x20` (' ', tj. blank) do `0x7E` ('∼'), kontrolni znakovi su: 0 (NUL) do `0x1F` (US) i `0x7F` (DEL).

Nekoliko primjera poziva funkcija za rad sa znakovima možemo vidjeti u doljnjem primjeu:

```
isdigit('0') = 1; isdigit('C') = 0;
isalpha('0') = 0; isalpha('C') = 1;
isxdigit('0') = 1; isxdigit('C') = 1;
```

Funkcije za pretvaranje mijenjaju **samo** slova. Ukoliko je ulaz znak koji nije slovo, taj znak vraćaju nepromijenjen.

```
int tolower(int c); /* Velika u mala */
int toupper(int c); /* Mala u velika */
```

Primjer 3.3.2

Program zasebno čita ime i prezime, svako u svom redu, uz preskakanje bjelina na početku reda. Zatim ih spaja u jedan string, s prazninom između njih, i ispisuje taj string.

```
#include <stdio.h>
#include <string.h>

int main(void) {
char ime[128], prezime[128];
char ime_i_prezime[256];

printf("Unesite␣ime:");
scanf("␣%127[^\n]", ime);

```

```
11 printf("Unesite␣prezime:");
12 scanf("␣%127[^\n]", prezime);
13
14 /* Spoji ime i prezime u jedan string */
15 strcpy(ime_i_prezime, ime);
16 strcat(ime_i_prezime, "␣");
17 strcat(ime_i_prezime, prezime);
18 printf("Ime␣i␣prezime:␣%s\n", ime_i_prezime);
19 return 0;
20 }
```

Ulaz: Marko Petar\n
Perić\n
Izlaz: Marko Petar Perić\n

Primjer 3.3.3

Implementiramo funkcije `isdigit`, `isalpha` i `toupper`.

```
1  int isdigit(int c) {
2      return ('0' <= c && c <= '9');
3  }
4
5  int isalpha(int c) {
6      return ('a' <= c && c <= 'z' || 'A' <= c
7                                       && c <= 'Z');
8  }
9
10 int toupper(int c) {
11     return ('a' <= c && c <= 'z') ?
12                          ('A' + c - 'a') : c; }
```

Pri implementaciji koristimo činjenicu da se znak može interpretirati kao kratki cijeli broj. Funkcije `isdigit` i `isalpha` koriste usporedbe cijelih brojeva, a funkcija `toupper` činjenicu da je poredak malih slova u ASCII tablici jednak poretku velikih slova, stoga udaljenost malog slova spremljenog u varijablu c do malog slova a odgovara udaljenosti traženog velikog slova do velikog slova A.

Primjer 3.3.4

Pišemo implementaciju funkcije `strtoupr` koja sva mala slova u zadanom stringu pretvara u velika.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

void strtoupr(char *s);

int main(void) {
char kolegij[] = "Programiranje␣2";
printf("␣Pocetni␣string:\n");
puts(kolegij);
strtoupr(kolegij);
printf("␣Obradjeni␣string:\n");
puts(kolegij);
return 0;
}

void strtoupr(char *s){
int i;

for (i = 0; s[i] != '\0'; ++i)
     if (islower(s[i]))
         s[i] = toupper(s[i]);
return;
}
```

Primjer 3.3.5

Funkcija `samoglasnik` provjerava je li zadani znak samoglasnik.

```
#include <ctype.h>
int samoglasnik(int c){
/*Pretvori c u malo slovo za ubrzanje provjere.*/
c = tolower(c);
return (c == 'a' || c == 'e' || c == 'i' || c == 'o' ||
        c == 'u');
}
```

Funkcija suglasnik provjerava je li zadani znak suglasnik. Znak je suglasnik ako i samo ako je slovo i nije samoglasnik.

```
1 int suglasnik(int c){
2 /* Koristi funkciju samoglasnik za ubrzanje provjere. */
3 return (isalpha(c) && !samoglasnik(c));
4 }
```

Primjer 3.3.6

Funkcija `samoglasnikS` vraća broj samoglasnika u stringu. Kroz varijabilni argument vraća prvi samoglasnik u stringu ukoliko postoji. Varijabilni argument ostaje nepromijenjen ukoliko ne postoji samoglasnik u ulaznom stringu.

```
1 int samoglasnikS(char *s, char *p_prvi){
2 int broj = 0, i;
3
4 for (i = 0; s[i] != '\0'; ++i)
5 if (samoglasnik(s[i]))
6         if (++broj == 1) *p_prvi = s[i];
7 return broj;
8 }
```

Primjer 3.3.7

Implementiramo funkciju `strlen` na dva način. Koristimo a) indekse, b) aritmetiku pokazivača.

```
1  int strlen(const char *s){//implementacija s indeksima
2  int n;
3
4  for (n = 0; s[n] != '\0'; ++n); //prazan for
5
6  return n;
7  }
8
9  int strlen(const char *s){//implementacija s pokazivacima
10 int n = 0;
11 //mozemo implementirati i koristenjem dva pokazivaca
12 while (*s++ != '\0') /* (*s++) */
```

```
13          ++n;
14  return n;
15  }
```

Primjer 3.3.8

Navodimo tri implementacije funkcije `strcpy`.

```
1   void strcpy(char *s, char *t){//koristenje indeksa
2           int i = 0;
3           while ((s[i] = t[i]) != '\0') ++i;
4   return;
5   }
6
7   void strcpy(char *s, char *t){//koristenje pokazivaca
8           while ((*s = *t) != '\0') {
9                   ++s; ++t; }
10  return;
11  }
12
13  void strcpy(char *s, char *t){//skraceni kod, oprez!
14              while ((*s++ = *t++) != '\0');
15  return;
16  }
```

U zadnjem primjeru, korištenje `*(++s) = *(++t)` ne bi bilo korektno!

Primjer 3.3.9

Implementiramo funkciju `strcat` korištenjem pokazivača i indeksa.

```
1   char *strcat(char *dest, const char *src){//pokazivaci
2   char *p = dest;
3   while (*p++); /* Pomak iza '\0' u dest. */
4           --p; /* Povratak na '\0'. */
5   while (*p++ = *src++); /* Kopiramo src u dest. */
6   return dest;
7   }
8
9   char *strcat(char s[], const char t[]){//indeksi
```

```
10 int i = 0, j = 0;
11 while(s[i] != '\0') ++i;
12 while((s[i++] = t[j++]) != '\0');
13 return s;
14 }
```

Primjer 3.3.10

Pišemo funkciju koja broji riječi u stringu koji funkcija prima kao argument. Riječ je niz znakova bez praznina, a riječi su odvojene bar jednom prazninom ili znakom `\t`.

Problem možemo riješiti na dva načina:

- koristimo varijablu `razmak` kojom pamtimo jesmo li pozicionirani ispred riječi (na razmaku), u tom slučaju je postavimo na istina, ili smo unutar riječi kada ju postavimo na laž. Svaka promjena vrijednosti varijable od laž prema istina označava jednu riječ (moramo paziti na zadnju riječ).
- koristimo varijablu `rijec` koju postavimo na istinu kada se pozicioniramo na riječ, a kada dođemo do razmaka nakon riječi, postavimo na laž. Svaka promjena varijable iz laž u istina označava jednu riječ.

```
1  int broj_rijeci(char *str){
2  int brojac = 0, razmak = 1;
3
4  while (*str != '\0') {
5        if (*str == '␣' || *str == '\t') {
6               if (!razmak) {/* Izlaz iz rijeci.*/
7                       ++brojac;
8                       razmak = 1; } }
9         else /* Dio rijeci. */
10              razmak = 0;
11        ++str; }
12
13 if (!razmak) ++brojac; /* Zadnja rijec! */
14
15 return brojac;
16 }
```

```c
int broj_rijeci(char *str){
int brojac = 0, rijec = 0;

for ( ; *str != '\0'; ++str)
      if (*str == ' ' || *str == '\t') {
          if (rijec)
              rijec = 0;  }
     else /* Dio rijeci. */
         if (!rijec) { /* Ulaz u rijec. */
              ++brojac;
              rijec = 1; }

return brojac;
}
```

Pripadni glavni program:

```c
#include <stdio.h>
#include <string.h>

int broj_rijeci(char *str);

int main(void) {
char s[] = "Ja sam mala Ruza , mamina sam kci.";

printf("String:\n");
printf("%s\n", s);
printf("Broj rijeci: %d\n", broj_rijeci(s));

return 0;
}
```

Izlaz za ulazni string s gore:
Broj rijeci: 7

Zadatak:

- Promijenite funkciju `broj_rijeci` tako da separatori budu i znakovi: ' ', '\t', '\v', '\n', '\r', '\f'.
- Dodatno i znakovi: ',', ':', ';', '.', '!', '?'.

- Promijenite funkciju `broj_rijeci` tako da broji samo riječi sastavljene od slova.
- Promijenite funkciju `broj_rijeci` tako da broji samo brojeve sastavljene od dekadskih znamenaka.
- Promijenite funkciju `broj_rijeci` tako da broji samo operatore po pravilima C-a.
- Nalazi najdulju riječ s određenim svojstvom i vraća pokazivač na početak te riječi (ili `NULL` ako je nema).

Primjer 3.3.11

Pišemo implementaciju funkcije `atoi` iz standardne biblioteke sa zaglavljem `<stdlib.h>`. Ova funkcija pretvara niz znakova (string), koji sadrži zapis cijelog broja, u njegovu numeričku vrijednost. Funkcija treba: preskočiti sve početne bjeline (kao u funkciji `scanf`, i redom pročitati sve znamenke broja, te ih pretvoriti u broj tipa `int`.

```
1  #include <ctype.h>
2
3  int f_atoi(const char s[]) {
4  int i, n, sign; /* Indeks, broj, predznak. */
5
6      /* Preskace sve bjeline, prazan for. */
7      for (i = 0; isspace(s[i]); ++i) ;
8      sign = (s[i] == '-') ? -1 : 1; /* Predznak! */
9
10     /* Preskoci predznak, ako ga ima. */
11     if (s[i] == '+' || s[i] == '-') ++i;
12
13    /* Hornerov algoritam za broj. */
14    for (n = 0; isdigit(s[i]); ++i)
15         n = 10 * n + (s[i] - '0');
16
17 return sign * n;
18 }
```

Koristimo funkcije `isspace`, koja ispituje je li znak praznina, i `isdigit`, koja ispituje je li znak znamenka. Funkcije su deklarirane u datoteci zaglavlja `<ctype.h>`.

U standardnoj biblioteci `<stdlib.h>` postoje funkcije za pretvaranje stringa u broj odgovarajućeg tipa:

```
14 double atof(const char *s) //pretvara s u double,
15 int atoi(const char *s) //pretvara s u int,
16 long atol(const char *s) //pretvara s u long.
```

Te funkcije dijele svojstva s funkcijom za čitanje broja odgovarajućeg tipa:

- preskaču se bjeline na početku stringa
- čita se najdulji niz znakova koji odgovara pravilu za pisanje broja tog tipa
- pročitani niz znakova se pretvara u broj

Pozive i povratne vrijednosti funkcija `atoi` i `atof` možemo vidjeti u donjem primjeru[9]:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5  printf("%d\n", atoi("␣123")); /* 123 */
6  printf("%d\n", atoi("␣12␣3")); /* 12 */
7  printf("%d\n", atoi("␣12abc3")); /* 12 */
8  printf("%d\n", atoi("abc12␣3")); /* 0 */
9  //ne moze konvertirati abc u vrijednost
10 //tipa int -> tada vraca 0
11 printf("%g\n", atof("␣12.5a4c␣3")); /* 12.5 */
12 printf("%g\n", atof("␣12.5e4c␣3")); /* 125000 */
13 return 0;}
```

Primjer 3.3.12

Pišemo implementaciju funkcije `itoa` koja pretvara cijeli broj u string (slično kao kod ispisa).

[9] U `<stdlib.h>` su definirane i funkcije za apsolutnu vrijednost `int abs(int n)` i `long labs(long n)`, dok je funkcija `double fabs(double d)` deklarirana u `math.h`.

```
1  void f_itoa(int n, char s[]){
2  int i = 0, sign; // Indeks, predznak.
3  //Zapamti predznak u sign i pretvori n u nenegativan broj.
4  if ((sign = n) < 0) n = -n;
5  // Generiraj znamenke u obratnom poretku.
6  do {
7       s[i++] = n % 10 + '0'; // Znamenka.
8  } while ((n /= 10) > 0); // Obrisi ju.
9
10 if (sign < 0)
11       s[i++] = '-'; // Dodaj minus na kraj.
12 s[i] = '\0'; // Kraj stringa.
13
14 invertiraj(s);
15 return;
16 }
```

Zadatak: Funkcija `f_itoa` ne radi korektno za najmanji prikazivi cijeli broj n. Razmislite zašto, te ispravite kod da ispravno radi za sve prikazive cijele brojeve.

Zadatak: Napišite implementaciju funkcije `atof` (`<stdlib.h>`) za pretvaranje niza znakova (stringa), koji sadrži zapis realnog broja, u njegovu numeričku vrijednost (tipa `double`). Realni broj smije biti napisan po svim pravilima C-a za pisanje realnih konstanti (točka, e).

Zadatak: Napravite implementaciju funkcije `ftoa` koja pretvara realni broj (tipa `double`) u string (kao kod ispisa u `printf`).

3.4 Rječnik, argumenti komandne linije i pokazivači na funkcije

U nastavku opisujemo korištenje rječnika, koji omogućuje učinkovito spremanje stringova, argumente komandne linije, koji omogućuju prijenos vrijednosti programu pri pozivu, te pokazivače na funkcije koji omogućavaju stvaranje široko primjenjivih funkcija.

3.4.1 Rječnik

Pri deklaraciji statičkih polja koje sadrže pokazivače moramo paziti na prioritete operatora [] i ∗. Primarni operator [] ima **viši prioritet** od unarnog operatora ∗. Zbog toga polje pokazivača ima deklaraciju:

```
1 tip_pod *ime[izraz];
2 int *ppi[10]; //polje od 10 pokazivaca
```

Prioritete možemo promijeniti korištenjem oblih zagrada. Promjenom prioriteta operatora se mijenjaju i svojstva deklariranih objekata.

```
1
2 int (*ppi)[10]; //pokazivac na polje od
3                 //10 elemenata
```

U odjeljku 3.3 smo vidjeli da pokazivač na `char` možemo inicijalizirati stringom. Tako dobijemo **konstantan** string. Isto vrijedi i za polje takvih pokazivača čime dobijemo **polje stringova**.

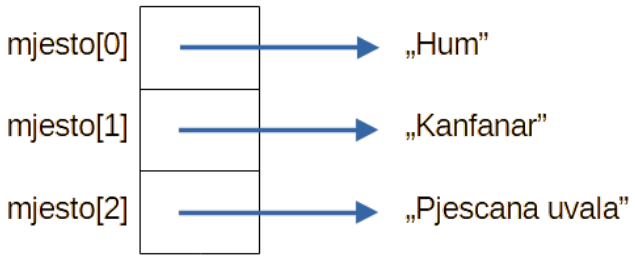
```
1 static char *gradovi[] = { "Osijek", "Rijeka",
2                            "Split", "Zagreb"};
```

Polja stringova deklarirana na takav način se često koriste za fiksna imena objekata (npr. dani u tjednu, mjeseci u godini). Stringove u takvim poljima nemamo namjeru mijenjati.

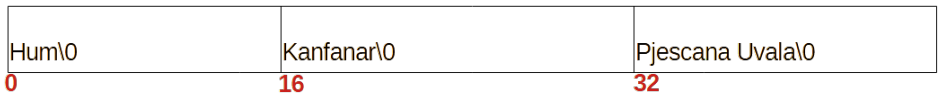
Postoji bitna razlika između polja pokazivača (na string) i dvodimenzionalnog polja znakova u koje spremamo stringove:

```
1 //polje pokazivaca na string
2 char *mjesto[] = { "Hum", "Kanfanar", "Pjescana␣Uvala" };
3
4 //dvodimenzionalno polje znakova sadrzi stringove
5 char amjesto[][16] = { "Hum", "Kanfanar",
6                        "Pjescana␣Uvala" };
```

Polje pokazivača:



Dvodimenzionalno polje znakova:



Primjer 3.4.1

Želimo učitavati riječi raznih duljina sa standardnog ulaza, leksikografski ih sortirati uzlazno i ispisati u sortiranom poretku. Riječi ćemo spremati kao stringove. Pretpostavit ćemo da svaka pojedina riječ stane u string od 80 znakova (u suprotnom možemo koristiti dinamičko alociranje stringova). Također, pretpostavljamo da ulaz sadrži po jednu riječ u svakom redu[10].

Problem možemo reprezentirati u memoriji na dva načina:

1. Stringove pamtimo u dvodimenzionalno polje te sortiramo niz redaka riječi. Navedeni pristup je loš iz dva razloga: a) spremanje je memorijski rastrošno, svi redci moraju biti duljine dovoljne za spremanje najveće riječi , b) sortiranje je vrlo sporo zbog zamjena stringova, dolazi do kopiranja stringova između redaka dvodimenzionalnog polja.

2. Polje pokazivača na stringove u kojem i-ti element sadrži pokazivač na i-tu riječ. Sortiranje riječi realiziramo zamjenama vrijednosti pokazivača u polju, a ne stringova. Zamjena vrijednosti pokazivača je iznimno brza operacija.

Predloženo rješenje će koristiti reprezentaciju predloženu u drugoj točki. Specifično, koristit ćemo dva jednodimenzionalna polja:

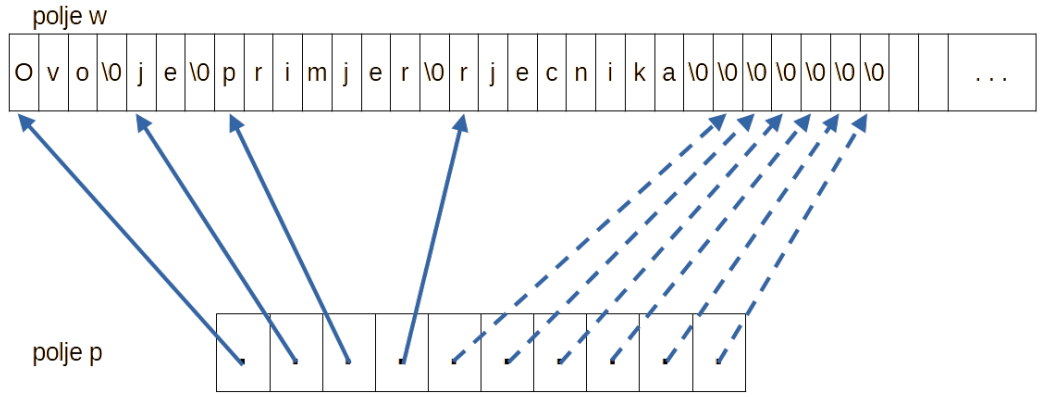
- polje znakova w koje sadrži sve riječi (kao stringove), jednu za drugom, bez praznina

[10] Ulaz u navedenom formatu je jednostavno čitati funkcijama `get_s` ili `fgets`

- polje pokazivača na znakove p tako da $p[i]$ sadrži pokazivač na početak $i+1$-e riječi u polju w

Pretpostavit ćemo da polje p sadrži maksimalno 100 riječi.

Na slici ispod je demonstrirana reprezentacija niza stringova "Ovo", "je", "primjer", "rjecnika".



Izvorni kod programa koji implementira opisanu funkcionalnost prvo definira dva globalna polja w i p, te funkciju za učitavanje riječi `unos`. Funkcija vraća ukupan učitani broj učitanih riječi ili -1 ukoliko je učitano više riječi od maksimalno dopuštenog broja. Slijedi implementacija funkcije za sortiranje `sort` koja koristi sortiranje izborom ekstrema[11], te implementacija funkcije za ispis riječi sadržanih u rječniku. Konačno, glavni program poziva funkcije za unos riječi, sortiranje i ispis rječnika.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  /* Sortiranje rjecnika. */
6  /* Simbolicke konstante:
7  MAXBROJ = max. broj rijeci,
8  MAXDULJ = max. duljina pojedine rijeci. */
9  #define MAXBROJ 100
10 #define MAXDULJ 80
11
12 /* Globalno polje znakova za rjecnik. */
```

[11] Više detalja možete vidjeti u materijalima kolegija Programiranje 1.

```
13 char w[MAXBROJ * MAXDULJ];
14 /* Globalno polje pokazivaca na
15 pojedine rijeci - stringove. */
16 char *p[MAXBROJ];
17
18 /* Stvarni broj rijeci u rjecniku. */
19 int broj_rijeci;
20
21 /* Ucitava rijeci i vraca broj rijeci.
22 Kraj ucitavanja je prazna rijec. */
23 int unos(char *p[]){
24 char *q = w;
25 int broj = 0, dulj;
26
27 while (1) {
28        if (broj >= MAXBROJ) return -1;
29        p[broj] = fgets(q, MAXDULJ, stdin);
30        if ((dulj = strlen(q)) == 0) break;
31        q += dulj + 1;
32        ++broj; }
33 return broj; }
34
35 /* Sortiranje rjecnika izborom ekstrema.
36 Dovodimo najmanji element na pocetak.
37 Rjecnik je zadan poljem pokazivaca na rijeci
38 (element polja je pokazivac na znak-string).
39 Koristimo samo zamjene pokazivaca. */
40
41 void sort(char *p[], int n){
42 int i, j, ind_min;
43 char *temp;
44
45 for (i = 0; i < n - 1; ++i) {
46        ind_min = i;
47        for (j = i + 1; j < n; ++j)
48              if (strcmp(p[j], p[ind_min]) < 0)
49                  ind_min = j;
50
51              if (i != ind_min) {
52                  temp = p[i];
```

```
53                  p[i] = p[ind_min];
54                  p[ind_min] = temp;
55               }
56            }
57 return;
58 }
59
60 /* Ispisuje sve rijeci u rjecniku. */
61 void ispis(char *p[]){
62 int i;
63
64 for (i = 0; i < broj_rijeci; ++i)
65            puts(p[i]);
66 return; }
67
68 /* Glavni program. */
69 int main(void){
70
71 if ((broj_rijeci = unos(p)) >= 0) {
72        printf("Broj rijeci = %d\n", broj_rijeci);
73        sort(p, broj_rijeci);
74        ispis(p); }
75 else
76       printf("Previse rijeci na ulazu.\n");
77 return 0; }
```

Zadaci: 1. U prethodnom programu, polja w i p imaju fiksnu duljinu (zadali smo maksimalan broj riječi i maksimalnu duljinu riječi). Promijenite program tako da se polja w i p **dinamički alociraju** i po potrebi **realociraju** (ako treba povećati duljinu nekog polja).

2. Umjesto polja w, koristite dinamičku alokaciju za svaku učitanu riječ tako da je ili zadana maksimalna duljina riječi (npr. 80 znakova) ili uz realokaciju riječi u malim blokovima ili *znak-po-znak*.

3. Prilagodite i iskoristite `QuickSort` algoritam za sortiranje rječnika. Pazite na to da funkcija `swap` mora zamijeniti vrijednosti pokazivača na znakove (stringove). Stoga argumenti funkcije swap moraju biti pokazivači na te pokazivače, tip argumenata je `char **`.

4. Za sortiranje rječnika iskoristite funkciju `qsort` iz standardne biblioteke `<stdlib.h>`. Treba paziti na tipove, kao za `swap`, i koristiti pokazivače na funkcije.

3.4.2 Argumenti komandne linije

Programi vrlo često koriste parametre koji se zadaju zajedno s imenom programa, te se učitavaju iz tog programa. Takvi parametri zovu se **argumenti komandne linije**:

```
cp ime1 ime2
```

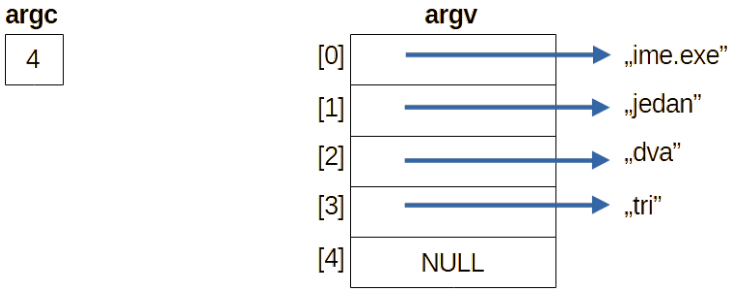
Ako želimo koristiti argumente komandne linije, moramo funkciju `main` deklarirati s dva formalna argumenta, a ne kao do sada s `void`. Standardna imena za te argumente su `argc` tipa `int` i `argv`, polje pokazivača na `char` (polje stringova). `argc` je skraćeno od eng. *argument count*. Broj `argc` − 1 je broj argumenata komandne linije. Ukoliko nema argumenata komandne linije, `argc` = 1. `argv`, skraćeno od eng. *argument value*, je polje pokazivača na argumente komandne linije. `argv[0]` uvijek pokazuje na string koji sadrži ime programa, kako je pozvan na komandnoj liniji. Ostali parametri smješteni su redom kojim su upisani. Uvijek je `argv[argc] = NULL`. Argumenti se učitavaju kao stringovi u funkciji `scanf`, bjeline su separatori. Kada argument zapisujemo u navodnicima, tada cijeli string uključujući praznine reprezentira taj ulazni argument.

```
1 int main(int argc, char *argv[])
2 { ... }
```

Ako program pozovemo s:

```
ime.exe jedan dva tri
```

tada je:



Uvijek je `argv[argc] = NULL`.

Program koji ispisuje argumente komandne linije prvo ispisuje vrijednost varijable `argc` kao cijeli broj s predznakom, zatim iterira po nizu stringova `argv`, te ispisuje svaki element kao string. `argv` sadrži pokazivače na stringove koji reprezentiraju proslijeđene ulazne parametre.

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[]){
4
5      printf("argc = %d\n", argc);
6
7      for (int i = 0; i < argc; ++i)
8          printf("argv[%d]: %s\n", i, argv[i]);
9
10 return 0;
11 }
```

Često u programima učitavamo jedan broj (recimo n), nakon čega učitavamo n brojeva, riječi, znakova itd. Ukoliko učitane elemente želimo spremiti, trebamo i alocirati memoriju, gdje opet koristimo zadanu vrijednost varijable n. Broj n možemo učitati i iz **komandne linije**, ali tada trebamo pretvoriti dobiveni `string` u `int` (koristiti funkciju `atoi`).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(int argc, char *argv[]){
4  ...
5  n = atoi(argv[1]);
6  ...
7  }
```

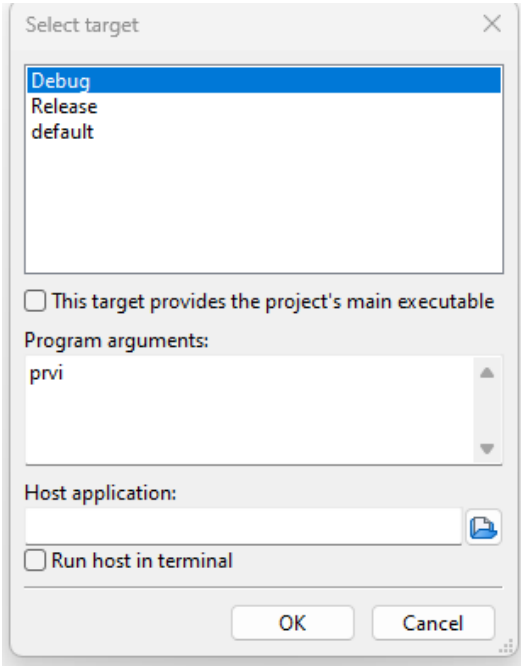
Ako program očekuje neke argumente na komandnoj liniji uvijek bi trebali napraviti njihovu provjeru.

- Jesu li zaista argumenti upisani pri pozivu programa? To možemo napraviti ispitivanjem vrijednosti varijable `argc`.
- Imaju li svi `argv[i]` korektan (očekivani) oblik? Provjeravamo format svakog ulaznog parametra.

U slučaju neispravnosti ulaznih parametara, treba vratiti pogrešku i prekinuti izvođenje programa. Korisna praksa pri obradi pogreške uzrokovane neispravnim ulaznim argumentima je opisati očekivani format ulaznih parametara, tako da korisnik može ispraviti ulaz i uspješno pokrenuti program.

```
1 if (argc < 2) {
2      printf("Broj␣elemenata␣nije␣zadan.\n");
3      exit(EXIT_FAILURE); } /* exit(1); */
4
5 n = atoi(argv[1]);
6 if (n <= 0) {
7     printf("Broj␣elemenata␣nije␣pozitivan.\n");
8     exit(EXIT_FAILURE); } /* exit(1); */
```

U `CodeBlocks`-u moramo koristiti projekte da bi učitali argumente zadane preko komandne linije u program. To radimo izabirom `Project`→`Set program's arguments...` Otvara se prozor kao na slici dolje (`CodeBlocks v20.03`):



U dio `Program arguments` navedemo ulazne argumente komandne linije koje želimo proslijediti programu. Projekti uvijek imaju `Debug` način izvođenja koji služi za testiranje projekta i `Release` način izvođenja koji je namijenjen produkciji (izvođenju od strane korisnika).

3.4.3 Pokazivači na funkcije

Pokazivač na funkciju deklarira se kao:

```
tip_pod (*ime)(tip_1 arg_1, ..., tip_n arg_n);
```

Ovdje je `ime` varijabla tipa pokazivač na funkciju, koja uzima n argumenata, tipa `tip_1` do `tip_n`, i vraća vrijednost tipa `tip_pod`. Slično kao i u prototipu funkcije, u deklaraciji ne treba pisati imena argumenata `arg_1` do `arg_n`.

```
int (*pf)(char c, double a);
int (*pf)(char, double);
```

Zagrade su nužne kod deklaracije pokazivača na funkciju zato što primarni operator () ima viši prioritet od unarnog operatora *.

Treba razlikovati funkciju koja vraća pokazivač na povratnu vrijednost nekog tipa (u primjeru `double`) od pokazivača na funkciju koja vraća element tipa `double`.

```
double *pf(double, double);
double *(pf(double, double));
//funkcije koje vracaju double*

double (*pf)(double, double);
//pokazivac na funkciju
//koja vraca vrijednost tipa double
```

Pokazivač na funkciju omogućava da jedna funkcija prima neku drugu funkciju kao argument. Realizacija ide tako da prva funkcija dobiva pokazivač na drugu funkciju.

```
int prva(int, int (*druga)(int));
```

U pozivu prve funkcije navodimo samo stvarno ime druge funkcije (koja negdje mora biti deklarirana s tim imenom), tj. ime funkcije je sinonim za pokazivač na tu funkciju.

```
prva(n, stvarna_druga);
```

Primjer 3.4.2

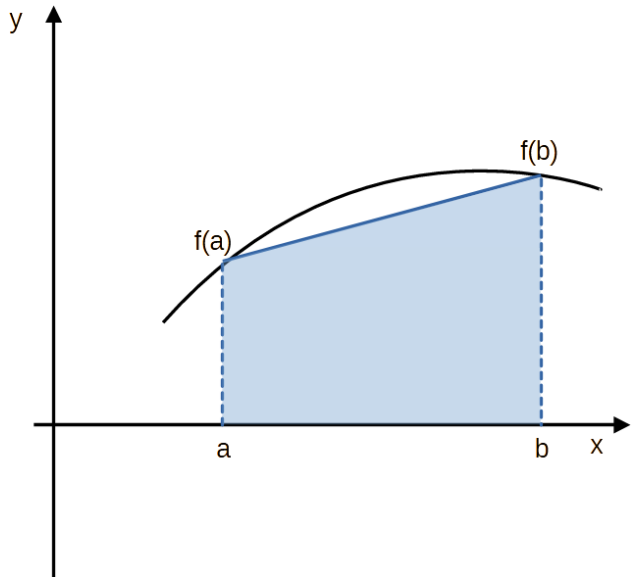
Radimo program za računanje vrijednosti integrala zadane (realne) funkcije f na segmentu $[a, b]$.

$$I = \int_a^b f(x)dx$$

Za računanje integrala obično se koriste približne (numeričke) formule. Slično Riemannovim sumama, te formule koriste vrijednosti funkcije u određenim točkama iz $[a, b]$. Funkcija za (približno) računanje integrala I onda mora imati (barem) 3 argumenta: granice integracije a,b i funkciju koja računa vrijednost $f(x)$ u zadanoj točki x.

Prvo ćemo napisati funkciju koja približno računa integral zadane funkcije, po tzv. **trapeznoj formuli**. Trapezna formula ima oblik (površina = srednjica puta visina trapeza):

$$\int_a^b f(x)dx \approx \frac{f(a) + f(b)}{2}(b - a)$$




```
1 include <stdio.h>
2 #include <math.h>
3
4 double integracija(double, double, double (*)(double));
5
6 int main(void) {
```

```
7
8       printf("Sin: %f\n", integracija(0, 1, sin));
9       printf("Cos: %f\n", integracija(0, 1, cos));
10
11  return 0;
12  }
13
14  double integracija(double a, double b,
15                                  double (*f)(double)) {
16     return 0.5 * (b - a) * ( (*f)(a) + (*f)(b) );
17  }//ova metoda nije jako tocna, provjerite!
```

Točniji način za računanje integrala funkcije definirane na nekom intervalu $[a, b]$ ćemo dobiti sljedeći korake:

- Izaberemo prirodan broj $n \in \mathbb{N}$
- Segment $[a, b]$ podijelimo na n podintervala točkama x_i, $i = 0, \ldots, n$ tako da je

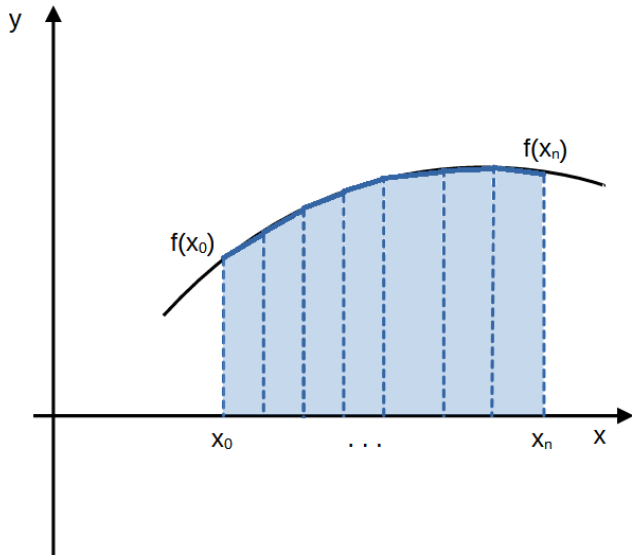
  $$a = x_0 < x_1 < \cdots < x_{n-1} < x_n = b$$

  Pripadni podintervali su $[x_{i-1}, x_i]$ za $i = 1, \ldots n$
- Vrijedi:

  $$\int_a^b f(x)dx = \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} f(x)dx$$

- Na svakom podintervalu $[x_{i-1}, x_i]$, za $i = 1, \ldots, n$, iskoristimo običnu trapeznu formulu i sve zbrojimo.

Obično se točke x_i uzimaju ekvidistantno, tako da svi podintervali imaju jednaku duljinu (h). Onda je **duljina** podintervala $h = \frac{b-a}{n}$, a točke su $x_i = a + i \cdot h, i = 0, \ldots, n$.



Običnom trapezoidnom formulom na intervalu $[x_{i-1}, x_i]$ dobivamo:

$$\int_{x_{i-1}}^{x_i} f(x)dx \approx \frac{f(x_{i-1}) + f(x_i)}{2}(x_i - x_{i-1}) = \frac{h}{2}(f(x_{i-1}) + f(x_i))$$

Zbrajanjem po svim podintervalima dobijemo:

$$I \approx I_n = \sum_{i=1}^{n} \frac{h}{2}(f(x_{i-1}) + f(x_i)) =$$

$$\frac{h}{2}(f(x_0) + 2f(x_1) + \cdots + 2f(x_{n-1}) + f(x_n))$$

$$= h(\frac{1}{2}(f(a) + f(b)) + \sum_{i=1}^{n-1} f(a + i \cdot h))$$

Povećavanjem broja n dobivamo sve točniju aproksimaciju integrala. Slučaj $n = 2$ je jednak običnoj trapeznoj formuli.

```
1 #include <stdio.h>
2 #include <math.h>
3 double integracija(double, double, int,
4                                 double (*)(double));
5
6 double integracija(double a, double b, int n,
```

```
7                                   double (*f)(double)){
8
9           double suma, h = (b - a) / n;
10          int i;
11
12          suma = 0.5 * ( (*f)(a) + (*f)(b) );
13
14          for (i = 1; i < n; ++i)
15                  suma += (*f)(a + i * h);
16  return h * suma;
17  }
18
19  int main(void){
20
21      double a = 0.0, b = 2.0 * atan(1.0); /* pi/2 */
22      int n = 1;
23
24      printf("Integral␣sinusa␣od␣0␣do␣pi/2:\n");
25
26      while (n <= 100000) {
27              printf("␣[n␣=␣%6d]:␣%13.10f\n", n,
28              integracija(a, b, n, sin));
29              n *= 10;}
30
31  return 0;
32  }
```

Adresni operator & ispred `sin` nije potreban (kao ni ispred imena polja), iako ga je dozvoljeno napisati:

```
1  integral = integracija(0, 1, &sin);//OK
2  integral = integracija(0, 1, sin);//OK
```

Slično smijemo napraviti i kod poziva funkcije zadane pokazivačem. Ne treba dereferencirati taj pokazivač.

```
1  double integracija(double a, double b,
2  double (*f)(double)) {
3  return 0.5 * (b - a) * ( (*f)(a) + (*f)(b) );
4  }//ILI
5
```

```
6 double integracija(double a, double b,
7 double (*f)(double)) {
8 return 0.5 * (b - a) * ( f(a) + f(b) ); }
```

Gornji kod je dozvoljen iako ne sasvim korektan po pitanju tipova. f je pokazivač na funkciju a $*f$ je funkcija nakon dereferenciranja.

Gornji kod možemo zapisati i kao:

```
1 double integracija(double a, double b,
2                                 double (*pf)(double)) {
3         double (*f)(double) = pf;
4 return 0.5 * (b - a) * ( f(a) + f(b) ); }
```

Zadaci: Napišite funkciju koja, kao argumente, prima string (tj. pokazivač na `char`) te pokazivač na funkciju za provjeru znakova, i radi sljedeće:

- vraća broj takvih znakova u stringu
- to isto, a kroz varijabilni argument vraća prvi takav znak u stringu, ako ga ima (u protivnom, ne mijenja taj argument)
- vraća pokazivač na prvi takav znak u stringu, ako ga ima (u protivnom, vraća `NULL`)
- to isto, kroz varijabilni argument vraća broj takvih znakova u stringu
- vraća pokazivač na zadnji takav znak u stringu, ako ga ima (u protivnom, vraća `NULL`).

3.5 Složene deklaracije, strukture, unije, polja bitova

U nastavku ćemo se upoznati sa složenim deklaracijama (interakcija polja, pokazivača i pokazivača na funkcije, definiranje novih jednostavnih i složenih tipova korištenjem ključne riječi `typedef`), definiranje složenih objekata korištenjem **struktura** te definiranje odgovarajućih tipova. Konačno, upoznat ćemo dva memorijski učinkovita objekta, **unije** koje omogućavaju reprezentiranje elemenata različitih tipova korištenjem dijeljene memorije fiksne duljine, te **polja bitova**, memorijski učinkovite reprezentacije polja zastavica.

3.5.1 Složene deklaracije

Deklaracije koje uključuju polja, pokazivače i pokazivače na funkcije mogu imati više značenja ovisno o prioritetima operatora. Stoga pravo značenje trebamo definirati korištenjem oblih zagrada.

Prisjetimo se razlike između **polja pokazivača** fiksne duljine i **pokazivača na polje** fiksne duljine.

```
int *a[10]; //polje duljine 10 koje sadrzi elemente
            //tipa int*
int (*b)[10]; //pokazivac na polje duljine 10
              //koje sadrzi elemente tipa int
int c[10];
b = &c;
(*b)[4] = 3; //postavlja c[4] na 3
a[4] = &b[4]; //sprema adresu elementa b[4]
              //u pokazivac a[4]
printf("%d\n",*(a[4])); //ispisuje 3
```

Pri deklaraciji funkcija i pokazivača na funkcije imamo više mogućnosti:

```
int f1(char a[10]); //deklaracija funkcije koja
                    //prima polje od 10 znakova
                   //a vraca vrijednost tipa int
int* f2(char a[10]); //deklaracija funkcije koja
                    //prima polje od 10 znakova
                   //a vraca vrijednost tipa int*
int* f3(char *a[10]); //deklaracija funkcije koja
                    //prima  polje od 10 pokazivaca
                    //na char a vraca vrijednost
                    //tipa int*
int f4(char (*a)[10]); //deklaracija funkcije koja
                    //prima  pokazivac na polje od
                    //10 znakova a vraca vrijednost
                    //tipa int
//int *f5(char *a)[10];//nije OK!
                      //f5 deklarirana kao funkcija
                      //koja vraca polje int-ova
int (*f6(char *a))[10]; //OK, funkcija koja prima
                       //pokazivac na char i
                       //vraca pokazivac na polje
```

```
                          //od 10 int-ova
int (*f7)(char (*a)[10]); //pokazivac na funkciju
                          //koja prima pokazivac
                          //na polje od 10 znakova
                          //i vraca int
int* (*f8)(char (*a)[10]); //pokazivac na funkciju
                          //koja prima pokazivac
                          //na polje od 10 znakova
                          //i vraca int*
int* (*f9[10])(char *a); //polje od 10 pokazivaca
                          //na funkciju koja
                          //prima pokazivac na char
                          //i vraca int*
```

Korištenjem ključne riječi `typedef` postojećim ili složenim tipovima podataka **dajemo nova imena**. Time ne kreiramo nove objekte ili varijable zadanog imena. Jednostavni oblik `typedef` deklaracije je:

```
typedef tip_podatka novo_ime_za_tip_podatka;
```

`novo_ime_za_tip_podataka` postaje sinonim za `tip_podataka` i smije se tako koristiti u svim kasnijim deklaracijama uz isto značenje.

Deklaracijom ispod identifikator `Masa` postaje sinonim za `double`.

```
typedef double Masa;
```

Varijable tipa `double` sada možemo deklarirati i kao:

```
Masa m1, m2, *pm1;//pm1 pokazivac na double
Masa elementi[10];//polje od 10 el. tipa double
```

Jednostavne deklaracije koristimo da bi **jednostavnije razumjeli ili čitali** programski kod i za **jednostavnije dokumentiranje** programa.

```
typedef int Metri, Kilometri;
Metri duljina, sirina;
Kilometri udaljenost;
```

U gornjem primjeru ime tipa sugerira jedinice u kojima su izražene određene vrijednosti.

Jednostavnije deklaracije su korisne i kada u programu koristimo **hijerarhiju** tipova, a veće prednosti se mogu dobiti definiranjem složenijih tipova. Npr. samoreferencirajuće strukture (vezane liste, stabla).

Deklaracija imena za složeniji tip počinje s `typedef`, a dalje ima isti oblik kao i deklaracija varijable tog imena i tog tipa. U tom slučaju `ime` nije varijabla tog tipa već sinonim za taj tip.

```
1 #define n 10
2 typedef double skalar;
3 typedef skalar vektor[n];
4 typedef vektor matrica[n];
```

U gornjem primjeru definiramo da je `skalar` novi tip koji je jednak (sinonim) tipu `double`. Koristeći tip `skalar` definiramo složeniji tip `vektor` kao niz od n (10) skalara, što je zapravo vektor od n (10) elemenata tipa `double`. `matrica` je ime tipa za polje od n (10) **vektora**, tj. sinonim za dvodimenzionalno polje **skalara**, sinonim za tip `double[n][n] = double[10][10]`.

Primjer 3.5.1

Koristeći ranije definirane tipove, možemo elegantno napisati funkciju za računanje umnoška $y = Ax$, kvadratne matrice A i vektora x:

```
1 void prod_mat_vek(matrica A, vektor x, vektor y){
2 int i, j;
3
4 for (i = 0; i < n; ++i) {
5       y[i] = 0.0;
6       for (j = 0; j < n; ++j)
7           y[i] += A[i][j] * x[j];
8       }
9 }
```

Izmijenite funkciju tako da n bude argument funkcije, a ne konstanta 10.

Kod rada sa **stringovima** možemo uvesti oznaku:

```
1 typedef char *string;
```

Ovdje je string sinonim za pokazivač na `char` (tip `char *`). Koristeći navedeni tip, funkcija `strcmp` za **uspoređivanje stringova** može se deklarirati:

```
1 int strcmp(string, string);
```

Pokazivač na `double` možemo nazvati **Pdouble**.

```
1 typedef double *Pdouble;
```

Nakon definicije tipa `Pdouble` možemo pisati:

```
1 Pdouble px; /* = double *px */
2 void f(Pdouble, Pdouble);
3 /* = void f(double *, double *); */
4 px = (Pdouble) malloc(100 * sizeof(double));
```

`typedef` možemo koristiti i za **kraće** zapisivanje **složenih** deklaracija. Npr. **pokazivač na funkciju** možemo zapisati kao:

```
1 typedef int (*PF)(char *, char *)
```

`PF` postaje ime za tip: **pokazivač na funkciju koja uzima dva pokazivača na `char` i vraća `int`**.

Nakon definicije tipa `PF`, umjesto deklaracije:

```
1 void f(double x, int (*g)(char *, char *)) { ... }
```

možemo pisati:

```
1 void f(double x, PF g) { ... }
```

Pri deklaracijama varijabli i tipova ponekad je dozvoljeno koristiti neke operatore . Npr.* (dereferenciranje), [] (polje), () (funkcija). Navedeni operatori djeluju samo na jedan pripadni argument, a ne na sve navedene.

U sljedećoj deklaraciji varijabli a i b:

```
1 int *a, b;
```

operator * djeluje samo na idenifikator a, a ne na tip `int`. Zato je a pokazivač na `int`, a b je varijabla tipa `int`. Operator u deklaraciji ima viši prioritet od navođenja tipa i djeluje na jednom argumentu, tj. varijabli. Ukoliko želimo deklarirati i varijablu a i b kao pokazivače na `int`, treba pisati:

```
1 int *a, *b;
```

Umjesto toga, možemo: deklarirati tip za pokazivač na `int`.

```
typedef int *pok_int;//deklarirati tip za int*
pok_int a, b;//deklarirati varijable tipa int*
```

Kod deklaracije argumenata funkcije tip se navodi **za svaki argument posebno**.

Za polja a i b duljine 10 trebamo **zasebno** navesti duljinu.

```
int a[10], b[10];
```

ili možemo uvesti i novi tip:

```
typedef int polje[10];
polje a, b;
```

Identifikator `polje` je ime tipa za `int[10]`.

3.5.2 Strukture

Struktura je **složeni tip podataka**, kao i polje. Za razliku od polja, koje služi grupiranju podataka istog tipa, strukture mogu grupirati podatke **različitih** tipova. Elementi (članovi) strukture mogu, ali ne moraju, biti različitog tipa i svaki element ima svoje posebno ime. Pri deklaraciji strukture moramo navesti ime i tip svakog člana.

Tip strukture možemo deklarirati na dva načina:

```
struct ime {
   tip_1 ime_1;
   tip_2 ime_2;
    ...
   tip_n ime_n;
};
```

`struct` je rezervirana riječ, a `ime` je ime stukture. Stvarni tip strukture je `struct ime`. Unutar vitičastih zagrada popisani su članovi strukture. Kao i kod polja, članovi strukture smješteni su u memoriji jedan za drugim, onim redom kojim su navedeni. Kod ovakve deklaracije tipa strukture, varijable tog tipa, općenito definiramo ovako:

```
mem_klasa struct ime var_1, var_2, ..., var_n;
```

var_1, var_2, ... , var_n su varijable tipa **struct ime**.

Definirat ćemo strukturu tocka koja definira točku u ravnini. Uzmimo da točka ima cjelobrojne koordinate, poput pixela na ekranu.

```
1 struct tocka {
2   int x;
3   int y;
4 };
```

Varijable tipa strukture tocka definiramo tako da:

Nakon gornje deklaracije strukture tocka (kao tipa) pišemo:

```
1 struct tocka t1, t2;
```

Ili pišemo:

```
1 struct tocka {
2    int x;
3    int y;
4 } t1, t2;
```

Postoji i bolji način deklaracije tipa strukture, koji olakšava definiciju varijabli tog tipa, koristeći `typedef`. **Prednost**: ne treba stalno navoditi riječ `struct` u deklaracijama varijabli.

```
1 typedef struct ime {
2    tip_1 ime_1;
3    tip_2 ime_2;
4    ...
5    tip_n ime_n;
6 } ime_tipa;
```

Na kraju deklaracije, cijelom tipu strukture dajemo ime `ime_tipa`. `ime_tipa` je sinonim za `struct ime` pa možemo koristiti oba način za deklariranje varijabli toga tipa. Ukoliko pišemo ime strukture, ono mora biti različito od svih ostalih imena (identifikatora) pa i od `ime_tipa`. Uobičajeno je kao ime strukture koristiti `_ime_tipa` (npr. `_osoba`). Ime strukture (odmah iza `struct`) smijemo i ispustiti ako ga nigdje nećemo koristiti. U donjem primjeru var_1, var_2, ... , var_n su varijable tipa ime_tipa, što je sinonim za `struct ime`.

```
1 mem_klasa ime_tipa var_1, var_2, ..., var_n;
```

Umjesto ranije definicije strukture za točku u ravnini, možemo uvesti tip `Tocka` za cijelu strukturu.

```
typedef struct {
   int x;
   int y;
} Tocka;
...
Tocka t1, t2, *pt1;
```

Identifikator `Tocka` je ime tipa za cijelu strukturu, a $t1$ i $t2$ su varijable tipa `Tocka`. Što je $pt1$?

Pri deklaraciji strukture koja odgovara tipu `Tocka` nismo navodili ime strukture iza `struct`, jer ga nećemo koristiti.

Varijablu tipa strukture možemo inicijalizirati pri definiciji:

```
mem_klasa struct ime var = {v_1, ..., v_n};
mem_klasa ime_tipa var = {v_1, ..., v_n};
```

Konstante v_1, v_2, ... , v_n pridružuju se navedenim redom odgovarajućim članovima strukture `var` - **član, po član**.

Ako je definirana struktura:

```
struct racun {
   int broj_racuna;
   char ime[80];
   float stanje;
}
```

Varijablu **kupac** možemo inicijalizirati:

```
struct racun kupac = { 12, "Ivo␣Ban", -200.00f};
```

Slično možemo inicijalizirati i čitavo polje struktura:

```
struct racun kupci[] = {
                  15, "Goga␣Markic", 45.00f,
                  18, "Josip␣Lovric", -23.00f,
                  23, "Martina␣Knezic", 0.00f };
```

Članovima strukture može se pojedinačno pristupiti korištenjem primarnog operatora točka (.). Operator točka (.) separira ime varijable i ime člana te strukture. Ako je `var` varijabla tipa strukture koja sadrži član `memb` onda je:

```
1 var.memb
```

vrijednost člana `memb` u strukturi `var`.

Ime člana je lokalno za svaku strukturu. Zato smijemo koristiti isto ime člana u raznim strukturama.

Operator točka (.) spada u najvišu prioritetnu grupu (primarni operatori) i ima asocijativnost $L \rightarrow D$. Zbog najvišeg prioriteta vrijedi:

$$++varijabla.clan \Leftrightarrow ++(varijabla.clan)$$

$$\&varijabla.clan \Leftrightarrow \&(varijabla.clan)$$

Član strukture (kao i element polja) smije pisati na lijevoj strani naredbe pridruživanja.

Za zadanu strukturu `tocka` i varijablu `ishodiste` tipa `struct tocka`:

```
1 struct tocka {
2      int x; /* prvi clan strukture */
3      int y; /* drugi clan strukture */
4 };
5 struct tocka ishodiste;
```

`ishodiste.x` je prvi član (prva komponenta) varijable ishodište, `ishodiste.y` je drugi član (druga komponenta) varijable `ishodiste`.

Analogno, za strukturu:

```
1 struct racun {
2      int broj_racuna;
3      char ime[80];
4      float stanje;
5 } kupac = { 1234, "Pero␣Peric", -1234.00f };
```

vrijedi:

```
1 kupac.broj_racuna = 1234,
2 kupac.ime = "Pero␣Peric",
3 kupac.stanje = -1234.00f.
```

Strukture mogu sadržavati druge strukture kao članove.

Pravokutnik paralelan koordinatnim osima možemo zadati parom dijagonalno suprotnih vrhova, npr. donjim lijevim (pt1) i gornjim desnim (pt2). Vrhovi su točke. Pri tome, struktura `struct tocka` (ili `Tocka`) mora biti deklarirana prije deklaracije strukture pravokutnik. U različitim strukturama mogu se koristiti ista imena članova.

```
struct pravokutnik {
      struct tocka pt1; /* ili Tocka pt1; */
      struct tocka pt2; /* ili Tocka pt2; */
};
```

Kad struktura sadrži polje kao član strukture, onda se pojedinim elementima tog polja (nazovimo ga `clan`) pristupa izrazom:

```
varijabla.clan[izraz]
```

Koristi se asocijativnost $L \rightarrow D$ za **primarne** operatore **točka** (.) i **indeksiranje** polja ([]). Operator . se primjeni na varijabli `varijabla` i dohvaća se član `clan`. Operator [] se primjenjuje na polju `clan` i dohvaća se element s indeksom koji se dobije evaluacijom izraza `izraz` u polju `clan`.

```
typedef struct {
      int broj_racuna;
      char ime[80];
      float stanje;
} Racun;
Racun kupac = { 12, "Ivo␣Ban",10.00f };
if (kupac.ime[0] == 'I') puts(kupac.ime);
```

Ako imamo polje struktura, svaki element polja je struktura. Nekom članu pripadne strukture pristupamo izrazom:

```
polje[izraz].clan
```

Opet je bitna asocijativnost: polje[izraz] je cijela struktura.

```
struct tocka {
int x;
int y;
} vrhovi[1024]; /* Polje tocaka. */
...
```

```
6 if (vrhovi[17].x == vrhovi[17].y) ...
```

Dozvoljene operacije nad strukturom su: a) pridruživanje, b) uzimanje adrese, primjena operatora `sizeof`. Nije dozvoljeno **uspoređivanje** cijelih struktura pošto prevodioc ne može znati kako ih uspoređivati.

Struktura može biti **argument** funkcije. Funkcija dobiva **kopiju** cijele strukture. Funkcija može **vratiti** strukturu. Ponašanje je kao kod varijabli osnovnog tipa, različito od polja.

Primjer 3.5.2

Pišemo strukturu `Tocka`, funkciju za sumiranje dvije varijable tipa `tocka` i pripadnu glavnu funkciju `main`.

```
1  #include <stdio.h>
2
3  typedef struct {
4     int x;
5     int y;
6  } Tocka;
7  Tocka t, ishodiste = {0, 0}, t1 = {1, 7};
8  //argumenti funkcije tipa Tocka
9  Tocka suma(Tocka p1, Tocka p2) {
10    p1.x += p2.x;
11    p1.y += p2.y;
12    return p1; }//povratna vrijednost tipa Tocka
13
14 int main(void) {
15    /* Dodjeljivanje struktura:
16    t i ishodiste moraju biti istog tipa */
17
18    t = ishodiste;
19    printf("Velicina = %u byteova\n", sizeof(t));
20
21    /* Zbroj tocaka, rezultat je tocka. */
22    t1 = suma(t1, t1);
23    printf("t1 = (%d, %d)\n", t1.x, t1.y);
24    return 0;
25 }
```

Izlaz programa:

```
Velicina = 8 byteova
t1 = (2, 14)
```

Primjer 3.5.3

Strukturu i funkciju za rad s kompleksnim brojevima možemo definirati kao:

```c
#include <stdio.h>
#include <math.h>

typedef struct {
    double re; /* ili x */
    double im; /* ili y */
} complex;

/* Napomena: cabs vec postoji u <math.h>! */
double zabs(complex a) {
   return sqrt( a.re * a.re + a.im * a.im );
}

int main(void){

complex a = {1,-3};

printf("%lg\n",zabs(a));

return 0;

}
```

Izlaz programa:

```
3.16228
```

U C99 standardu postoje tipovi i odgovarajuće funkcije za kompleksne brojeve (zaglavlje <complex.h>).

Pokazivač na strukturu definira se kao i pokazivač na druge tipove objekata.

```c
struct tocka {
```

```
    int x;
    int y;
} p1, *pp1;
...
pp1 = &p1;
(*pp1).x = 13; /* Zagrade su NUZNE! */
(*pp1).y = 27;
*pp1.x = 13; /* GRESKA!
*pp1.x je isto sto i *(pp1.x) */
```

Primarni operator strelica ($\rightarrow$) omogućava jednostavno dohvaćanje člana strukture preko pokazivača na tu strukturu. Asocijativnost operatora $\rightarrow$ je $L \rightarrow D$. Ako je `ptvar` pokazivač na strukturu, a `clan` je neki član te strukture, onda je:

$$ptvar \rightarrow clan \Leftrightarrow (*ptvar).clan$$

```
struct tocka p1, *pp1 = &p1;
pp1->x = 13;
pp1->y = 27;
```

Pokažimo pristup koordinatama vrhova pravokutnika `r` izravno i preko pokazivača `pr`.

```
struct pravokutnik {
    struct tocka pt1;
    struct tocka pt2;
} r, *pr = &r;

//ekvivalentni izrazi za dohvacanje x-koordinate vrha pt1
r.pt1.x // Operatori . i ->
pr->pt1.x // imaju isti prioritet.
(r.pt1).x // Asocijativnost im je
(pr->pt1).x // L -> D.
```

3.5.3 Unije

Unija je složeni tip podataka sličan strukturi, jer sadrži članove različitog tipa. Za razliku od strukture kod koje su članovi smješteni u memoriji jedan

za drugim, kod unije svi članovi počinju **na istom mjestu u memoriji**, tj. na istoj lokaciji. Svi članovi dijele zajednički dio memorije ovisno o veličini članova unije. Ukupna rezervirana memorija za varijablu tipa unije dovoljno je velika da u nju stane najveći član unije. **Ideja** je da zajednički dio memorije možemo **interpretirati na razne načine**, kao **vrijednosti različitih tipova**. Od tuda dolazi i ime **unija tipova**. Osnovna svrha unije nije ušteda memorijskog prostora, iako se može koristiti i u tu svrhu. Osim navedenih razlika u rezervaciji memorije, nema drugih razlika između strukture i unije u C-u. Umjesto ključne riječi `struct` za strukture, pišemo ključnu riječ `union` za unije. Deklaracija tipa unije ima isti oblik kao i za tip strukture, međutim umjesto `struct`, pišemo `union`. Kao i kod struktura, preporučuje se koristiti `typedef` za deklaraciju tipa unije.

```
union ime {
    tip_1 ime_1;
    ... ...
    tip_n ime_n;
};
```

Varijable x i y unije iz gornje definicije mogu se deklarirati:

```
union ime x, y;
```

`u.i` i `pu->i` iz donje deklaracije su varijable tipa `int`, a `u.x` i `pu->x` tipa `float`. Članovi i (tipa `int`) i x (tipa `float`) počinju na istoj lokaciji u memoriji. Standardno zauzimaju po 4 bytea, tj. dijele isti prostor.

```
union podatak {
   int i;
   float x;
} u, *pu;
```

Uniju možemo iskoristiti za ispis heksadecimalnog oblika prikaza realnog broja tipa `float` u računalu.

```
u.x = 0.234375f;
printf("0.234375␣binarno␣=␣%x\n", u.i);
```

Zadamo `u.x` kao `float`, te ista 4 byte-a čitamo i pišemo kao `u.i` tipa `int`. Pravi binarni prikaz možemo dobiti algoritmom koji ispisuje binarni prikaz

cijelog broja[12].

Primjer 3.5.4

Pišemo program koji učitava realni broj tipa `double` i piše **binarni** prikaz tog broja u računalu.

Broj tipa `double` standardno zauzima 8 byteova = 64 bita. Taj prostor možemo reprezentirati kao polje od 2 cijela broja tipa `int` (2 riječi).

Bitovi u `IEEE` prikazu za `double` imaju sljedeći raspored po byteovima (na `IA-32` i `Intel®` `64`):

- 1. byte = bitovi $7-0$ (donji bitovi),
- 2. byte = bitovi $15-8$ (donji bitovi),
- ...
- 8. byte = bitovi $63-56$ (gornji bitovi).

Globalno deklariramo tip unije za jedan `double` i polje od 2 `int`-a.

```
1  #include <stdio.h>
2  /* Binarni prikaz realnog broja tipa double. */
3  typedef union {
4     double d; /* 8 byteova = 64 bita. */
5     int i[2]; /* 2 rijeci od po 4 bytea. */
6  } Double_bits;
7
8  //funkcija za prikaz cijelog broja
9  void prikaz_int(int broj){
10    int nbits, bit, i;
11    unsigned mask;
12 /* Broj bitova u tipu int. */
13    nbits = 8 * sizeof(int);
14 /* Pocetna maska ima bit 1 na najznacajnijem */
15    mask = 0x1 << nbits - 1;//mjestu.
16
17     for (i = 1; i <= nbits; ++i) {
18        /* Maskiranje odgovarajuceg bita. */
19         bit = broj & mask ? 1 : 0;
```

[12]Navedeni algoritam je obrađen na kolegiju programiranje 1

```
20        /* Ispis i blank nakon svaka 4 bita,
21         osim zadnjeg. */
22         printf("%d", bit);
23         if (i % 4 == 0 && i < nbits) printf("␣");
24         /* Pomak maske za 1 bit udesno. */
25        mask >>= 1; }
26      printf("\n");
27
28      return;
29  }
30
31  void prikaz_double(double d){
32       Double_bits u;
33
34       u.d = d;
35
36       printf("␣1.␣rijec:␣");
37       prikaz_int( u.i[0] );
38       printf("␣2.␣rijec:␣");
39       prikaz_int( u.i[1] );
40
41       return;
42  }
43
44  int main(void){
45
46   double d;
47
48   printf("␣Upisi␣realni␣broj:␣");
49   scanf("%lf", &d);
50   printf("␣Prikaz␣broja␣%10.3f␣u␣racunalu:\n", d);
51   prikaz_double(d);
52
53   return 0;
54  }
```

Za ulaz 1.0 dobivamo:

```
Prikaz broja 1.000 u racunalu:
1. rijec: 0000 0000 0000 0000 0000 0000 0000 0000
2. rijec: 0011 1111 1111 0000 0000 0000 0000 0000
```

Za ulaz 0.1 dobivamo:

```
Prikaz broja 0.100 u racunalu:
1. rijec: 1001 1001 1001 1001 1001 1001 1001 1010
2. rijec: 0011 1111 1011 1001 1001 1001 1001 1001
```

Primijetite da je došlo do zaokruživanja mantise (signifikanda) u zadnja 2 bita prve riječi (bitovi 0 i 1).

Zadaci:

a) Napišite varijantu gornjeg programa za realni broj tipa `float`.

b) Preuredite oba programa tako da pregledno ispisuju sve bitne dijelove u prikazu realnog broja:

- bit predznaka,
- bitove karakteristike (eksponenta),
- bitove značajnog dijela (signifikanda/mantise).

3.5.4 Polja bitova

Polja bitova (eng. *bit-fields*) omogućavaju rad s pojedinim bitovima unutar jedne riječi u računalu. Jedno polje bitova je član (ili element) strukture ili unije. Sprema se u bloku susjednih bitova u memoriji računala, a zadaje se brojem bitova koje zauzima. Susjedna polja spremaju se u bloku susjednih bitova. **Svrha** polja bitova je spremanje 1-bitnih zastavica (engl. *flag*) u jednu riječ. Npr. u aplikacijama kao što je tablica simbola za prevodioc. Komunikacija s vanjskim uređajima kada treba postaviti ili očitati samo dijelove riječi. Deklaracija jednog polja bitova, kao člana strukture ili unije, ima sljedeći oblik (član, znak : i broj bitova):

```
1 struct ime { /* ili: union ime */
2    ...
3    tip_polja ime_polja : broj_bitova;
4    ...
5 }
```

Ograničenja (svi detalji ovise o implementaciji):

- `tip_polja` mora biti `int`, `unsigned int` ili `signed int`

- `ime_polja` je identifikator (kao i za ostale članove)
- `broj_bitova` **mora** biti **nengativan cijeli broj** (**nula** ima posebno značenje, tada se `ime_polja` smije ispustiti).

Gore opisani način deklariranja člana `ime_polja` predstavlja jedno polje uzastopnih bitova u računalu, duljine `broj_bitova`. Međutim, možemo deklarirati i uzastopne članove tog oblika. Kod običnih članova strukture, svaki član započinje u novoj riječi i zauzima cijeli broj riječi, ovisno o poravnanju riječi (eng. *byte/word/memory alignment*). Susjedno deklarirana polja bitova spremaju se u bloku uzastopnih bitova, bez praznina, tj. nastavljaju se jedan do drugog (potencijalno i unutar iste riječi). Poredak spremanja $\leftarrow$ ili $\rightarrow$ u riječi i eventualni prijelom sljedećih članova između riječi ovise o implementaciji.

```
1  struct primjer {
2      unsigned int a : 1;
3      unsigned int b : 3;
4      unsigned int c : 2;
5      unsigned int d : 1;
6  };
7  struct primjer v;
8  ...
9  if (v.a == 1) ...
10     v.c = 2;
```

Deklariramo strukturu sastavljenu od četiri uzastopna polja bitova: `a`, `b`, `c` i `d`. Ta polja imaju duljinu 1, 3, 2 i 1 bit (ukupno zauzimaju 7 bitova - spremaju se u bloku). Pojedine članove, koji su polje bitova, dohvaćamo na isti način kao i obične članove strukture (`v.a`, `v.b` itd.).
Ako broj bitova, deklariran u polju bitova nadmašuje duljinu jedne riječi u računalu, za pamćenje tog polja bit će upotrebljeno više riječi. Isto vrijedi i za blok uzastopnih polja bitova.

```
1  #include <stdio.h>
2  int main(void){
3    struct primjer {
4        unsigned int a : 5;
5        unsigned int b : 5;
6        unsigned int c : 5;
7        unsigned int d : 5;
```

```
  };
  struct primjer v = {1, 2, 3, 4};
  printf("␣v.a␣=␣%d,␣v.b␣=␣%d,␣v.c␣=␣%d,"
  "␣v.d␣=␣%d\n", v.a, v.b, v.c, v.d);
  printf("␣sizeof(v)␣=␣%u\n", sizeof(v));
  return 0;}
```

Izlaz:

```
v.a = 1, v.b = 2, v.c = 3, v.d = 4
sizeof(v) = 4
```

Na IA-32 i Intel® 64, cijela struktura v zauzima jednu riječ (4 byte-a). Kod ispisa, vrijednosti članova (polja bitova) se pretvaraju u `int` (standardna konverzija kratkih cjelobrojnih tipova za `printf`).

Raspored bitova unutar riječi može se kontrolirati korištenjem neimenovanih članova pozitivne duljine, unutar bloka uzastopnih polja bitova.

```
struct primjer {
    unsigned int a : 5;
    unsigned int b : 5;
    unsigned int : 5; /* Razmak 5 bitova. */
    unsigned int c : 5;
};
struct primjer v;
```

Neimenovani član duljine 0 bitova označava da sljedeće polje iz bloka treba smjestiti u sljedeću riječ.

```
#include <stdio.h>

int main(void) {
  struct primjer {
     unsigned int a : 5;
     unsigned int b : 5;
     unsigned int : 0; /* Idi u novu rijec. */
     unsigned int c : 5;
  };
  struct primjer v = {1, 2, 3};
  printf("␣v.a␣=␣%d,␣v.b␣=␣%d,␣v.c␣=␣%d\n",
  v.a, v.b, v.c);
```

```
13    printf("␣sizeof(v)␣=␣%u\n", sizeof(v));
14    return 0;
15  }
```

Izlaz:

```
v.a = 1, v.b = 2, v.c = 3
sizeof(v) = 8
```

Na IA-32 i Intel® 64, struktura v sad zauzima dvije riječi (8 byteova).

3.6 Vezane liste

Vezane liste su bitan predstavnik tzv. samoreferencirajućih struktra. Nakon kratkog uvoda u samoreferencirajuće strukture, koji će sadržavati opis vezanih ista i binarnih stabala, implementirat ćemo glavne operacije nad vezanim listama, te pokazati primjerima neke prednosti korištenja vezanih lista.

3.6.1 Uvod u samoreferencirajuće strukture

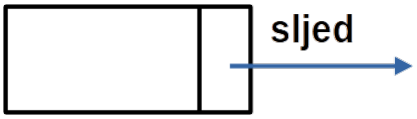
Pokazivač na objekt nekog tipa smije biti **član strukture**. To **omogućava povezivanje objekata različitih tipova**, ovisno o tipu pokazivača. Dozvoljeno je da pokaziva, član strukture, **pokazuje na istu takvu strukturu**, odnosno sadrži pokazivač na element istog tipa. Struktura **ne može** rekurzivno sadržavati samu sebe.

Struktura koja sadrži jedan ili više članova koji su pokazivači na strukturu tog istog tipa zove se **samoreferencirajuća** struktura. Samoreferencirajuće strukture su bitna posljedica korištenja pokazivača kao elementa strukture. Ovisno o broju i svrsi pokazivača možemo implementirati različite složene tipove podataka s rekurzivnom definicijom.
Svaki element samoreferencirajuće strukture je struktura koja ima dva bitna dijela: a) sadržaj, odnosno jedan ili više članova nekog tipa, b) jedan ili više pokazivača na element istog tipa, odnosno strukturu. Članovi strukture koji sadrže pokazivače na tu strukturu obično imaju standardna imena koja sugeriraju značenje pokazivača.

Primjeri samoreferencirajućih struktura su **vezane liste** i **binarna stabla**.

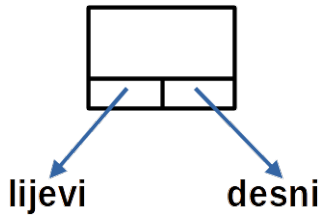
Element vezane liste, osim sadržaja, ima jedan pokazivač na element istog tipa. Taj pokazivač interpretiramo kao pokazivač na sljedeći element u listi.

Često korištena imena za taj pokazivač, član strukture, su **sljed**, **veza**, **next**, **link**.



Ako zamislimo cijelu listu elemenata, gdje su elementi kao na gornjoj slici, koji se nalaze iza prikazanog, pokazivač možemo interpretirati i kao listu sljedbenika ovog elemenata. To nam daje rekurzivnu definiciju.

Element binarnog stabla obično se zove čvor stabla. Osim sadržaja, ima dva pokazivača na element istog tipa. Te pokazivače interpretiramo kao pokazivače na lijevo i desno dijete tog čvora. Standardna imena za te pokazivače su **lijevi**, **desni** ili **left**, **right**.



Pokazivač možemo interpretirati kao lijevo i desno podstablo prikazanog elementa, što nam ponovo daje rekurzivnu definiciju.

Deklaraciju tipa za element samoreferencirajućih struktura možemo dobiti sljedeći smjernice:

- Koristimo `typedef` za deklaraciju tipa cijele strukture za element. Time izbjegnemo stalno pisanje riječi `struct` pri deklaraciji.
- Pretpostavimo da se sadržaj elementa sprema u jedan član strukture `info` tipa `sadrzaj`. Neka je tip `sadrzaj` ranije deklariran.
- Ovisno o vrsti strukture, u deklaraciju treba dodati jedan ili više pokazivača na takav element.

Deklaracija tipa za element vezane liste čiji podatci su sadržani u varijabi `info` tipa `sadrzaj`:

```
typedef struct _element {
        sadrzaj info; /* Sadrzaj. */
        struct _element *sljed; /* Pokazivac. */
} element;
```

Član `sljed` je pokazivač na `struct _element`, a ne struktura, stoga definicija strukture **nije rekurzivna**. Bez znaka * bi imali pokušaj deklaracije strukture koja sadrži samu sebe što **nije dozvoljeno**. U trenutku deklaracije pokazivača `sljed`, tip `struct _element` još nije potpuno određen. Međutim, memorija potrebna za spremanje pokazivača ne ovisi o tipu na koji pokazuje, stoga je definicija moguća. Pošto tip `element` nije definiran unutar strukture, za deklaraciju tipa pokazivača nužno trebamo koristiti ime strukture `_element`.

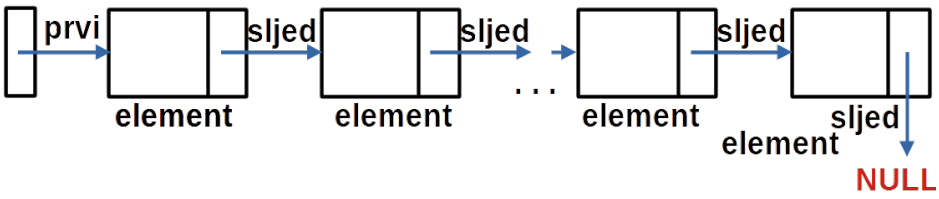
Tip za vezanu listu čiji elementi sadržavaju polje od 80 znakova definiramo:

```
typedef struct _element {
        char ime[80]; /* Sadrzaj. */
        struct _element *sljed; /* Pokazivac. */
} element;
```

Ovdje koristimo ime `sljed` za pokazivač na sljedeći element liste.

Pošto vezana lista može biti i **prazna**, imati nula elemenata, ona se ne zadaje prvim elementom već pokazivačem na prvi element ako postoji. Lista je prazna ako i samo ako je vrijednost tog pokazivača `NULL`.

Standardna imena za pokazivač na prvi element liste su `prvi`, `glava`, `first`, `head`.



Pokazivač na **prvi** element liste možemo definirati kao:

```
struct _element *prvi; /* Pocetak liste. */
```

Gornju deklaraciju smijemo pisati i prije deklaracije tipa `element`.

Nakon deklaracije tipa `element` smijemo pisati:

```
element *prvi; /* Pocetak liste. */
```

Poželjno je uvesti i deklaraciju tipa za pokazivače na elemente liste.

```
typedef struct _element *lista;
```

Gornju deklaraciju smijemo pisati i **prije** deklaracije tipa `struct _element`.

Skraćenu deklaraciju tipa ne smijemo napisati prije deklaracije tipa `element`:

```
typedef element *lista;
```

Potrebne deklaracije tipova za vezanu listu su:

```
typedef struct _element *lista;

typedef struct _element {
        sadrzaj info; /* Sadrzaj. */
        lista sljed; /* Pokazivac. */
} element;
```

Listu deklariramo pokazivačem `prvi` kojega **inicijaliziramo** na `NULL`, što označava praznu listu.

```
lista prvi = NULL; /* Pocetak liste. */
```

Nakon deklaracija, možemo definirati nekoliko varijabli tipa `element` i povezati ih u vezanu listu.

```
element a, b, c; /* Elementi liste. */
...
prvi = &a;
strcpy(a.ime, "prvi"); /* NE: a.ime = "prvi"; */
a.sljed = &b;
strcpy(b.ime, "drugi");
b.sljed = &c;
strcpy(c.ime, "treci");
c.sljed = NULL;
```

U praksi se uglavnom kreiranje vezane liste radi drugačije, učitavanjem ili računanjem elemenata, spremanjem u dinamički alocirani objekt te povezivanjem tih objekata u listu.

Vezane liste i slične strukture su idealne za:

- **dinamičku** promjenu veza među elementima
- **dodavanje novih** i **izbacivanje postojećih** elemenata

Elementi takvih struktura se uglavnom kreiraju **dinamičkom alokacijom** memorije.

```
prvi = (lista) malloc(sizeof(element));

if (prvi == NULL) {
     printf("Alokacija nije uspjela.\n");
     exit(EXIT_FAILURE); /* exit(1); */
}

strcpy(prvi->ime, "prvi");
prvi->sljed = NULL;
```

Deklaracija **tipa** za element **binarnog stabla**:

```
typedef struct _cvor {
      sadrzaj info; /* Sadrzaj. */
      struct _cvor *lijevi; /* Pokazivac. */
      struct _cvor *desni; /* Pokazivac. */
} cvor;
...
cvor *korijen; /* Pokazivac na korijen. */
```

Početni element stabla obično se zove **korijen**. On nema roditelja u stablu, tj. nije dijete niti jednog elementa. U dinamičkoj strukturi, to je pokazivač na početni element stabla.

```
typedef struct _treenode *Treeptr;

typedef struct _treenode {
... /* Sadrzaj. */
     Treeptr left; /* Pokazivac. */
     Treeptr right; /* Pokazivac. */
} Treenode;
...
Treeptr root; /* Pokazivac na korijen. */
```

Binarno stablo može biti prazno, u tom slučaju je `root = NULL`.

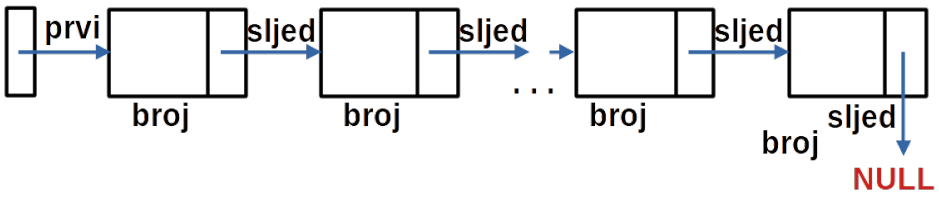
3.6.2 Operacije nad vezanim listama

Osnovne operacije nad vezanom listom su: a) kreiranje i uništavanje elemenata, b) dodavanje novog elementa u listu, c) brojanje elemenata liste, d) ispis liste, e) pretraživanje liste ili prolaz kroz listu, f) izbacivanje ili brisanje elementa iz liste, g) spajanje ili konkatenacija dvije liste, h) sortiranje liste

Navedene operacije ćemo implementirati u sljedećim primjerima.

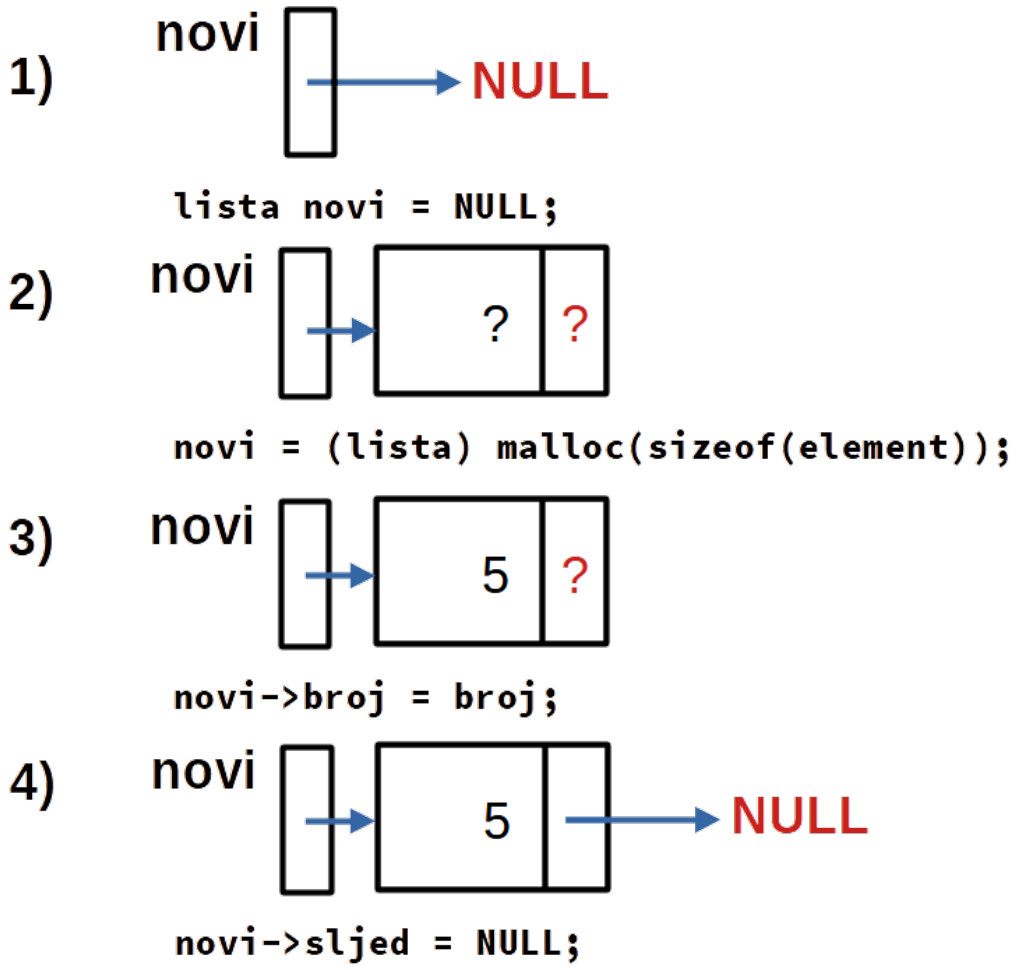
U primjerima ćemo koristiti vezanu listu **cijelih brojeva**.

```
1  /* Tip za pokazivac na element liste. */
2  typedef struct _element *lista;
3
4  /* Tip za element liste. */
5  typedef struct _element {
6       int broj; /* Sadrzaj je broj. */
7       lista sljed; /* Pokazivac na sljedeci */
8  } element; /* element u listi. */
9
10 lista prvi; /* Pokazivac na pocetak liste. */
```



Operacije **dodavanja** i **izbacivanja** elemenata najlakše se rade na **početku** liste, zbog sekvencijalnog pristupa elementima. Pristup proizvoljnom elementu liste (kod nepraznih listi), moguć je preko pokazivača `prvi`. Potrebno je iterirati do elementa pomoćnim pokazivačem `pom`, koristeći pokazivače `sljed`. Sve **funkcije** koje rade s elementima vraćaju **pokazivač** na element, objekt tipa `lista`.

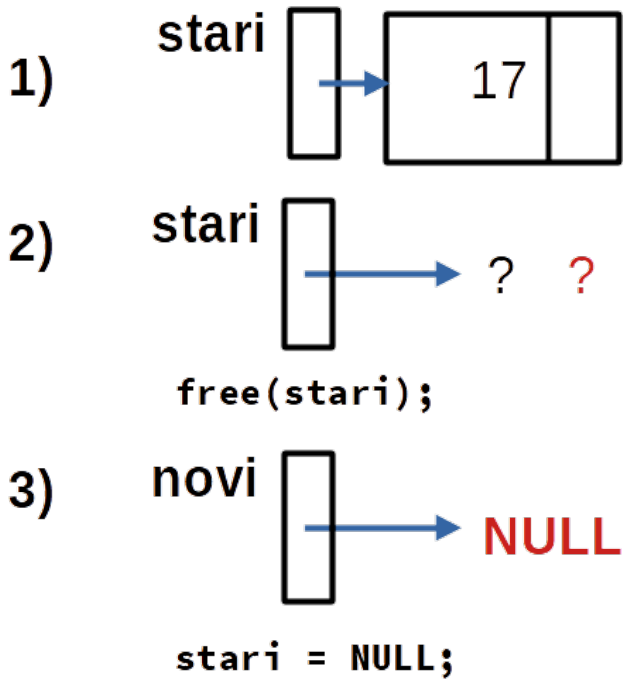
Postupak kreiranja novogelementa liste je opisan na slici dolje i u pripadnom kodu:




```
1  lista kreiraj_novi(int broj){
2
3    lista novi = NULL;
4    novi = (lista) malloc(sizeof(element));
5
6    if (novi == NULL) {
7         printf("Alokacija nije uspjela.\n");
8         exit(EXIT_FAILURE); /* exit(1); */
9    }
10
11   novi->broj = broj;
12   novi->sljed = NULL;
```

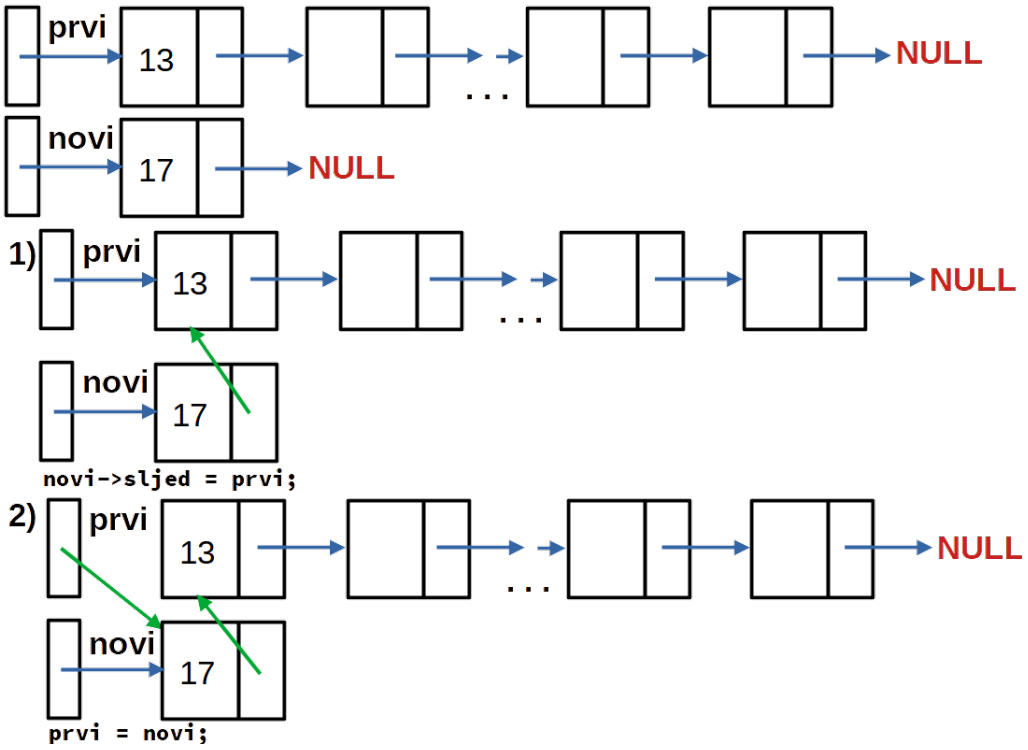
```
13   return novi;
14 }
```

Postupak brisanja elementa liste je opisan na slici dolje i u pripadnom kodu:




```
1 lista obrisi_element(lista stari){
2     free(stari);
3     return NULL; /* Umjesto stari = NULL; */
4 }
```

Postupak ubacivanja novog elementa na početak postojeće liste je opisan na slici dolje i u pripadnom kodu:



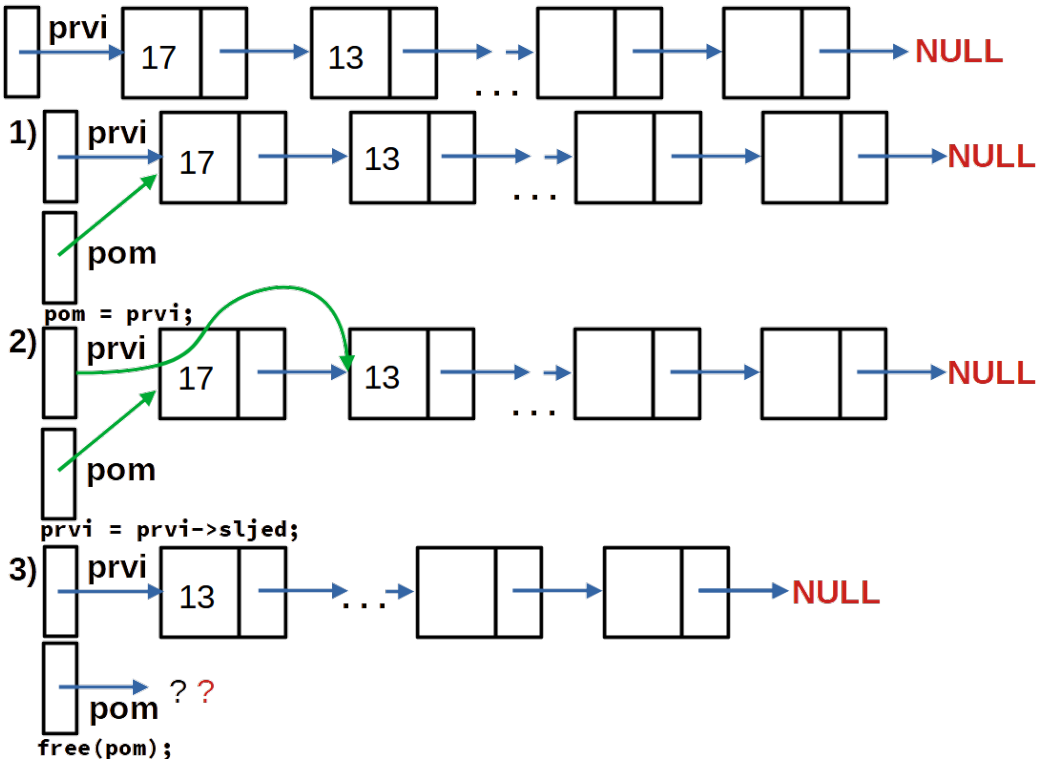

```
1 lista ubaci_na_pocetak(lista prvi, lista novi){
2 /* Ne provjerava novi != NULL. */
3     novi->sljed = prvi;
4     prvi = novi;
5     return prvi;
6 }
```

Poziv svih funkcija za rad s listom će biti oblika:

`prvi = funkcija_na_listi(prvi, ...);`

Dozvoljeno je da je lista **prazna** na ulazu i izlazu.

Postupak brisanja prvog elementa iz postojeće vezane liste je opisan na slici dolje i u pripadnom kodu:




```
1  lista obrisi_prvog(lista prvi){
2
3       lista pom;
4
5       if (prvi != NULL) {
6            pom = prvi;
7            prvi = prvi->sljed;
8            free(pom);
9           /* Ne treba pom = NULL; */
10        }
11
12      return prvi;
13 }
```

Operacije `kreiraj_novi` i `ubaci_na_pocetak` ima smisla spojiti u jednu operaciju `kreiraj_sprijeda`. Tada pomoćni pokazivač **novi** postaje **lokalni** objekt u funkciji.

```
1  lista kreiraj_sprijeda(lista prvi, int broj){
```

```
2
3       lista novi = NULL;
4       novi = (lista) malloc(sizeof(element));
5
6       if (novi == NULL) {
7          printf("Alokacija␣nije␣uspjela.\n");
8          exit(EXIT_FAILURE); /* exit(1); */
9        }
10
11      novi->broj = broj;
12      novi->sljed = prvi;
13      return novi; }
```

Brisanje liste se može implementirati kao brisanje prvog elementa dok `prvi!= NULL`.

```
1  lista obrisi_listu(lista prvi){
2
3       lista pom;
4
5       while (prvi != NULL) {
6              pom = prvi;
7              prvi = prvi->sljed;
8              free(pom);
9         }
10
11     return NULL; /* <=> return prvi; */
12 }
```

Broj elemenata liste treba **izračunati**, kao i kod stringa, prolaskom do kraja liste.

```
1  int broj_elemenata(lista prvi){
2    lista pom;
3    int brojac = 0;
4
5    for (pom = prvi; pom != NULL; pom = pom->sljed)
6           ++brojac;
7
8    return brojac;
9  }
```

```
void ispisi_listu(lista prvi){

  lista pom;
  int brojac = 0;

  for(pom = prvi; pom != NULL; pom = pom->sljed){
           printf("␣Element␣%2d,␣broj␣=␣%2d\n",
          ++brojac, pom->broj);
    }

  return;
}
```

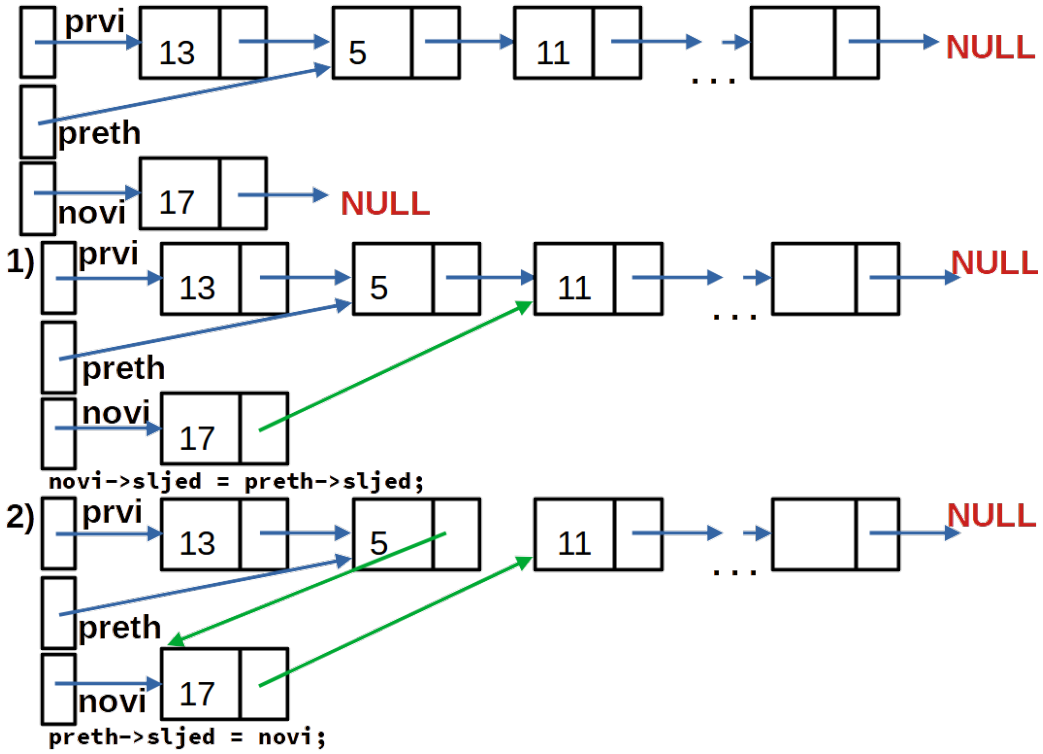
Funkcija za traženje zadanog broja vraća pokazivač na prvi element koji sadrži zadani broj, ili `NULL` ako takvog elementa nema u listi. Uočite **skraćeno računanje** uvjeta u `while` petlji.

```
lista trazi_broj(lista prvi, int broj){

     lista pom = prvi;

     while (pom != NULL && pom->broj != broj)
            pom = pom->sljed;
    return pom;
}
```

Funkcija `trazi_zadnji` vraća pokazivač na zadnji element u listi ili vraća `NULL` ako takvog elementa nema. Ukoliko postoji, zadnji element liste ima `sljed` jednak `NULL`.

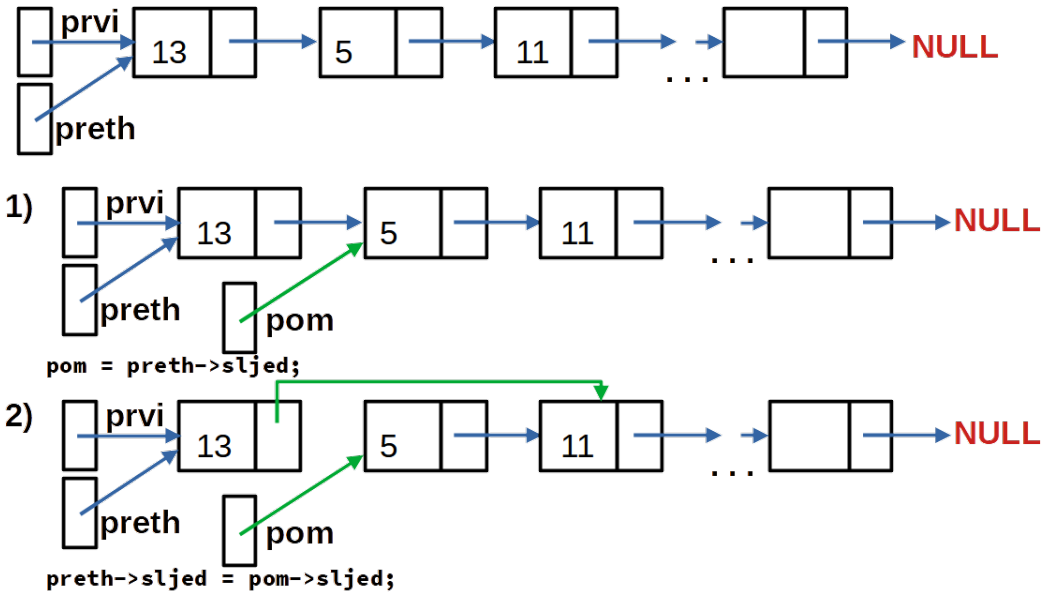
```
lista trazi_zadnji(lista prvi){
      lista pom;

      if (prvi == NULL) return NULL;

      for (pom = prvi; pom->sljed != NULL;
             pom = pom->sljed);

      return pom;
}
```

Funkcija koja ubacuje element u postojeću listu na proizvoljnu poziciju iza prve je objašnjena u koracima na slici dolje i u pripadnom kodu:




```
1  lista ubaci_iza(lista pr, lista pre, lista novi){
2    /* Ne provjerava novi != NULL. */
3    /* Ako je pre == NULL, ubacujemo na pocetak. */
4                if (pre == NULL) {
5                     novi->sljed = pr;
6                     pr = novi;
7                   }
8                else {
9                     novi->sljed = pre->sljed;
10                    pre->sljed = novi;
11                  }
12               return pr;
13 }
```

Brisanje elementa iz postojeće liste s proizvoljne pozicije iza prve je objašnjeno u koracima na slici dolje i u pripadnom kodu:



Ako izbačeni element, na koji pokazuje `pom`, zaista želimo obrisati, pozivamo `pom = obrisi_element(pom);` ili `free(pom);`. Međutim, s tim elementom možemo raditi i druge operacije.

```
1  lista obrisi_iza(lista prvi, lista preth){
2
3          lista pom;
4  /* Ako je preth == NULL, brisemo prvi element. */
5          if (preth == NULL) {
6              pom = prvi;
7              prvi = prvi->sljed;
8             }
9          else {
10            pom = preth->sljed;
11            preth->sljed = pom->sljed;
12        }
13
14        free(pom);
15        return prvi;
16 }
```

Implementacija funkcije ima dva nedostatka:

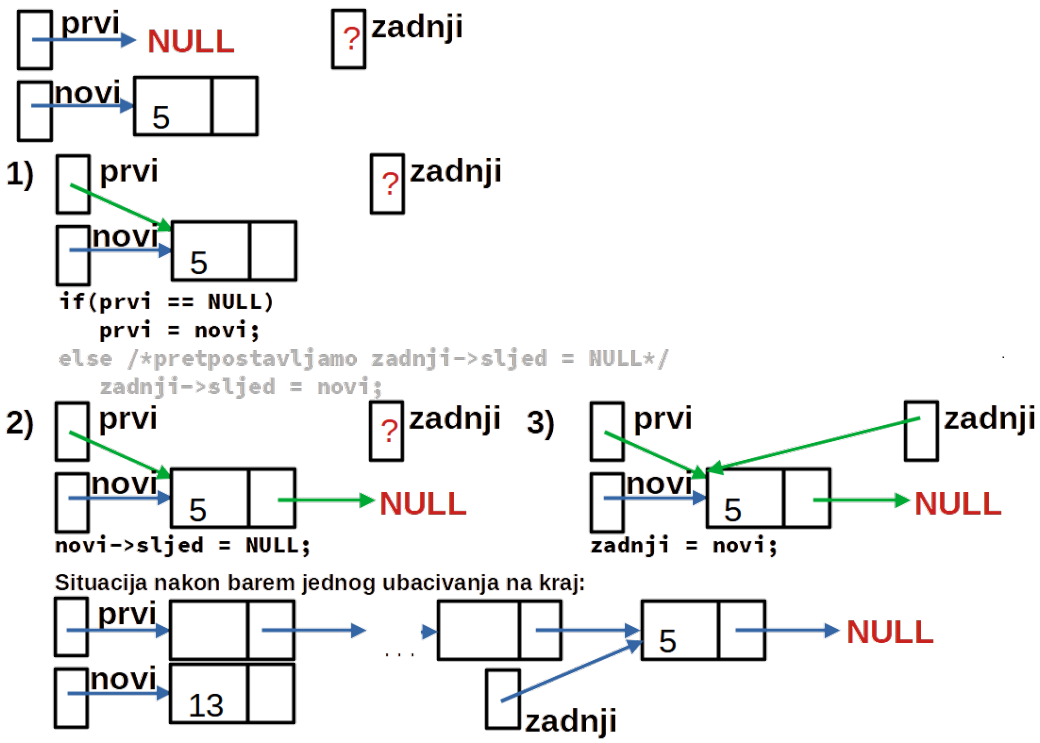
- ne provjerava je li ulazna lista prazna, tj. ne testira uvjet `prvi == NULL`

- ne pazi na kraj liste, slučaj `preth == zadnji`, tj. `preth->sljed == NULL`

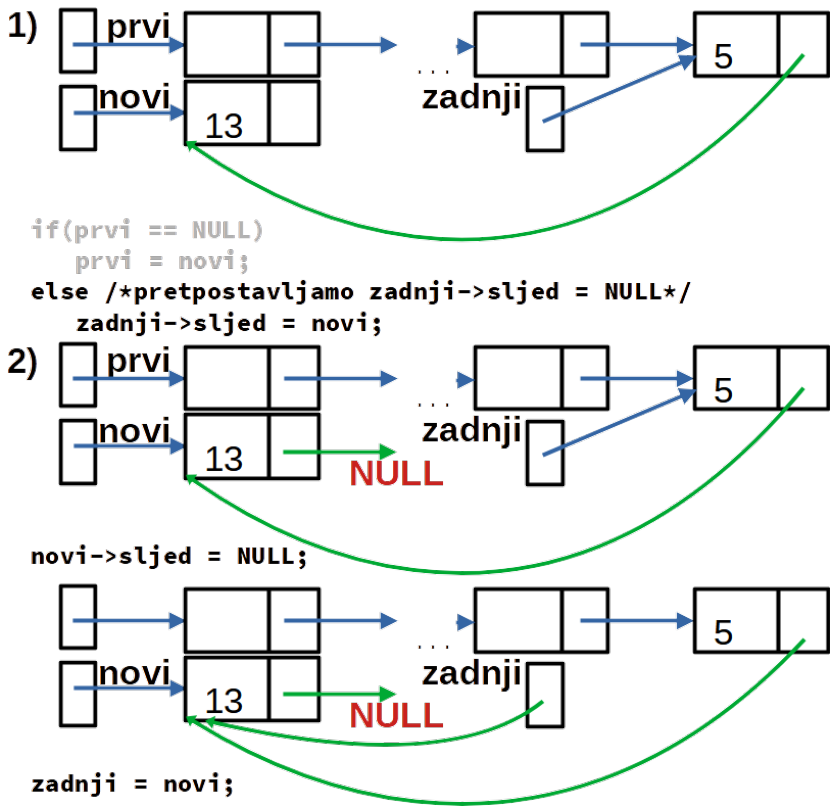
Uz navedeno, pedantna varijanta provjerava pokazuje li `preth` na neki element liste zadane pokazivačem `prvi`.

Zadatak: **Popravite** navedene nedostatke funkcije.

Ubacivanje na kraj postojeće prazne liste uz pamćenje zadnjeg elementa liste je objašnjeno u koracima na slici dolje i u pripadnom kodu:



Ubacivanje na kraj postojeće ne prazne liste uz pamćenje zadnjeg elementa liste je objašnjeno u koracima na slici dolje i u pripadnom kodu:




```
1  lista ubaci_na_kraj(lista prvi, lista *p_zadnji,
2                                          lista novi){
3  /* Ne provjerava novi != NULL. */
4  /* Vraca zadnji kroz varijabilni argument  p_zadnji. */
5
6        lista zadnji = *p_zadnji;
7
8       /* Moze: prvi == NULL || zadnji == NULL. */
9        if (prvi == NULL)
10            prvi = novi;
11       else /* Ocekujemo zadnji->sljed == NULL. */
12           zadnji->sljed = novi;
13       novi->sljed = NULL;
14 /*Ne treba: zadnji = novi; *p_zadnji = zadnji;*/
15       *p_zadnji = novi; /* Vrati novi zadnji! */
16       return prvi; }
```

Zadatak: Napišite funkciju `kreiraj_straga` po ugledu na funkciju `kreiraj_sprijeda` sa zaglavljem:

```
lista kreiraj_straga(lista prvi, lista *p_zadnji,
                                          int broj)
```

koja kreira novi element za zadani broj, ubacuje ga na kraj liste i korektno vraća pokazivače na prvi i zadnji element.

Primjer 3.6.1

U zadanu, uzlazno sortiranu, vezanu listu cijelih brojeva, zadanu pokazivačem `prvi` treba ubaciti jedan element, zadan pokazivačem `novi` tako da nova lista također bude uzlazno sortirana. Polazna lista smije biti i prazna.

Rješenje problema ima dva ključna koraka: a) pronalaženje odgovarajuće pozicije na koju treba ubaciti novi element (traženje po listi) i b) ubacivanje zadanog elementa na tu poziciju u listu.

Poziciju za ubacivanje novog elementa (pošto je lista sortirana) tražimo tako da preskačemo sve elemente čiji sadržaj je manji od sadržaja novog, a stajemo na prvom elementu čiji sadržaj je veći ili jednak sadržaju novog (ako takav postoji). Time nalazimo element **ispred** kojeg treba ubaciti novi element. Zato koristimo dva pokazivača: `pom` - na element čiji sadržaj testiramo i `preth` - na element ispred tog (ako postoji). Umjesto `pom` možemo koristiti i `preth->sljed`.

Ubacivanje iza elementa `preth` radimo kao u funkciji `ubaci_iza`.

- Novom elementu postavimo sljedeći.
- Prethodnom elementu (ako postoji) promijenimo sljedeći u `novi`, a ako prethodnog nema, pokazivač `prvi` postavimo na `novi` (u tom slučaju je `novi` prvi element u listi).

```
lista sortirano_ubaci(lista prvi, lista novi){
 /* Insertion sort za jedan element. */
 /* Ne provjerava novi != NULL. */
 lista preth, pom;
 pom = prvi;
 while (pom != NULL && pom->broj < novi->broj) {
       preth = pom;
       pom = preth->sljed; /* pom = pom->sljed;*/
```

```
9       }
10
11  /* Ispod je ubaci_iza(prvi, preth, novi) s
12  promjenom uvjeta. To radi i za prvi == NULL.*/
13
14   if (pom == prvi) { /* preth ne treba. */
15      novi->sljed = prvi;
16      prvi = novi;
17   }
18   else { /* Tu je pom == preth->sljed. */
19      novi->sljed = preth->sljed;
20      preth->sljed = novi;
21   }
22
23   return prvi; }
```

Primijetimo da `pom` uvijek pokazuje na listu sljedbenika elementa `novi`. Zato možemo pisati `novi->sljed = pom;`

Funkciju `sortirano_ubaci` možemo kraće zapisati:

```
1   lista sortirano_ubaci(lista prvi, lista novi){
2
3    lista preth, pom;
4    pom = prvi;
5    while (pom != NULL && pom->broj < novi->broj) {
6        preth = pom;
7        pom = preth->sljed; /* pom = pom->sljed;*/ }
8
9    novi->sljed = pom;
10
11   if (pom == prvi) prvi = novi;
12   else preth->sljed = novi;
13
14   return prvi; }
```

Sortirano ubacivanje jednog elementa u vezanu listu možemo koristiti kao dio algoritma za sortiranje niza podataka.

Sortiranje ubacivanjem ili Insertion sort:

- krećemo od prazne liste (koja će sadržavati sortirane elemente),

- sortirano ubacujemo jedan po jedan element iz niza kojeg treba sortirati
- postupak ponavljamo dok ne ubacimo sve elemente

Vezana lista je prikladna struktura za realizaciju algoritma zbog jednostavnosti ubacivanja elemenata bez potrebe za realokacijom memorije. U algoritmu nije bitno kako nastaju pojedini elementi sortirane liste. Možemo redom čitati brojeve (sadržaje) i kreirati pripadne elemente ili krenuti od postojeće nesortirane liste elemenata iz koje izbacujemo elemente nekim redosljedom (npr. s početka ili kraja).

Složenost sortiranja *ubacivanjem* je $\mathcal{O}(n^2)$. Zato se u praksi koristi samo za vrlo kratke liste. Postoje puno bolji algoritmi za sortiranje liste kao npr. `MergeSort`, o kojem ćemo reći nešto više u nastavku.

Prije implementacije `MergeSort` algoritma, opisat ćemo dva jednostavnija problema koja se javljaju pri izvođenju samog algoritma: a) spajanje dvije vezane liste, odnosno specifičan problem b) sortirano spajanje dvije sortirane vezane liste.

Primjer 3.6.2

Zadane su dvije vezane liste cijelih brojeva pokazivačima `prvi_1` i `prvi_2` na prvi element odgovarajuće liste. Te dvije liste treba spojiti u jednu listu, tako da druga lista bude iza prve (poredak elemenata u pojedinoj listi se ne mijenja). Ova operacija se još zove i konkatenacija po analogiji s konkatenacijom dva stringa (funkcija `strcat`). Obje polazne liste smiju biti prazne. U spojenoj listi, sljedbenik zadnjeg elementa prve liste mora biti prvi element druge liste.

Problem rješavamo slijedeći dva glavna koraka:

- pronalazimo zadnji element u prvoj listi (kao u funkciji `trazi_zadnji`)
- postavljamo njegovog sljedbenika na prvi element druge liste

Ako je prva lista prazna, rezultat je druga lista (koja također može biti prazna).

```
1 lista spoji_dvije(lista prvi_1, lista prvi_2){
2     lista pom;
3
4     if (prvi_1 == NULL) return prvi_2;
```

```
5         /* Nadji zadnjeg u prvoj i spoji drugu. */
6     for (pom = prvi_1; pom->sljed != NULL;
7                          pom = pom->sljed);
8     pom->sljed = prvi_2;
9
10    return prvi_1;
11 }
```

Konkatenaciju možemo realizirati i rekurzivnom funkcijom:

```
1  lista spoji_dvije_r(lista prvi_1, lista prvi_2){
2
3       if (prvi_1 == NULL) return prvi_2;
4       if (prvi_1->sljed == NULL) /* = zadnji_1. */
5               prvi_1->sljed = prvi_2;
6       else /* Skrati prvu listu (bez prvog).
7            Ne trebamo vrijednost funkcije! */
8           spoji_dvije_r(prvi_1->sljed, prvi_2);
9
10      return prvi_1;
11 }
```

Za dvije ulazne liste: `1 -> 3 -> 5 -> 7 -> NULL` i `2 -> 4 -> 6 -> 8 -> NULL`, lista nakon konkatenacije je: `1 -> 3 -> 5 -> 7 -> 2 -> 4 -> 6 -> 8 -> NULL`.

Ukoliko imamo dvije sortirane liste, spajanje ima smisla raditi tako da rezultantna lista ostane sortirana.

Primjer 3.6.3

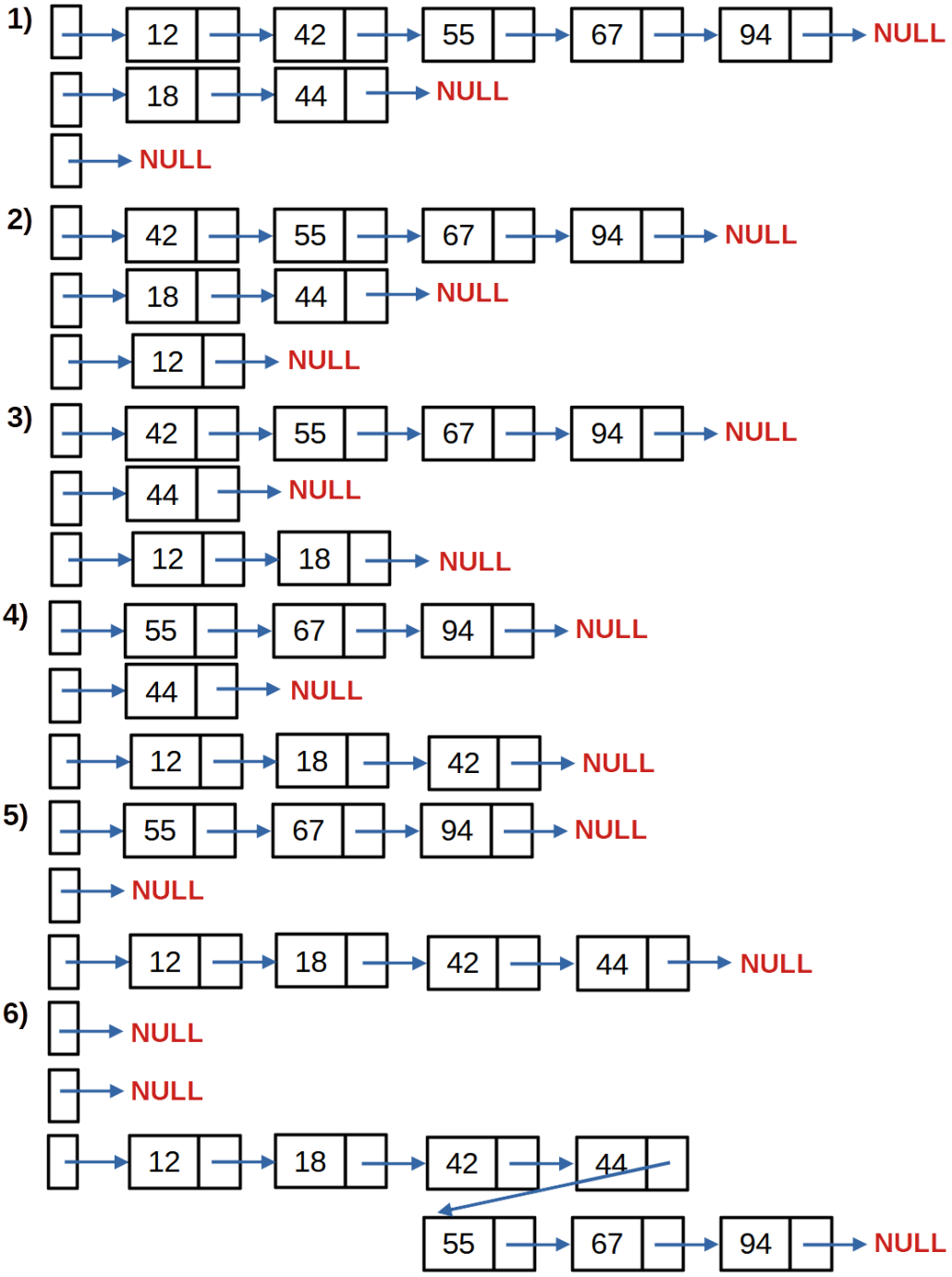
Rješavamo problem u kojem imamo dvije sortirane (uzlazno od početka prema kraju) vezane liste cijelih brojeva, zadane pokazivačima `prvi_1` i `prvi_2` na prvi element odgovarajuće liste. Te dvije liste treba spojiti u jednu listu, tako da spojena lista također bude uzlazno sortirana. Ova operacija se zove **sortirano spajanje** (eng. *merge*). Obje polazne liste smiju biti prazne.

- Ako je barem jedna od dvije liste prazna, rezultat je ona druga lista (može također biti prazna).

- U protivnom su obje liste neprazne te koristimo *pametni Insertion sort*. Na početku je spojena lista prazna. Zatim gradimo spojenu listu (element po element, tako da stalno bude uzlazno sortirana).

Iz ulaznih sortiranih listi rezultantnu sortiranu listu možemo kreirati na učinkovitiji način od korištenja funkcije `sortirano_ubaci`. Postupak je opisan u doljnjem kodu:

```
lista merge(lista prvi_1, lista prvi_2){
/* Sortirano spaja dvije liste (merge). */
      lista prvi = NULL, zadnji, pom;
/* Ako je jedna lista prazna, rezultat je druga lista. */
      if (prvi_1 == NULL) return prvi_2;
      if (prvi_2 == NULL) return prvi_1;
/* U nastavku obrade je barem jedna lista neprazna. */
/* Prolazimo po listama dok su obje neprazne. */

     while (prvi_1 != NULL && prvi_2 != NULL) {
/* Nadji manjeg od prvih iz obje liste.
Izbaci ga s pocetka njegove liste. */

         if (prvi_1->broj <= prvi_2->broj) {
             pom = prvi_1;
             prvi_1 = prvi_1->sljed;
          }
         else {
            pom = prvi_2;
            prvi_2 = prvi_2->sljed;
          }
          /* Ubaci ga na kraj sortirane liste. */
          if (prvi == NULL)
              prvi = pom;
          else
              zadnji->sljed = pom;
          zadnji = pom;
   }

/* Spoji ostatak na kraj sortirane liste. */
   if (prvi_1 == NULL) zadnji->sljed = prvi_2;
   if (prvi_2 == NULL) zadnji->sljed = prvi_1;
   return prvi; }
```



Promotrimo primjer sortiranja dvije ulazne sortirane liste: `13 -> 42 ->`

`55 -> 67 -> 94 -> NULL` i `18 -> 44 -> NULL`. Sortiranim spajanjem dobivamo listu `12 -> 18 -> 42 -> 44 -> 55 -> 67 -> 94 -> NULL`.

Slijedeći gore opisani postupak, u prvom koraku biramo najmanji element iz ulaznih listi. Pošto su ulazne liste uzlazno sortirane, najmanji element u svakoj listi je prvi element liste. Najmanji element pronalazimo izabirom manjeg elemenat između prvog elementa prve ulazne liste i prvog elementa druge ulazne liste. Pronađeni najmanji element (12 iz prve ulazne liste u našem primjeru), dodajemo u izlaznu listu i izbacujemo iz prve ulazne liste. Postupak ponavljamo dodavanjem najmanjeg prvog elementa ulaznih listi (18 iz druge ulazne liste u našme primjeru), doajemo u izlaznu listi i izbacujemo iz druge ulazne liste. Postupak ponavljamo do kraja 5) u kojem je druga ulazna lista prazna. Sada na kraj izlazne liste dodamo ostatak prve ulazne liste i dobivamo sortiranu izlaznu listu koja sadrži elemente obje ulazne liste.

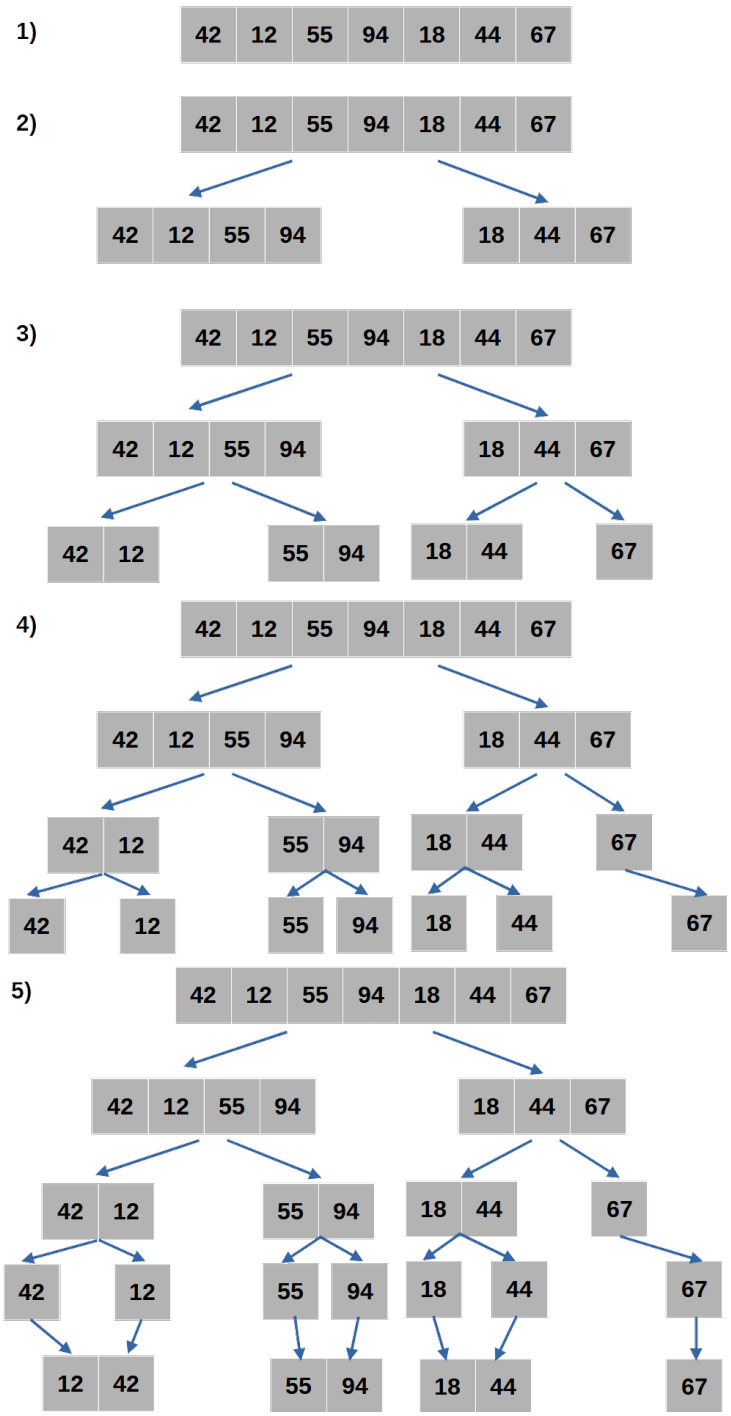
Sada znamo riješiti sve potprobleme potrebne za učinkovito rješavanje problema sortiranja vezanih listi.

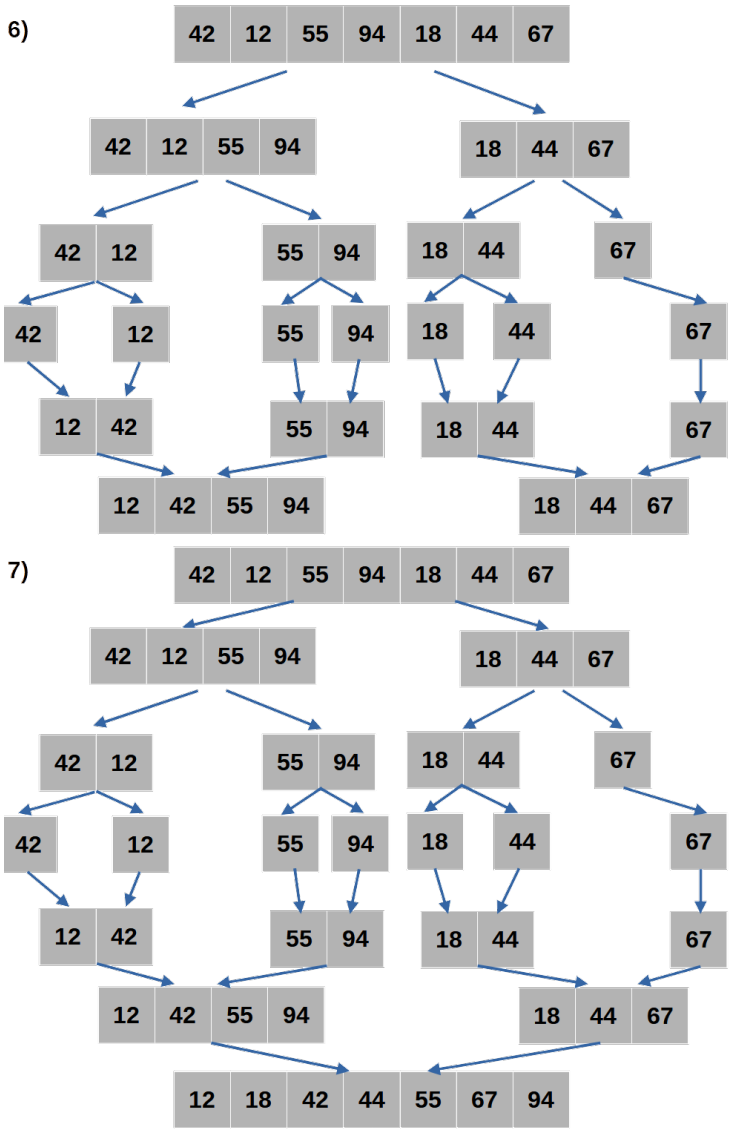
Problem: Zadana je jedna vezana lista cijelih brojeva pokazivačem `prvi` na prvi element liste. Listu treba uzlazno sortirati po sadržaju elemenata (od početka prema kraju liste) i vratiti pokazivač na početak sortirane liste. Polazna lista smije biti prazna. Sortiranje liste treba napraviti samo promjenama veza elemenata (pokazivača).

Za sortiranje liste koristimo `MergeSort` algoritam, koji je razvio **John von Neumann**, 1945. godine. To je bio **prvi** program napisan za računalo (`EDVAC`) koje sprema i podatke i programe (von Neumann-ov model računala). `MergeSort` je rekurzivni algoritam za sortiranje niza podataka, baziran na operaciji `merge` (sortiranom spajanju već sortiranih nizova podataka). Osnovna ideja `MergeSort` algoritma je: a) podijeliti nesortirani niz (listu) podataka na dva dijela (podniza) podjednake duljine, b) rekurzivno sortirati svaki od nastalih podnizova, c) sortirano spojimo (`merge`) ta dva već sortirana podniza u jedan sortirani niz. Rekurzivno raspolavljanje i sortiranje radimo sve dok ne dobijemo trivijalni niz (prazan ili jednočlan). Pošto su trivijalni nizovi već sortirani, jedino što trebamo napraviti je vratiti pokazivač na takve liste. Primjenom sortiranog spajanja od trivijalnih nizova (lista) dobijemo sortirane liste veće duljine, na koje opet primijenimo postupak sortiranog spajanja.

Demonstrirat ćemo ideju `MergeSort` algoritma na primjeru soriranja liste:

42 -> 12 -> 55 -> 94 -> 18 -> 44 -> 67.





Raspolavljanje liste možemo realizirati na dva načina: a) preko pokazivača, pažljivim iteriranjem po listi (elegantnije ali sporije), b) brojanjem elemenata u listi (brže, potreban dodatan ulazni argument).

Složenost `MergeSort` algoritma u najgorem slučaju je $\mathcal{O}(n\ log(n))$.

Dokaz slijedi iz prethodne slike. Neka je n broj elemenata u nizu i neka je $2^{k-1} < n \leq 2^k$ (2^k je najmanja potencija broja 2 veća ili jednaka n), $k = \lceil log_2 n \rceil$. Na slici je $n = 7$ i $k = 3$.

Uzmimo da polazni, nesortirani niz, ima razinu (nivo) 0. Nakon toga, imamo točno k horizontalnih razina raspolavljanja podnizova i sortiranog spajanja podnizova. Broj razina je $2k = 2log_2 n$.

Svako raspolavljanje i svaki `merge` traju linearno u duljini većeg odgovarajućeg niza (svaki element u većem niz prolazimo najviše jednom). Zbrajanjem operacija na istoj razini (imamo ih maksimalno n) zaključujemo da na svakoj razini imamo $\mathcal{O}(n)$ operacija. Množenjem broja operacija na razini i broja razina, dobivamo složenost algoritma $\mathcal{O}(n\ log(n))$.

```
1  lista merge_sort(lista prvi){
2  /* Sortira listu Merge_Sort algoritmom. */
3       lista zadnji, prvi_2;
4  /* Test na praznu ili jednoclanu listu. */
5       if ((prvi == NULL) || (prvi->sljed == NULL))
6            return prvi;
7  /* U nastavku lista ima bar dva elementa. */
8  /* Raspolovi listu. Pokazivac zadnji pokazuje
9  na zadnjeg u prvom dijelu. Pokazivac prvi_2 je
10 pomocni i sluzi za raspolavljanje liste. */
11      zadnji = prvi;
12      prvi_2 = prvi->sljed;
13
14 /* Pomicemo zadnjeg za JEDNO mjesto, prvi_2 za
15 DVA mjesta, dok prvi_2 ne bude na kraju liste. */
16      while ((prvi_2 != NULL) &&
17            (prvi_2->sljed != NULL)) {
18              zadnji = zadnji->sljed;
19              prvi_2 = prvi_2->sljed->sljed;
20         }
21 /* Pokazivac zadnji sad korektno pokazuje na
22 zadnjeg u prvom dijelu. Pokazivac prvi_2 zadamo
23 kao prvog u drugom dijelu (prvi iza zadnjeg) i
24 korektno zavrsavamo prvi dio. */
25     prvi_2 = zadnji->sljed;
26     zadnji->sljed = NULL;
27 /* Rekurzivno sortiranje i merge. */
28     prvi = merge(merge_sort(prvi),
29     merge_sort(prvi_2));
30    return prvi; }
```

Za zadanu ulaznu listu brojeva `42 -> 12 -> 55 -> 94 -> 18 -> 44 -> 67 -> NULL`, nakon poziva funkcije `MergeSort` dobivamo listu `12 -> 18 -> 42 -> 44 -> 55 -> 67 -> 94 -> NULL`.

Zadatak: Napišite funkciju za `MergeSort` liste koja kao argument prima broj elemenata u listi.

Zadatak: Napišite funkciju za `MergeSort` na polju. Za `merge` (spajanje) smijete koristiti jedno dodatno pomoćno polje! Probajte koristiti što manje kopiranja iz jednog polja u drugo.

Primjer 3.6.4

Iz vezane liste cijelih brojeva, zadane pokazivačem `prvi` na prvi element, želimo obrisati prvi element s parnim brojem (kao sadržajem). Ako takvog elementa nema, lista se ne mijenja. Polazna lista smije biti prazna.

Opet koristimo princip *izbaci* ili *obriši iza nekog* kao u funkciji `obrisi_iza`.

```
1  lista obrisi_prvi_parni(lista prvi){
2
3        lista preth, pom;
4        if (prvi == NULL) return NULL;
5        pom = prvi;
6
7        while (pom != NULL && pom->broj % 2 != 0) {
8                  preth = pom;
9                  pom = preth->sljed;}
10
11       if (pom != NULL) {
12          if (pom == prvi)
13             prvi = prvi->sljed;
14          else
15            preth->sljed = pom->sljed;
16         free(pom);
17         return prvi;
18       }
```

Za ulaz: `2->4->5->6->8->9->10->NULL` i 6 poziva funkcije `obrisi_prvi_parni` dobijemo listu `5->9->NULL`. Zadnji poziv funkcije **ne mijenja** listu jer u tom trenutku lista više nema parnih elemenata.

Zadaci:

1. Napišite funkciju `izbaci_iza` sa zaglavljem: `lista izbaci_iza(lista prvi, lista preth, lista *p_izbacen);`

   Funkcija treba iz liste zadane pokazivačem `prvi`, koji pokazuje na prvi element, izbaciti element koji se nalazi iza elementa na kojeg pokazuje `preth`. Za razliku od funkcije `obrisi_iza`, izbačeni element se ne briše već treba vratiti pokazivač na njega kroz varijabilni argument `*p_izbacen` (slično kao u funkciji `ubaci_na_kraj`).

   Vrijednost funkcije je **pokazivač** na **prvi** element dobivene liste.

   Funkcija treba raditi u svim slučajevima:

   - `prvi == NULL` - ne radi ništa, vraća `NULL` i `*p_izbacen = NULL`,
   - `preth == NULL` - izbacuje **prvi** element liste
   - `preth == zadnji`, tj. `preth->sljed == NULL` - ne izbacuje **ništa** i vraća `*p_izbacen = NULL`.

2. Vezana lista cijelih brojeva zadana je pokazivačem `prvi` na prvi element. Treba napisati funkciju koja rastavlja tu listu u dvije liste, tako da prva lista sadrži samo elemente s parnim sadržajem, a druga lista sadrži samo elemente s neparnim sadržajem iz polazne liste. Funkcija treba vratiti pokazivače na te dvije liste kroz varijabilne argumente (ili vrijednost i varijabilni argument).

   Varijacije:

   - relativni poredak elemenata u dobivenim listama smije biti bilo koji - obratan od polaznog
   - mora biti isti kao u polaznoj listi

   Rastav liste treba napraviti samo promjenama veza elemenata (pokazivača), **zabranjeno je mijenjati sadržaj elemenata, alocirati i dealocirati dodatnu memoriju, tj. koristiti pomoćne elemente, polja itd.** Navedeno vrijedi u svim zadacima s vezanim listama, uključujući kolokvije (osim ako je u zadatku navedeno drugačije).

3. Vezana lista cijelih brojeva zadana je pokazivačem `prvi` na prvi element. Napišite funkciju koja preuređuje tu listu, tako da na početku

liste budu svi elementi s parnim sadržajem, a na kraju liste (iza svih parnih) budu svi elementi s neparnim sadržajem iz polazne liste. Funkcija treba vratiti pokazivač na prvi element preuređene liste. Preuređenje liste treba napraviti samo promjenama veza elemenata (pokazivača).

Varijacije: relativni poredak elemenata u parnom i neparnom dijelu dobivene liste

- može biti proizvoljan
- mora biti isti kao u polaznoj listi

Unutar funkcije smijete koristiti rastav liste u dvije liste, a onda spajanje dvije liste - konkatenacija.

Probajte zadatak riješiti bez toga, tj. unutar jedne liste.

4. Vezana lista brojeva zadana je pokazivačem `prvi` na prvi element. Napišite funkciju `okreni_listu`: `lista okreni_listu(lista prvi);` koja preuređuje tu listu tako da cijelu listu okreće naopako, tj. invertira poredak elemenata u listi.

   Funkcija treba vratiti pokazivač na prvi element okrenute liste (to je zadnji element u polaznoj listi, ako postoji). Invertiranje liste treba napraviti **samo promjenama veza elemenata (pokazivača)**.

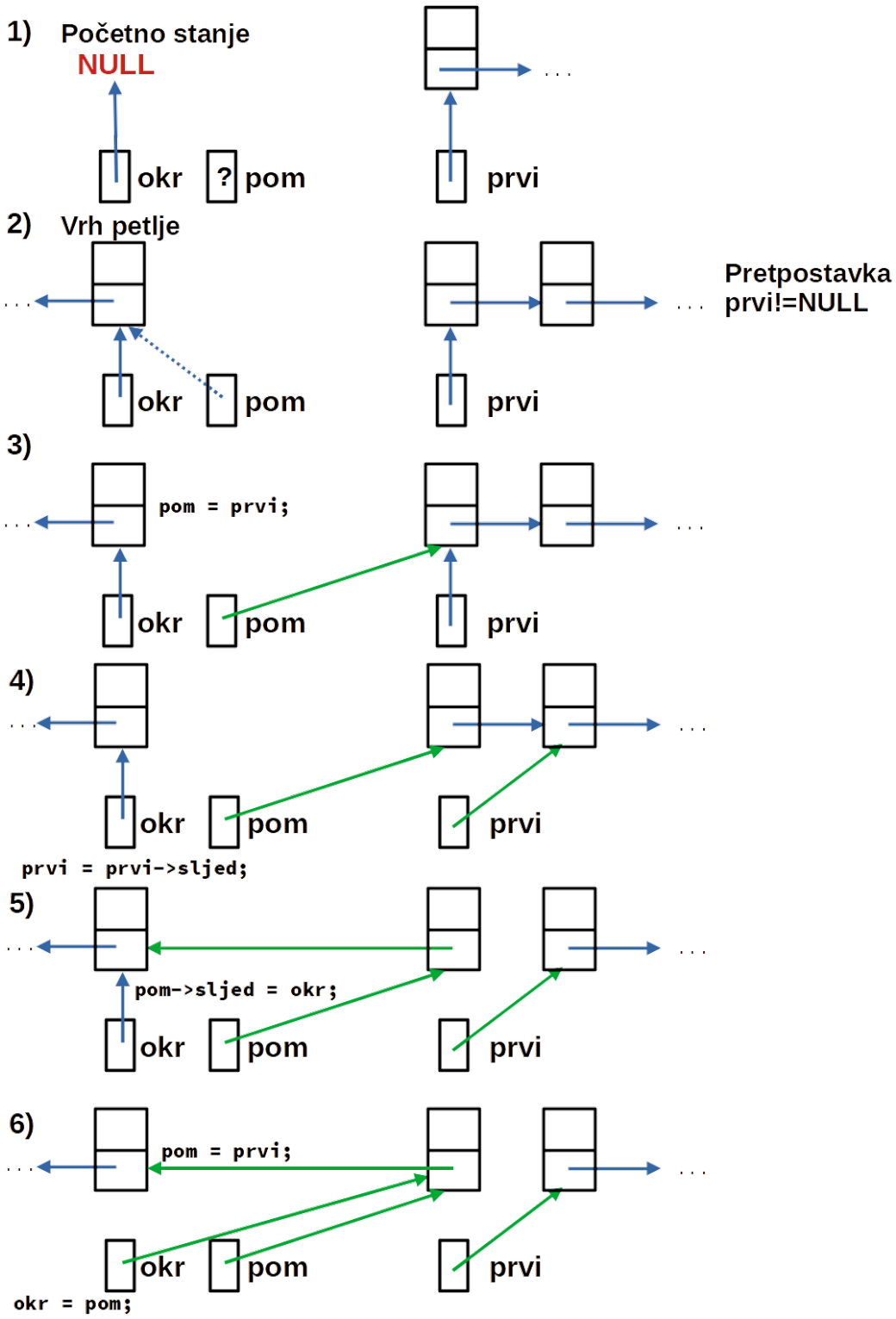
Primjer 3.6.5

Zadana je vezana lista pokazivačem `prvi` na prvi element. Napišite funkciju koja preuređuje tu listu tako da cijelu listu okreće naopako, tj. invertira poredak elemenata u listi. Funkcija treba vratiti pokazivač na prvi element okrenute liste (zadnji element u polaznoj listi, ako postoji). Invertiranje liste treba napraviti samo promjenama veza elemenata (pokazivača).

Na početku je okrenuta lista prazna (`okr = NULL`). Prolazimo kroz zadanu listu i u svakom koraku:

- izbacimo prvi element iz preostale neokrenute liste
- ubacimo ga na početak okrenute liste (zadane s `okr`).

Postupak ponavljamo dok preostala ulazna lista nije prazna.


1)
Početno stanje
NULL
okr
?
pom
prvi
2)
Vrh petlje
okr
pom
prvi
Pretpostavka
prvi!=NULL
3)
pom = prvi;
okr
pom
prvi
4)
okr
pom
prvi
prvi = prvi->sljed;
5)
pom->sljed = okr;
okr
pom
prvi
6)
pom = prvi;
okr
pom
prvi
okr = pom;

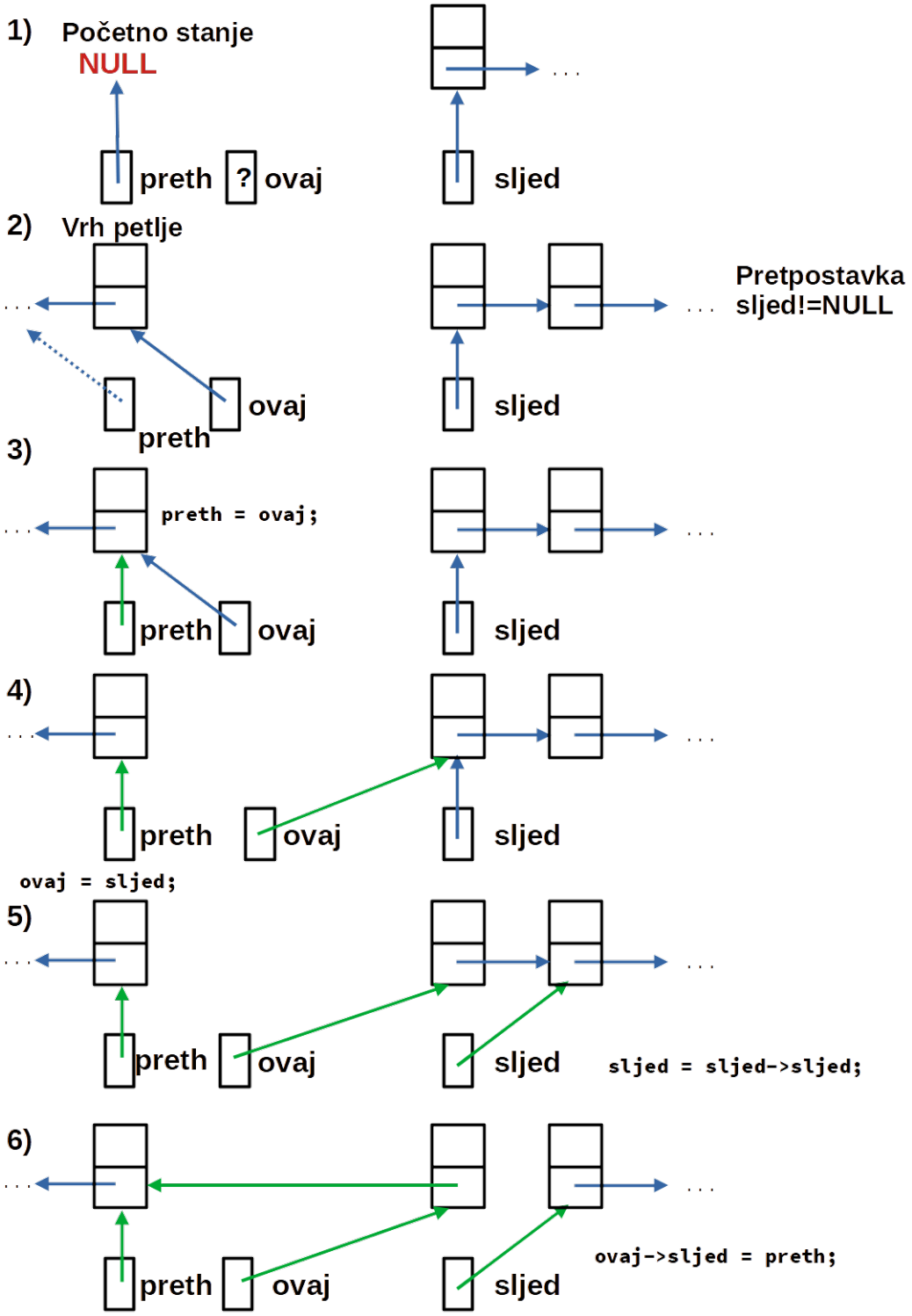
```
lista okreni_listu(lista prvi){

      lista okr = NULL, pom;

      while (prvi != NULL) {
/* Izbaci s pocetka stare. */
           pom = prvi;
           prvi = prvi->sljed;
/* Ubaci na pocetak okrenute. */
           pom->sljed = okr;
           okr = pom;
        }
      return okr;
  }
```

Problem možemo riješiti i prozorom od nekoliko susjednih pokazivača koji prolaze kroz listu. U svakom koraku pomaknemo prozor za jedno mjesto unaprijed i okrenemo (usmjerimo pokazivač `sljed` u drugu stranu) jedan element u tom prozoru. Dovoljna su tri pokazivača:

- `preth` - pokazuje na prethodno okrenutu listu, u trenu kad ubacujemo novi element (pomoćni pokazivač)
- `ovaj` - pokazuje na element kojeg prebacujemo, to će biti (novi) početak okrenute liste
- `sljed` - pokazuje na početak preostalih elemenata originalne liste

Okretanje (izbacivanje/ubacivanje) odgovara naredbi: `ovaj->sljed = preth`.

```
lista okreni_listu(lista prvi){
      lista preth, ovaj = NULL, sljed = prvi;

      while (sljed != NULL) {
           preth = ovaj;
           ovaj = sljed;
           sljed = sljed->sljed;
           ovaj->sljed = preth; }

      return ovaj; }
```


1)
Početno stanje
NULL
preth
?
ovaj
sljed
2)
Vrh petlje
Pretpostavka
sljed!=NULL
ovaj
preth
sljed
3)
preth = ovaj;
preth
ovaj
sljed
4)
preth
ovaj
sljed
ovaj = sljed;
5)
preth
ovaj
sljed
sljed = sljed->sljed;
6)
preth
ovaj
sljed
ovaj->sljed = preth;

`sljed` možemo pamtiti i u varijabli `prvi`. Time dobijemo sličan algoritam kao u prvom rješenju.

```
1  lista okreni_listu(lista prvi){
2
3        lista preth, ovaj = NULL;
4
5        while (prvi != NULL) {
6            preth = ovaj;
7            ovaj = prvi;
8            prvi = prvi->sljed;
9            ovaj->sljed = preth;
10        }
11
12       return ovaj;
13 }
```

Poglavlje 4

Trajna memorija i dodaci

U ovom poglavlju ćemo naučiti kako koristiti datoteke spremljene u trajnu memoriju (tvrdi disk, SSD). Za razliku od radne memorije (RAM) i registara procesora iz kojih se informacije brišu nakon gašenja računala, podaci spremljeni u trajnu memoriju ostaju trajno sačuvani. Naučit ćemo raditi s dvije vrste datoteka: a) tekstualnim datotekama - koje su formatirane i stoga pogodne čitanju od strane ljudi, ali zahtjevaju vremenski skuplje računarske operacije i b) binarne datoteke - koje se zapisuju kao nizovi znakova (bajtova) i nisu čitljivi ljudima, ali omogućavaju relativno brze operacije od strane računala. Naučit ćemo kako mjeriti vrijeme izvršavanja C-programa, što će omogućiti izvođenje empirijskih analiza algoritama, koje za razliku od teorijskih izračuna vremenske složenosti ovise o komponentama računala, operacijskom sustavu i trenutnom opterećenju procesora. Detaljnije ćemo se upoznati s radom pretprocesora i opisati strukturu standardne C biblioteke s naglaskom na matematičku biblioteku sa zaglavljem `math.h`.

4.1 Datoteke

Naučili smo da računalo sadrži hijerarhiju privremenih i trajnih memorija koje služe sa spremanje međurezultata, pomoćnih informacija potrebnih za računanje i konačnih izlaza programa. Od privremenih memorija smo upoznali *cache* memoriju procesora (malu ali izuzetno brzu i kvalitetnu memoriju) koja služi za spremanje informacija potrebnih za procesiranje trenutnog izračuna i radnu (RAM) memoriju koja je veća, sporija i često manje kvalitetna od *cache* memorije proesora i služi za spremanje međurezultata, pomoćnih i

privremenih rezultata potrebnih za uspješno izvršavanje programa. Privremena memorija je konstruirana na način da se sve informacije zapisane u nju brišu kod gašenja računala, a kod prekida izvršavanja programa operacijski sustav pokrene čišćenje zazete memorije. Informacije koje želimo trajno spremiti, trebamo pohraniti u trajnu memoriju. Tipični mediji za spremanje podataka (zastarjeli/ne koriste se, izlaze iz upotrebe/povremeno se koriste, novi/koriste se) su trake, kazete, disketa, optički mediji (CD, DVD), flash-memorije, disk (HDD, SSD) itd. Svaka od navedenih tehnologija za spremanje podataka ima određene nedostatke, dok je kod većine tehnologija problem mogućnost gubitka podataka kod dugotrajnog skladištenja. Iz tog razloga i zbog jednostavnosti korištenja, danas je jedan od češćih načina trajne pohrane informacija, pohrana u oblaku (eng. *cloud*). Kod tog načina pohrane se podaci spremaju u podatkovni centar o kojem se brine određena institucija ili privatna kompanija. Institucija kopira podatke na niz uređaja za trajno skladištenje podataka i kroz vrijeme mijenja istrošene uređaje novima čuvajući integritet podataka kontinuiranim višestrukim kopiranjem podataka. Korisnik za spremanje podatka uglavnom plaća naknadu i za pristup podacima mora imati vezu s Internetom[1]. Informacije se na uređajima za trajno spremanje podataka spremaju u obliku datoteka (cjelina bajtova koje sadrže spremljenu informaciju). Operacijski sustav računala koristi sustav datoteka (*file-system*) za organizaciju datoteka u računalu. Unutar tog sustava datoteka, svaka datoteka ima svoje ime koje koristimo za pristup datoteci. Prednost upotrebe datoteka pri programiranju je što se jedna datoteka može koristiti od strane više programa (omogućava ponovnu upotrebu ili komuniciranje između različitih programa), omogućava učitavanje ulaznih podataka pri testiranju programa (eliminira potrebu za konstantnim učitavanjem ulaza) i omogućava trajan zapis izlaza (rezultata) nekog programa.

Pravila za pisanje (tvorbu) imena datoteka specifična su za pojedini operacijski sustav. To posebno vrijedi za puno ime datoteke, koje sadrži i putanju (*path*) do te datoteke u sustavu datoteka. Osnovno ime datoteke (bez putanje) do nje, standardno ima oblik `ime.ekstenzija` (Unix, Windows). Ekstenzija označava vrstu sadržaja datoteke.

- `sort.c` - izvorni kod C programa,
- `math.lib` - biblioteka prevedenih funkcija pripremljena za linker,

[1] `https://www.enciklopedija.hr/clanak/internet`

- `sort.exe` - izvršni (binarni) kod programa (Windows).

Točka na početku imena datoteke na Unix sustavu označava skrivenu datoteku. Osnovne operacije s datotekama su (iz perspektive programa koji obrađuje datoteku):

- čitanje podataka iz datoteke - ulaz podataka u program
- pisanje podataka u datoteku - izlaz podataka iz programa

Postoje dvije podjele datoteka, prema tome što se događa u gore navedenim operacijama:

- po načinu pristupa podacima u datoteci: a) slijedne (sekvencijalne), b) direktne datoteke
- po interpretaciji sadržaja podataka u datoteci pri čitanju i pisanju: a) formatirane, b) neformatirane.

Postoje dva bitno različita načina pristupa podacima u datoteci pri čitanju i pisanju (dva načina realizacije ovih operacija).

- Slijedni ili sekvencijalni pristup čita i piše samo u jednom smjeru (unaprijed), podatak za podatkom, kao na traci. To je standardni način pristupa podacima u C-u.
- Direktnim pristupom čitamo i pišemo bilo gdje u datoteci, slično kao u polju. Realizira se posebnim funkcijama za pozicioniranje u datoteci.

Neki podatak, npr. cijeli broj, možemo zapisati u datoteku na dva bitno različita način:

- **formatirano** - u obliku **tekstualnog zapisa** podatka, kao da pišemo funkcijom `printf`,
- **neformatirano** - u obliku **interne reprezentacije** tog podatka u računalu (kopiranjem sadržaja memorije koju taj podatak zauzima).

Identična stvar vrijedi i kod čitanja. Stoga, po načinu zapisa ili po interpretaciji sadržaja, datoteke možemo podijeliti na **formatirane** i **neformatirane**. Obje vrste zapisa možemo realizirati u C-u odgovarajućim funkcijama za **čitanje** i **pisanje** (može i na istoj datoteci). Iako u C-u nema izravne podjele na formatirane i neformatirane datoteke, po ANSI standardu postoje dvije vrste datoteka: **tekstualne** i **binarne**.

- **Binarna datoteka** - niz podataka tipa `char` (niz znakova).
- **Tekstualna datoteka** ima dodatnu strukturu, kao tekst. Reprezentira se kao niz znakova podijeljenih u linije (redove), a svaka linija sadrži nula ili više znakova, iza kojih slijedi znak `\n` za kraj linije (reda).

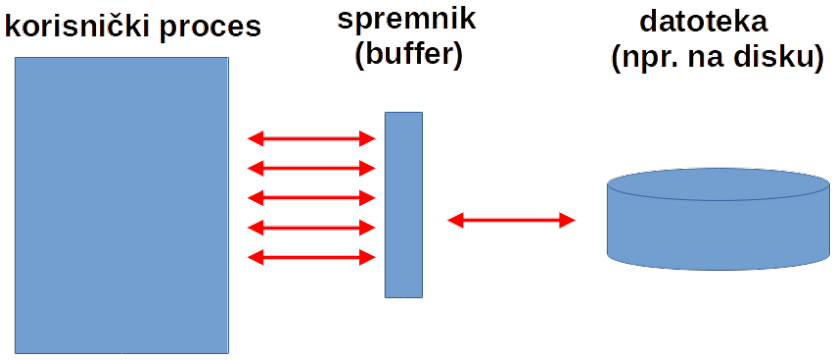
Razlika između binarnih i tekstualnih datoteka ovisi samo o standardnoj oznaci za kraj linije u odgovarajućem operacijskom sustavu.

- Unix - znak `\n` (line feed ili newline).
- Windows - dva znaka: `\r` (carriage return) i `\n`.
- Mac OS - znak `\r`.

Funkcije koje prepoznaju kraj linije (poput `fgets` i `fputs`) korektno pretvaraju standardni kraj linije (u datoteci) u znak `\n` (u C programu) i obratno. Zato na Unixu nema nikakve razlike između binarnih i tekstualnih datoteka, a na ostalim navedenim sustavima ima.

Sve operacije s datotekama u C-u, uključujući i ulazno-izlazne, tj. stvarno čitanje i pisanje podataka, realizirane su standardnim funkcijama, deklariranim u standardnoj datoteci zaglavlja `<stdio.h>`. Te funkcije su visoka razina ulazno-izlaznih operacija, jer skrivaju niz detalja vezanih uz konkretni operacijski sustav i lako se prenose s jednog sustava na drugi.

Prije upotrebe tih funkcija, trebamo razumjeti svrhu i način rada spremnika (*buffer*-a). Stvarna komunikacija između korisničkog programa i datoteke vrši se preko posebnog prostora u memoriji računala kojeg zovemo spremnik (*buffer*). Razlog je razlika u brzini između centralnih dijelova računala (procesor, memorija) i vanjskih ulazno-izlaznih jedinica (npr. disk). Posljedica hijerarhijske građe memorije. U spremnik se privremeno pohranjuju sve informacije koje se šalju u datoteku ili primaju iz datoteke.



Svrha spremnika je: a) smanjiti komunikaciju s vanjskom memorijom (npr. diskom) na način da transfer podataka ide u blokovima, b) povećati efikasnost ulazno–izlaznih funkcija. Ovaj spremnik je dio operacijskog sustava za rad s datotekama. Većina ulazno-izlaznih uređaja ima još i svoj vlastiti spremnik (katkad i *cache*) s identičnom svrhom (tzv. *double-buffer* komunikacija).

4.1.1 Građa datoteke i osnovne funkcije za rad s datotekama

Standardna datoteka zaglavlja `<stdio.h>` sadrži deklaraciju strukture posebnog tipa koji se zove `FILE`. U strukturi tipa `FILE` opisan je spremnik i svi ostali podaci potrebni za komunikaciju s datotekom, koji ovise o operacijskom sustavu. Ovu strukturu katkad isto zovemo **spremnik** ili **file buffer** po jednom dijelu njezinog sadržaja. Tu se nalaze svi detalji implementacije datoteka koje korisnici ne moraju znati.

Struktura `FILE` sadrži: a) osnovne informacije o datoteci, b) vrstu operacije, čitanje ili pisanje (tzv. `file_mod`), c) status operacija koji pamti je li došlo do greške ili smo došli do kraja datoteke (`ferror`, `feof`), d) stvarnu trenutnu poziciju u datoteci koja pamti poziciju gdje ide sljedeće čitanje ili pisanje (`ftell`) dok nula označava početak datoteke, e) stvarnu lokaciju spremnika za komunikaciju, f) trenutnu poziciju u spremniku koja pamti koju informaciju trebamo učitati iz spremnika ili upisati iz spremnika u datoteku.

Svakoj datoteci s kojom radimo u programu pridružen je odgovarajući objekt tipa `FILE`. To je spremnik za tu datoteku. Pošto je on dinamički objekt, do njega dolazimo preko pokazivača (tzv. *file pointer*). Deklaracija pripadnog pokazivača na `FILE` se radi naredbom `FILE *fp;`. Na početku rada s datotekom, *file pointer* moramo kreirati operacijom otvaranja datoteke, funkcijom `fopen`. Tom naredbom se kreira pripadni spremnik (alocira se memorija) i uspostavlja komunikacija sa stvarnom datotekom u operacijskom sustavu. Na kraju rada s datotekom, moramo osloboditi memoriju za spremnik operacijom zatvaranja datoteke, funkcijom `fclose`. Datoteka **mora biti otvorena** prije prve operacije pisanja ili čitanja.

Otvaranje datoteke vrši se pozivom funkcije `fopen FILE *fopen(const char *ime, const char *tip);` gdje je:

- `ime` - pravo ime datoteke koja se otvara (string)

- `tip` - string koji kaže kako treba otvoriti tu datoteku (način rada ili `file_mod`)

Funkcija `fopen` vraća:

- Pokazivač na strukturu `FILE`, povezanu s tom datotekom ako je datoteka uspješno otvorena
- `NULL`, ako datoteka nije mogla biti otvorena (greška).

Nakon otvaranja datoteke treba uvijek provjeriti vraćeni pokazivač.

```
1 FILE *fp;
2 ...
3 fp = fopen(ime, tip);
4 if (fp == NULL) { /* Reakcija na gresku. */
5 printf("Greska u otvaranju datoteke!\n");
6 ...
7 }
```

`ime` je pravo ime datoteke (ime u operacijskom sustavu), npr. *podaci.dat*. Drugi string `tip` je jedan od predefiniranih stringova oznake tipa.

Za otvaranje tekstualne datoteke, tj. za tekstualni način rada s datotekom koriste se sljedeći tipovi:

- `r` - otvaranje postojeće datoteke samo za čitanje,
- `w` - kreiranje nove datoteke samo za pisanje,
- `a` - otvaranje postojeće datoteke za dodavanje teksta,
- `r+` - otvaranje postojeće datoteke za čitanje i pisanje,
- `w+` - kreiranje nove datoteke za čitanje i pisanje,
- `a+` - otvaranje postojeće datoteke za čitanje i dodavanje teksta.

`r = read`, `w = write`, `a = append` (dodavanje na kraj).
Kod tipova za otvaranje datoteka vrijede sljedeća pravila:

- Čitanje (`r` ili `r+`) očekuje postojeću datoteku, ne kreira novu (greška).
- Pisanje (`w` ili `w+`) briše sadržaj postojeće datoteke (ukoliko postoji) i pisanje počinje od početka.

- Dodavanje (`a` ili `a+`) kreira datoteku ukoliko ona ne postoji i pisanje kreće od početka, ako datoteka postoji novi tekst će biti dodan na kraj te datoteke.

Za otvaranje binarne datoteke, tj. za binarni način rada s datotekom, u odgovarajući tekstualni tip treba dodati slovo `b`.

- `rb` - binarno čitanje iz postojeće datoteke,
- `wb` - binarno pisanje, kreiranje nove binarne datoteke,
- `ab` - binarno dodavanje,
- `rb+` ili `r+b` - binarno čitanje i pisanje iz postojeće binarne datoteke,
- `wb+` ili `w+b` - binarno čitanje i pisanje, kreiranje nove binarne datoteke,
- `ab+` ili `a+b` - binarno čitanje i dodavanje.

Unix ima samo jedan tip datoteka (binarno = tekstualno).

Na kraju rada s datotekom, datoteku treba obavezno zatvoriti pozivom funkcije `fclose`. U funkciji `int fclose(FILE *fp);`, `fp` označava pokazivač na strukturu `FILE`, povezanu s tom datotekom (pripadni spremnik).

Funkcija `fclose` vraća:

- nulu, ako je datoteka uspješno zatvorena,
- `EOF` u slučaju greške.

Zatvaranje datoteke je nužno, u protivnom kod pisanje može doći do gubitka podataka, ako program završi greškom. Moguće je i da se dio sadržaja datoteke ne učita iz spremnika datoteke.

Funkcija `fclose` radi sljedeće:

- prazni spremnik i ako treba piše u datoteku ono što dotad nije napisano iz spremnika,
- završava komunikaciju s datotekom u operacijskom sustavu,
- oslobađa memoriju za spremnik.

Funkcija `fclose` se primijenjuje na pokazivač na datoteku `fp` kao `fclose(fp);` Nakon poziva funkcije `fclose`, pokazivač `fp` se ne mijenja, ali pokazuje na dealociranu memoriju, stoga je korektno taj pokazivač postaviti na `NULL` (kao što to radimo i nakon poziva funkcije `free`).

Prikazujemo programski isječak koji otvara za pisanje i zatvara datoteku `primjer.dat`.

```
#include <stdio.h>
#include <stdlib.h>
...
FILE *fp;

if ((fp = fopen("primjer.dat", "w")) == NULL) {
          printf("Ne mogu otvoriti datoteku!\n");
          exit(EXIT_FAILURE); /* exit(1); */
}
/* Rad s datotekom (pisanje u nju). */
...
fclose(fp);
```

4.1.2 Standardne datoteke, veza standardnog ulaza i datoteka, primjeri rada s tekstualnim datotekama

Svakom C programu u trenutku izvršavanja stoje na raspolaganju tri standardne, od operacijskog sustava automatski otvorene datoteke:

- **standardni ulaz** - tipkovnica računala,
- **standardni izlaz** - ekran računala,
- **standardni izlaz za greške** - ekran računala.

U datoteci zaglavlja `<stdio.h>` deklarirani su konstantni pokazivači na `FILE` strukture, povezane s tim datotekama. Ti pokazivači imaju sljedeća imena:

- `stdin` - za standardni ulaz,
  emphitem stdout - za standardni izlaz,
- `stderr` - za standardni izlaz za greške.

Neki operacijski sustavi (*Unix*, *DOS*, *Windows*, ...) imaju mogućnost preusmjeravanja datoteka (*redirection*). Pri pozivu programa, na komandnoj liniji, možemo standardne datoteke `stdin` i `stdout` preusmjeriti na neke druge datoteke.

```
demo <demo.in >demo.out
```

Znak < preusmjerava `stdin` na datoteku `demo.in`, pa se čitanje vrši iz datoteke `demo.in`. Znak > preusmjerava `stdout` na datoteku `demo.out`, pa se pisanje vrši u datoteku `demo.out`.

Standardni ulaz/izlaz	Datoteke
`getchar`	`fgetc`, `getc`
`putchar`	`fputc`, `putc`
`gets`	`fgets`
`puts`	`fputs`
`printf`	`fprintf`
`scanf`	`fscanf`

Funkcionalnost funkcija za čitanje iz standardnog ulaza i pisanje na standardni izlaz se može ostvariti direktnim ili blago modificiranim pozivom generalnijih funkcija za čitanje iz i pisanje u datoteke, dok su neke funkcije kao `getchar` i `putchar` implementirane pozivom odgovarajućih funkcija `getc`, odnosno `putc`. Više o tome ćemo reći u nastavku teksta. Tablica iznad navodi funkcije za čitanje sa standardnog ulaza i pisanje u standardni izlaz te analogne funkcije za rad s datotekama. Sve funkcije u desnom stupcu kao argument primaju pokazivač na `FILE`. To je zadnji argument u prva četiri reda, a prvi argument za zadnje dvije funkcije.

Deklaracija (prototip) funkcija `fgetc` i `getc` je `int fgetc(FILE *fp);` i `int getc(FILE *fp);`. Navedene funkcije vraćaju:

- sljedeći znak iz datoteke na koju pokazuje `fp`,
- `EOF` u slučaju greške ili kraja datoteke.

Vraćeni znak je tipa `unsigned char`, pretvoren u `int`. `EOF` je simbolička konstanta definirana u `<stdio.h>`. Najčešće je `EOF = -1` (ne smije biti legalni znak u datoteci) i zato je izlazni tip `int`. Razlika između `fgetc` i `getc` je da se `getc` može implementirati i kao makro naredba dok isto nije

moguće za `fgetc`. `getc` treba oprezno upotrebljavati ako se definira kao makro naredba jer postoji mogućnost višestrukog evaluiranja argumenta `fp`. Funkcija `getchar()` za standardni ulaz **implementira se kao** `getc(stdin)`. Obrada datoteke čitanjem znak po znak, tipično se radi na sljedeći način:

```
FILE *fp; int ch; /* Ne: char ch! */
if ((fp = fopen("podaci.txt", "r")) == NULL) {
        printf("Ne mogu otvoriti datoteku!\n");
        exit(EXIT_FAILURE); /* exit(1); */
}

/* Obrada datoteke - znak po znak). */
while ((ch = fgetc(fp)) != EOF) {
... /* Obradi znak ch. */
}
...
fclose(fp);
```

U gornjem kodu radimo s tekstualnom datotekom, međutim identičan kod (do na otvaranje datoteke) se može koristiti za rad s binarnom datotekom.

Primjer 4.1.1

Pišemo program koji broji znakove u datoteci. Ime datoteke zadaje se kao argument komandne linije.

```
#include <stdio.h>
#include <stdlib.h>
/* Broj znakova u datoteci. */
int main(int argc, char *argv[]){
 FILE *fp;
 int ch, brojac = 0;

 if (argc == 1) { /* Nema imena datoteke! */
    printf("Uporaba: %s ime\n",argv[0]);
    exit(1); }

 if ((fp = fopen(argv[1], "r")) == NULL) {
    printf("Neuspjesno otvaranje %s!\n",argv[1]);
    exit(2); }

 while ((ch = fgetc(fp)) != EOF) ++brojac;
```

```
17          fclose(fp);
18  printf("Broj␣znakova␣%s:␣%d\n",argv[1], brojac);
19  return 0; }
```

Zbog razlika u označavanju kraja reda na različitim operacijskim sustavima, možemo dobiti nešto drugačiji broj znakova. Funkcija `fgetc` će, kada datoteku čitamo kao tekstualnu, dva znaka `\r`, `\n`, koji označavaju kraj reda na Windowsima, pretvoriti u znak `\n`. Ukoliko datoteku promatramo i otvorimo kao binarnu, tada navedene konverzije nema pa dobijemo isti rezultat bez obzira na korišteni operacijski sustav.

Funkcija `ungetc`, s prototipom `int ungetc(int c, FILE *fp);`, vraća znak c (pretvoren u `unsigned char`) natrag u spremnik iz `FILE` strukture na koju pokazuje `fp` i čini taj znak dostupnim za ponovo čitanje. Taj znak će se ponovo pročitati kod sljedećeg čitanja. Izlazna vrijednost funkcije je:

- c ako je uspješno vraćen u spremnik,
- `EOF` u slučaju greške.

Po standardu, u svakom trenutku dozvoljeno je vratiti najviše jedan znak (različit od `EOF`) u spremnik za datoteku. Vraćanje unatrag u spremniku za čitanje je često korisno kod obrade gramatički strukturiranog teksta (npr. riječi). Kraj neke vrste takvih objekata prepoznamo učitavanjem prvog znaka sljedećeg objekta. Tada, umjesto da posebno pamtimo suvišno učitani znak, vratimo se za poziciju unatrag u spremniku za čitanje te učitavamo sljedeći objekt. Praznina (blank) je tipičan primjer znaka koji označava kraj riječi u tekstu.

```
1  int c; /* Bolje od char c. */
2  ... /* Citamo znak po znak. */
3  c = fgetc(fp);
4  ... /* Prepoznamo kraj. */
5  ungetc(c, fp); /* Vratimo c u spremnik. */
6  ...
7  c = fgetc(fp); /* Opet procitamo c. */
```

Funkcije `fputc` i `putc`, s prototipom `int fputc(int c, FILE *fp);` i `int putc(int c, FILE *fp);`, upisuju zadani znak c (pretvoren u `unsigned char`) u datoteku na koju pokazuje `fp`. Izlazna vrijednost je:

- *c* ako je uspješno upisan u datoteku,
- `EOF` u slučaju greške.

Jedina razlika između `fputc` i `putc` je što `putc` možemo koristiti i kao makro naredbu dok `fputc` ne možemo. Kao i kod `getc`, `putc` treba oprezno upotrebljavati ako se definira kao makro naredba (mogućnost višestrukog evaluiranja argumenta `fp`). Funkcija `putchar(c)` za standardni izlaz **implementira se kao** `putc(c,stdout)`.

Primjer 4.1.2

Pišemo program koji kopira sadržaj jedne datoteke u drugu, znak po znak. Imena datoteka zadaju se kao argumenti komandne linije (`odakle`, `kamo`). Poruke o greškama pišemo na `stderr` (standardni izlaz za greške) funkcijom `fprintf`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[]){
5    FILE *in, *out;
6    int c; /* Ne: char c! */
7    if (argc != 3) { /* Nema imena datoteka! */
8        fprintf(stderr, "Uporaba:␣%s␣ime1␣ime2\n", argv[0]);
9        exit(1); }
10
11  if ((in = fopen(argv[1], "r")) == NULL) {
12      fprintf(stderr, "Ne␣mogu␣citati:␣%s!\n", argv[1]);
13      exit(2); }
14
15  if ((out = fopen(argv[2], "w")) == NULL) {
16      fprintf(stderr, "Ne␣mogu␣pisati:␣%s!\n", argv[2]);
17      exit(3); }
18
19  while ((c = fgetc(in)) != EOF)
20         fputc(c, out);
21
22  fclose(in);
23  fclose(out);
24  return 0; }
```

Kopiranje sadržaja neke datoteke u drugu datoteku je često praktično implementirati u obliku funkcija.

Primjer 4.1.3

Pišemo funkciju koja kopira sadržaj jedne datoteke u drugu znak po znak. Obje datoteke, tj. pokazivači na njih su argumenti funkcije (smatramo da su obje već otvorene).

```
1 void kopiraj_datoteku(FILE *in, FILE *out){
2     int c; /* Ne: char c! */
3
4     while ((c = fgetc(in)) != EOF)
5             fputc(c, out);
6
7     return; }
```

Kopiranje datoteke `byte-po-byte` je najsporiji način kopiranja. Prednost je što je jednostavno, i sigurno. Bitno brže je kopirati u većim blokovima.

Funkcija za čitanje podataka iz datoteke, liniju po liniju, je:
`char *fgets(char *str, int n, FILE *fp);`

- `str` - pokazivač na dio memorije (*buffer*) u koji će ulazna linija biti spremljena kao string,
- `n` - veličina memorije na koju pokazuje prvi argument = maksimalni broj znakova koji želimo spremiti u polje `str`,
- `fp` - pokazivač na datoteku iz koje se učitava.

Funkcija `fgets` će iz datoteke na koju pokazuje `fp` pročitati liniju od najviše $n-1$ znakova, a najdalje do prvog sljedećeg znaka za kraj linije (’`\n`’), uključujući i njega, ili do kraja datoteke, i na kraj učitanog stringa dodati nul-znak ’`\0`’. Ako je ulazna linija dulja od $n-1$ znakova, ostatak se ne čita. Može se pročitati kasnije, npr. sljedećim pozivom funkcije `fgets`. Izlazna vrijednost je:

- pokazivač `str` ako je sve uspješno pročitano,

- `NULL` u slučaju greške ili ako se na početku čitanja odmah došlo do kraja datoteke.

Funkcija `gets`, s prototipom `char *gets(char *str);`, čita string sa standardnog ulaza `stdin`. `gets` ne prima veličinu *buffer*-a `str` kao argument, stoga se može dogoditi da je ulazna linija dulja od za nju rezervirane memorije u `str`. Iz tog razloga se ne preporuča korištenje ove funkcije!. `gets` je izbačena iz jezika C od standarda $C11$. Umjesto `gets(str)` bolje je koristiti `fgets(str, n, stdin)`. Dodatna razlika između `fgets` i `gets` je u tome što `fgets` učitava i znak '`\n`' (bez zamjene) dok `gets` učita '`\n`' i zamjenjuje ga znakom '`\0`'.

Funkcija za pisanje podataka u datoteku, liniju po liniju ima prototip:
`int fputs(const char *str, FILE *fp);` Ova funkcija:

- ispisuje znakovni niz (string) na kojeg pokazuje `str` u datoteku na koju pokazuje `fp`,
- nul-znak na kraju stringa se ne ispisuje.

Ako želimo prijelaz u novi red, string mora sadržavati znak '`\n`' (ne piše se automatski na kraju stringa).

Izlazna vrijednost je:

- nenegativan broj - ako je ispis uspio,
- `EOF` u slučaju greške.

Funkcija `puts`, s prototipom `int puts(const char *str);`, piše string na standardni izlaz `stdout`. Razlika između `fputs` i `puts`:

- `fputs` ne dodaje znak '`\n`' na kraju ispisa
- `puts` dodaje znak '`\n`' na kraj (umjesto znaka '`\0`').

Razlike u ponašanju s obzirom na znak '`\n`', između:

- `fputs(str, stdout)` i `puts(str)`
- `fgets(str, n, stdin)` i `gets(str)`

odgovaraju jedna drugoj, tako da kopiranje datoteke liniju po liniju, odgovarajućim parom funkcija radi korektno.

Za formatirano čitanje iz datoteke koristimo funkciju:
`int fscanf(FILE *fp, const char *format, ...);`, dok za formatirano pisanje u datoteku koristimo funkciju:
`int fprintf(FILE *fp, const char *format, ...);` Ove funkcije rade identično kao ranije funkcije `scanf`, `printf`, s tim da je ovdje prvi argument: pokazivač `fp` - na datoteku s kojom se radi operacija (čitanje ili pisanje). Pravila za string formata i ostale argumente su ista kao prije.

- `fscanf(stdin, ...)` **je ekvivalentno** sa `scanf(...)`,
- `fprintf(stdout,...)` **je ekvivalentno** s `printf(...)`.

Funkcija `fscanf` vraća:

- nenegativan broj učitanih objekata,
- `EOF` ako je došlo do greške ili do kraja datoteke, prije prve konverzije, tj. čitanja vrijednosti prvog objekta.

Funkcija `fprintf` vraća:

- nenegativan broj napisanih znakova,
- negativan broj - u slučaju greške

Preporučljivo je ispitivati povratne vrijednosti funkcija `fscanf` i `fprintf`.

Primjer formatiranog pisanja u datoteku možemo vidjeti u doljnjem isječku koda:

```
1  //formatirano pisanje u datoteku
2  int kolicina = 50;
3  double cijena = 7.50;
4  ...
5  fprintf(fp, "%5d, %10.2f\n", kolicina, cijena);
6
7  //formatirano citanje iz datoteke
8  int kolicina;
9  double cijena;
10 ...
11 fscanf(fp, "%d, %lf\n", &kolicina, &cijena);
```

Čitanje i pisanje liniju po liniju (`fgets`, `fputs`), te formatirano čitanje i pisanje liniju po liniju (`fscanf`, `fprintf`) **ima smisla raditi samo nad tekstualnim datotekama** koje sadrže format, zapisane su po redcima i time omogućuju lako čitanje sadržaja ljudima.

Formatirano čitanje i pisanje možemo raditi i sa stringovima funkcijama:

```
int sscanf(char *s, const char *format, ...);
int sprintf(char *s, const char *format, ...);
```

Ove funkcije rade identično kao i funkcije `fscanf`, `fprintf`, s tim da je ovdje prvi argument **pokazivač** *s* na string s kojim se radi operacija. Kod pisanja, funkcija `sprintf` dodaje nul-znak '\0' na kraj stringa, ali ga ne broji kad vraća broj napisanih znakova. String *s* mora biti dovoljno velik za cijeli ispis (nema kontrole).

Primjer 4.1.4

Pišemo program koji formatirano čita tekst iz stringa i piše u string.

```
1  #include <stdio.h>
2  int main(void){
3      char *ulaz = "␣50,␣7.50\n"; /* Ulaz! */
4      char izlaz[80];
5      int kolicina;
6      double cijena;
7      sscanf(ulaz, "%d,%lf\n", &kolicina, &cijena);
8      printf("%5d,␣%10.2f\n", kolicina, cijena);
9      sprintf(izlaz, "␣kolicina␣=␣%5d\n"
10         "␣cijena␣=␣%10.2f\n", kolicina, cijena);
11     printf("%s\n", izlaz);
12     return 0; }
```

Kod možemo upotrebljavati za brzo pretvaranje broja u niz znakova i obratno.

Primjer 4.1.5

Pišemo program koji s komandne linije učitava imena dviju datoteka: ulazne i izlazne. Broj argumenata ne treba provjeravati.

Pretpostavljamo da ulazna datoteka već postoji i:

- sastoji se iz riječi odvojenih bjelinama,

- svaka riječ može sadržavati bilo koje druge znakove,
- riječi nisu dulje od 128 znakova.

Program treba iz ulazne datoteke u novu izlaznu datoteku prepisati riječi s više od 4 znaka, koje ne sadrže niti jedno slovo. Svaku riječ treba napisati u novi red.

Opisani zadatak je **filtriranje** ulaza po nekom pravilu.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <ctype.h>
4  #include <string.h>
5  int main(int argc, char *argv[]){
6      char s[129];
7      FILE *ulaz, *izlaz;
8      int uvjet, n, i;
9  /* Kontrolni ispis na stdout. */
10     printf("Ulazna datoteka: %s\n", argv[1]);
11     printf("Izlazna datoteka: %s\n", argv[2]);
12     if ((ulaz = fopen(argv[1], "r")) == NULL) {
13         printf(" Greska na ulazu!\n");
14         exit(2); }
15     if ((izlaz = fopen(argv[2], "w")) == NULL) {
16         printf(" Greska na izlazu!\n");
17         exit(3); }
18
19 /* Uociti test u while: " ... == 1". */
20    while (fscanf(ulaz, "%128s", s) == 1) {
21           n = strlen(s);
22 /* Kontrolni ispis na stdout. */
23    printf("duljina = %d, rijec = %s\n", n, s);
24    if (n > 4) {
25       uvjet = 1;
26       i = 0;
27 /* U uvjetu petlje moze i s[i] != '\0'. */
28       while (i < n && uvjet) {
29            uvjet = uvjet && !isalpha(s[i]);
30            ++i; }
31       if (uvjet)
32          fprintf(izlaz, "%s\n", s); } }
```

```
33    fclose(ulaz);
34    fclose(izlaz);
35    return 0; }
```

Uz pretpostavke:

- ulazna datoteka se zove `fzad1.in`,
- izlazna datoteka se zove `fzad1.out`.

Komandna linija za izvršavanje programa ima oblik:

```
fzad1 fzad1.in fzad1.out
```

Za ulaz:

```
12345a 12345 a 1111
333333 aaaaa 222222
```

Pripadna izlazna datoteka je:

```
12345
333333
222222
```

ZADACI:

1. Prepravite unutarnju `while` petlju tako da ima samo jedan uvjet i koristi naredbu `break` (ubrzanje). Ideja: ako je `s[i]` slovo, stavmio uvjet na 0 i prekinemo petlju.
2. Probajte staviti drugačije uvjete i naredbe za čitanje u vanjskoj petlji. Pažljivo testirajte za razne ulazne podatke:
   - bjeline ispred prve riječi u redu (liniji),
   - bjeline iza zadnje riječi u redu (liniji),
   - prazne linije,
   - linija koja ima samo bjeline,
   - linija koja ima samo bjeline i javlja se na samom kraju ulazne datoteke.

Funkcije za čitanje podataka iz datoteke vraćaju `EOF` ili `NULL` (kod `fgets`) u dva slučaja:

- ako je došlo do greške prilikom čitanja,
- ako je kod čitanja odmah detektiran kraj datoteke.

Funkcije: `int ferror(FILE *fp);` i `int feof(FILE *fp);` služe za razlikovanje ta dva slučaja. Funkcija `ferror` vraća broj različit od nule (istina) ako je došlo do greške, a nulu (laž) ako nije. Funkcija `feof` vraća broj različit od nule (istina) ako smo naišli na kraj datoteke prilikom čitanja, a nulu (laž) u suprotnom. Provjeru vrijednosti ovih funkcija treba napraviti odmah **nakon operacije čitanja iz datoteke**. Pisanje nije uspješno samo u slučaju greške, tada `ferror` vraća istinu.

Svaka otvorena datoteka ima dva indikatora (zastavice, *flag*) statusa u pripadnoj strukturi tipa `FILE`: a) `eof` (end of file) koja označava kraj datoteke, b) `error` koja označava grešku prilikom operacije (npr. disk se napuni prilikom pisanja). Funkcije `ferror` i `feof` testiraju stanje zastavica (indikatora) i vraćaju odgovarajuću logičku vrijednost. Otvaranje datoteke `fopen(...)` briše stanje oba indikatora. Odmah nakon uspješnog otvaranja datoteke je: `feof(...) == 0` i `ferror(...) == 0`. Indikatore postavljaju funkcije za ulazne i izlazne operacije na datoteci (stanje indikatora odnosi se na prethodnu operaciju). Zbog toga provjeru stanja treba napraviti odmah nakon operacije. Indikator `error` signalizira grešku i kod čitanja i kod pisanja, dok `feof` treba testirati samo nakon čitanja. Indikatore datoteke na koju pokazuje pokazivač `fp` možemo obrisati funkcijom: `void clearerr(FILE *fp);`

U doljnjem isječku koda kopiramo sadržaj datoteke `in` u datoteku `out`, znak po znak. Pretpostavimo da nema grešaka pri čitanju i pisanju (ne provjeravamo `ferror`, već samo `feof`). Ukoliko kopiranje realiziramo:

```
1 do {
2    c = fgetc(in);
3    fputc(c, out);
4 } while (!feof(in));
```

Znak `c == EOF` upišemo prije testa `!feof(int)` u `while` petlji. Istu pogrešku dobijemo i implementacijom prikazanom u isječku ispod:

```
while (!feof(in)) {
       c = fgetc(in);
       fputc(c, out); }
```

Ispravan način provjerava i znak c prije upisa u datoteku `out`.

Primjer 4.1.6

Pišemo funkciju koja kopira sadržaj jedne tekstualne datoteke u drugu liniju po liniju (obje datoteke su otvorene). Funkcija treba prepoznati i javiti greške prilikom čitanja i pisanja. Za prepoznavanje greške prilikom čitanja koristimo funkciju `ferror` (vraćamo 0 ili kod greške).

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_LINE 129

int copy_file(FILE *in, FILE *out){
    char buf[MAX_LINE]; /* Ulazni buffer. */
    while (fgets(buf, MAX_LINE, in) != NULL)
          if (fputs(buf, out) == EOF) {
               fprintf(stderr,
                    "\nGreska u pisanju.\n");
              return 1; }

     if (ferror(in)) {
        fprintf(stderr, "\nGreska u citanju.\n");
        return 2; }
      return 0;
}
```

4.1.3 Osnovne funkcije za rad s binarnim datotekama

Izbor tipa datoteke i način otvaranja ovise o potrebama, efikasno spremanje i pristup ili mogućnost vizualnog pregledavanja.

Kao što smo ranije naveli, ulazno-izlazne operacije koje se izvode liniju po liniju `fgets`, `fputs`, ili formatirano liniju po liniju `fscanf`, `fprintf` se prirodno izvode s tekstualnim datotekama. Spomenutim funkcijama možemo čitati i

pisati, formatirano ili neformatirano, samo znakove, vrijednosti ostalih standardnih tipova možemo čitati ili pisati samo formatirano. Kod čitanja znak po znak funkcijama `fgetc` i `fputc` može doći do razlika ovisno o tipu datoteke (načinu zapisa) i operacijskom sustavu. Čitanje i pisanje objekata ostalih, izvedenih, tipova ne možemo raditi direktnim pozivom ranije navedenih funkcija, već ih moramo zapisivati po komponentama što je često nepraktično. Složene tipove možemo učinkovito spremati i čitati koristeći **binarno čitanje i pisanje**.

U praksi često trebamo datoteke koje sadrže niz struktura određenog tipa ili niz podataka standardnog tip (`int`, `double` itd.) u internoj binarnoj reprezentaciji (bez pretvaranja u tekst). Tako izbjegavamo greške zaokruživanja koje nastaju pri formatiranom čitanju i pisanju realnih vrijednosti. Opisanu funkcionalnost ostvarujemo otvaranjem datoteke kao binarne. Neformatirane ulazno-izlazne operacije realiziraju se posebnim funkcijama `fread` i `fwrite`, koje kopiraju sadržaj zadanog bloka byte-ova kao niz znakova, odnosno niz podataka tipa `unsigned char`.

Funkcije za binarno (neformatirano) čitanje i pisanje su: `size_t fread(void *ptr, size_t size, size_t nobj, FILE *fp);` i `size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);`.

Argumenti funkcija su:

- `ptr` - pokazivač na varijablu (ili polje) u koju `fread` upisuje, odnosno, iz koje `fwrite` čita,
- `size` - veličina pojedinog objekta,
- `nobj` - broj objekata koje treba učitati/ispisati,
- `fp` - pokazivač na datoteku iz koje se čita (`fread`) ili u koju se piše (`fwrite`).

Funkcija `fread` čita iz datoteke na koju pokazuje `fp` niz od `nobj` objekata, svaki veličine `size`, i sprema ih u varijablu (polje) na koju pokazuje `ptr`. Izlazna vrijednost funkcije je broj učitanih objekata iz datoteke, koji može biti i manji od `nobj`, ako je došlo do greške ili kraja datoteke. Treba koristiti funkcije `feof` i `ferror` za provjeru statusa nakon operacije.

Funkcija `fwrite` piše u datoteku na koju pokazuje `fp` niz od `nobj` objekata, svaki veličine `size`, iz varijable (polja) na koju pokazuje `ptr`. Izlazna vrijed-

nost funkcije je broj napisanih objekata u datoteku, koji može biti i manji od `nobj`, ako je došlo do greške. Kod pisanja nema smisla testirati kraj datoteke.

Ove funkcije ne rade konverziju iz binarnog zapisa u znakovni (ASCII) zapis i obratno. Čitaju odnosno pišu blok od `nobj * size` znakova, kao interna reprezentacija podataka u tom računalu.

Čitanje cijelog polja cijelih brojeva iz binarne datoteke možemo napraviti pozivom:

```
int polje[10];
...
fread(polje, sizeof(int), 10, fp);
```

Pisanje cijelog polja cijelih brojeva u binarnu datoteku možemo ostvariti pozivom:

```
int polje[10] = { ... };
...
fwrite(polje, sizeof(int), 10, fp);
```

Pisanje vrijednosti iz zadane strukture podataka u binarnu datoteku, uz provjeru korektnosti upisa, možemo ostvariti naredbama:

```
typedef struct {
      int broj_racuna;
      char ime[80];
      double stanje;
} Racun;

Racun kupac = { 47, "Pero␣Peric", 200.00 };
fp = fopen("novi.dat", "wb");
...
if (fwrite(&kupac, sizeof(Racun), 1, fp) != 1) {
        fprintf(stderr, "Greska␣pri␣upisu.\n");
        exit(1); }
```

Prednosti binarnog ulaza i izlaza:

- brzina - nema pretvaranja u tekst ili iz teksta,
- manja veličina zapisa - npr. zapis `int`-a u 4 byte-a, umjesto do 10 znakova i potencijalno predznak.

Nedostaci binarnog ulaza i izlaza:

- ovisnost o arhitekturi računala i prevoditelju,
- nije čitljiv ljudima - binarna datoteka se ne može editirati običnim editorom za obradu teksta.

U kombinaciji s funkcijama za pozicioniranje u datoteci (`ftell`, `fseek`), funkcije `fread` i `fwrite` služe i za direktni pristup podacima.

Ulazno-izlazne operacije koje smo radili do sada, koristile su sekvencijalni pristup podacima u datoteci. Nakon otvaranja datoteke, početna pozicija ovisi o načinu otvaranja datoteke. Kod pisanja (`"w"`) i čitanja (`"r"`), pozicionirani smo na početak datoteke, a kod dodavanja (`"a"`) na kraj datoteke (iza svega što postoji u datoteci - ovo je ovisno o implementaciji). Nakon toga, svaka sljedeća ulazno-izlazna operacija nastavlja raditi točno tamo gdje je prethodna operacija završila - tj. stalno se krećemo u datoteci unaprijed. Za svaku datoteku, u pripadnoj `FILE` strukturi, pamti se i trenutna pozicija u datoteci (tzv. `file_pos`), do koje smo stigli prethodnim operacijama na toj datoteci. Trenutna pozicija se računa kao broj znakova (byte-ova) od početka datoteke (slično kao indeksi polja znakova). Nula označava početak datoteke (ispred prvog znaka). Standardni tip za tu vrijednost je `long`, odnosno `long int`. Na nekim sustavima taj tip može biti i veći od `long`, ovisno o dozvoljenoj veličini datoteke. Svaka ulazno-izlazna operacija pomiče poziciju u datoteci unaprijed (u odnosu na trenutnu). Kad god napravimo operaciju čitanja ili pisanja, trenutna pozicija se povećava upravo za broj pročitanih ili napisanih znakova (byteova). Ako sami ne mijenjamo trenutnu poziciju, dobivamo sekvencijalno čitanje i pisanje (svaka operacija počinje gdje je prethodna stala). Trenutna pozicija se mijenja sama. Dozvoljeno je promijeniti vrijednost trenutne pozicije u datoteci, međutim kod tekstualnih datoteka je promjena vrijednosti trenutne pozicije **znatno ograničena**. Promjenu pozicije radimo zadavanjem mjesta u datoteci na kojem želimo da počne sljedeća ulazno-izlazna operacija. Na taj način, kod binarnih datoteka, možemo čitati i pisati podatke bilo gdje u datoteci, tj. svakom znaku (byte-u) u datoteci pristupamo direktno, slično kao u polju. Zato se ovaj način rada s datotekom zove **direktni ili slučajni pristup podacima**.

Realizacija direktnog pristupa slična je indeksiranju kod polja. Prije operacije zadajemo poziciju u datoteci, posebnom funkcijom za pozicioniranje u datoteci. Za direktni pristup podacima u C-u koristimo dvije funkcije:

`ftell` koja vraća trenutnu poziciju u datoteci i `fseek` koja mijenja trenutnu poziciju u datoteci na zadanu poziciju.

Funkcija `ftell`, s prototipom `long int ftell(FILE *fp);`, vraća trenutnu poziciju u već otvorenoj datoteci na koju pokazuje `fp` (broj znakova, byte-ova, od početka te datoteke). Izlazna vrijednost je:

- nenegativan broj u slučaju uspjeha
- $-1L$ u slučaju greške.

Vrijednost $0L$ znači da se nalazimo na početku datoteke. Dakle, povratna vrijednost je udaljenost od početka datoteke (u znakovima, byte-ovima).

Kod otvaranja datoteke s `"r"` ili `"w"` dobivamo da je trenutna pozicija $0L$, tj. početak. Ako datoteku otvorimo za dodavanje (`"a"`), dobivena vrijednost ovisi o implementaciji. Na Windowsima (Intel C, Code::Blocks) - trenutna pozicija je početak datoteke ($0L$), na Linuxu - trenutna pozicija je kraj datoteke (iza zadnjeg znaka, duljina datoteke u byte-ovima).

Argumenti funkcije `fseek`, s prototipom `int fseek(FILE *fp, long offset, int origin);`, su:

- `fp` - pokazivač na već otvorenu datoteku,
- `offset` - zadani pomak u broju znakova (byte-ova),
- `origin` - indikator položaja ili ishodište od kojeg se broji pomak. Zadaje se jednom od tri simboličke konstante (definirane u `<stdio.h>`):
  - `SEEK_SET` - od početka datoteke,
  - `SEEK_CUR` - od trenutne pozicije u datoteci,
  - `SEEK_END` - od kraja datoteke.

Funkcija `fseek` postavlja trenutnu poziciju u datoteci na koju pokazuje `fp` na `offset` znakova od zadanog ishodišta `origin`.

Izlazna vrijednost funkcije je:

- nula - ako je uspješno postavila zadanu poziciju,
- broj različit od nule - u slučaju greške.

Nekoliko poziva funkcije `fseek` za pozicioniranje u binarnoj datoteci zadanoj pokazivačem `fp` možemo vidjeti u programskom isječku dolje:

```
fseek(fp, 0L, SEEK_SET); /* POCETAK datoteke. */
fseek(fp, 0L, SEEK_END); /* KRAJ datoteke. */
fseek(fp, 2L, SEEK_SET); /* 2 znaka IZA
pocetka datoteke. */
fseek(fp, 2L, SEEK_CUR); /* 2 znaka IZA
trenutne pozicije. */
fseek(fp, -2L, SEEK_END); /* 2 znaka ISPRED
kraja datoteke. */
```

Zadana pozicija mora biti unutar granica datoteke.

Kod poziva funkcije `fseek` za **tekstualne datoteke**, standard postavlja sljedeće ograničenje: `offset` mora biti nula, ili vrijednost koju vrati poziv funkcije `ftell` (označimo ju s `ftell_pos`). To znači da su dobro definirani jedino pozivi oblika:

- `fp`, `0L`, `SEEK_SET` - idi na početak datoteke,
- `fp`, `0L`, `SEEK_END` - idi na kraj datoteke,
- `fp`, `0L`, `SEEK_CUR` - ostani na trenutnoj poziciji,
- `fp`, `ftell_pos`, `SEEK_SET` - idi na poziciju koju je dao prethodni poziv `ftell`.

Pozicioniranje na početak datoteke možemo napraviti pozivom funkcije `void rewind(FILE *fp);`. Ovaj poziv ekvivalentan je s:

```
fseek(fp, 0L, SEEK_SET);
clearerr(fp);
```

Osim pozicioniranja na početak datoteke, brišemo i indikatore za kraj datoteke i za grešku.

Primjer 4.1.7

Pretpostavimo da postoji datoteka koja sadrži 4 znaka:

```
a b c d
```

Prvo otvorimo tu datoteku za (tekstualno ili binarno) čitanje, a zatim 6 puta ponovimo sljedeće: a) nađemo trenutnu poziciju u toj datoteci (funkcija `ftell`) i ispišemo ju (na `stdout`), b) učitamo sljedeći znak iz te datoteke i ispišemo ga (na `stdout`).

```
1 /* Sekvencijalno citamo tu datoteku. */
2 for (i = 0; i < 6; ++i) {
3     printf("␣Pozicija:␣%ld,", ftell(fp));
4
5     if ((c = fgetc(fp)) >= 0)
6         printf("␣znak␣=␣%2c\n", c); /* Znak. */
7     else
8         printf("␣znak␣=␣%2d\n", c); /* Broj. */
9     }
```

Izlaz programa je:

```
Pozicija: 0, znak =  a
Pozicija: 1, znak =  b
Pozicija: 2, znak =  c
Pozicija: 3, znak =  d
Pozicija: 4, znak = -1
Pozicija: 4, znak = -1
```

Znak -1 je uobičajena vrijednost za `EOF`.

ZADACI:

1) Modificirajte program tako da ispisuje trenutne pozicije prije pisanja svakog znaka (treba zapisati i početnu poziciju).

2) Nakon kreiranja zadane datoteke s 4 znaka, treba:

   - otvoriti tu datoteku za dodavanje (`"a"`),
   - u nju napisati još 2 znaka: 'e', 'f' (na kraj),
   - zatvoriti datoteku.

3) Zatim treba otvoriti tu datoteku za čitanje i 8 puta ponoviti operaciju čitanja sljedećeg znaka (kao u primjeru).

4) Nakon kreiranja zadane datoteke s 4 znaka treba:

- otvoriti tu datoteku za čitanje i dodavanje (`"a+"`),
- 6 puta ponoviti operaciju čitanja sljedećeg znaka (kao u primjeru),
- u datoteku napisati još 2 znaka: 'e', 'f' (na kraj).

Primjer 4.1.8

Pišemo program koji naopako kopira sadržaj jedne datoteke u drugu.

Npr. koristeći datoteku:

```
a b c d
```

trebamo stvoriti datoteku:

```
d c b a
```

Kopiranje radimo znak po znak, tako da po prvoj datoteci idemo unatrag — od kraja datoteke, prema početku, koristeći direktni pristup podacima.

Zbog načina pristupa podacima, obje datoteke treba otvoriti kao **binarne**, a ne kao tekstualne! Varijabla `pomak` broji pomak od kraja datoteke, tj. ishodište je `SEEK_END`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(void){
4
5    char *in_name = "freverse.in";
6    char *out_name = "freverse.out";
7
8    FILE *in, *out;
9    long file_pos, pomak = 0L;
10   if ((in = fopen(in_name, "rb")) == NULL) {
11       fprintf(stderr, "Ne mogu citati iz: %s!\n",
12                   in_name);
13       exit(1); }
14
15   if ((out = fopen(out_name, "wb")) == NULL) {
16       fprintf(stderr, "Ne mogu pisati u: %s!\n",
17                   out_name);
```

```
18        exit(2); }
19
20    do {/* Datoteku kopiramo naopako. */
21  /* Pomak unazad od kraja. */
22        if (fseek(in, --pomak, SEEK_END)) break;
23  /* Zapamti poziciju i ucitaj znak. */
24        file_pos = ftell(in);
25        fputc(fgetc(in), out);
26  /* Sad je pozicija narasla za 1L. */
27    } while (file_pos != 0L);
28
29    fclose(in);
30    fclose(out);
31    return 0; }
```

Kod pomaka unatrag od kraja datoteke, test:
`if (fseek(in, -pomak, SEEK_END)) break;`
je zaštita od prazne ulazne datoteke. Bez tog testa program ne radi.

Za ulaznu datoteku od 33 znaka:

```
Ja sam mala Ruza, mamima sam kci.
```

izlazna datoteka ima 33 znaka:

```
.ick mas amimam ,azuR alam mas aJ
```

Složenost ovog algoritma je linearna u duljini datoteke, zbog direktnog pristupa podacima.

ZADATAK: Napravite istu stvar sekvencijalnim pristupom podacima. Složenost je tada kvadratna u duljini datoteke.

Obje datoteke moraju biti otvorene kao binarne. U protivnom, čim ulazna datoteka ima bar jedan znak za kraj reda, program ne radi dobro kod sustava kod kojih postoji razlika između binarnih i tekstualnih datoteka (npr. Windows). Neće dobro raditi niti program koji naopako ispisuje sadržaj binarne datoteke na `stdout`, te preusmjerava ispis u izlaznu datoteku. Na Windowsu se znak za kraj linije'`\n`' zapisuje kao dva znaka '`\r`' '`\n`', stoga dolazi do umjetnog dodavanja znakova u izlaznu datoteku.

Datoteku možemo otvoriti tako da je dozvoljeno i čitanje iz datoteke i pisanje u tu datoteku. Takav način rada s datotekom definiramo tako da u `file_mod` stringu navedemo znak `+`. Treba biti oprezan pri prijelazu s čitanja na pisanje i obratno, tada treba korektno isprazniti spremnik za komunikaciju između programa i datoteke. Pri prijelazu s čitanja na pisanje (između operacija) treba pozvati funkciju za pozicioniranje u datoteci (`fseek`, `rewind` ili `fsetpos`[2]), osim ako je čitanje stiglo do kraja datoteke ili pozvati funkciju `fflush` za pražnjenje (pisanje) sadržaja spremnika u datoteku.

Ako `fp` pokazuje na izlaznu datoteku (tj. zadnja operacija je bila pisanje u tu datoteku), `fflush`, s prototipom `int fflush(FILE *fp);`, piše u tu datoteku onaj dio sadržaja spremnika koji do tada nije bio fizički zapisan u nju (prazni spremnik u datoteku). Ako `fp` pokazuje na ulaznu datoteku (tj. zadnja operacija je čitanje iz te datoteke), efekt poziva funkcije `fflush` nije definiran (nema smisla). Poziv oblika `fflush(NULL);` prazni spremnike za sve izlazne datoteke u tom trenutku.

Izlazna vrijednost funkcije `fflush` je:

- nula - ako je uspješno ispraznila spremnik,
- `EOF` - ako je prilikom pisanja došlo do greške.

Regularan završetak programa (uključujući pozivom funkcije `exit`) automatski prazni spremnike za sve otvorene (izlazne) datoteke.

Primjer 4.1.9

Telefonski račun opisan je strukturom tipa `Racun`:

```
1 typedef struct {
2    int tel_broj;
3    char vlasnik[20];
4    double stanje;
5 } Racun;
6
7 int size = sizeof(Racun);
```

Podaci o računima korisnika spremljeni su u binarnoj datoteci koja sadrži niz takvih struktura. Svaki zapis u datoteci je jedna struktura tipa `Racun`, veličine `size`. Treba napisati funkciju `dodaj_bonus` sa zaglavljem oblika:

[2] https://cplusplus.com/reference/cstdio/fsetpos/

```
void dodaj_bonus(const char *f_name, int n);
```

String `f_name` je ime (postojeće) binarne datoteke koja sadrži niz struktura tipa `Racun`. Funkcija treba n-tom zapisu u datoteci dodati bonus od 100.0 na stanje računa, ako je stanje prije toga bilo pozitivno. Brojanje zapisa u datoteci počinje od 1. Za rješenje koristimo direktni pristup podacima u datoteci. Uzmimo da se pokazivač na tu datoteku zove `racuni`. Za čitanje n-tog zapisa treba preskočiti prvih $n-1$ zapisa od početka datoteke, tj. od ishodišta `SEEK_SET`. Odgovarajuće pozicioniranje je (pretvaranje u `long`):

```
file_pos = (long) (n - 1) * size;
fseek(racuni, file_pos, SEEK_SET);
```

Nakon dodavanja bonusa, za pisanje novog n-tog zapisa treba se vratiti natrag (za jedan zapis od trenutne pozicije u datoteci), tj. od ishodišta `SEEK_CUR`.

```
fseek(racuni, -size, SEEK_CUR);
```

Funkcija `dodaj_bonus` glasi:

```
void dodaj_bonus(const char *f_name, int n){
   FILE *racuni;
   Racun kor;
   long file_pos;
   const double bonus = 100.0;

   if ((racuni = fopen(f_name, "r+b")) == NULL) {
       fprintf(stderr, "Ne mogu otvoriti: %s!\n",
                                          f_name);
   exit(1); }

/* Pozicioniranje ispred n-tog zapisa. */
  file_pos = (long) (n - 1) * size;

  if (fseek(racuni, file_pos, SEEK_SET)) {
        fprintf(stderr,
                "Greska u fseek, n = %d.\n", n);
  printf("Greska u fseek, n = %d.\n", n);
  fclose(racuni);
  return;}

```

```
22  /* Ucitaj zapis u strukturu. */
23   if (fread(&kor, size, 1, racuni) != 1)
24        if (ferror(racuni)) {
25           fprintf(stderr, "Greska u citanju.\n");
26           exit(2); }
27        else if (feof(racuni)) {
28            fprintf(stderr,
29            "Kraj datoteke, n = %d.\n", n);
30        printf("Kraj datoteke, n = %d.\n", n);
31        fclose(racuni);
32        return; }
33
34 /* Azuziraj stanje i napisi novi zapis. */
35    if (kor.stanje > 0) {
36       kor.stanje = kor.stanje + bonus;
37       fseek(racuni, -size, SEEK_CUR);
38    if (fwrite(&kor, size, 1, racuni) != 1) {
39       fprintf(stderr, "Greska u pisanju.\n");
40       exit(3); } }
41    fclose(racuni);
42    return; }
```

Pretpostavimo da u datoteci `racuni.dat` imamo sadržaj:

```
zapis 1: 384907, Tihana Glasnovic, 92.00
zapis 2: 622744, Goga Trubic, 456.27
zapis 3: 918235, Josip Mobitelic, -234.49
zapis 4: 436702, Martina Lajavic, 74.12
zapis 5: 739417, Pero Bacilova, -1017.12
zapis 6: 208143, Mirna Sutljivic, 48.50
```

Zatim, dodajemo bonus sljedećim zapisima:

```
dodaj_bonus(argv[1], 3);
dodaj_bonus(argv[1], 6);
dodaj_bonus(argv[1], 1);
```

Nova datoteka `racuni.dat` ima sadržaj:

```
zapis 1: 384907, Tihana Glasnovic, 192.00
zapis 2: 622744, Goga Trubic, 456.27
```

```
zapis 3: 918235, Josip Mobitelic, -234.49
zapis 4: 436702, Martina Lajavic, 74.12
zapis 5: 739417, Pero Bacilova, -1017.12
zapis 6: 208143, Mirna Sutljivic, 148.50
```

Primjer 4.1.10

Pišemo program koji okreće naopako (invertira) sadržaj jedne zadane binarne datoteke, međutim invertirani sadržaj upisuje u tu istu datoteku (bez korištenja pomoćne datoteke).

Ako datoteka na početku ima oblik:

```
'a' 'b' 'c' 'd'
```

nakon izvođenja programa treba imati oblik:

```
'd' 'c' 'b' 'a'
```

Datoteku treba otvoriti tako da je dozvoljeno istovremeno čitanje i pisanje u toj datoteci. Postupamo slično kao u polju ili stringu. Jedini problem je naći polovište datoteke, tj. prepoznati kada smo gotovi. Najlakši način je pronaći duljinu datoteke (odlaskom na kraj) te raspoloviti dobivenu duljinu.

```
fseek(dat, 0L, SEEK_END);
file_pola = ftell(dat) / 2L;
```

Uz to trebamo raditi pažljivo pozicioniranje s odgovarajućim pomakom.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5     char *dat_name = "finvert.dat";
6     FILE *dat;
7     long file_pola, pomak = 0L;
8     int ch_1, ch_2;
9
10    if ((dat = fopen(dat_name, "rb+")) == NULL) {
11       fprintf(stderr, "Ne mogu citati iz: %s!\n",
12                                          dat_name);
```

```
13      exit(1); }
14 /* Duljina i poloviste datoteke. */
15    fseek(dat, 0L, SEEK_END);
16    file_pola = ftell(dat) / 2L;
17
18 /* Datoteku invertiramo. */
19    while (pomak < file_pola) {
20 /* Pomak unaprijed od pocetka.
21 Ucitaj prvi znak. */
22       fseek(dat, pomak, SEEK_SET);
23       ch_1 = fgetc(dat);
24       /* Pomak unazad od kraja.
25 Ucitaj drugi znak. */
26       fseek(dat, -pomak - 1L, SEEK_END);
27       ch_2 = fgetc(dat);
28       /* Pomak za jedno mjesto unazad od
29 trenutnog. Napisi prvi znak. */
30       fseek(dat, -1L, SEEK_CUR);
31       fputc(ch_1, dat);
32
33 /* Pomak unaprijed od pocetka.
34 Napisi drugi znak. */
35       fseek(dat, pomak, SEEK_SET);
36       fputc(ch_2, dat);
37       ++pomak; /* Povecaj pomak! */
38      }
39       fclose(dat);
40       return 0; }
```

Za ulaznu datoteku sadržaja:

```
Ja sam mala Ruza, mamima sam kci.
```

Promijenjena datoteka ima sadržaj:

```
.ick mas amimam ,azuR alam mas aJ
```

Ukoliko kod izvršimo dva puta, dobijemo polaznu datoteku.

Složenost programa je linearna u duljini datoteke.

ZADACI:

1) Napišite program koji okreće ili invertira datoteku `racuni.dat`, koja sadrži niz struktura tipa `Racun`, iz ranijeg primjera.

   Nemojte učitati cijeli niz iz datoteke u neko polje, tamo ga okrenuti a onda napisati niz u datoteku!

   - Pronađite broj računa u datoteci (duljina datoteke podijeljena s veličinom svake strukture),
   - Daljnji postupak je kao kod okretanja datoteke (pojedinačnih znakova) osim što svaki pojedini objekt ima duljinu `size`, pa čitanje vršimo funkcijom `fread` (a ne `fgetc`), a pisanje vršimo funkcijom `fwrite` (a ne `fputc`).

2) Napišite program koji (uzlazno) sortira datoteku `racuni.dat`, koja sadrži niz struktura tipa `Racun` po nekom zadanom kriteriju (telefonskom broju, imenu vlasnika ili stanju računa).

   Nemojte učitati cijeli niz iz datoteke u neko polje, tamo ga sortirati nekim algoritmom, a onda napisati niz u datoteku.

   Izaberite neki algoritam sortiranja na polju (proizvoljan) koji koristi jedno polje, zatim sve osnovne operacije realizirajte malim funkcijama na datoteci.

   Potencijalno korisne funkcije: a) broj objekata u zadanoj datoteci, b) citanje i-tog objekta iz datoteke (u strukturu), c) pisanje i-tog objekta (iz strukture) u datoteku, d) usporedba dva objekta (strukture), e) zamjena dva objekta u datoteci (pažljivim čitanjem i pisanjem - može zasebnom funkcijom kao u invertiranju).

3) **Napredno:** implementirajte `QuickSort` algoritam na datoteci.

4.2 Pretprocesor, standardna biblioteka, mjerenje vremena

U odjeljku ćemo prvo opisati i navesti neke značajke rada pretprocesora, zatim ćemo istražiti strukturu standardne biblioteke s fokusom na matematičku biblioteku i konačno opisati kako mjeriti vrijeme izvršavanja C-programa.

4.2.1 Pretprocesor

Prije prevođenja izvornog koda u objektni ili izvršni kod, izvršavaju se **pretprocesorske naredbe**. Svaka linija izvornog koda koja započinje znakom # (osim u komentaru) čini jednu pretprocesorsku naredbu. Pretprocesorska naredba završava krajem linije, a ne znakom ;. Glavni razlog je što pretprocesorske naredbe nisu sastavni dio jezika C, stoga ne podliježu sintaksi jezika. Opći oblik pretprocesorske naredbe je: `#naredba parametri`

Pretprocesorske naredbe su npr. `#include`, `#define`, `#undef`, `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif`.

Naredba `#include` se javlja u dva oblika: a) `#include <ime_datoteke>` ili b) `#include "ime_datoteke"`. U oba slučaja pretprocesor briše liniju s `#include` naredbom i uključuje sadržaj datoteke `ime_datoteke` u izvorni kod, na mjestu `#include` naredbe. Ta datoteka smije imati svoje pretprocesorske naredbe koje će se također obraditi. Ovisno o načinu navođenja `ime_datoteke`, pretprocesor će tražiti datoteku:

- na mjestu određenom operacijskim sustavom za sistemske datoteke, ako je ime navedeno između znakova <>,
- u direktoriju (folderu) u kojem se nalazi program, ako je ime navedeno unutar navodnika ".

Naredba `#define` ima oblik: `#define ime tekst_zamjene`. Ona definira simbol `ime` kao objekt. Ako je zadan i `tekst_zamjene` onda `ime` postaje *makro-naredba*. Pretprocesor će od mjesta `#define` do kraja datoteke, svaku pojavu imena `ime` zamijeniti tekstom `tekst_zamjene`. Do zamjene neće doći unutar znakovnih nizova (string konstanti), unutar para dvostrukih navodnika " i unutar susjednog para znakova /* i */.

U parametriziranoj makro-naredbi, simboličko ime i `tekst_zamjene` sadrže formalne argumente. `#define ime(argumenti) tekst_zamjene`. Argumenti se pišu u zagradama () i odvajaju zarezom. Prilikom poziva makro-naredbe, formalni argumenti se zamjenjuju stvarnim argumentima navedenim u pozivu. Zamjena se vrši tako da se tekst argumenta zamijeni tekstom stvarne vrijednosti navedene u pozivu. Makro-naredba je efikasnija od funkcije (jer nema prijenosa argumenata), ali može izazvati neželjene efekte.

Makro naredba za nalaženje većeg od dva argumenta je:

```
1 #define max(A, B) ((A) > (B) ? (A) : (B))
```

A i B su formalni argumenti.
Pretprocesor će naredbu:

```
1 x = max(a1, a2);
```

zamijeniti naredbom:

```
1 x = ((a1) > (a2) ? (a1) : (a2));
```

Formalni argumenti (parametri) A i B zamijenjeni su stvarnim argumentima $a1$ i $a2$. Zamjena se izvršava **supstitucijom teksta**.

Naredba:

```
1 x = max(a1, a1 - a2);
```

će biti zamijenjena naredbom:

```
1 x = ((a1) > (a1 - a2) ? (a1) : (a1 - a2));
```

Sličnost makro-naredbe i funkcije može zavarati. Kod makro-naredbe nema kontrole tipa argumenata. Argumenti se doslovno supstituiraju tekstualno, što može izazvati neželjene efekte.

Ako makro–naredbu `max` pozovemo na sljedeći način:

```
1 x = max(i++, j++);
```

nakon supstitucije dobivamo:

```
1 x = ((i++) > (j++) ? (i++) : (j++));
```

Posljedica: veća varijabla bit će inkrementirana dva puta!

Zaglavlje `<stdio.h>` sadrži neke makro-naredbe (npr. `getchar`, `putchar`), a funkcije u `<ctype.h>` su također uglavnom izvedene kao makro-naredbe.

Primjer 4.2.1

Promotrimo različite makro–naredbe za kvadriranje argumenta:

```
1 #include <stdio.h>
2 #define SQ1(x) x*x
3 #define SQ2(x) (x)*(x)
```

```
4  #define SQ3(x) ((x)*(x))
5
6  int main(void){
7     printf("%d\n", SQ1(1+1));
8     printf("%d\n", 4/SQ2(2));
9     printf("%d\n", 4/SQ3(2));
10    return 0; }
```

Izlaz programa je:

```
3
4
1
```

Nakon poziva `SQ1(1+1)` se dogodi supstitucija $1+1*1+1 = 1+1+1 = 3$. Nakon poziva $4/$`SQ2(2)` se dogodi supstitucija $4/(2)*(2) = 4/2*2 = 2*2 = 4$. Nakon poziva $4/$`SQ3(2)` se dogodi supstitucija $4/((2)*(2)) = 4/(2*2) = 4/4 = 1$.

Uočavamo da samo treća makro naredba iz primjera daje korektan rezultat u svim slučajevima. Zato se kod definicije parametriziranih makro-naredbi koristi puno zagrada.

U `#define` naredbi, `tekst_zamjene` obuhvaća tekst do kraja linije. Ako želimo da `ime` bude zamijenjeno s više linija teksta, moramo koristiti obratnu kosu crtu (\) na kraju svakog reda, osim posljednjeg. Oznaka \ označava da se red teksta nastavlja na početku sljedećeg.

Makro naredbu za inicijalizaciju polja možemo definirati:

```
1  #define INIT(polje, dim) for(int i=0;\
2  i < (dim); ++i) \
3  (polje)[i] = 0.0;
```

Definicija nekog imena može se poništiti korištenjem `#undef` naredbe. Nakon naredbe `#undef ime`, simbol `ime` više nije definiran.

Pretpostavimo da želimo redefinirati konstantu `M_PI` koja reprezentira vrijednost od π u tipu `double`. To možemo napraviti doljnjim prograskim isječkom:

```
1  #include <math.h>
2  /* math.h definira M_PI kao 3.14... */
```

```
3 #undef M_PI
4 #define M_PI (4.0*atan(1.0))
```

Provjeru je li simbol `ime` definiran možemo napraviti naredbama `#ifdef` odnosno `#ifndef`.

Pretprocesorske naredbe `#if`, `#endif`, `#else`, `#elif` služe za uvjetno uključivanje (ili isključivanje) pojedinih dijelova programa. Uvjetno uključivanje koda naredbama `#if`, `#endif` ima oblik:

```
1 #if uvjet
2    blok naredbi
3 #endif
```

Ako je uvjet ispunjen, tada će blok naredbi između `#if uvjet` i `#endif` biti uključen u izvorni kod, koji će biti predan prevoditelju za prevođenje. Ako uvjet nije ispunjen, blok neće biti uključen (prevoditelju će biti proslijeđen kod bez tog bloka koda). `#if` nije zamjena za `if` naredbu jezika C, već radi na razini teksta izvornog koda. Uvjet u `#if` naredbi je konstantan cjelobrojni izraz (0 - laž, $\neq 0$ - istina). Najčešća svrha uključivanja/isključivanja je uključiti neku datoteku zaglavlja, ako neki simbol ili pretprocesorska varijabla nije bila definirana ranije. Provjeru definiranosti možemo izvršiti naredbom `defined(ime)` koja vraća 1 ako je ime definirano, a 0 ako nije. Kod provjere nedefiniranosti imena možemo koristiti i operator negacije (!), `!defined(ime)`.

Provjeru nedefiniranosti imena za uključivanje odgovarajuće datoteke zaglavlja `datoteka.h` možemo napraviti naredbama:

```
1 #if !defined(__datoteka.h__)
2     #include "datoteka.h"
3 #endif
```

uz definiranje imena `__datoteka.h__` unutar datoteke `datoteka.h` naredbom: `#define __datoteka.h__`.

To je standardna tehnika kojom se izbjegava višestruko uključivanje `.h` datoteka (i potencijalna beskonačna rekurzija). Konstrukcije `#if defined` i `#if !defined` se vrlo često pojavljuju u praksi, stoga postoje kraći oblici `#ifdef` i `#ifndef`.

Prethodnu provjeru možemo napisati u obliku:

```
#ifndef __datoteka.h__
     #include "datoteka.h"
#endif
```

uz definiranje imena `__datoteka.h__` unutar datoteke `datoteka.h` naredbom: `#define __datoteka.h__`.

Složene `if` naredbe za uključivanje ili isključivanje pojedinih dijelova koda (u pretprocesoru) grade se pomoću naredbi `#else` i `#elif`. `#else` ima isto značenje kao `else` u C-u, dok `#elif` ima isto značenje kao `else if`.

Treba imati na umu da su `#else` i `#elif` naredbe koje rade na nivou teksta programa!

Primjer 4.2.2

Kod izrade koda koji treba raditi na raznim operacijskim sustavima, testiramo koji je operacijski sustav u pitanju, kroz ime (simbol) `SYSTEM` da bi se uključilo ispravno zaglavlje.

```
#if SYSTEM == SYSV
    #define DATOTEKA "sysv.h"
#elif SYSTEM == BSD
    #define DATOTEKA "bsd.h"
#elif SYSTEM == MSDOS
    #define DATOTEKA "msdos.h"
#else
    #define DATOTEKA "default.h"
#endif
#include DATOTEKA
```

Primjer 4.2.3

U fazi razvoja programa korisno je ispisivati razne međurezultate za kontrolu korektnosti izvršavanja programa. U završnoj verziji programa, sav taj suvišan ispis treba eliminirati. Za to koristimo standardni simbol `DEBUG`.

```
...
scanf("%d", &x);
#ifdef DEBUG
printf("Debug: x = %d\n", x); // testiranje
```

```
5 #endif
```

Svi standardni prevoditelji imaju `-Dsimbol` opciju koja omogućava da se simbol definira na komandnoj liniji. Pretpostavimo da je program koji sadrži prikazani dio koda smješten u `prog.c`. Tada će prevođenje naredbom: `cc -o prog prog.c` proizvesti program u kojem ispis varijable x nije uključen. Prevođenje naredbom `cc -DDEBUG -o prog prog.c` će stvoriti izvršni kod koji uključuje `printf` naredbu jer je simbol `DEBUG` definiran. Mogućnost `-Dsimbol` ima i `Code::Blocks`, ali je nužno napraviti projekt.

4.2.2 Standardna C biblioteka

Standardna C biblioteka sadrži niz funkcija, tipova i makro naredbi. Pripadne deklaracije nalaze se u sljedećim standardnim zaglavljima:

```
<assert.h> <float.h>  <math.h>   <stdarg.h>
<stdlib.h> <ctype.h>  <limits.h> <setjmp.h>
<stddef.h> <string.h> <errno.h>  <locale.h>
<signal.h> <stdio.h>  <time.h>
```

Zaglavlje `assert.h` sadrži makro naredbu `assert` koja omgućava provjeru pretpostavki programera o vrijednosti varijabli i izraza na nekom mjestu u kodu. Naredba izbacuje poruku u slučaju da je pretpostavka programera o vrijednosti pogrešna. Konstrukcija se najčešće koristi u svrhu testiranja programa.

Primjer 4.2.4

`assert` u liniji 8 neće izbaciti nikakvu poruku jer je pretpostavka programera da je vrijednost varijable y u tom trenutku izvođenja programa jednaka 2 točna. Međutim, pretpostavka o vrijednosti izraza u liniji 9 je pogreška, stoga će `assert` izbaciti pogrešku.

```
1 #include <stdio.h>
2 #include <assert.h>
3
4 int main(void){
5
6     int y = 2;
7
```

```
8       assert(y == 2);
9       assert((y+5) == 9);
10
11      return 0;
12  }
```

```
Assertion failed!

Program: ...\AssertPrimjer.exe
File: ...\AssertPrimjer.c, Line 9

Expression: (y+5) == 9
```

Zaglavlje `float.h` sadrži niz konstanti i makro naredbi za rad s realnim vrijednostima dok zaglavlje `limits.h` sadrži niz konstanti i makro naredbi za rad s cjelobrojnim vrijednostima, npr. `char`, `int`, `long`. Zaglavlje `math.h` sadrži niz definicija matematičkih funkcija, konstanti i makro naredbu koje omogućavaju implementiranje složenijih matematičkih operacija. Zaglavlje `<stdarg.h>` sadrži tip `va_list` i makro naredbe koje omogućuju definiranje funkcija s varijabilnim brojem argumenata.

Primjer 4.2.5

Radimo funkciju f s varijabilnim brojem argumenata. Prvi argument definira broj argumenata funkcije. Pretpostavka je da su svi argumenti cijeli brojevi. Funkcija ispisuje vrijednosti svih ulaznih argumenata osim prvog u komandni prozor. Generalniju funkciju možemo napraviti po uzoru na `printf`/`scanf`, gdje je prvi argument format koji definira broj i tip ulaznih argumenanta.

```
1   #include <stdio.h>
2   #include <stdarg.h>
3
4   void f(int broj, ...) {
5       va_list argumenti;//deklaracija liste argumenata
6       va_start(argumenti, broj);//pristup argumentima
7                                 //nakon imenovanog broj
8
9       for (int i = 0; i < broj; ++i) {
10        int argument = va_arg(argumenti, int);//dohvcanje
```

```
11                                                  //argumenta
12          printf("%d␣",argument);
13      }
14      va_end(argumenti);//brisanje liste argumenti
15  }
16
17  int main(void){
18
19  f(5,6,9,2,4,2);
20
21  return 0;
22
23  }
```

IZLAZ:

```
6 9 2 4 2
```

Zaglavlje `stdlib.h` sadrži niz makro naredbi i zaglavlja funkcija za razne namjene kao što su alokacija memorije (npr. `malloc`, `calloc`, `free`), konverziju znakova i stringova u cjelobrojni ili realni tip (npr. `atof`, `atoi`, `strtod`), funkcije `exit`, `bsearch`, `qsort`, tip `size_t` i mnogi drugi. Zaglavlje `ctype.h` sadrži deklaraciju niza funkcija za rad sa znakovima, `string.h` sadrži deklaraciju niza funkcija za rad sa stringovima, `stdio.h` sadrži deklaracije niza funkcija za realizaciju unosa i ispisa podataka iz C programa. Zaglavlje `setjmp.h` definira makro `setjmp`, funkciju `longjmp` i tip varijable `jmp_buf` koji omogućuju skok na određenu predefiniranu liniju tijekom izvršavanja programa.

Primjer 4.2.6

```
1  #include <stdio.h>
2  #include <setjmp.h>
3
4  jmp_buf mjesto_skoka; //varijabla za skok
5
6  void f(int broj)
7  {
8     printf("f(%d)\n", broj);
9     longjmp(mjesto_skoka, broj + 1);//skoci na mjesto skoka
```

```
10                                          //vrati broj+1
11 }
12
13 int main(void)
14 {
15    int brojac = 0;
16    if (setjmp(mjesto_skoka) != 5) //definiraj mjesto skoka
17        f(++brojac);
18    return 0;
19 }
```

IZLAZ:

```
f(1)
f(2)
f(3)
f(4)
```

Zaglavlje `stddef.h` definira razne tipove varijabli i makroe kao što su `ptrdiff_t`. Zaglavlje `errno.h` definira varijablu `errno` koju postavljaju neki sistemski pozivi i funkcije iz standardne biblioteke. Varijabla omogućava detekciju vrste pogrešaka ukoliko do njih dođe. Zaglavlje `locale.h` sadrži postavke specifične za geografsku lokaciju, kao što su format datuma, oznaku valute i slično. Zaglavlje `signal.h` sadrži deklaraciju funkcija, varijabli i makro naredbi za obradu signala[3] prijavljenih tijekom izvršavanja programa. Zaglavlje `time.h` sadrži tipove varijabli, makro definicije i deklaracije funkcija za rad s datumom i vremenom. Postoji i zaglavlje `complex.h`, dodano u standardu `C99`, koje sadrži definicije strutura i funkcija potrebnih za rad s kompleksnim brojevima.

Datoteka zaglavlja `math.h`

U nastavku detaljnije opisujemo funkcije sadržane u datoteci zaglavlja `math.h`. x i y će u nastavku po konvenciji biti varijable tipa `double`, a n varijabla tipa `int`. Sve funkcije vraćaju rezultat tipa `double`.

[3]Signali omogućuju postavljanje stanja ili dojavu informacija procesu od strane korisnika ili operacijskog sustava. Poznatiji signali za prekid procesa na Linux-u su `SIGINT`, nastaje pristiskom tipki `CTRL + C` od strane korisnika, te `SIGKILL` od strane korisnika ili operacijskog sustava

- $sin(x)$ - `sin x`, $cos(x)$ - `cos x`, $tan(x)$ - `tg x`
- $asin(x)$ - `arcsin(x)`$\in [-\frac{\pi}{2}, \frac{\pi}{2}]$, $x \in [-1, 1]$
- $acos(x)$ - `arccos(x)`$\in [0, \pi]$, $x \in [-1, 1]$
- $atan(x)$ - `arctg(x)`$\in [-\frac{\pi}{2}, \frac{\pi}{2}]$
- $atan2(y, x)$ - (x, y) su koordinate točke u ravnini, vraća $arctg\frac{y}{x} \in [-\pi, \pi]$ jer prepoznaje kvadrant. Mjeri kut između x osi i polupravca koji kreće od ishodišta i prolazi kroz točku (x, y).
- $sinh(x)$ - `sh x`, $cosh(x)$ - `ch x`, $tanh(x)$ - `th x`
- $exp(x)$ - e^x
- $log(x)$ - `ln x`, $x > 0$
- $log10(x)$ - $log_{10}x$, $x > 0$
- $pow(x, y)$ - x^y - greška ako $x = 0$ i $y \leq 0$ ili $x < 0$ i y nije cijeli broj
- $sqrt(x)$ - $\sqrt{x}$, $x \geq 0$
- $ceil(x)$ - $\lceil x \rceil$, tipa `double`, najmanji cijeli broj $\geq x$
- $floor(x)$ - $\lfloor x \rfloor$, tipa `double`, najveći cijeli broj $\leq x$
- $fabs(x)$ - $|x|$
- $ldexp(x, n)$ - $x \cdot 2^n$
- $frexp(x, int * exp)$ - ako je $x = y \cdot 2^n$, uz $y \in [\frac{1}{2}, 1 >$, vraća y, a eksponent n sprema na memorijsku lokaciju na koju pokazuje `exp`.
- $modf(x, double * ip)$ - rastavlja x na cjelobrojni i razlomljeni dio, oba istog predznaka kao x. Razlomljeni dio vrati, a cjelobrojni dio spremi na memorijsku lokaciju na koju pokazuje `ip`.
- $fmod(x, y)$ - realni (floating-point) ostatak dijeljenja x/y, istog znaka kao x. Ako je $y = 0$, rezultat ovisi o implementaciji.

`atan(x)` vraća $arctg\ x \in [-\frac{\pi}{2}, \frac{\pi}{2}]$, a `atan2(y, x)` interpretira argumente kao koordinate točke (x, y) u ravnini i vraća $arctg\frac{y}{x} \in [-\pi, \pi]$ jer prepoznaje kvadrant u kojem je točka.

```
double atan(double x);
double atan2(double y, double x);
```

Razliku vrijednosti između `atan` i `atan2` možemo vidjeti u doljnjem kodu:

```
#include <stdio.h>
#include <math.h>
int main(void) {
    double x = -1.0, y = -1.0;
    printf("%f\n", atan(y / x));//0.785398
    printf("%f\n", atan2(y, x));//-2.356194
return 0; }
```

Točni rezultati su $\pi/4$ i $-3\pi/4$.

U nastavku promatramo funkcije:

```
double floor(double x);
double ceil(double x);
```

Rezultat ispisa sljedećeg dijela koda:

```
printf("%g\n", floor(5.2));
printf("%g\n", floor(-5.2));
printf("%g\n", ceil(5.2));
printf("%g\n", ceil(-5.2));
```

je $5, -6, 6, -5$ (zbog %g formata).

Funkcija `double fmod(double x, double y);` vraća realni ostatak pri dijeljenju x/y, gdje ostatak ima isti predznak kao x. Princip je isti kao kod cjelobrojnog dijeljenja: $x =$ cjelobrojni kvocijent $\cdot\ y +$ ostatak, s tim da je $|\text{ostatak}| < |y|$ ili ostatak $= x - ((int)(x/y)) \cdot y$.

Sljedeći odsječak koda ispisuje:

```
printf("%g,␣", fmod(5.2, 2.6)); //0
printf("%g,␣", fmod( 5.57, 2.51)); //0.55
printf("%g,␣", fmod( 5.57, -2.51)); //0.55
printf("%g,␣", fmod(-5.57, 2.51)); //-0.55
printf("%g\n", fmod(-5.57, -2.51)); //-0.55
```

zbog $(int)(5.57/2.51) = 2$ i $5.57 = 2 \cdot 2.51 + 0.55$, itd.

Funkcija `frexp` rastavlja broj x na binarnu mantisu i binarni eksponent.

```
1 double frexp(double x, int *exp);
```

Upotreba funkcije se može vidjeti u doljnjem isječku koda:

```
1 double x = 8.0;
2 int exp_2;
3 printf("%f,␣", frexp(x, &exp_2));//0.500000,
4 printf("%d\n", exp_2);//4
```

Funkcije:

```
1 double exp(double x);
2 double log(double x);
3 double log10(double x);
4 double pow(double x, double y);
```

se upotrebljavaju na dolje opisani način:

```
1 printf("%g\n", log(exp(22)));
2 printf("%g\n", log10(pow(10.0, 22.0)));
```

ispisuje:
22
22
Vidimo da je `log` inverzna funkcija, funkcije `exp`, a `log10(x)` je inverz funkcije `pow(10.0,x)`.

Datoteka zaglavlja `stdlib.h`

Datoteka zaglavlja `<stdlib.h>` sadrži funkcije:

- `qsort` - QuickSort algoritam za općenito sortiranje niza,
- `bsearch` - Binarno traženje zadanog podatka u već sortiranom nizu.

U pozivu gore navedenih funkcija kao stvarni argument moramo zadati funkciju za uspoređivanje podataka u nizu. Funkcija za usporedbu, nazovimo ju `comp`, prima pokazivače na dva objekta koje treba usporediti i vraća rezultat tipa `int`.

```
int comp(const nesto *a, const nesto *b);
```

Rezultat određuje medusobni poredak objekata `*a` i `*b`. Povratna vrijednost je kao kod funkcije `strcmp` za usporedbu stringova.

Funkcija `qsort` sortira niz uzlazno, prema zadanom poretku. Za obrnuti (*silazni*) poredak, treba zamijeniti predznak rezultata funkcije `comp`.

Funkcija za sortiranje niza `QuickSort` algoritmom ima prototip:

```
void qsort(void *base, size_t n, size_t size,
int (*comp) (const void *, const void *));
```

Uzlazno sortiranje stringova je opisano doljnjim programskim isječkom:

```
char rjecnik[3][20] = {"po", "ut", "sri"};
int i;
qsort(rjecnik, 3, 20, strcmp);
for (i = 0; i < 3; ++i)
     puts(rjecnik[i]);
```

Funkcija za binarno traženje zadanog podatka u sortiranom nizu, prema poretku zadanom funkcijom `comp`, ima prototip:

```
void *bsearch(const void *key, const void *base,
size_t n, size_t size,
int (*comp) (const void *, const void *));
```

Funkcija vraća pokazivač na nađeni podatak ako postoji, inače `NULL`.

```
printf("%s\n",
bsearch("ut", rjecnik, 3, 20, strcmp));
```

Pripadni indeks možemo izračunati aritmetikom pokazivača, samo treba paziti na tipove. Pošto smo funkcijama `bsearch` i `qsort` proslijedili funkciju `strcmp` koja kao parametre prima `char *` umjesto `void *`, dobit ćemo upozorenje prevoditelja. Problem možemo popraviti stvaranjem nove funkcije korektnog prototipa unutar koje, uz pretvorbu tipova, pozovemo funkciju `strcmp`.

```
int usporedi(const void *a, const void *b){
return strcmp((char *) a, (char *) b); }
```

Poziv funkcije `qsort` sada glasi:

```
qsort(rjecnik, 3, 20, usporedi);
```

Analogno radimo i poziv funkcije `bsearch`, uz eksplicitnu pretvorbu povratne vrijednosti u `char *`.

```
printf("%s\n",
(char*) bsearch("ut", rjecnik, 3, 20, usporedi));
```

Elegantnije i čitljivije rješenje dobijemo pretvorbom tipa funkcije u zadani pri pozivu funkcija `qsort` i `bsearch`.

```
qsort(rjecnik, 3, 20,
  (int (*) (const void*, const void*)) strcmp);
```

Možemo uvesti i novi tip za funkciju:

```
typedef int (*Comp_f) (const void*, const void*);
qsort(rjecnik, 3, 20, (Comp_f) strcmp);
```

Kod sortiranja rječnika zamjenama pokazivača treba **paziti na tipove u usporedi**.

Primjer 4.2.7

Implementiramo program za sortiranje polja cijelih brojeva.

```
typedef int (*Comp_f) (const void*, const void*);

int usporedi(const int *p_a, const int *p_b){
/* Nije dobro: return *p_a - *p_b;
jer rezultat ne mora biti korektno prikaziv! */
  if (*p_a < *p_b)
     return -1;
  else if (*p_a > *p_b)
     return 1;
  else
    return 0; }

int main(void) {
     int i, polje[4] = {1, 3, -4, 2};
     qsort(polje, 4, sizeof(int),
       (Comp_f) usporedi); /* Cast funkcije */
```

```
17      for (i = 0; i < 4; ++i)
18         printf("%d\n", polje[i]);
19      return 0; }
```

Pri generiranju slučajnih cijelih brojeva koristimo dvije osnovne funkcije:

```
1 int rand(void);
2 void srand(unsigned int seed);
```

Funkcija `rand()` vraća psudo-slučajni cijeli broj u rasponu od 0 do `RAND_MAX`, s tim da `RAND_MAX` mora biti barem $2^{15} - 1 = 32767$ (16-bitni int bez negativnih brojeva). Funkcija `srand(seed)` postavlja sjeme za generator pseudo-slučajnih brojeva na zadanu vrijednost `seed`[4]. Standardno sjeme je 1, ako ga ne postavimo sami. Način korištenja funkcija `rand` i `seed` možemo vidjeti u doljnjem programskom isječku:

```
1 unsigned int seed; int i;
2 printf("%d\n", RAND_MAX); /* 32767 */
3 scanf("%u", &seed);
4 srand(seed);
5 for(i = 1; i <= 10; ++i) printf("␣%6d\n",rand());
```

Generatori pseudoslučajnih brojeva se često koriste kod heurističkih algoritama za brzo ali nepotpuno pretraživanje velikih prostora rješenja. Često se koriste i kod implementacije raznih igara itd.

Funkciju `srand` često koristimo u kombinaciji s funkcijom `time` iz zaglavlja `time.h`. Time možemo omogućiti da se naš program ponaša različito kod svakog pokretanja. Naime, pseudo-slučajan generator će generirati identičan niz pseudo-slučajnih brojeva svaki puta kada koristimo isto sjeme. Takvo ponašanje nije prihvatljivo kod implementacije raznih aplikacija. Jedna važna skupina su video igre (zamislimo da nam oblici padaju uvijek u istom poretku u tetrisu ili da labirint u zmiji izgleda uvijek isto). Zato kao sjeme koristimo

[4]Algoritmi za generiranje pseudo-slučajnih brojeva često kao ulaz primaju sjeme - cijeli broj bez predznaka i na temelju njega generairaju niz brojeva koji imaju određena svojstva slučajno generiranih brojeva. Generirani niz je u potpunosti određen sjemenom i generator će za isto sjeme uvijek generirati isti pseudo-slučajni niz brojeva. Unatoč tome što postoje bolji pristupi za generiranje slučajnih brojeva, pseudo-slučajni generatori su brzi i jednostavni za korištenje, stoga se često upotrebljavaju u primjenama.

broj koji dobijemo kao broj sekundi koji je protekao od 01.01.1970. do paljenja programa. Pošto ćemo program nužno paliti u različitim vremenskim trenucima na istom računalu, korišteno sjeme će biti uvijek različito, pa će generirani pseudo-slučajni niz biti različit kao i ponašanje našeg programa.

Povećanje raspona pseudoslučajnih brojeva na dvostruki kvadratni, ako je prikaziv, možemo postići funkcijom:

```
1 int randint(void) {
2 return RAND_MAX * rand() + rand(); }
```

Raspon 0 do $(RAND\_MAX + 1)^2 - 1$ možemo dobiti funkcijom:

```
1 int randint(void) {
2 static int RAND_BASE = RAND_MAX + 1
3 return RAND_BASE * rand() + rand(); }
```

Raspon možemo dodatno povećati koristeći operator pomaka:

```
1 int randint(void) { /* int od 0 do 2^30 - 1 */
2     return ( rand() << 15 ) + rand(); }
```

Skaliranje nenegativnog slučajnog cijelog broja na zadani interval $[l, u]$ postižemo funkcijom:

```
1 int randint_interval(int l, int u) {
2     return l + randint() % (u - l + 1); }
```

Za dobivanje uniformno distribuiranih realnih brojeva iz intervala $[0, 1]$, bitno je znati `RANDINT_MAX` (za naš `randint` $= 2^{30} - 1$):

```
1 double rand_double() {
2        return (double) randint() / RANDINT_MAX; }
```

Napomena: postoje puno bolji generatori pseudo-slučajnih brojeva od navedenih, npr. Mersenne Twister[5].

U datoteci zaglavlja `<time.h>` deklarirani su tipovi i funkcije za manipulaciju danima, datumima i vremenom. Detaljnije ćemo opisati smo funkcije za vrijeme s ciljem stvaranja jednostavne štoperice za vrijeme izvršavanja pojedinih dijelova programa.

[5] `http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/emt.html`

Deklarirana su dva aritmetička tipa za prikaz vremena: a) `time_t` za prikaz stvarnog kalendarskog vremena i b) `clock_t` za prikaz procesorskog vremena. Stvarno kalendarsko vrijeme (razlika od danog trenutka do $01.01.1970, 00 : 00$) mjeri se funkcijom `time_t time(time_t *tp);` u sekundama. Funkcija vraća navedenu razliku ili -1 ako vrijeme nije dostupno. Ako pokazivač `tp` nije `NULL` onda se izlazna vrijednost sprema i u `*tp`. Štopericu za mjerenje vremena izvršavanja dijela programa implementiramo kao razliku vremena dobivenog funkcijom `time` na kraju i na početku izvođenja zadanog dijela programa. Razliku možemo izračunati funkcijom: `double difftime(time_t time2, time_t time1);` koja vraća `time2 - time1` izraženu u sekundama.

Primjer 4.2.8

Radimo jednostavni program za testiranje mjerenja vremena. Program treba trajati dovoljno dugo da možemo izmjeriti vrijeme izvršavanja, zato koristimo dvije cjelobrojne petlje. U unutarnjoj petlji pozivamo funkciju `rand()`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  int main(void){
5      int i, j;
6      time_t t1, t2;
7      time(&t1); /* Moze i t1 = time(NULL); */
8      for (i = 1; i < 100000; ++i)
9         for (j = 1; j < i; ++j)  rand();
10     time(&t2); /* Moze i t2 = time(NULL); */
11     printf("%g\n", difftime(t2, t1));
12     return 0; } //44s Intel(R) Core(TM) i7-10750H
```

Procesorsko vrijeme mjeri se u broju otkucaja procesorskog sata. Funkcija za očitanje procesorskog vremena je: `clock_t clock(void);`. Funkcija vraća procesorsko vrijeme (u broju otkucaja sata) od početka izvršavanja programa ili -1 ako vrijeme nije dostupno. Simbolička konstanta `CLOCKS_PER_SEC` sadrži broj otkucaja procesorskog sata u jednoj sekundi. Štopericu implementiramo po istom principu razlike vremena, a pretvaranje u sekunde realiziramo funkcijom `dsecnd`.

Primjer 4.2.9

Mjerimo vrijeme izvršavanja programa koji u ugnježdenoj petlji generira slučajne brojeve koristeći procesorsko vrijeme.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

double dsecnd (void) {
return (double)( clock() ) / CLOCKS_PER_SEC;}

int main(void){
    int i, j;
    double t1, t2, time;

    t1 = dsecnd();
    for (i = 1; i < 100000; ++i)
        for (j = 1; j < i; ++j)
            rand();
    t2 = dsecnd();
    time = t2 - t1;
    printf("%g\n", time);
    return 0; } //44.903s,
    //Intel(R) Core(TM) i7-10750H
```

Primjer 4.2.10

Mjerimo vrijeme izvršavanja algortima `qsort` iz `<stdlib.h>` na slučajno generiranom polju od 10^7 cijelih brojeva.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAXN 10000000 /* 10^7 */
int x[MAXN];

double dsecnd (void) {
   return (double)( clock() ) / CLOCKS_PER_SEC; }
//slucajni broj iz [0,2^30-1]
int randint(void) {
   return ( rand() << 15 ) + rand(); }
```

```
12
13 int intcomp(const int *p_a, const int *p_b){
14        if (*p_a < *p_b) return -1;
15        else if (*p_a > *p_b) return 1;
16        else return 0; }
17
18 typedef int (*Comp_fun) (const void*, const void*);
19
20 int main(void){
21     int n = MAXN; /* Broj elemenata u polju. */
22     int i;
23     double start, stop;
24     for (i = 0; i < n; ++i)
25          x[i] = randint();
26
27     start = dsecnd();
28     qsort(x, n, sizeof(int), (Comp_fun) intcomp);
29     stop = dsecnd();
30
31     for (i = 1; i < n; ++i)
32        if (x[i-1] > x[i])
33          printf("␣Greska␣na␣[%d,␣%d]\n", i-1, i);
34
35    printf("␣n␣=␣%8d,␣vrijeme␣=␣%9.3f␣[s]\n",
36                  n, stop - start);
37    return 0; }
```

Vrijeme izvršavanja: $1.67s$ na `Intel(R) Core(TM) i7-10750H` uz `MinGW64` na Windows-u 11.

Bibliografija

[1] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

Index