Ljiljana Despalatović

Programske metode i apstrakcije

IZDAVAČ

Sveučilište u Splitu

Sveučilišni odjel za stručne studije

RECENZENTI

doc. dr. sc. Ljiljana Šerić

Karmen Klarin, pred.

dr. sc. Toma Rončević, viši pred.

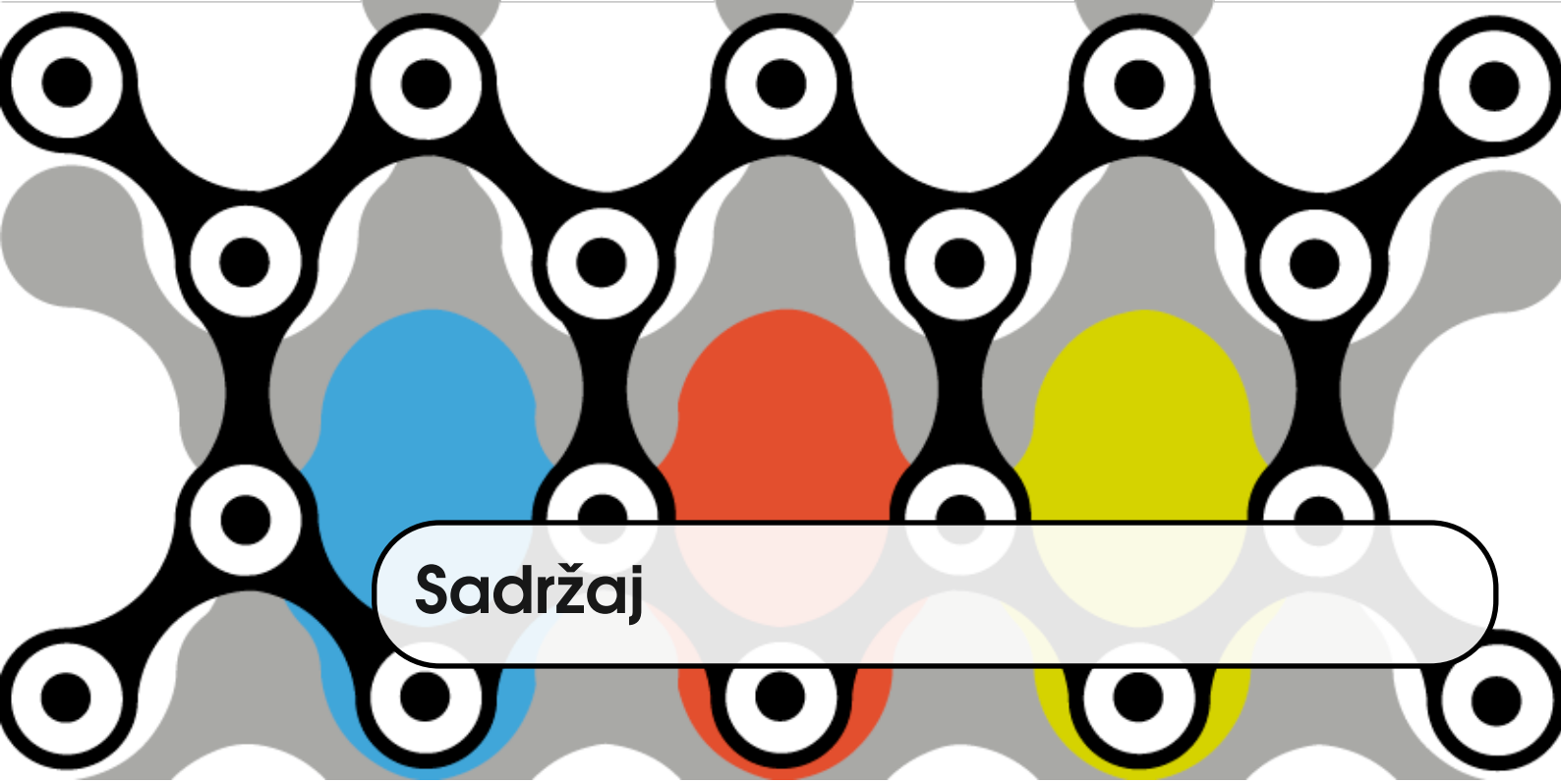
LEKTORICA

Ivanka Kuić

NASLOVNICA

Petar Reić

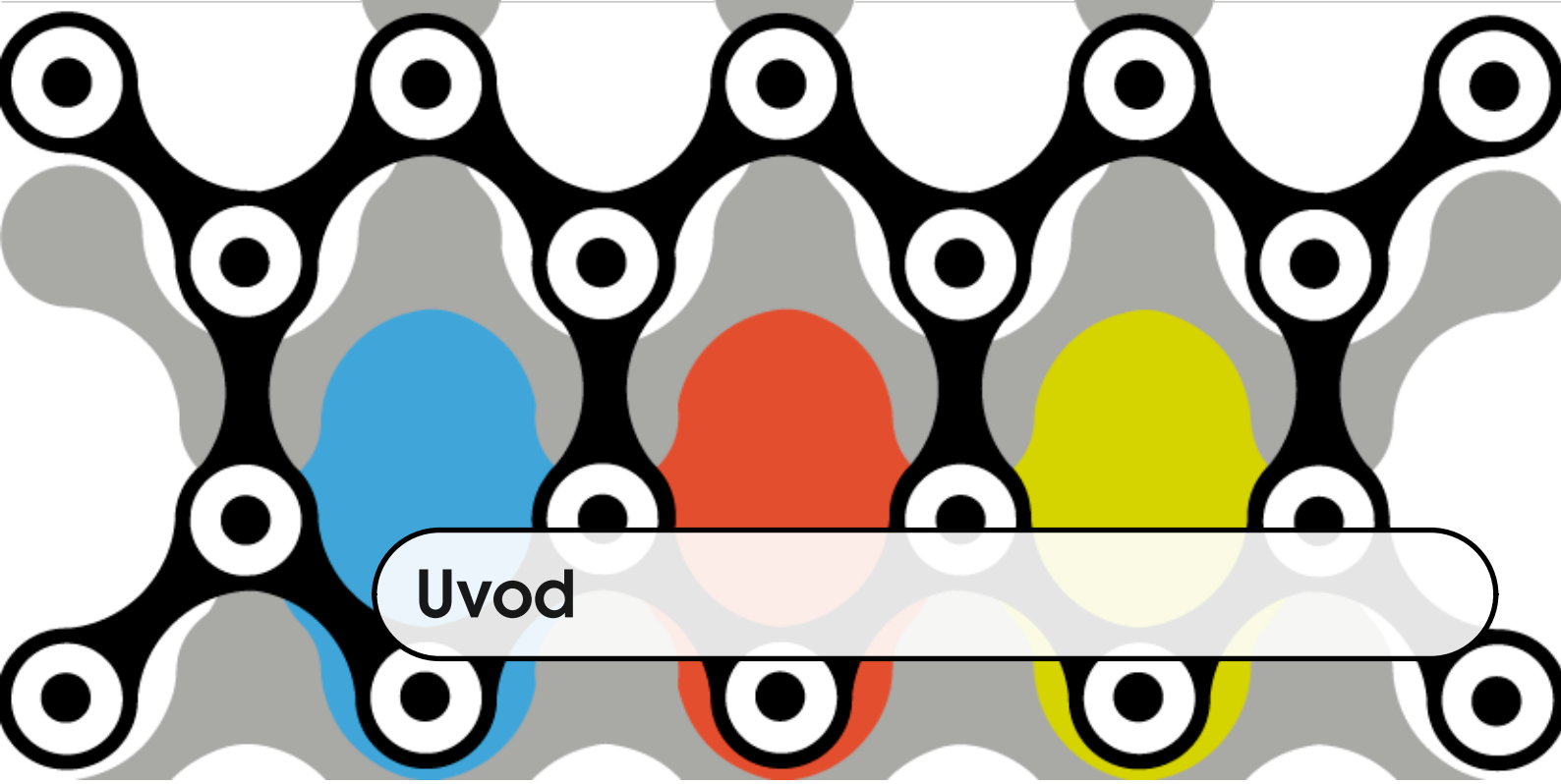
ISBN 978-953-7220-26-6



Sadržaj

Popis slika

Popis tablica



Uvod

1.1 Što je programski jezik?

Svaki program pokrenut na računalu napisan je u nekom programskom jeziku. Tipičnog korisnika ne zanima u kojem je jeziku napisan *browser*, ni operativni sustav, ni program za reprodukciju glazbe. Bitno mu je da radi ono za što je predviđen. Međutim, programeru je bitno i kako radi. Da li je kôd razumljiv, krši li se program i zašto, da li je program prenosiv i da li je brz? Programiranje ne znači naprosto naučiti sintaksu nekog jezika, već i shvatiti kako neki problem pretočiti u "kompjuterski jezik". A što je taj kompjuterski jezik i što kompjuter razumije možda je prvi korak u razumijevanju procesa pisanja programa.

IZVORNI KÔD

Generalno, proces programiranja je sljedeći: napiše se izvorni kôd (engl. *source code*), koji je obična tekstualna datoteka uobičajeno s nastavkom `.c`, prevede se u strojni jezik i pokrene. U trenutku pokretanja program postaje proces: dobiva svoj dio memorije u koji se upisuje, te ulazi u red čekanja na izvršavanje. Procesor dohvaća instrukcije programa i izvršava ih jednu po jednu kada na njih dođe red. Iako se čini da mi prilikom pisanja programa imamo određenu

slobodu, važno je prihvatiti da procesor neće izvršiti ništa drukčije od onoga što je napisano. Doslovno tako. Instrukciju po instrukciju. Ali jednako tako treba znati da problem koji napadamo možemo riješiti na mnogo načina. Način na koji ćemo riješiti problem ovisi o poznavanju programskih jezika, poznavanju algoritama, poznavanju uzoraka koji se ponavljaju, o praksi koju imamo, o našem mentalnom sklopu i raspoloženju. Programski jezik samo je alat, metoda, ostalo je nadogradnja - iskustvo, obrazovanje, razmišljanje, znatiželja.

KOMPAJLER

Programski jezik je skup definiranih pravila koje obrađuje prevoditelj ili kompajler. Kompajler (engl. *compiler*) je program koji naše "ljudskim jezikom" napisane naredbe pretvara u naredbe razumljive procesoru. Pravila su uglavnom određena standardom. Zadnji standard programskog jezika C krije se iza kratice ISO/IEC 9899:2011. Standarde programskih jezika donosi ISO/IEC komisija s oznakom JTC1/SC22. Standard programskog jezika C nalazi se unutar grupe WG14. Standardizacija je važna piscima prevoditelja, ali i ostalim programerima. Pišući svoje programe u skladu sa standardom, jamčimo da će oni biti korektno prevedeni kompajlerima koji su u skladu sa standardom. To je važno jer ne želimo pisati programe koji će biti prevođeni i izvršavati se samo na računalima na kojima su instalirani isti operativni sustavi i isti kompajler kao na našem računalu. Želimo da i netko drugi bez problema prevede i izvrši naš kôd.

STANDARD

U ovom će udžbeniku svi primjeri biti napisani u skladu sa standardom C89 (ANSI C). Na taj način bit ćemo sigurni da će se programi prevoditi i kompajlerima koji su napisani u skladu sa standardima C99 i C11 koji su doneseni kasnije i predstavljaju nadogradnju na C89 standard.

1.2 Podjela programskih jezika

Kada rješavamo neki problem postavlja se pitanje u kojem programskom jeziku pisati. To ovisi o mnogo parametara. Naprimjer, koja nam je ciljana platforma/operativni sustav, je li nam bitna brzina, kolika i kakva je podrška za taj programski jezik i za ciljanu platformu. Međutim, programski jezici se ne dijele

po takvim parametrima. Glavna je podjela programskih jezika prema načinu na koji se prevode. Tu su u igri kompajleri i interpreteri. Osim toga postoji i podjela prema načinu definiranja tipova (*type system*). Nadalje, programski jezici mogu biti proceduralni, objektno-orijentirani, funkcionalni i logički. Neki podržavaju višenitno izvođenje koje je dosta važno u vrijeme višejezgrenih procesora i upotrebe grafičkih procesora za paralelno izvršavanje. Ipak, možda najvažniji parametar pri odabiru programskog jezika predstavlja količina (standardnih) biblioteka koje implementiraju potrebne dijelove programa omogućivši nam pisanje kôda na višoj razini. Pa, krenimo redom...

1.2.1 Kompajleri i interpreteri

Bazična je podjela programskih jezika na jezike koji se kompajliraju i na jezike koji se interpretiraju. Postoje i jezici koji su na određeni način između te dvije grupe. U biti se svaki jezik može implementirati pomoću kompajlera ili interpretera, a čest je slučaj da kompajler prevodi kôd u *bytecode*, koji onda interpreter prevodi u trenutku izvršavanja u izvršni kôd. C/C++ je programski jezik čije su implementacije najčešće kompajlerske.

Kompajliranje znači da se program potpuno prevodi u strojni kôd prije izvršavanja. Kompajler je program kojem je ulaz naš kôd, a izlaz prevedeni program u obliku izvršne datoteke. Na Windows operativnom sustavu izvršne datoteke imaju ekstenziju `.exe`, dok na Unix/Linux operativnim sustavima sama ekstenzija nije određena, ali datoteka mora imati metaoznaku izvršne datoteke (*file permission*). Kompajliranje se obavlja za svaku platformu posebno. Program kompajliran za Windows operativni sustav ne može se izvršavati na Linux operativnom sustavu i obratno.

KOMPAJLER

Interpreter je program koji izvršava programe naredbu po naredbu u trenutku izvođenja. Primjeri interpreterskih jezika su PHP i Perl.

INTERPRETER

Mnoštvo jezika stoji između te dvije mogućnosti. Programi napisani u Pythonu ili Smalltalku mogu se prevoditi putem interpretera, ali i prevesti kompaj-

lerom u *bytecode* koji će se onda izvršavati pomoću *bytecode* interpretera. Takav pristup ima određene prednosti: program se prevodi u *bytecode* univerzalno za sve platforme, a onda se izvodi na različitim platformama koristeći se interpreterima napisanim za tu određenu platformu. U Java programskom jeziku interpreter se krije iza imena *Java Virtual Machine*.

1.2.2 Sustav tipova

SISTEM TIPOVA

Sustav tipova (engl. *type system*)sistem tipova je skup pravila koji definira ponašanje varijabli, vrijednosti i izraza u programu.

Programski se jezici mogu podijeliti na jezike koji imaju tipove (engl. *typed languages*) i jezike koji nemaju (engl. *untyped languages*). U prvom slučaju za svaki tip postoji interval (*range*) u kojem varijabla, vrijednost ili izraz mogu imati vrijednost. Primjerice, varijabla tipa `unsigned char` može poprimiti vrijednost u intervalu [0, 255]. U drugom slučaju podaci su naprosto nizovi bitova različite duljine. Jezici sa tipovima mogu imati eksplicitno definirane tipove, ako su tipovi dio sintakse, kao što je to slučaj u C-u, ili implicitne tipove inače [4].

Provjera tipova se u programskim jezicima sa tipovima može izvršavati u vremenu kompajliranja (engl. *static check*) ili u vremenu izvođenja (engl. *dynamic check*). C je programski jezik u kojem se provjera tipova u programu događa u vrijeme kompajliranja.

Nadalje, za programske se jezike kaže da imaju čvrstu (engl. *strong*) ili labavu (engl. *weak*) provjeru tipova. Čvrsta provjera tipova bi značila da se neće dogoditi greške u izvođenju ako je provjera tipova prilikom prevođenja uspješno izvedena. Češći slučaj je labava provjera tipova, a C pripada toj porodici jezika. Naime, iako provjeru tipova izvodi se *typechecker* program, programer može zaobići ograničenja konverzijom.

TYPECHECKER

1.2.3 Programske paradigme

Klasifikacija programskih jezika je složen posao i može se obaviti s obzirom na razne parametre: tipovi, prevođenje, programska podrška i slično. Međutim, najčešća podjela programskih jezika je s obzirom na paradigmu koju predstavljaju. Postoje četiri osnovne paradigme. To su proceduralna, objektno orijentirana, funkcionalna i logička paradigma. Proceduralni jezici, kakav je i C, sastoje se od procedura/funkcija/metoda koje se sastoje od naredbi koje se sekvencijalno izvode od kojih neke mogu biti pozivi druge procedure/funkcije/metode.

PROCEDURALNI JEZICI

Objektno orijentirani jezici uvode koncept klase i objekta kao instance klase, koji se sastoji od podataka i metoda koje se mogu izvršavati nad njima. Primjeri takvih jezika su Smalltalk, C++ i Java.

OBJEKTNO ORIJENTIRANI JEZICI

Programi pisani u funkcionalnim jezicima računaju izraze, a ne izvršavaju naredbe koje mijenjaju stanje podataka. Temeljni konstrukt u tim jezicima je funkcija u matematičkom smislu koja nema popratne efekte (engl. *side effects*). Primjeri takvih jezika su Haskell, Clojure i Common Lisp. I na kraju, logički jezici, poput Prologa, bazirani su na matematičkoj logici.

FUNKCIONALNI JEZICI

LOGIČKI JEZICI

1.3 Kratka povijest C jezika

Priča o nastanku C jezika ne može se razdvojiti od priče o nastanku UNIX operativnog sustava. Iza oba projekta stoji AT&T Bell Telephone Laboratories, iako u vrijeme samog nastanka nije izgledalo da bilo tko stoji iza nekolicine entuzijasta zaposlenih u toj kompaniji. Razlog za to bio je propast njihovog prethodnog Multics operativnog sustava koji je bio zajednički projekt Bell Labsa, MIT i General Electricsa, i u koji se slilo dosta novaca, a završio je kao promašeni i napušteni projekt.

KEN THOMPSON I DENNIS RITCHIE

Spomenuti entuzijasti su Ken Thompson, Dennis Ritchie, Malcolm Doug McIlroy i Joe Osanna. Bell Labs u to vrijeme (1969.g.) više nije imao interesa za razvoj operativnog sustava i njihovi su zahtjevi za potrebnim hardverom bili

KEN THOMPSON

odbijeni. Bili su hrpa geekova koji nisu imali podršku upravljačkih stuktura kompanije. Ken Thompson je počeo pisati operativni sustav za GE-645, dizajnirati datotečni sustav koji će postati srce Unixa, te je razvio i igru *Space Travel*. Igra je originalno napisana za Multics, zatim prevedena za GECOS (operativni sustav GE mašine) i predstavljala je simulaciju gibanja tijela sunčevog sustava u kojoj je igrač putovao s tijela na tijelo [10]. Međutim, osim lošeg prikaza igre, samo izvođenje na GECOS-u je bilo poprilično skupo, pa je Thompson odlučio prepisati igricu za PDP-7 kojeg su imali i služio im je kao grafički terminal. U međuvremenu je postalo jasno da će Bell Labs uskoro odustati od GE-645, pa je Thompson nastavio razvijati operativni sustav na PDP-7. Space Travel poslužio je kao test aplikacija za novi operativni sustav. Samo portanje igre pokazalo se prilično zahtjevno - da bi uspješno portali igricu morali su napisati paket za floating point aritmetiku, prikaz točaka i debugger. Zajedno sa datotečnim sustavom u razvoju počeo je razvoj Unixa[1].

PDP-7 je bio prilično zastario, pa je skupina zatražila novoizašli PDP-11 koji je bio red veličine jeftiniji od prethodno traženog kompjutera ($65000). Ovaj put su se poslužili i trikom - naveli su da će izraditi sustav za editiranje i formatiranje teksta (*word-processing system*). Operativni sustav nisu spominjali.

Iako je Thompson htio koristiti viši programski jezik, originalni Unix kôd za PDP-7 napisan je u asembleru. Od prvobitne ideje da Fortran bude sistemski programski jezik brzo je odustao, te je kreirao novi programski jezik B temeljen na jeziku BCPL [11]. Samo ime B derivirano je iz imena jezika Bon koje je Thompson razvio u Multics danima, a koji je dobio ime po njegovoj ženi Bonnie ili, po tvrdnji autora, po religiji čiji rituali uključuju mrmljanje magičnih formula. Na žalost,

DENNIS RITCHIE

jezik se pokazao neodgovarajućim, pa je **Dennis Ritchie** krenuo u razvoj jezika C. Do početka 1973. godine napisan je programski jezik C i kompajler koji je omogućio pisanje Unix kernela za PDP-11.

Prvi dokument koji opisuje programski jezik C je *reference manual* koji je Dennis Ritchie objavio kao interni memorandum Bell Labsa [9]. Međutim, prva

[1] www.bell-labs.com/usr/dmr/www/spacetravel.html

prava knjiga objavljena je 1978. godine zvala se „The C Programming Language", a napisali su je Brian Kerningan i Dennis Ritchie. Ta je služila kao standard jezika sve do stvarne standardizacije i poznata je i pod nazivom "*K&R book*".

BEN KERNIGHAN

U cilju donošenja pravog standarda, ANSI (*American National Standards Institute*) je formirao komisiju X3J11 čiji je zadatak bio definirati standard jezika kako bi se pisci kompajlera držali zajedničkih pravila, a programeri bili u mogućnosti pisati portabilan kôd. Napokon je 1989. godine objavljen standard danas poznat pod imenom ANSI C ili C89.

ANSI

C89

Vrlo brzo je ISO/IEC (*International Organization for Standardization/International Electrotechnical Commission*) JTC1/SC22/WG14 grupa za standardizaciju prihvatila taj standard pod imenom ISO/IEC 9899:1990 ili C90. Sljedeća veća izmjena standarda donesena je 1999. godine i taj je standard poznat pod imenom C99. Konačno, zadnja izmjena standarda bila je 2011. godine. Standard je objavljen pod imenom ISO/IEC 9899:2011, a poznat pod imenom C11. Sve ove informacije dostupne su na `http://www.open-std.org/jtc1/sc22/wg14/www/standards`.

C90

C11

1.4 Osnovni ciklus u kodiranju

Većina programa koji danas postoje napisana je u nekome od viših programskih jezika (engl. *high level programming language*). Za razliku od nižih programskih jezika (engl. *low level programming language*) viši programski jezici omogućuju određeni stupanj apstrakcije dijelova računala. Programer ne mora znati detalje upravljanja memorijom ili stanje registara da bi napisao korektan program. Viši programski jezici koriste se elementima običnog jezika i razumljiviji su.

Programski jezik C smatra se višim programskim jezikom, iako se među višim jezicima nalazi među jezicima s najnižom razinom apstrakcije. Programirajući u C-u i dalje moramo razmišljati o memoriji, premda joj sada pristupamo s malo razumljivijim rječnikom. Strojni jezik (engl. *machine code*) je jedini jezik

STROJNI JEZIK

koje mikroprocesor može izvršiti. Svaki procesor ima svoj stojni jezik, iako su oni relativno slični. To je razlog zašto se programi trebaju prevoditi za točno određenu platformu [6]. Dio strojnog kôda može se vidjeti u sljedećem ispisu:

```
1010   0011   0001   1001  (strojni kod)
 ADD    R3,    R1,    R9   (assembler)
```

Da bi se neki program napisan u višem programskom jeziku mogao izvršiti treba ga prevesti u strojni kôd. Kompajler to može napraviti potpuno ili može prevesti u neki jezik između (engl. *intermediate language*), kao što je već spomenuti *bytecode*. Kompajleri napisani za programski jezik C najčešće potpuno prevode program iz višeg jezika u strojni kôd. Taj kôd se naziva **objektni kôd**.

OBJEKTNI KÔD

Strojni kôd može biti izvršni (engl. *executable*) ili se može povezati (engl. *link*) sa nekim dugim objektnim kôdom (bibliotekama). Osim prevedenoga programa, objektni kôd sadrži i neke druge informacije: relokacijske infomacije kojima se koristi linker, programske simbole i informacije za debugiranje.

Osnovni ciklus u programiranju može se predočiti slikom 1.1.

Ovo je krajnje pojednostavljeni prikaz ciklusa. Neposredno prije kompajliranja provodi se predprocesiranje, o kojemu će kasnije biti više riječi. Samo kompajliranje sastoji se od više koraka: leksička analiza, sintaksna analiza, semantička analiza. Ta tri koraka pripadaju fazi analize. Prilikom analize kompajler hvata greške. Nakon te faze dolazi faza sinteze u kojoj se vrši optimizacija kôda, te generiranje kôda [2].

FAZA ANALIZE

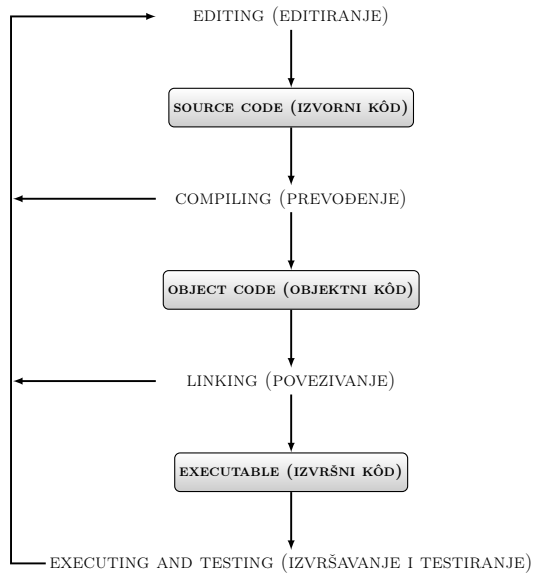
FAZA SINTEZE

1.5 Alati

Da bismo mogli pisati programe u bilo kojem programskom jeziku, potreban nam je editor. Za pisanje programa dovoljan nam je bilo kakav tekst

EDITOR

[2] izvor: http://www.durofy.com/computing/phases-of-compiler-design/



Slika 1.1: Osnovni ciklus u kodiranju

editor (notepad, vi, nano), ali naravno, bolje će nam poslužiti tekst editor koji će znati prepoznati programski kôd i, recimo, bojati ključne riječi određenog jezika. Takvih editora ima mnogo (notepad++, editpad, kate, ...). Izvorni kôd u programskom jeziku C sprema se u datoteke koje imaju nastavak `.c`.

Osim što program trebamo napisati, potrebno ga je i kompajlirati i linkati. Programi koji se nude u tu svrhu su Microsoft Visual C++, Turbo C, pcc (Portable C Compiler), gcc (GNU C compiler), orange c compiler, MinGW itd. Sjetimo se, C kompajler je program kojemu je ulaz ASCII datoteka ili više ASCII datoteka, a izlaz kôd napisan u strojnom jeziku. Sve što nam treba za taj posao je C kompajler.

Kompajler je program koji možemo nabaviti u obliku izvršne datoteke, pa pri- KOMPAJLER
tom trebamo paziti za koju platformu je namjenjen. Ako planiramo kompajlirati

programe za Windows operativni sustav, naći ćemo kompajler za Windowse. To može biti MinGW. Programi napisani za ovaj udžbenik kompajlirani su pomoću GNU gcc kompajlera. Sa Linux operativnim sustavom obično dolazi gcc kompajler, dok s Windows operativnim sustavom ne, pa je potrebno neki kompajler instalirati na računalo. Upotreba GNU gcc kompajlera opisana je u Dodatku A.

Opremljeni editorom i kompajlerom možemo krenuti na posao. Naši programi, barem u početnoj fazi, bit će jednostavni i lako ćemo pisati programe i ispravljati greške. No, međutim, kada programi narastu i počnemo se koristiti složenijim konceptima, dobro će nam doći alati kojima ćemo u vremenu izvođenja (engl. *runtime*) provjeravati stanje svojih varijabli i izraza. Alati koji će nam u tome pomoći su *debuggeri*. Primjer takvih program su gdb (GNU debugger) i idb (Intel debugger).

DEBUGGERI

Sve ovo što je navedeno; editori, kompajleri, linkeri i *debuggeri*, najčešće se koriste zajedno, u obliku IDE (Integrated Programming Environment) programa. Primjer takvih su Microsoft Visual Studio, Code::Blocks ili QtCreator.

IDE

1.6 Prvi C program

U nastavku (primjer 1.1) je dan najmanji C program napisan po ANSI/ISO C standardu. U literaturi je moguće naći i varijante zaglavlja `main` funkcije u kojoj je tip povratne vrijednosti `void`. Međutim, nije sigurno da će se takav program kompajlirati sa svim kompajlerima. Štoviše, ako kompajliramo takav program sa gcc kompajlerom i opcijom `-Wall` dobit ćemo upozorenje "*warning: return type of 'main' is not "int [-Wmain]*".

Primjer 1.1 Najmanji C program

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

Prva linija u primjeru 1.1 je zaglavlje funkcije `main`. Tip povratne vrijednosti funkcije je `int`. Naredbe funkcije se nalaze između para vitičastih zagrada, a ključnom riječju `return` vraćamo vrijednost koja po tipu odgovara tipu povratne vrijednosti definirane u zaglavlju.

Dva su moguća načina kako se može deklarirati `main` funkcija. Jedan je

```
int main(void);
```

a drugi

```
int main(int argc, char **argv);
```

S obzirom na to da je za objašnjenje drugog načina potrebno poznavati složenije koncepte C jezika, mi ćemo se za sada zadržati na prvom načinu.

Ključna riječ `void` na mjestu predviđenom za argumente funkcije može se protumačiti kao "prazno", odnosno možemo reći da funkcija `main` ne prima nikakve argumente.

Ključna riječ `int` s lijeve strane imena funkcije označava tip povratne vrijednosti funkcije. U gornjem slučaju, tip povratne vrijednosti je `int`, te zbog toga funkcija treba vratiti, pomoću ključne riječi `return`, cijeli broj. Običaj je da se u slučaju uspješnoga okončanja programa vrati cijeli broj `0` onome tko je pozvao funkciju. U slučaju `main` funkcije, ona je pozvana od strane operativnoga sustava u trenutku pokretanja programa.

Ovaj program ne radi ništa. Ipak, on je primjer regularnog C programa.

Primjer 1.2 uobičajeni je prvi primjer u većini udžbenika koji se bave C programskim jezikom. Ispisuje pozdravnu poruku korisniku, koristeći se funkcijom

za ispis standardne biblioteke `printf`. Da bismo se koristili `printf` funkcijom ili bilo kojom drugom funkcijom za unos ili ispis iz standardne biblioteke, potrebno je na početku uključiti datoteku `stdio.h` u kojoj su navedeni prototipovi ulazno/izlaznih funkcija standardne biblioteke. Naime, unos i ispis nisu dio jezika samog, već su omogućeni pozivanjem funkcija iz standardne biblioteke.

Primjer 1.2 Pozdrav svijete!

```
#include <stdio.h>
/* hello world */
int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

KOMENTAR

Sljedeća linija u primjeru je komentar. Komentari se pišu između znakova `/*` i `*/` i mogu se protegnuti na više linija. Jednolinijski komentar koji počinje znakovima `//` nije standardni ANSI C komentar, ali ga prihvaća većina kompajlera.

Pisanje komentara dobra je praksa. Komentari se obično pišu na početku programa kako bi objasnili što program radi, na početku funkcije što funkcija radi i unutar kôda da bi objasnili neki dio kôda koji nije jasan.

ESCAPE SEKVENCE

Kombinacija znaka backslash (\) i određenih znakova ili niza znamenki koju vidimo u pozivu funkcije `printf` zove se ***escape* sekvenca**. Takva kombinacija može označavati novi red, tab, navodnike i smatra se jednim znakom. U tablici 1.1 opisane su najčešće *escape* sekvence.

oznaka	opis
`\n`	novi red
`\t`	horizontalni tab
`\v`	vertikalni tab
`\'`	apostrof

\"	dvostruki navodnik
\\	*backslash* znak
\?	upitnik

Tablica 1.1: Escape sekvence

Pomoću funkcije `printf` mogu se na standardni izlaz (konzolu) ispisati i vrijednosti varijabli u određenom formatu. O tipovima i formatima bit će riječi u poglavlju 1.7.

Primjer 1.3 Ispis vrijednosti dvije varijable.

```
#include <stdio.h>
int main(void)
{
    int a = 4, b = 2;
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

Unos sa tastature se obavlja pomoću funkcije `scanf`. Funkcije `printf` i `scanf` su primjeri funkcija koje imaju varijabilni broj parametara. Prvi parametar je format string u kojem se navodi što će biti ispisano, u slučaju funkcije `printf` ili uneseno u slučaju funkcije `scanf`. Unutar format stringa može biti nula ili više oznaka koje predstavljaju format specifikatore koji na prvom mjestu ima znak %, a nakon toga kombinaciju slova koja predstavlja određeni format. Formati su dani u tablici 1.2. U funkciji `scanf` unutar format stringa dan je oblik stringa koji se očekuje od korisnika. Ako u format stringu ima drugih znakova osim oznaka formata, korisnik treba unijeti te znakova.

`scanf`
`printf`

FORMAT STRING

FORMAT SPECIFIKATOR

Funkcija `scanf` kao parametre od drugog mjesta nadalje ima pokazivače, pa stoga ispred imena varijable možemo vidjeti znak & koji predstavlja adresni operator. To možemo promatrati na način da funkcija `scanf` na adresu varijable

ADRESNI OPERATOR

zapisuje vrijednosti koje korisnik unosi s tastature.

Primjer 1.4 Unos vrijednosti dvije varijable.

```
#include <stdio.h>
int main(void)
{
    int a, b;
    printf("input a, b\n");
    scanf("%d %d", &a, &b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

1.7 Varijable i njihove vrijednosti

VARIJABLA

TIP VARIJABLE

Varijabla je objekt u koji možemo pohraniti neku vrijednost. To je zapravo simboličko ime za memorijsku lokaciju čija interpretacija ovisi o **tipu varijable**. Tip varijable određuje kako će varijabla biti spremljena u memoriji, koliko će mjesta zauzimati, koji je opseg vrijednosti koje se može u nju zapisati i koje se operacije mogu nad njom izvršavati.

DEKLARACIJA

DEFINICIJA

Prije upotrebe varijable, varijabla mora biti deklarirana. **Deklariranje** je određivanje tipa varijable, a **definiranje** varijable je rezerviranje memorijskih lokacija u kojima će biti zapisane vrijednosti varijabli. Deklaracija je podatak potreban kompajleru koji će detektirati ako postoji više varijabli s istim imenom i tipom unutar istog bloka programa. Definicijom se određuje gdje se spremaju vrijednosti varijable. Kod lokalnih varijabli oba ova koraka se odvijaju istodobno, te se najčešće zajedno zovu samo deklaracija. Lokalne varijable se deklariraju nakon otvorene vitičaste zagrade (po C89 standardu). Prilikom deklaracije varijabla ne dobiva neku određenu vrijednost, već je njena vrijednost prethodno stanje tih memorijskih lokacija. Količina memorije koja će se dodjeliti nekoj varijabli ovisi o tipu varijable.

Razlika između deklaracije i definicije izražena je kod globalnih varijabli i o tome će više riječi biti u poglavlju 10.

Programski jezik C je jezik sa statičkim tipovima. Jednom deklarirana varijabla kao varijabla određenog tipa ne mijenja svoj tip za vrijeme čitavog života. Nadalje, provjera tipova odvija se u vremenu kompajliranja (engl. *compile time*), a ne u vremenu izvođenja (engl. *runtime*). Programski jezik C je jezik sa slabom tipizacijom (engl. *weakly typed*), što znači da dopušta operacije među različitim tipovima.

PROVJERA TIPOVA

Jednostavni tipovi su cjelobrojni i realni (brojevi s pokretnim zarezom). Osnovni cjelobrojni tipovi su `char` i `int`, a realni `float` i `double`. Dodavanjem određenih modifikatora nastaju novi jednostavni tipovi. Dodavanjem modifikatora, tj. ključne riječi `short` nastaje tip `short int` ili kraće `short`. Dodavanjem modifikatora `long` nastaje tip `long int` ili kraće `long`. Po standardu C99 postoje još `long long` i `long double` jednostavni tipovi.

JEDNOSTAVNI TIPOVI

Ako ispred tipova `int`, `short` i `long` ne stoji nikakav modifikator, pretpostavlja se da je tip s predznakom `signed`. To znači da će se najznačajniji bit (engl. *most significant bit*, msb) tumačiti kao bit predznaka.

`signed`

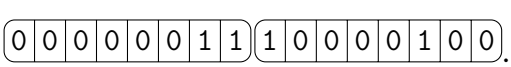
MOST SIGNIFICANT BIT

Dodavanjem modifikatora `unsigned` ispred imena cjelobrojnog tipa dobije se tip bez predznaka. Objekti ovog tipa mogu poprimiti samo pozitivne vrijednosti. Standard ne propisuje da li je tip `char signed` ili `unsigned`. Ispitivanje da li je char `signed` ili `unsigned` može se provesti sljedećim programom.

`unsigned`

Primjer 1.5 Prikaz varijable tipa `short` uz pretpostavku da je veličina koju zauzima objekt tipa `short` dva bajta.

Broj `short a = 900` se zapisuje kao



Vrijednost varijable `a` je pozitivna, jer je bit predznaka jednak `0`.

Pogledajmo kako bi zapisali broj `signed short b = -900`. Vodeći bit treba

biti jedan, a ostatak broja možemo dobiti dualnim komplementom broja 900. Zamjene se sve jedinice u zapisu broja s nulama i na takvu vrijednost doda se broj 1. Novi broj je

$$\mathrm{b}\,\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{0}.$$

Najveći broj koji se može zapisati kao `short` je broj kojem je bit predznaka 0, a sve ostale vrijednosti 1. To je broj $2^{15}-1$, odnosno 32767.

$$\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}.$$

Najmanji broj koji se može zapisati kao `short` je broj kojem je bit predznaka 1, a svi ostali bitovi su nula. To je broj -2^{15} odnosno -32768.

$$\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}.$$

Primjer 1.6 Granice `unsigned char` tipa.

Najmanji broj koji se može zapisati kao `unsigned char` je 0, tj.

$$\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0},$$

a najveći 255 tj.

$$\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}.$$

tip	veličina	raspon	format
`char`	1 byte	$[-2^7, 2^7-1]$	%c
`unsigned char`	1 byte	$[0, 2^8-1]$	%hhu
`short`	2 byte	$[-2^{15}, 2^{15}-1]$	%hd
`unsigned short`	2 byte	$[0, 2^{16}-1]$	%hu
`int`	4 byte	$[-2^{31}, 2^{31}-1]$	%d
`unsigned int`	4 byte	$[0, 2^{32}-1]$	%u
`long`	4 byte	$[-2^{63}, 2^{63}-1]$	%ld
`unsigned long`	4 byte	$[0, 2^{64}-1]$	%lu
`float`	4 byte	$[-3.4 \times 10^{38}, 3.4 \times 10^{38}]$	%f
`double`	8 byte	$[-1.8 \times 10^{308}, 1.8 \times 10^{308}]$	%lf

Tablica 1.2: Veličine jednostavnih tipova 64-bit Linux, GNU gcc kompajler

Veličine tipa nisu konstantne nego se mogu mijenjati s vremenom i platformama. Primjerice, `int` je nekad bio velik 16 bajta, a danas je 32 i možemo pretpostaviti da će se ta vrijednost i u budućnosti mijenjati. Zato se nikada ne pretpostavlja veličina određenog tipa, nego se izračunava upotrebom operatora `sizeof` koji vraća količinu memorije u bajtovima koje zauzima varijabla danog tipa, a maksimalna i minimalna vrijednost koju varijabla može poprimiti se čita iz datoteka `limits.h` ili `float.h` iz predefiniranih vrijednosti (tablica 1.3). Jedino što standard garantira je da je `sizeof(char)` $\leq$ `sizeof(short)` $\leq$ `sizeof(int)` $\leq$ `sizeof(long)`.

tip	veličina	raspon	format
`char`	`sizeof(char)`	[CHAR_MIN,CHAR_MAX]	%c
`signed char`	`sizeof(signed char)`	[SCHAR_MIN,SCHAR_MAX]	%c
`unsigned char`	`sizeof(unsigned char)`	[0,UCHAR_MAX]	%hhu
`short`	`sizeof(short)`	[SHRT_MIN,SHRT_MAX]	%hd
`unsigned short`	`sizeof(unsigned short)`	[0,USHRT_MAX]	%hu
`int`	`sizeof(int)`	[INT_MIN,INT_MAX]	%d
`unsigned int`	`sizeof(unsigned int)`	[0,UINT_MAX]	%u
`long`	`sizeof(long)`	[LONG_MIN,LONG_MAX]	%ld
`unsigned long`	`sizeof(unsigned long)`	[0,ULONG_MAX]	%lu
`float`	`sizeof(float)`	[FLT_MIN,FLT_MAX]	%f
`double`	`sizeof(double)`	[DBL_MIN,DBL_MAX	%lf

Tablica 1.3: Veličine jednostavnih tipova

Primjer 1.7 Program koji ispisuje vrijednosti varijabli svih jednostavnih tipova.

```
#include <stdio.h>
int main(void)
{
    char c = 65;
    signed char sc = 65;
    unsigned char uc = 65;
    short sh = 256;
    unsigned short ush = 256;
    int i = -1;
    unsigned ui = -1;
    long l = 10485761;
    unsigned long ul = 1048576ul;
    float f = 3.141592f;
    double d = 10e10;

    printf("char %c\n", c);
    printf("integer signed %hhd %hd %d %ld\n",
                                        c, sh, i, l);
    printf("integer unsigned %hhu %hu %u %lu\n",
                                      uc, ush, ui, ul);
    printf("realni %f %lf\n", f, d);
    return 0;
}
```

Primjer 1.8 Program koji ispituje da li je `char` tip `signed` ili `unsigned`.

```
#include <stdio.h>
int main (void)
{
    char c = 255;
    if (c > 128)
        printf ("char is unsigned (c = %d)\n", c);
    else
        printf ("char is signed (c = %d)\n", c);
  return 0;
}
```

Primjer 1.9 Program koji ispisuje sve granice intervala svih jednostavnih cjelobrojnih tipova, te veličinu u bajtovima. Broj u oznaci formata označava broj mjesta rezerviranih za ispis, a predznak definira poravnanje.

```
#include <stdio.h>
#include <limits.h>
int main(void)
{

    printf("char: %lu [%20hhd, %-20hhd]\n",
                sizeof(char), CHAR_MIN, CHAR_MAX);
    printf("unsigned char: %lu [%20hhu, %-20hhu]\n",
                sizeof(unsigned char), 0, UCHAR_MAX);
    printf("signed char: %lu [%20hhd, %-20hhd]\n",
                sizeof(signed char), SCHAR_MIN, SCHAR_MAX)
                    ;
    printf("short: %lu [%20hd, %-20hd]\n",
                sizeof(short),SHRT_MIN, SHRT_MAX);
    printf("unsigned short: %lu [%20hu, %-20hu]\n",
                sizeof(unsigned short), 0, USHRT_MAX);
    printf("int: %lu [%20d, %-20d]\n",
                sizeof(int), INT_MIN, INT_MAX);
    printf("unsigned int: %lu [%20u, %-20u]\n",
                sizeof(unsigned), 0, UINT_MAX);
    printf("long: %lu [%20ld, %-20ld]\n",
                sizeof(long), LONG_MIN, LONG_MAX);
    printf("unsigned long: %lu [%20d, %-20lu]\n",
                sizeof(unsigned long), 0, ULONG_MAX);
    return 0;
}
```

Primjer 1.10 Program koji ispisuje sve granice intervala svih jednostavnih realnih tipova, te veličinu u bajtovima.

```
#include <stdio.h>
#include <float.h>
int main(void)
{
```

```
        printf("float: [%e, %e], velicina %lu\n",
                        FLT_MIN, FLT_MAX, sizeof(float));
        printf("double:[%e, %e], velicina %lu\n",
                        DBL_MIN, DBL_MAX, sizeof(double));
        return 0;
    }
```

Ispisivati se može i u formatu koji nije prirodan za određeni tip. Funkcija `printf` dohvaća niz argumenata zapisanih od drugog mjesta nadalje, te vrijednosti varijabli umeće u string koji je prvi argument funkcije, tako da redom umeće vrijednosti varijable na mjesta u stringu određena specijalnim znakom % i oznakom formata. Na taj način može se napisati sljedeće:

Primjer 1.11 Ispis vrijednosti `unsigned` varijable u `unsigned` i `int` formatu.

```
#include <stdio.h>
int main(void)
{
    unsigned n = -1;
    printf("%u, %d\n", n, n);
    return 0;
}
```

U primjeru 1.11 varijabla `n` dobije vrijednost `-1`. Literal `-1` ima u bitovnom zapisu sve jedinice. Međutim, kako je varijabla `n` tipa `unsigned` taj broj se interpretira tako da prvi bit ne predstavlja bit predznaka, točnije u varijablu `n` zapisat će se broj $2^{32} - 1$, tj. `4294967295`. Kad se taj broj ispiše u `%u` formatu, dobije se `4294967295`, a kad se ispiše u `%d` formatu dobije se `-1`.

IMENA VARIJABLI

mogu se sastojati od malih i velikih slova engleskog alfabeta, znamenki i znaka podvlačenja (engl. *underscore*) s tim da znamenka ne smije biti na prvom mjestu. Razlikuju se mala i velika slova, pa su recimo imena `n` i `N` različita.

Običaj je da se imena varijabli u C-u pišu malim slovima, a ako se sastoje od više riječi odvajaju se znakom podvlačenja, npr. `block_id`.

1.7.1 Literali

LITERALI

ili **literalne konstante** (engl. *literal constants*) su vrijednosti koje se ne mogu mijenjati i koje nisu adresabilne (iako je literalna konstanta spremljena na nekoj memorijskoj lokaciji, nema smisla pristupati njenoj adresi). Svaki literal ima vrijednost i tip. Postoje cjelobrojni literali, realni, znakovni i string literali.

Tip **cjelobrojnog literala** je određen oblikom, vrijednošću i sufiksom. Oblik može biti decimalni, oktalni ili heksadecimalni.

```
100 //decimalni oblik

015 //oktalni oblik

0x1f //heksadecimalni oblik
```

Tip cjelobrojnog literala bez sufiksa po *defaultu* je `int`. Međutim, ako je vrijednost literala veća od `INT_MAX`, tip literala će biti `long` za cjelobrojne literale, a `unsigned`, `long`, `unsigned long` za oktalne i heksadecimalne literale (ovisno o vrijednosti). Sufiks `u` ili `U` određuje `unsigned` vrijednosti, a `l` ili `L` `long` vrijednosti. Po C99 standardu postoji još i `ll` ili `LL` sufiks koji označava `long long int` tip.

Primjer 1.12 `0x1ful` je `unsigned long` literal vrijednosti 31.

Realni literali po *defaultu* su tipa `double`, osim ako se ne doda sufiks `f` ili `F` koji označava tip `float` (po C99 standardu `l` ili `L` označava `long double`). Realni literali mogu imati decimalni oblik (`3.141592`) ili biti zapisani u znanstvenoj notaciji (engl. *scientific notation*) (`3.2e-4`).

Znakovni literali pišu se u jednostrukim navodnicima i njihov je tip `int`. Vrijednost znakovnih literala je pripadajuća ASCII vrijednost znaka. Npr. vrijednost literala `'0'` je `48`.

1.7.2 Konstante

KONSTANTE

Varijable mogu imati konstanstne vrijednosti. Možemo ih deklarirati ključnom riječi `const` koja osigurava da se vrijednost varijable neće promijeniti.

Primjer 1.13 Vrijednost `const` varijabli ne može se promjeniti.

```
int main(void)
{
    const int b = 10;
    float const pi = 3.14;
    b = 14;                    // error!
    return 0;
}
```

1.8 Zadaci

1. Instalirajte neku od IDE aplikacija, kreirajte projekt te izvedite sve zadatke iz prethodnih poglavlja.
2. Kreirati C projekt te napisati program koji ispisuje "Finally PMA!".

```c
#include <stdio.h>
int main(void)
{
    printf("Finally PMA!\n");
    return 0;
}
```

3. Napisati program u kojem se unose dva cijela broja, zbrajaju i ispisuje rezultat.

```c
#include <stdio.h>
int main(void)
{
    int a, b, c;
    printf("Input two integers\n");
    scanf("%d %d", &a, &b);
    c = a + b;
    printf("sum %d\n", c);
    return 0;
}
```

4. Napisati program koji zamjenjuje vrijednosti dviju varijabli unesenih putem tipkovnice. Zamijenjene vrijednosti ispisati.
5. Napisati program koji računa kvocijent dva unesena decimalna broja. Ukoliko je drugi broj nula, ispisati poruku "Zabranjeno je dijeliti s nulom!!!".
6. Prepisati i izvršiti sljedeći kôd:

```c
#include <stdio.h>
#include <math.h>
int main(void)
{
```

```
    char ch1, ch2;
    printf("Input two lowercase letters:\n");
    scanf(" %c %c", &ch1, &ch2);
    printf("Distance between %c and %c is %d",
                    ch1, ch2, abs(ch1 - ch2));
    printf("%c and %c ASCII values are %d
                    and %d\n", ch1, ch2, ch1, ch2);
    return 0;
}
```

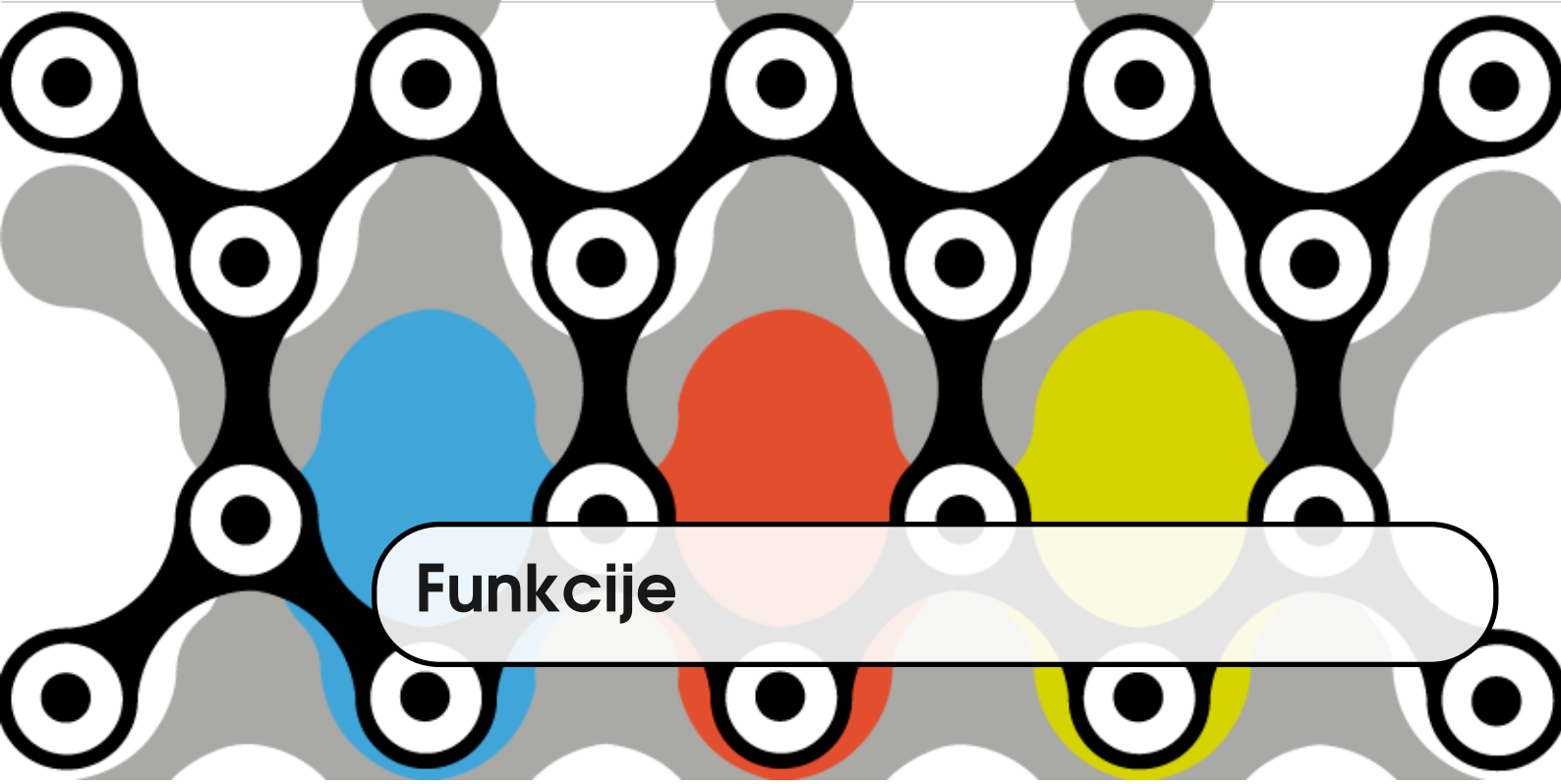
7. Napisati program u kojem se unose dva broja i jedan znak (’+’, ’-’, ’*’ , ’/’) te se ovisno o znaku izvršava odgovarajuća operacija.

```
#include <stdio.h>
int main(void)
{
    char ch;
    float x, y;
    printf("Input two numbers:\n");
    scanf("%f %f", &x, &y);
    printf("Input char:\n");
    scanf(" %c", &ch);
    if (ch == '+')
      printf("sum %f\n", x + y);
    else if (ch == '-')
      printf("difference %f\n", x - y);
    else if (ch == '*')
      printf("product %f\n", x * y);
    else if (ch == '/' && y != 0) /* undefined
                                  division by zero */
      printf("quotient %f\n", x / y);
    else
      printf("Incorrect input!\n");
    return 0;
}
```

Uočite prazninu ispred prvog znaka %c u pozivu `scanf` funkcije. Uklonite je i ponovo pokrenite program. Što primjećujete?

8. Napisati program za pretvaranje vrijednosti varijable tipa `int` u `short` vrijednost. Ako je `int` broj izvan opsega `short` vrijednosti, dodijeliti mu vrijednost `SHRT_MIN` i `SHRT_MAX` (tj.-32768, 32767).

9. Napisati program koji generira pseudoslučajni broj iz zatvorenog intervala `[a, b]`, gdje varijable `a` i `b` unosi korisnik. Za generiranje pseudoslučajnog broja treba se koristiti funkcijom `rand()` definiranoj u `stdlib.h`, koja generira broj u rasponu od 0 do `RAND_MAX`. Za transformaciju generiranog broja `x` u broj `y` iz intervala `[a, b]` koristite `y = a + rand()%(b-a+1)`.



Funkcije

2.1 Značenje pojma funkcija u računarstvu

Pisanje programa unutar jednog tzv. glavnog programa, što je, u slučaju programskog jezika C, `main` funkcija, možda je početnicima najlakše. Međutim, takav pristup vrlo nas brzo dovodi do gomile kôda u kojem će se dijelovi ponavljati, bit će nepregledan i težak za održavanje. Takve probleme možemo smanjiti pisanjem procedura ili funkcija koje će nam omogućiti da dijelove programa koji čine neku manju zaokuženu cjelinu izdvojimo u dijelove koje ćemo po potrebi pozivati i uključivati u naš kôd.

Nekoliko je naziva za takvu cjelinu i najčešće ne odražavaju razlike u koncepciji nego dolaze iz različitih programskih jezika i stilova programiranja. To su **funkcija**, **procedura**, **subrutina**, **podprogram** i **metode**. Metode se razlikuju od ostala četiri pojma po tome što tako nazivamo funkcije koje su članovi klase u objektno orijentiranim jezicima. Ostala četiri pojma označavaju isto: dio kôda koji prima `0` ili više parametara i vraća `0` ili više rezultata. U nekim jezicima uobičajeno je procedurama zvati funkcije koje nemaju povratne vrijednosti ili subrutinama funkcije koje vraćaju više vrijednosti, ali to nisu konceptualno

FUNKCIJA
PROCEDURA
SUBRUTINA
PODPROGRAM
METODE

značajne razlike. U programskom jeziku C zovemo ih naprosto funkcije.

Pojmom funkcije ne koriste se samo proceduralni jezici, već se njime u velikoj mjeri koriste i objektni jezici uvelike koriste, obično kao metode objekata, ali i kao globalne funkcije. Funkcionalni jezici, što se i u samom imenu dâ naslutiti, funkcije zaista stavljaju u prvi plan (*first class citizents*) i približavaju ih matematičkom pojmu funkcije. U čistim funkcionalnim jezicima funkcije nemaju popratne efekte (engl. *side effects*), što znači da ona neće mijenjati vrijednosti izvan svog dosega ili ispisivati rezultat na ulazno/izlazne uređaje. U C-u funkcije ne moraju biti čiste funkcije (engl. *pure function*). One imaju deterministička svojstva: računaju povratne vrijednosti u matematičkom smislu funkcije, ali istodobno omogućavaju popratne efekte. Nitko nam ne brani napisati funkciju koja računa kvadrat decimalnog broja i za svaki broj vrati uvijek isti razultat, a da uz to ispisuje na terminal sam broj ili njegov kvadrat. Ukratko, funkcije koje pišemo u proceduralnom jeziku mogu biti čiste, mogu biti kombinacija (računati rezultat, ali i imati popratne efekte) ili mogu imati samo popratne efekte.

POPRATNI EFEKTI

ČISTE FUNKCIJE

Na najjednostavnijem nivou o funkcijama razmišljamo tako da želimo kôd napraviti čitljivim i ne želimo promjene u kôdu raditi na više mjesta nego što je potrebno. Stoga kada nam se dogodi da isti niz naredbi trebamo napisati više puta, taj niz naredbi nam je dobar kandidat za izdvajanje u funkciju koju ćemo po potrebi pozivati. Osim toga, dobra praksa je u funkciju izdvojiti dio kôda koji čini neku funkcionalnu cjelinu. Primjerice, ako pišemo funkciju koja za neki cijeli broj vraća njegovu prvu znamenku i znamo da ćemo taj broj dobiti tako što ćemo podijeliti broj s najvećom potencijom broja deset koja je manja od broja, a koju ćemo dobiti tako da od broja znamenki u broju oduzeti broj `1`, onda nam je prirodno napisati funkciju koja računa broj znamenki danog broja.

Primjer 2.1 Funkcija koja vraća broj znamenki u broji potrebna nam je za računanje prve znamenke broja.

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

```
int count_digits(int n)
{
    int cnt = 0;
    while(n != 0)
    {
        cnt++;
        n = n / 10;
    }
    return cnt;
}

int first_digit(int n)
{
    return n / pow(10, count_digits(n) - 1);
}

int main(void)
{
    int n;
    printf("input number\n");
    scanf("%d", &n);
    printf("first digit: %d\n", first_digit(n));
    return 0;
}
```

Treba uočiti da za računanje broja znamenki možemo koristiti i logaritamsku funkciju, jer brojevi između dviju potencija od 10 imaju isti cjelobrojni dio logaritma tj. za $10^k \leq n < 10^{k+1}$, n ima $k+1$ tj. $(int)log_{10}(n)+1$ znamenki.

```
int count_digits(int n)
{
    if (n <= 0)     /* log racunamo samo za pozitivne br.
       */
        return -1;
    return (int)log10(n)+1;
}
```

Primjer 2.2 Funkcija koja računa umnožak dva broja. Iako trivijalan, primjer

nam pokazuje kako jednom napisanu funkciju možemo pozivati više puta, ovisno o kontekstu.

```
#include <stdio.h>
int multi(int x, int y)
{
    return x * y;
}

int main(void)
{
    int m, n;
    scanf("%d %d", &m, &n);
    printf("the product is %d\n", multi(m, n));
    printf("the sum of squares is %d\n",
                   multi(m, n) + multi(m, n));
    printf("the square sum is %d\n", multi(m+n, m+n));
    return 0;
}
```

Funkcija je imenovani blok naredbi koji prima vrijednosti, izvršava naredbe unutar tijela funkcije (bloka), te vraća vrijednost i kontrolu onom dijelu kôda koji ju je pozvao. Može se pozivati iz drugih dijelova C programa.

Svaki C program sastoji se od barem jedne funkcije. To je funkcija `main` koja predstavlja ulaznu točku u program. Uobičajeno je da se za svaku logičku cjelinu napiše funkcija.

Primjer 2.3 Program koji se sastoji od više funkcija.

```
#include <stdio.h>

int is_odd(int k)
/* returns 1 if k is even, 0 otherwise */
{
    return k%2 == 0;
```

```
}

float series(int n)
/* 1 - 1/2 + 1/3 - 1/4 + ... +- 1/n */
{
    float s = 0, i;
    for (i = 1; i <= n; ++i)
        if (is_odd(i))
            s = s - 1./i;
        else
            s = s + 1./i;
    return s;
}
```

```
int main(void)
{
    int n;
    float s;
    printf("input number ");
    scanf("%d", &n);
    s = series(n);
    printf("%f\n", s);

    return 0;
}
```

2.2 Definicija funkcije

Opći oblik definicije funkcije je:

DEFINICIJA FUNKCIJE

```
type_r function_name(type_1 a1, type_2 a2,...,type_n an)
{
    function body
}
```

gdje je

- `type_r` tip povratne vrijednosti
- `function_name` ime funkcije
- `type_1,...,type_n` tipovi argumenata
- `a1,...,an` argumenti

Za imena funkcija vrijedi isto pravilo kao i za imena varijabli. Ime se može sastojati od malih i velikih slova engleskog alfabeta, znamenki i znaka podvlačenja, s tim da znamenka ne smije biti na prvom mjestu. Uobičajeno je imena funkcija pisati malim slovima, a riječi u imenu odvajati znakom podvlačenja kao u primjeru 2.3.

2.3 Tip povratne vrijednosti

Tip povratne vrijednosti može biti `void`, ako funkcija ne vraća nikakvu vrijednost, ili bilo koji jednostavni tip, struktura, unija ili pokazivač. Ako funkcija ima definiran tip povratne vrijednosti, unutar tijela funkcije svaki tok izvršavanja mora završiti naredbom `return izraz;`, gdje izraz treba po tipu odgovarati tipu povratne vrijednosti ili se može napraviti implicitna konverzija u traženi tip.

Primjer 2.4 Funkcija za računanje apsolutne vrijednosti decimalnog broja.

```
1 float absolut(float a)
2 {
3     if (a < 0)
4         return -a;
5     else
6         return a;
7 }
```

U primjeru 2.4 za negativne vrijednosti varijable `a`, funkcija vraća vrijednost `-a`, a u suprotnom vraća `a`. Kako naredba `return` označava izlazak iz funkcije, `else` se može izostaviti (vidi primjer 2.5). Ako je vrijednost varijable `a` nenega-

tivna neće biti izvršena naredba u liniji 4, već će se izvršavanje nastaviti od linije 5.

Primjer 2.5 Funkcija za računanje apsolutne vrijednosti cijelog broja (kraći zapis).

```
1 float modul(float a)
2 {
3     if (a < 0)
4         return -a;
5     return a;
6 }
```

Funkcija koja ima za tip povratne vrijednosti `void` ne vraća ništa. U tom slučaju može se i ne mora pisati naredba `return;`.

Primjer 2.6 Funkcija bez povratne vrijednosti.

```
void print_positive(int a)
{
    if (a <= 0)
        return;
    printf("%d\n", a);
}
```

2.4 Poziv funkcije i prijenos argumenata

Funkciju pozivamo navodeći njeno ime i odgovarajući broj (i tip) argumenata između okruglih zagrada. Prilikom komapjliranja, kompajler provjerava postoji li prototip funkcije sa navedenim imenom i odgovarajućim brojem i tipom argumenata. Kada program u izvršavanju dođe do poziva funkcije, izvršit će se

prijenos argumenata, te naredbe iz tijela pozvane funkcije.

FORMALNI ARGUMENTI
PARAMETRI
STVARNI ARGUMENTI

Argumenti navedeni u zaglavlju funkcije zovu se **formalni argumenti** ili **parametri** funkcije. Izrazi koji se prilikom poziva funkcije nalaze na mjestu argumenata zovu se **stvarni argumenti**.

CALL-BY-VALUE

U programskom jeziku C argumenti se prenose po vrijednosti (**call-by-value**). To znači da se prilikom poziva funkcije vrijednosti stvarnih argumenata **kopiraju** u (pozicijom) određene formalne argumente. Iz toga slijedi da funkcija **ne može mijenjati vrijednosti stvarnih argumenata**.

Primjer 2.7 Funkcija koja ne radi ništa.

```
1  void swap(int a, int b)
2  {
3      int t = a;
4      a = b;
5      b = t;
6  }
7  int main(void)
8  {
9      int x = 24, y = 42;
10     printf("x = %d, y = %d\n", x, y);
11     swap(x, y);
12     printf("x = %d, y = %d\n", x, y);
13     return 0;
14 }
```

U primjeru 2.7 varijable `x` i `y` u pozivu funkcije `swap` (linija 11) su stvarni argumenti. Njihove se vrijednosti (`24` i `42`) kopiraju u odgovarajuće formalne argumente funkcije `swap`. Tako formalni argument `a` dobiva vrijednost `24`, a formalni argument `b` dobija vrijednost `42`. U tijelu funkcije `swap` vrijednosti varijabli `a` i `b` se zamjenjuju. Nakon završetka izvođenja funkcije program se nastavlja izvoditi od mjesta nakon poziva funkcije. Varijable `x` i `y` ostaju nepromjenjene.

Primjer 2.8 Poziv funkcije sa povratnom vrijednošću. Kako se izrazi oblika `a = izr;` izvršavaju tako da se prvo izvrši desna strana izraza, a onda vrijednost desne strane pridruži varijabli sa lijeve strane, to će se u liniji 8 prvo izvršiti poziv funkcije `sum(x, y)`, a onda će se povratna vrijednost funkcije `sum` zapisati u varijablu `s`.

```
1  float sum(float a, float b)
2  {
3      return a + b;
4  }
5  int main(void)
6  {
7      float s, x = 24, y = 42;
8      s = sum(x, y);
9      printf("sum: %f\n", s);
10     return 0;
11
12 }
```

Povratna vrijednost funkcije se može i odbaciti. Npr. funkcija `printf` kao povratnu vrijednost vraća broj ispisanih znakova. No, najčešće se taj broj odbacuje.

Mehanizam poziva funkcije

MEHANIZAM POZIVA FUNKCIJE

Prilikom poziva funkcije događa se sljedeće [13]:

1. Izračunavaju se vrijednosti argumenata.
2. Dodjeljuje se memorijski prostor (*stack frame*) za sve lokalne varijable i parametre funkcije.
3. Vrijednost svakog stvarnog argumenta se kopira u odgovarajući formalni argument funkcije (standard ne propisuje redoslijed). Ako je potrebno, izvršava se konverzija tipova.
4. Izvršava se tijelo fukcije sve dok se ne izvrši naredba `return` ili dok se ne

izvrši posljednja naredba u tijelu funkcije.

5. Vrijednost `return` izraza se izračunava i vraća kao povratna vrijednost funkcije. Obavlja se konverzija, ako je potrebna.
6. Oslobađa se *stack frame* memorija. Lokalne varijable više ne postoje.
7. Povratna vrijednost funkcije se zapisuje na mjesto poziva funkcije, a dio programa iz kojeg je pozvana funkcija se nastavlja izvoditi.

2.5 Deklaracija funkcije

DEKLARACIJA
PROTOTIP

Prije poziva funkcija treba biti deklarirana. Deklaracija (ili prototip) je potpis funkcije koja kompajleru omogućava provjeru ispravnosti poziva funkcije.

```
type_r function_name(type_1 a1, type_2 a2,...,type_n an);
```

Imena argumenata `a1,...,an` mogu biti izostavljena.

Ako se definicija funkcije nalazi u istoj datoteci u kojoj se nalazi i poziv funkcije, i to prije prvog poziva, definicija funkcije je ujedno i deklaracija.

Ako se definicija funkcije nalazi nakon poziva funkcije, potrebno je dodati deklaraciju ispred prvog poziva.

Primjer 2.9 Definicija funkcije `quotient` nalazi se nakon poziva funkcije. Potreban je prototip prije poziva.

```
#include <math.h>
float quotient(int br, int naz);
int main(void)
{
    int x = 3, y = 0;
    float kv = quotient(x, y);
    if (kv != kv) /* provjera da li je kv nan */
        printf("not a number\n");
```

```
        else
            printf("kvocijent je %f\n", kv);
        return 0;
    }
    float quotient(int br, int naz)
    {
        if (naz == 0)
            return NAN;
        return (float)br/naz;
    }
```

Mogu se koristiti i funkcije koje nisu predhodno deklarirane. Kompajler pretpostavlja da je tip povratne vrijednosti takve funkcije `int`, a ne pretpostavlja ništa o broju i tipu argumenata.

Primjer 2.10 Testirajte sljedeće programe:

```
int main(void)
{
    float pi = 3.1459f;

    printf("cjelobrojni dio je\n");
    printf("%d\n", cijeli(pi));
    printf("%d\n", cijeli(2.87));
    printf("%d\n", cijeli(2));
    return 0;
}
int cijeli(double x)
{
    return (int)x;
}
```

```
int cijeli(double x);
int main(void)
{
    float pi = 3.1459f;
```

```
        printf("cjelobrojni dio je\n");
        printf("%d\n", cijeli(pi));
        printf("%d\n", cijeli(2.87));
        printf("%d\n", cijeli(2));
        return 0;
    }
    int cijeli(double x)
    {
        return (int)x;
    }
```

2.6 Rekurzija

Rekurzija je koncept koji označava svođenje problema na manji problem istog tipa. Taj koncept postoji u matematici (rekurzivne relacije), umjetnosti (M. C. Escher), lingvistici (Chomsky, Interfaces+recursion=language), u prirodi (fraktali), u dječjem svijetu ("Pošla koka u dućan", babuške), legendama (Hanoi tornjevi) i raznim drugim područjima. U računarstvu pričamo o rekurzivnim strukturama podataka (liste, stabla), rekurzivnim algoritmima (backtracking, divide and conquer) i, naravno, rekurzivnim funkcijama. REKURZIJA

Ideja rješavanja problema pomoću rekurzije svodi se na to da se problem podijeli na manje podprobleme istog tipa, zatim se svaki podproblem podijeli na podpodprobleme istog tipa i tako dalje, sve dok se pod...podproblem ne može rješiti na jednostavan način. Taj način se zove **bazni slučaj**.

Pogledajmo jednostavan primjer rekurzije: dobili ste hrpu numeriranih sličica (npr. 64) koje želite poredati po broju, ne bi li ih brže zalijepili u album sa sličicama. Kako bi to napravili?

Iterativni postupak, usporedi svaku sa svakom sličicom i ako nisu u dobrom redoslijedu zamijeni ih, obavio bi (ako ne ponavljamo usporedbe i ne uspoređujemo ih sa samim sobom) oko 2000 usporedbi i, u najgorem slučaju, isto toliko zamjena. Generalno, složenost ovakvog algoritma je reda veličine n^2. S druge strane, možemo pokušati složiti sličice malo drukčije. Odaberemo jednu sličicu (pivot) i sve koje su manjeg broja stavimo na lijevu stranu, a sve većeg broja na desnu. Odabranu sličicu pridružimo bilo kojoj hrpi.

Svaku hrpu sortiramo na isti način: odaberemo jednu sličicu i sve sličice s manjim brojem stavimo na jednu hrpu, a sve s većim na drugu hrpu. Nastavljamo postupak dok hrpa ne postane toliko mala da je možemo jednostavno složiti. Kada bi ovo pretočili u program, dijeljenje bi se zaustavilo kada bi hrpa imala samo jednu sličicu. Složenost ovog algoritma je $nlogn$. Zašto? U svakom smo koraku prošli sve sličice (`n`) ne bi li ih razvrstali sa prave strane pivota. Nadalje, u svakom koraku smo hrpu podijelili na dva dijela. U idealnom slučaju (dobro

odabran pivot) je u prvom prolazu bilo 64 sličice u hrpi, u drugom 32, pa 16, 8, 4, 2 i na kraju 1. Dakle, napravili smo 6 podjela. To je $log_2 64$. Dakle, sve skupa $nlog_2 n$ operacija.

REKURZIVNE FUNKCIJE

Rekurzivne funkcije su funkcije koje pozivaju same sebe. Pritom pri svakom pozivu funkcija dobiva novi dio *stack* memorije, tj. *stack frame*. Kako memorija nije beskonačne veličine, postupak u nekom trenutku treba zaustaviti tj. potrebno je odrediti bazni slučaj: slučaj kada prestaje pozivanje funkcije. Inače dolazi do prekoračenja steka (engl. *stack overflow*).

Tri su pravila kojih se treba držati prilikom pisanja rekurzivnih funkcija:

1. rekurzivni algoritam mora imati bazni slučaj,
2. rekurzivni algoritam mijenja stanje parametara prema baznom slučaju,
3. rekurzivni algoritam poziva sam sebe.

Primjer 2.11 Isprobajte sljedeći program. Što možete zaključiti o ovako napisanoj rekurzivnoj funkciji `fun`?

```
#include <stdio.h>
void fun(int n)
{
    printf("poziv funkcije za %d\n", n);
    fun(n-1);
}
int main(void)
{
    fun(5);
    return 0;
}
```

Dakle, želimo zaustaviti pozivanje funkcije u nekom trenutku. Uvjet za zaustavljanje ovisi o problemu koji modeliramo. Pretpostavimo u prethodnom primjeru da želimo da se funkcija poziva samo za pozitivne brojeve, te da se zaustavlja kad vrijednost broja `n` dođe do nula.

Primjer 2.12 Isprobajte sljedeći program. Što će se dogoditi ako korisnik unese negativan broj? Kako modelirati problem da ne dođe do prekoračenja steka.

```
#include <stdio.h>
void fun(int n)
{
    if (!n) return;
    printf("poziv funkcije za %d\n", n);
    fun(n-1);
}
int main(void)
{
    int n;
    printf("unesi n\n");
    scanf("%d", &n);
    fun(n);
    return 0;
}
```

U prethodnom primjeru funkcija je samo ispisivala vrijednosti parametra. Najčešće funkcije ipak računaju nekakve vrijednosti. Recimo, želimo izračunati sumu kvadrata brojeva do `n`. Modificirajmo prethodni primjer. U svakom koraku sumi dodajemo vrijednost kvadrata parametra tj. `n*n`.

Primjer 2.13 Rekurzivna funkcija koja računa sumu kvadrata brojeva do `n`.

```
#include <stdio.h>
int fun(int n)
{
    if (!n) return 0;
    printf("poziv funkcije za %d\n", n);
    return n*n + fun(n-1);
}

int main(void)
{
```

```
    int n;
    printf("unesi n\n");
    scanf("%d", &n);
    if (n >= 0)
       printf("suma kvadrata je %d\n",fun(n));
    return 0;
}
```

Primjer se može napisati i na drugi način. Prođite debuggerom kroz sve pozive funkcije i promotrite tok izvršavanja.

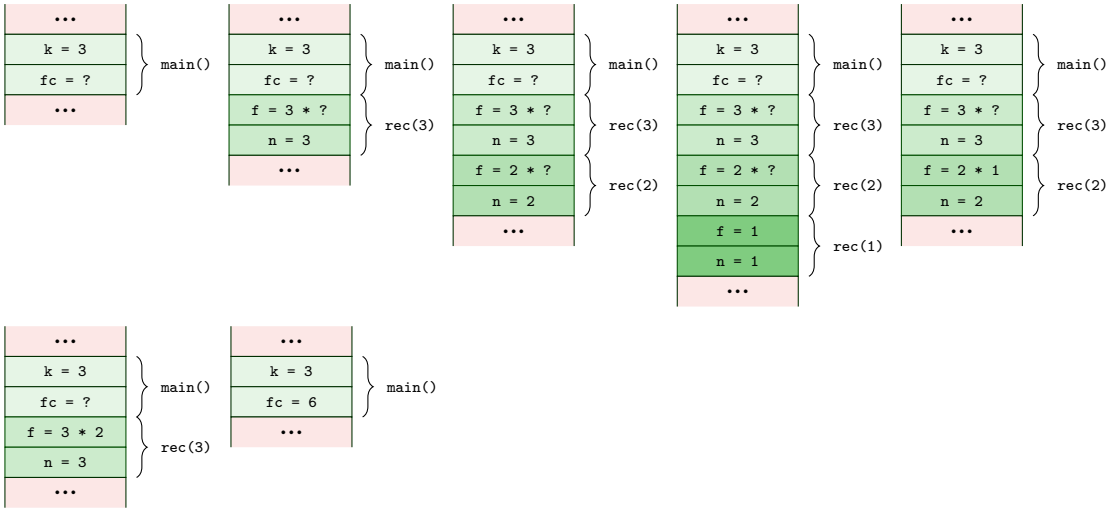
```
int fun(int n)
{
    int k = 0;
    if (!n) return 0;
    printf("poziv funkcije za %d, k je %d\n", n, k);
    k = n*n + fun(n-1);
    printf("parcijalna suma je %d\n", k);
    return k;
}
```

Iz prethodnog primjera lako je vidjeti kako bismo modelirali neke slične probleme, primjerice rekurzivno pomnožili prvih `n` prirodnih brojeva. Umnožak prvih `n` prirodnih brojeva zove se **faktorijel** i označava se `n!`. Vrijedi $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n$.

```
1 unsigned fact(unsigned n)
2 {
3     unsigned f = 1;
4     if (n <= 1)
5         return f;
6     f = n * fact(n - 1);
7     return f;
8
9 }
```

```
10 int main(void)
11 {
12     unsigned k = 3;
13     unsigned fc = fact(k);
14     printf("%u! = %u\n", k, fc);
15     return 0;
16 }
```

Pogledajmo kako se puni (i prazni) *stack* prilikom poziva funkcije `fact` u sljedećem, vrlo pojednostavljenom prikazu.



2.6.1 Rekurzija vs. iteracija

Rekurzija spada u koncepte koji su programerima početnicima teški za usvajanje. Međutim, kada se usvoji, postaje moćna strategija za rješavanje različitih problema. Po Robertsu [13] postoje dva načina na koja se može promatrati program (ili bilo koji objekt). Jedan je *redukcionistički* - da bi shvatili cjelinu, moramo shvatiti djelove, a drugi *holistički* pretpostavlja da je cjelina puno više od djelova koji je sačinjavaju. Kod programiranja je stalna izmjena ova dva principa, međutim za shvaćanje rekurzivnih principa, a i za kreiranje algoritama općenito, naglasak je na holističkom pristupu. Od tuda i naziv našeg kolegija "Programske metode i apstrakcije": metode su detalji, implementacija, a apstrakcija je obuhvat-

nija slika nekog problema. Kod rekurzije redukcionistički princip lako dovodi do toga da se problem teže razumijeva. Pokušaj praćenja poziva rekurzivne funkcije (*call stack*) je jednostavan dok se radi o linearnoj rekurziji (samo jedan rekurzivni poziv unutar funkcije). Međutim, čim rekurzija nije linearna, praćenje poziva se zakomplicira.

Primjer 2.14 Što vraća ova funkcija za `n = 6` i `c = 2`?

```
float sto_radim(unsigned n, unsigned c)
{
    if (n<=4) return c;
    return c*n + sto_radim(n/2, c) + sto_radim(n/4, c);
}
```

Način na koji treba promatrati rekurzivni algoritam je holistički. Ne treba sagledati sve detalje izvršavanja algoritma, već pokušati sagledati algoritam "odozgo", pretpostavljajući da algoritam, što se tiče detalja, naprosto radi. To Roberts naziva *leap of faith*. Ako je algoritam dobro postavljen i implementiran, a bazni slučajevi dobro definirani, nema potrebe za proučavanjem detalja implementacije.

Hoće li se neki problem riješiti rekurzivnim ili iterativnim putem, ovisi o samom problemu. Sljedeća dva primjera usporedit će oba pristupa i izdvojiti njihove prednosti i mane.

Fibonaccievi brojevi su problem poznat kao "problem zečeva". Spomenut je u knjizi Leonarda Fibonaccia *Liber Abbaci* 1202. godine i bavi se rastom populacije zečeva tijekom određenog perioda. Pitanje koje postavlja je "koliko će parova zečeva biti nakon određenog perioda" ako se zečevi razmnožavaju pod sljedećim (idealnim) preduvjetima:

1. počinjemo s jednom ženkom i s jednim mužjakom zečeva (ti zečevi upravo su rođeni),

2. zec će postati zreo za oplodnju nakon mjesec dana života,
3. gestacijski period zeca je jedan mjesec,
4. kad dostigne zrelost, ženka će se okotiti svakog mjeseca,
5. ženka će uvijek okotiti jednog muškog i jednog ženskog zeca,
6. zečevi nikad ne umiru.

Ako označimo sa f_n broj parova zečeva u n-tom mjesecu možemo zaključiti sljedeće. U početnom mjesecu $n = 0$ broj parova zečeva je 1. Nakon mjesec dana i dalje je broj parova zečeva 1, jer je zečica tek postala zrela za oplodnju. Nakon mjesec dana (za $n = 2$) originalni par je dobio mladunčad, te je ukupan broj parova $f_2 = 2$. Sljedeći mjesec mladi zečevi su tek spremni za oplodnju, ali su originalni zečevi stigli napraviti još jedan par mladunaca, te je broj parova porastao na $f_3 = 3$. U četvrtom mjesecu prvi par mladunaca je dobio mladunčad, i originalni par zečeva je dobio mladunčad, pa na tri postojeća para dodajemo još ta dva para. U tom trenutku možemo generalizirati problem i reći da je na postojeći broj parova zečeva iz prethodnog mjeseca dodan još broj parova zečeva koji su postojali dva mjeseca ranije, jer su oni u međuvremenu dobili svaki po još par mladunaca. Dakle, problem se može zapisati kao $f_n = f_{n-1} + f_{n-2}$. To je rekurzivna relacija. Uz početne uvjete $f_0 = 1$ i $f_1 = 1$, relaciju lako pretvorimo u algoritam:

Algoritam 1: Rekurzivni algoritam za problem zečeva

Data: n - broj mjeseci od rođenja prvog para
Result: broj zečeva nakon n mjeseci
if *ako je prošlo nula ili jedan mjesec od rođenja prvih zečeva* **then**
    broj parova zečeva je 1;
else
    broj parova zečeva je jednak broju parova zečeva iz prošlog mjeseca,
    uvećan za broj parova zečeva koji su postojali prije dva mjeseca;
end

Problem se može riješiti i iterativno. U svakom trenutku ukupni broj parova zečeva jednak je broju parova zečeva koji su postojali prošli mjesec, plus broj

parova koji su postojali prije dva mjeseca, a koji su sada dobili još po par zečeva. Pogledajmo algoritam.

Algoritam 2: Iterativni algoritam za problem zečeva

Data: n - broj mjeseci od rođenja prvog para,
x - broj parova prije dva mjeseca,
y - broj parova prije mjesec dana,
z - broj parova danas
Result: broj zečeva nakon n mjeseci
x = 1 je stanje nakon 0 mjeseci;
y = 1 je stanje nakon 1 mjeseca;
while *dok ne dođemo do n mjeseci* **do**
    if *ako je prošlo nula ili jedan mjesec od rođenja prvih zečeva* **then**
        broj parova zečeva je 1;
    else
        rezultat z je x + y;
        x = y;
        y = z;
    end
end

Usporedimo implementaciju ova dva algoritma.

Iterativno:

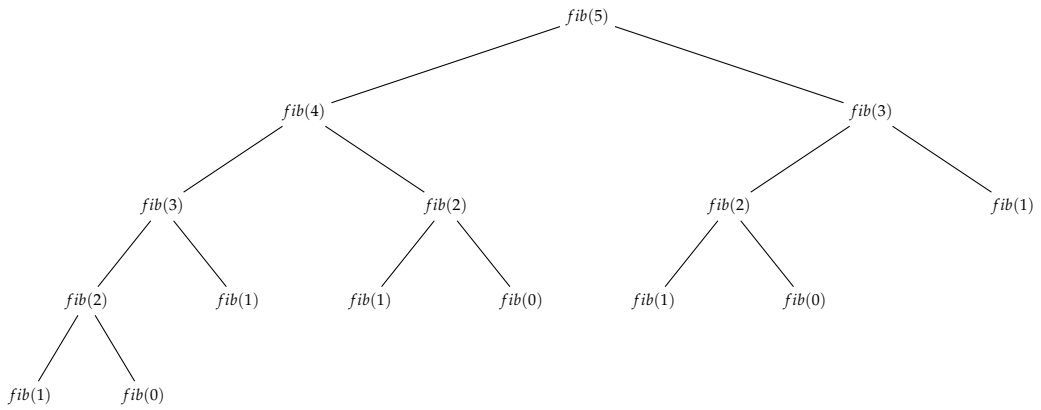
```
unsigned fib_it(unsigned n)
{
    unsigned x, y, z, i = 2;
    x = y = z = 1;
    while (i <= n)
    {
        z = x + y;
        x = y;
        y = z;
        i = i + 1;
    }
    return z;
}
```

Rekurzivno:

```
unsigned fib(unsigned n)
{
    if (n <= 1)
        return 1;
    return fib(n - 1) + fib(n - 2);

}
```

Usporedimo li gornji kôd možemo zaključiti da je rekurzivno rješenje prirodnije i elegantnije. Međutim, ovakvo rekurzivno rješenje nije efikasno. Naime, rekurzivni pozivi za većinu parametara se ponavljaju, te na donjoj slici vidimo da se kod računanja petog fibonaccijevog broja, funkcija `fib` za vrijednost parametara 1 poziva 5 puta.



Taj primjer se često koristi kao argument protiv rekurzije. Međutim, nije problem u rekurziji, već u algoritmu. Pogledajmo sljedeću rekurzivnu funkciju:

```
int fib(int n, int a, int b)
{
    if (n == 0) return a;
    if (n == 1) return b;
    return fib(n - 1, b, a + b);
}
```

Primjer 2.15 Za sumu prvih n prirodnih brojeva najefikasniji način je upotreba matematičke formule. Usporedite!

```
unsigned suma_it(unsigned n)
{
    unsigned s = 0, i;
    for (i = 1; i <= n; ++i)
        s += i;
    return s;
}

unsigned suma_rek(unsigned n)
{
    if (n == 0)
        return 0;
    return n + suma(n-1);
}

unsigned suma_mat(unsigned n)
{
    return n * (n + 1) / 2;
}
```

2.7 Riješeni zadaci

1. Napisati funkciju koja računa sumu prvih `n` trokutnih brojeva. Trokutni brojevi su brojevi oblika $\frac{1}{2} * \texttt{i} * \texttt{(i + 1)}$, za `i` prirodan broj. Problem riješiti iterativno i rekurzivno.

```
unsigned triangle_nums_iter(unsigned n)
/* iterativno rjesenje */
{
    unsigned i, s = 0;

    for (i = 1; i <= n; i++)
        s += i * (i + 1) / 2;
    return s;
}

unsigned triangle_num_rek(unsigned n)
/* rekurzivno rjesenje */
{
    /* ako je n jednak 0 ili 1 - kraj rekurzije*/
    if (n <= 1) return n;

    return n * (n + 1) /2 + triangle_num_rek(n - 1) ;
}
int main(void)
{
    unsigned n;
    printf("unesi n\n");
    scanf("%u",&n);
    printf("suma trokutnih iterativno %u\n",
        triangle_num_iter(n));
    printf("suma trokutnih rekurzivno %u\n",
        triangle_num_rek(n));
    return 0;
}
```

2. Napisati funkciju koja za dva unesena znaka vraća koliko daleko se nalaze u engleskom alfabetu (udaljenost je pozitivan broj). Ako jedan od znakova nije slovo, funkcija vraća -1.
   Napomena: Za provjeru je li znak slovo možete se koristiti funkcijom `int`

`isalpha(int c);` iz `ctype.h`, a za pretvaranje znaka u veliko slovo funkciju `int toupper(int c);`.

```
#include <ctype.h>
int distance(char a, char b)
{
    int d;
    if (!isalpha(a) || !isalpha(b))
        return -1;
    d = toupper(a) - toupper(b);
    if (d < 0 )
        return -d;
    return d;

}
```

3. Napisati funkciju koja generira pseudoslučajni broj iz zatvorenog intervala `[a, b]`, gdje su `a` i `b` argumenti funkcije.
   Za generiranje pseudoslučajnog broja koristiti se funkcijom `rand()` definiranom u `stdlib.h`, koja generira broj u rasponu od 0 do `RAND_MAX`.
   Za transformaciju generiranog broja `x` u broj `y` iz intervala `[a, b]` upotrebljavajte `y = a + rand()%(b-a+1)`

```
#include <stdlib.h>
#include <time.h>
int generate(int a, int b)
{
    if (a > b)
    {
        int t = a; a = b; b = t;
    }
    srand(time(NULL));
    return a + rand()%(b - a + 1);
}
```

4. Napisati funkciju koja za dva broja ispituje je li u zapisu jedan završetak drugog.
   Primjer: Za brojeve 126 i 32126, funkcija vraća 1, a za brojeve 123 i 32125

vraća 0.

Ideja: Jedan broj (manji) je završetak drugog (većeg) ako je on ostatak pri djeljenju većeg broja sa prvom potencijom broja 10 koja je veća od manjeg broja. Npr. 32125 % 1000 je 125, pa je 125 završetak broja 32125.

```
#include <math.h>
int count(int m)
/* broji znamenke u broju*/
{
    if (n <= 0)
        return -1;
    return (int)log10(n)+1;
}

int zavrsetak(int m, int n)
{
    int k, t;
    if (m < n) n ^= m ^= n ^= m; // zamjena
       vrijednosti
    k = (int)pow(10, count(n));
    if (m % k == n) return 1;
    else return 0;

}
```

5. Napisati rekurzivnu funkciju za ispisivanje prirodnog broja u binarnom obliku.

```
void print_bin(unsigned n)
{
    if (!n) return;
    print_bin(n/2);
    printf("%d", n%2);
}
int main(void)
{
    unsigned n;
    printf("unesi broj\n");
    scanf("%u", &n);
    print_bin(n);
    printf("\n");
```

```
    return 0;
}
```

6. Napisati rekurzivnu funkciju koja vraća sumu prvih n kvadrata prirodnih brojeva.

```
/* s = 1**2 + 2**2 + 3**2 + ... + n**2 */
/* s(n) = s(n-1) + n**2 */

unsigned sum_of_squares(unsigned n)
{
    if (n == 1) return 1;
    return n * n + sum_of_squares(n - 1);
}

int main(void)
{
    unsigned n;
    scanf("%u", &n);
    printf("suma kvadrata je %u\n", sum_of_squares(n));
    return 0;
}
```

7. Napisati rekurzivnu funkciju koja vraća sumu svih brojeva do n koji su djeljivi sa 3, a nisu sa 5.

```
/* s = 3 + 6 + 9 + 12 + 18*/
/* s(n) = s(n-1) + a(n)
    a(n) = n , ako je djeljiv s 3, nije sa 5
    a(n) = 0
*/
unsigned sum_divisible(unsigned n)
{
    int k = 0;
    if (n < 3) return 0;
    if (n % 3 == 0 && n % 5 != 0) k = 1;
    return k * n + sum_divisible(n - 1);
}

int main(void)
{
```

```
    unsigned n;
    scanf("%u", &n);
    printf("suma djeljivih sa 3, a ne sa 5 je %u\n",
       sum_divisible(n));
    return 0;
}
```

8. Napisati rekurzivnu funkciju koja provjerava da li prirodni broj u binarnom zapisu ima sve jedinice, npr. $111_2 = 3_{10}$ ima sve jedinice.

```
int all_ones(unsigned n)
{
    unsigned k = n & 1;
    if (!n) return 1;
    if (k) return all_ones(n >> 1);
    else return 0;
}

unsigned count_ones(unsigned n)
/* broji jedinice rekurzivno */
{
    unsigned k = n & 1, s;
    if (!n) return 0;
    s = count_ones(n >> 1);
    if (k) return s + 1;
    else return s;
}
void print_bin(unsigned n)
/*ispisuje sve znamenke - rekurzivna*/
{
    unsigned k = n & 1;
    if (!n) return;
    print_bin(n >> 1);
    printf("%u",k);
}

int main()
{
    unsigned n;
    while (1)
    {
        printf("Unesi poz. cijeli broj, nulu za kraj\n");
```

```
        scanf("%u", &n);
        if (!n) break;
        printf("binarni zapis\n");
        print_bin(n);
        printf("\nbroj jedinica %u\n", count_ones(n));
        all_ones(n) ? printf("sve jedinice\n") : printf("
           nisu sve jedinice\n");
    }
    return 0;
}
```

9. Napisati rekurzivnu funkciju koja provjerava ima li prirodni broj u heksadecimalnom zapisu sve jedinice.

```
int all_ones(unsigned n)
/* za strogo pozitivne brojeve */
{
    if (!n) return 1;
    if (n%16 == 1) return all_ones(n/16);
    else return 0;
}
void print_hexa(unsigned n)
/*ispisuje sve znamenke*/
{
    int k;
    if (!n) return;
    print_hexa(n/16);
    k = n%16;
    if (k > 9)
        printf("%c", k + 'A' - 10);
    else
        printf("%u", k);
}

int main()
{
    unsigned n;
    while (1)
    {
        printf("Unesi poz. cijeli broj, nulu za kraj\n");
        scanf("%u", &n);
        if (!n) break;
```

```
            print_hexa(n);
            /* ispis hexa moze i ovako*/
            printf ("\n%x\n", n);
            all_ones(n) ? printf("ima sve jednice u hexa\n")
                : printf("nema sve jedinice u hexa\n");
        }
        return 0;
    }
```

10. Napisati rekurzivnu finkciju koja vraća umnožak kvadrata znamenki prirodnog broja.

```
unsigned product_digits_square(unsigned n)
{
    int zn;
    if (!n) return 1;
    zn = n%10;
    return zn * zn * product_digits_square(n/10);
}

int main(void)
{
    unsigned n;
    do
    {
        scanf("%u", &n);
    }while(!n);
    printf("produkt kvdrata znamenki je %u\n",
       product_digits_square(n));
    return 0;
}
```

11. Napisati rekurzivnu funkciju koja računa broj koji je obratan od originalnog, npr. za 12345 obratni broj je 54321.

```
#include <stdio.h>
/*  funkcija koji racuna broj koji je obratan od
    originalnog,
    npr. za 12345 obratni broj je 54321
*/
int reverse(int n, int novi)
```

```
{
    if(!n)
        return novi;
    return reverse(n/10, novi*10 + (n%10));
}
/*
    test na broju 12343
    1. poziv - reverse(1234, 3)
    2. poziv - reverse(123, 34)
    3. poziv - reverse(12, 343)
    4. poziv - reverse(1, 3432)
    5. poziv - reverse(0, 34321)
    n == 0 -> vrati novi
*/

int main()
{
    unsigned n;
    scanf("%u", &n);
    n == reverse(n, 0) ? printf("palindrom\n") : printf(
        "nije palindrom\n");
    return 0;
}
```

12. Napisati rekurzivnu funkciju koja ispisuje sve znamenke svaku u svom redu. Ne zna se unaprijed broj znamenki u danom broju.

    Primjer: Za broj 2342 ispisuje

    2

    3

    4

    2

```
#include <stdio.h>
void print_dec(unsigned n)
{
    unsigned zn = n%10;
    if (!n) return;
    print_dec(n/10);
    printf("%u\n",zn);
}
```

```
int main()
{
    unsigned n;
    scanf("%u", &n);
    print_dec(n);
    return 0;
}
```

13. Napisati rekurzivnu funkciju koja za uneseni broj *n* ispisuje vrijednost prve znamenke.

```
void print_first_sec(unsigned n)
{
  if (!n) return;
  if (n/100 == 0 && n > 9)
  {
      printf("prva i druga %u\n", n);
      return;
    }
  print_first_sec(n/10);
}
```

14. Za prirodni broj `n` funkcija ispisuje niz brojeva koji se dobiju tako da se počevši sa `n` parni prepolove, a neparni pomnože sa 3 i doda im se jedan. Niz završava jedinicom (Collatzova slutnja). Problem riješiti rekurzivno. Funkcija treba računati u koliko se koraka došlo do jedinice.

```
#include <stdio.h>

int collatz(int n)
{
    printf("%d\t", n);
    if (n == 1) return 1;
    if(n % 2 == 0) return 1 + collatz(n/2);
    else return 1 + collatz(3 * n + 1);

}
int main()
{
```

```
    int n;
    scanf("%d", &n);
    printf("\n%d\n", collatz(n));
    return 0;
}
```

15. Napisati rekurzivnu funkciju koja proverava da li je uneseni broj koračni broj tj. da li mu je razlika susjednih znamenki 1.

```
int stepping_num(unsigned n)
{
  if (n < 10) return 1;
  if (n%10 - (n/10)%10 == 1 || n - (n/10)%10 == -1)
     return stepping_num(n/10);
  return 0;
}
```

16. Napisati rekurzivnu funkciju koja za uneseni broj n ispisuje vrijednost prve dvije znamenke.

```
int print_first_digit(unsigned n)
{
  if (n < 10 && n > 0)
       return n;
  return print_first_digit(n/10);
}
```

2.8 Zadaci za vježbanje

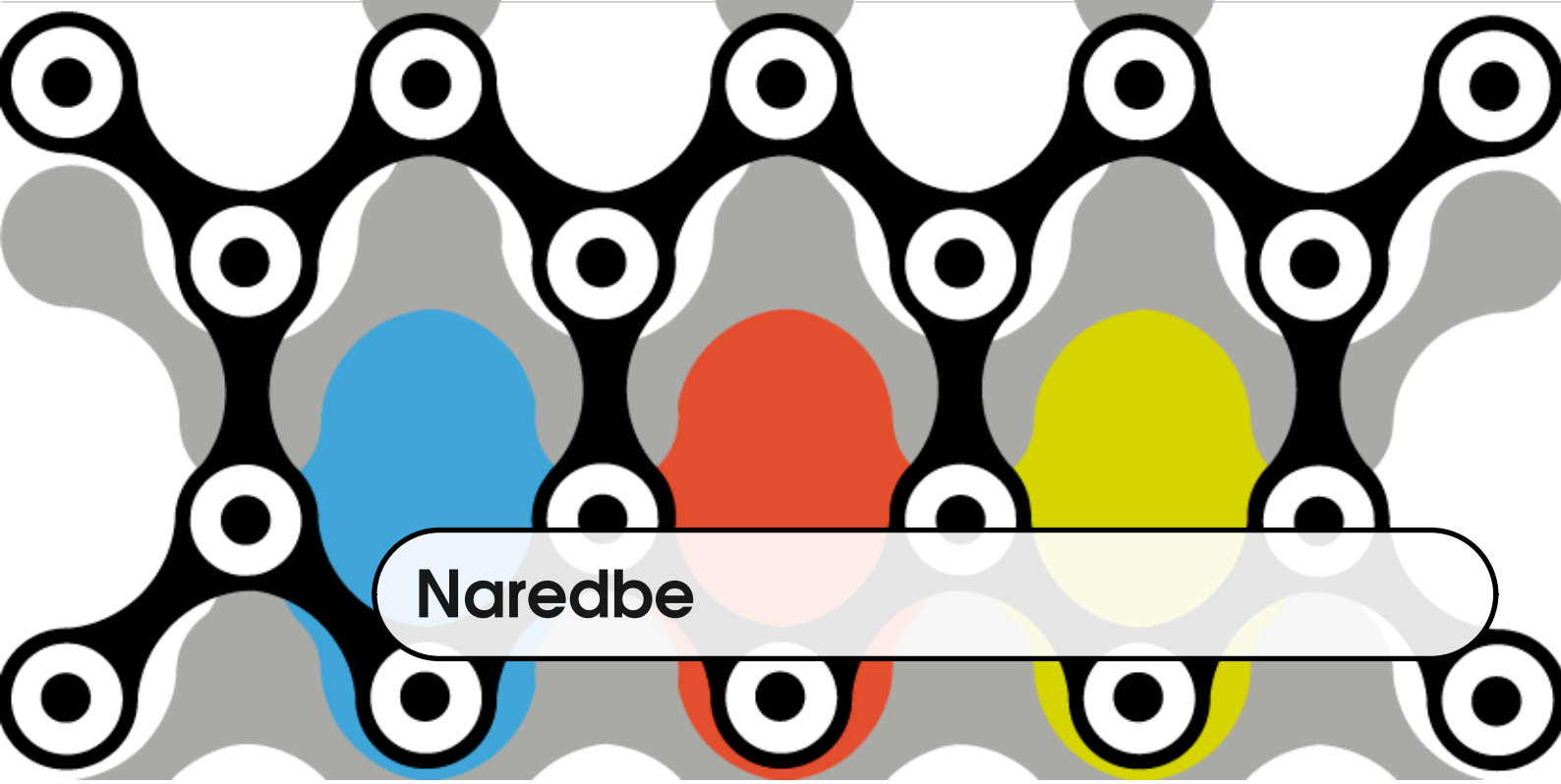
1. Napisati funkciju koja za dva unesena znaka vraća koliko daleko se nalaze u engleskom alfabetu (udaljenost je pozitivan broj). Ako jedan od znakova nije slovo, funkcija vraca -1.
   Napomena: Za provjeru je li znak slovo možete se koristiti funkcijom `int isalpha(int c);` iz `ctype.h`, a za pretvaranje znaka u veliko slovo funkciju `int toupper(int c);`.
2. Napisati funkciju koja generira pseudoslučajni broj iz zatvorenog intervala `[a, b]`, gdje su `a` i `b` argumenti funkcije.
   Za generiranje pseudoslučajnog broja koristiti se funkcijom `rand()` definiranom u `stdlib.h`, koja generira broj u rasponu od 0 do `RAND_MAX`.
   Za transformaciju generiranog broja `x` u broj `y` iz intervala `[a, b]` upotrebljavajte `y = a + rand()%(b-a+1)`
3. Napisati funkciju koja iterativno računa umnožak prvih `n` neparnih brojeva.
4. Napisati funkciju koja rekurzivno računa umnožak prvih `n` neparnih brojeva.
5. Napisati funkciju koja ispisuje broj `n` u bazi `k`, za $1 \leq k \leq 16$.
6. Napisati funkciju koja računa `n`-ti član aritmetičkog niza $a_1, a_2, ..., a_n$, sa zadanim prvim članom a_1 i udaljenošću među članovima `d`. Formula je $a_n = a_1 + (n-1)d$. Zadatak riješiti iterativno i rekurzivno.
7. Napisati rekurzivnu funkciju koja ispituje da li je prirodni broj palindrom.
8. Napisati funkciju koja rekurzivno računa sumu prvih `n` brojeva djeljivih sa 3.
9. Napisati funkciju koja rekurzivno računa umnožak prvih `n` parnih brojeva.
10. Napisati funkciju koja rekurzivno računa zbroj parnih brojeva manjih od `n`.
11. Napisati funkciju koja rekurzivno računa `k`-tu potenciju broja `n`.
12. Napisati rekurzivnu funkciju za računanje `n`-tog člana geometrijskog niza za dani a_1 i q. `n`-ti član geometrijskog niza je $a_n = a_{n-1} * q$.
    Primjer: za $a_1 = 1$ i $q = 2$, geometrijski niz je oblika $1, 2, 4, 8, 16, ...$
13. Napisati funkciju koja rekurzivno računa sumu pozitivnih cijelih brojeva

između `a` i `b`.

14. Napisati funkciju koja rekurzivno računa sumu recipročnih vrijednosti prirodnih brojeva do n.
15. Napisati rekurzivnu funkciju za ispisivanje prirodnog broja u bazi k, $2 \leq$ `k` ≤ 10.
16. Napisati rekurzivnu funkciju koja računa sumu prvih n prostih brojeva.
17. Napisati rekurzivnu funkciju koja ispisuje unesene znakove u obratnom redoslijedu odvojene zarezom. Unos traje dok korisnike ne unese nula.
18. Napisati rekurzivnu funkciju koja za dani broj n ispisuje sve neparne brojeve unazad do 1.
19. Napisati rekurzivnu funkciju koja za dani broj n ispisuje sve neparne brojeve do n.
20. Napisati rekurzivnu funkciju koja za dani broj n ispisuje sve prirodne brojeve unazad do 1, pa onda opet do n.
21. Napisati rekurzivnu funkciju koja za dane brojeve m i n ispisuje sve prirodne brojeve između m i n.
22. Napisati rekurzivnu funkciju koja broji u koliko se koraka od broja `n` množeći znamenke dolazi do jednoznamenkastog broja.
    Primjer: za `n=727` umnožak znamenki je `98`
    za `n=98` umnožak znamenki je `72`
    za `n=72` umnožak znamenki je `14`
    za `n=14` umnožak znamenki je `4`
    rezultat je 5 koraka.
23. Napisati rekurzivnu funkciju koja ispituje da li uneseni prirodni broj ima u binarnom zapisu jednaki broj nula i jedinica. Primjer: broj 9 (1001) ima jednak broj nula i jedinica.
24. **Repunit** je broj čije su sve znamenke 1. Napisati funkciju koja za prirodni broj ispituje da li je on repunit.
25. Napisati funkciju koja rekurzivno traži najveći među n unesenih brojeva (korisnik definira n).
26. Napisati funkciju koja vraća sumu reda $4(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7}...)$. Zadatak riješiti

iterativno i rekurzivno.

27. Napisati program u kojem korisnik unosi brojeve sve dok ne unese broj 1000. Program svaki uneseni broj ispiše naopako.
    Primjer: za uneseni broj 2345, ispisat će se 5432



Naredbe

Naredbe su konstruktivni elementi programa kojima se izvršavaju određene akcije i koje određuju način izvršavanja programa. U programskom jeziku C naredbe završavaju znakom ';' ili krajem vitičaste zagrade koja im pripada.

3.1 Prazna naredba

Prazna naredba se koristi kada sintaksa jezika zahtijeva naredbu, ali naredba ne treba izvršavati ništa.

Primjer 3.1 Program koji uneseni negativni broj pretvara u broj 1, a uneseni pozitivni broj uvećava za 1.

```
#include <stdio.h>
int main(void)
{
    int a;
    scanf("%d", &a);
    while (a++ < 0)
        ;
```

```
    printf("%d\n", a);
    return 0;
}
```

Posljedica postojanja prazne naredbe je da kompajler neće prijaviti grešku ako se slučajno napiše dva znaka ';' na kraju linije.

3.2 Naredba izraz

IZRAZ

Izraz je bilo koja kombinacija operanada i operatora koja za rezultat daje broj. Npr. `a = b + c` ili `a < b` su izrazi. Ako se izrazu na kraj stavi znak ';', izraz postaje naredba-izraz. Npr. `a = b + c;` je naredba-izraz.

3.3 Blok naredba

Blok naredba je sekvencijalni niz naredbi unutar vitičastih zagrada. U sljedećem primjeru linije 6-9 predstavljaju blok naredbu.

Primjer 3.2 Primjer blok naredbe.

```
1  int main(void)
2  {
3      int a, b;
4      scanf("%d %d", &a, &b);
5      if (a < b)
6      {
7          a = b - a;
8          printf("novi a je %d\n", a);
9      }
10     return 0;
11 }
```

3.4 Naredba `if`

Opći oblik naredbe `if` je

```
if (expression)
    statement1
```

ili

```
if (expression)
    statement1
else
    statement2
```

Pritom se izračunava izraz u zagradi i ako je izraz **različit od nule** izvršit će se `statement1`.

3.5 Naredba `for`

Naredba `for` sastoji se od tri izraza: prvi dio se izvršava na početku, samo jednom. Drugi dio naredbe se izračunava i ako je on različit od nule ulazi se u tijelo naredbe `for`. Nakon toga izvršava se treći izraz. Dalje se ponavlja ciklus: izračunavanje drugog izraza, ulaz u tijelo naredbe (ako je drugi izraz različit od nule), te izvršavanje trećeg izraza.

```
for (expression1; expression2; expression3)
    statement
```

Unutar okruglih zagrada naredbe `for` obavezno se nalaze dva znaka `';'`. Izrazi se mogu izostaviti, ali se znakovi `';'` ne smiju izostaviti i treba ih biti točno dva.

Primjer 3.3 Proučite i izvršite sljedeći kôd.

```
#include <stdio.h>
int main(void)
{
    int a, b;
    for (a = 0, b = 10; a < b; a = a + 2, b = b/2)
        printf("a = %d, b = %d\n", a, b);
    printf("izlazak iz naredbe for\n");
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

3.6 Naredba `while`

Naredba `while` ima oblik

```
while(expression)
    statement
```

Ako je vrijednost izraza različit od nule, izvršava se naredba. Zatim se ponovno ispituje vrijednost izraza.

Primjer 3.4 Suma znamenki cijelog broja.

```
unsigned sum_digits(unsigned n)
{
    unsigned s = 0;
    while(n)
    {
        s = s + n%10;
        n = n/10;
    }
    return s;
```

```
}
```

3.7 Naredba do .. while

Naredba do .. while ima oblik

```
do
   naredba
while (izraz);
```

Redoslijed izvođenja je takav da se prvo izvršava naredba, a onda ispituje je li izraz različit od nule. Ako je različit od nule iteracija se nastavlja, a ako je nula, izvođenje naredbe završava.

Primjer 3.5 Suma znamenki prirodnog broja.

```
unsigned sum_digits(unsigned n)
{
    unsigned s = 0;
    do
    {
        s = s + n%10;
        n = n/10;
    }while(n);
    return s;
}
```

Iako u predhodnim primjerima izgleda da je svejedno hoće li će se koristiti naredba while ili do..while, to baš nije tako. U naredbi while prvo se testira vrijednost izraza, a u naredbi do..while prvo se izvršava naredba, a tek potom se ispituje je li izraz različit od nule.

Primjer 3.6 Usporedite sljedeća dva primjera.

```
void test_while(void)
{
    unsigned n = 1;
    while(n < 1)
    {
        printf("%u\n", n);
        n = n + 1;
    }
}
```

```
void test_do_while(void)
{
    unsigned n = 1;
    do
    {
        printf("%u\n", n);
        n = n + 1;
    }while(n < 1);
}
```

3.8 Naredba `break`

Naredba `break` koristi se za izlazak iz `for`, `while`, `do while` ili `switch` naredbe.

Primjer 3.7 Napisati funkciju koja računa sumu unesenih prirodnih brojeva. Korisnik unosi brojeve sve dok ne unese nulu.

```
unsigned sum(void)
{
    unsigned s = 0, n;
    while(1)
```

```
    {
        scanf("%u", &n);
        if (n == 0)
            break;
        s = s + n;
    }
    return s;
}
```

3.9 Naredba `continue`

Efekt naredbe `continue` je prelazak na sljedeću iteraciju naredbe `for`, `do while` ili `while` unutar koje se nalazi.

Primjer 3.8 Napisati funkciju koja računa sumu unesenih cijelih brojeva. Korisnik unosi brojeve sve dok ne unese nulu. Negativni brojevi ne ulaze u sumu.

```
1  int suma(void)
2  {
3      int s = 0, n;
4      while(1)
5      {
6          scanf("%d", &n);
7          if (n == 0)
8              break;
9          if (n < 0)
10             continue;
11         s = s + n;
12     }
13     return s;
14 }
```

U gornjem primjeru vidimo da, ako korisnik unese negativan broj, nakon linije 10, izvršit će se linija 4.

3.10 Naredba `switch`

Naredbom `switch` možemo modelirati situaciju u kojoj tijek izvršavanja programa ovisi o izrazu koji može poprimiti različite cjelobrojne vrijednosti. Za razliku od naredbe `if` u kojoj je tok izvršavanja ovisio o tome da li je izraz jednak nula ili je različit od nule, naredbom `switch` omogućava se višestruko grananje. Opći oblik naredbe `switch` je

```
switch(izraz)
{
    case const_1: naredba_1
    case const_2: naredba_2
    ...
    case const_n: naredba_n
    default: naredba_def
}
```

Prilikom izvršavanja `switch` naredbe izračunava se izraz. Vrijednost izraza mora biti cjelobrojan (`long`, `int` ili `char`). Ako postoji `case const_k` takva da je `const_k` jednaka vrijednosti izraza, izvršavanje se nastavlja prelaskom na prvu naredbu nakon `case const_k`. Nakon toga izvršavaju se se sve naredbe nakon `naredba_k` osim ako tijek programa nije promijenjen eksplicitno naredbom `break`. Ako takva konstanta nije navedena u `case` dijelovima, izvršavanje se nastavlja prelaskom na prvu naredbu nakon `default` ako postoji ili nakon kraja naredbe `switch` ako `default case` ne postoji.

Primjer 3.9 Tijek izvršavanja koristeći naredbu `break`.

```
int main()
{
    int m = 1;
    switch (m)
    {
        case 0: printf("nula\n"); break;
```

```
            case 1: printf("jedan\n"); break;
            case 2: printf("dva\n"); break;
            default: printf("default\n"); break;
        }
        return 0;
    }
```

```
jedan
```

Ako izostavimo break tijek izvršavanja se ne mijenja. To se zove propadajući case (engl. *falling throught*).

Primjer 3.10 Propadajući case.

```
int main()
{
    char ch = 'E';
    switch(tolower(ch))
    {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            printf("vowel\n");
            break;
        default:
            printf("consonant\n");
    }
    return 0;
}
```

```
vowel
```

3.11 Riješeni zadaci

1. Napisati funkciju koja sumira prvih n trokutnih brojeva. **Trokutni brojevi** su brojevi koji su oblika $\frac{1}{2}i(i+1)$.

```
unsigned sum_triangle_nums(unsigned n)
{
    unsigned i, s = 0;     /* trokutni brojevi *
                            * su sigurno cijeli */
    for (i = 1; i <= n; ++i)
        s += i * (i + 1) / 2;
    return s;
}
int main(void)
{
    unsigned n;
    printf("unesi n\n");
    scanf("%u", &n);
    printf("suma prvih %u trokutnih je %u\n", n,
        sum_triangle_nums(n));
    return 0;
}
```

2. Napisati program koji računa umnožak kvadrata znamenki unesenog prirodnog broja.

```
unsigned square_digits(unsigned n)
{
    unsigned p = 1, zn;
    while(n)
    {
        zn = n % 10;  /* zadnja znamenka */
        p = p * zn * zn;
        n /= 10;      /* ukloni zadnju znamenku */
    }
    return p;
}
int main(void)
{
    unsigned n, p;
```

```
    printf("unesi n\n");
    scanf("%u", &n);
    p = square_digits(n);
    if (p == 0)
        printf("broj je negativan ili prevelik\n");
    else
        printf("produkt kvadrata znamenki je %u\n", p);
    return 0;
}
```

3. Napisati funkciju koja ispituje da li uneseni prirodni broj ima u binarnom zapisu jednaki broj nula i jedinica.
   Primjer: broj 9 (1001) ima jednak broj nula i jedinica.

```
int is_equal01(unsigned n)
{
    unsigned br0 = 0, br1 = 0, zn;
    while(n)
    {
        zn = n % 2; /* zadnja znamenka u binarnom zapisu
           */
        if (zn == 0)
            br0 = br0 + 1;
        else
            br1 = br1 + 1;
        n = n / 2;
    }
    if (br0 == br1)
        return 1;
    return 0;
}
int main(void)
{
    int n, p;
    printf("unesi n\n");
    scanf("%u", &n);
    p = is_equal01(n);
    if (p == 0)
        printf("nema jednak broj nula i jedinica\n");
    else
        printf("ima jednak broj nula i jedinica\n");
    return 0;
```

```
}
```

4. Napisati funkciju koja za uneseni cijeli broj n provjerava da li je **palindrom**.

```
unsigned reverse(unsigned n)
{
    unsigned k = 0, pot = 1, zn;
    while(n)
    {
        zn = n % 10;
        k = k * pot + zn;
        pot = 10;
        n = n / 10;
    }
    return k;
}
int palindrom(unsigned n)
{
    if (n == reverse(n))
        return 1;
    return 0;
}
int main(void)
{
    int n, p;
    printf("unesi n\n");
    scanf("%u", &n);
    p = palindrom(n);
    if (p == 0)
        printf("nije palindrom\n");
    else
        printf("je palindrom\n");
    return 0;
}
```

5. Napisati funkciju koja ispisuje sve brojeve do n koji imaju točno 4 prava djelitelja. **Pravi djelitelj** je djelitelj broja koji nije on sam i nije 1.

```
unsigned non_trivial_divisor(unsigned n)
{
    unsigned cnt = 0, i;
```

```
        for (i = 2; i <= n/2; ++i)
            if (n % i == 0)
                ++cnt;
        return cnt;
    }
    void print_4_divisor(unsigned n)
    {
        unsigned i;
        for (i = 1; i <= n; ++i)
            if (non_trivial_divisor(i) == 4)
                printf("%u\n", i);
    }
    int main(void)
    {
        print_4_divisor(1000);
        return 0;
    }
```

6. Napisati funkciju koja vraća sumu reda:

$$red(x,n) = x + \frac{x^2}{2} + \frac{x^3}{3} + ... + \frac{x^n}{n}$$

Funkcija ima prototip

```
double red(double x, int n);
```

```
#include <math.h>
float sum_series(float x, unsigned n)
{
    unsigned i;
    float s = 0;
    for (i = 1; i <= n; ++i)
        s += pow(x, i) / i;
    return s;
}
int main(void)
{
    float x;
    unsigned n;
    printf("unsei x i n\n");
    scanf("%f %u", &x, &n);
```

```
    printf("suma reda je %f\n", sum_series(x, n));
    return 0;
}
```

7. Napisati funkciju koja vraća predzadnju znamenku produkta prvih `n` prostih brojeva.
   Napomena: Prost broj je broj koji je djeljiv samo sa jedinicom i sa samim sobom (`1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ...` su prosti brojevi).

```
int is_prime(unsigned n)
{
    unsigned i;
    for (i = 2; i <= n/2; ++i)
        if (n % i == 0)
            return 0;
    return 1;
}
unsigned penultimate_digit(unsigned n)
{
    unsigned p = 1, i = 2, br = 1;
    while (br < n)
    {
        if (is_prime(i))
        {
            p = p * i;
            ++br;
        }
        ++i;
    }
    p = p / 10;
    return p%10;
}
int main(void)
{
    unsigned n, p;
    printf("unesi n\n");
    scanf("%u", &n);
    printf("%u\n", penultimate_digit(n));
    return 0;
}
```

8. Koračni broj (engl. *step number*) je broj u kojem je razlika dviju susjednih znamenki 1. Napisati funkciju koja provjerava da li je prirodan broj n koračni broj.

```
int is_step_number(unsigned n)
{
    unsigned zz, pz;
    while(n)
    {
        if (n < 10)
            return 1;
        zz = n % 10;
        n = n / 10;
        pz = n % 10;
        if (abs(zz - pz) != 1)
            return 0;
    }
}
int main(void)
{
    int n, p;
    printf("unesi n\n");
    scanf("%u", &n);
    p = is_step_number(n);
    if (p == 0)
        printf("nije\n");
    else
        printf("je\n");
    return 0;
}
```

3.12 Zadaci za vježbanje

1. Napisati funkciju koja sumira sve prirodne brojeve do `n` koji su djeljivi s `3`, a nisu djeljivi s `5`.
2. Napisati funkciju koja za unesene prirodne brojeve računa razliku najvećeg i najmanjeg broja. Brojeve unosi korisnik sve dok ne unese `0`.
3. Napisati program u kojem korisnik unosi znakove dok ne unese znak `'.'`. Ako je znak malo slovo, ispisuje se poruka "malo slovo" i odgovarajuće veliko slovo. Ako je znak veliko slovo, ispisuje se slovo. Ako je znak znamenka, ispisuje se njena vrijednost, a ako znak nije ni slovo, ni znamenka, ne ispisuje se ništa. Riješiti problem korištenjem `switch` naredbe.
4. Napisati funkciju koja za prirodni broj `n` vraća najmanji prost broj koji je veći od `n`.
   Primjer: Za `n = 10` funkcija vraća `11`, a za `n = 23` funkcija vraća `29`.
   Napomena: Prost broj je broj koji je djeljiv samo s jedinicom i sa samim sobom (`1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ...` su prosti brojevi).
5. Napisati funkciju koja za uneseni broj `n` provjerava je li on **trimorfan**, tj. završava li njegov kub istim znamenkama. Napisati program u kojem korisnik unosi brojeve sve dok ne unese nulu, te se za uneseni broj poziva funkcija i ispisuje rezultat.
   Primjer:
   $\mathbf{49}^3 = 117649$
   $\mathbf{125}^3 = 1953125$
6. Napisati funkciju koja vraća promijenjen peteroznamenkasti broj na način da svaku znamenku uveća za `1`. Ukoliko broj nije peteroznamenkast, funkcija vraća nepromjenjeni broj.
   Primjer: Za broj `43992` funkcija vraća `54003`.
7. Napisati program koji će za uneseni `n` ispisati mrežu `n`×`n` sljedećeg oblika (primjer za `n=4`):

```
*
*  *
*  *  *
```

```
*  *  *  *
*  *  *
*  *
*
```

8. Napisati program koji će za uneseni `n` ispisati mrežu `n`×`n` sljedećeg oblika (primjer za `n=4`):

```
      *
   *     *
*     *     *
   *     *     *
```

9. Napisati program koji će za uneseni `n` ispisati mrežu `n`×`n` sljedećeg oblika (primjer za `n=4`). Brojevi u mreži su binomni koeficijenti $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

```
      1
   1     1
1     2     1
   3     3     1
1     4     6     4     1
```

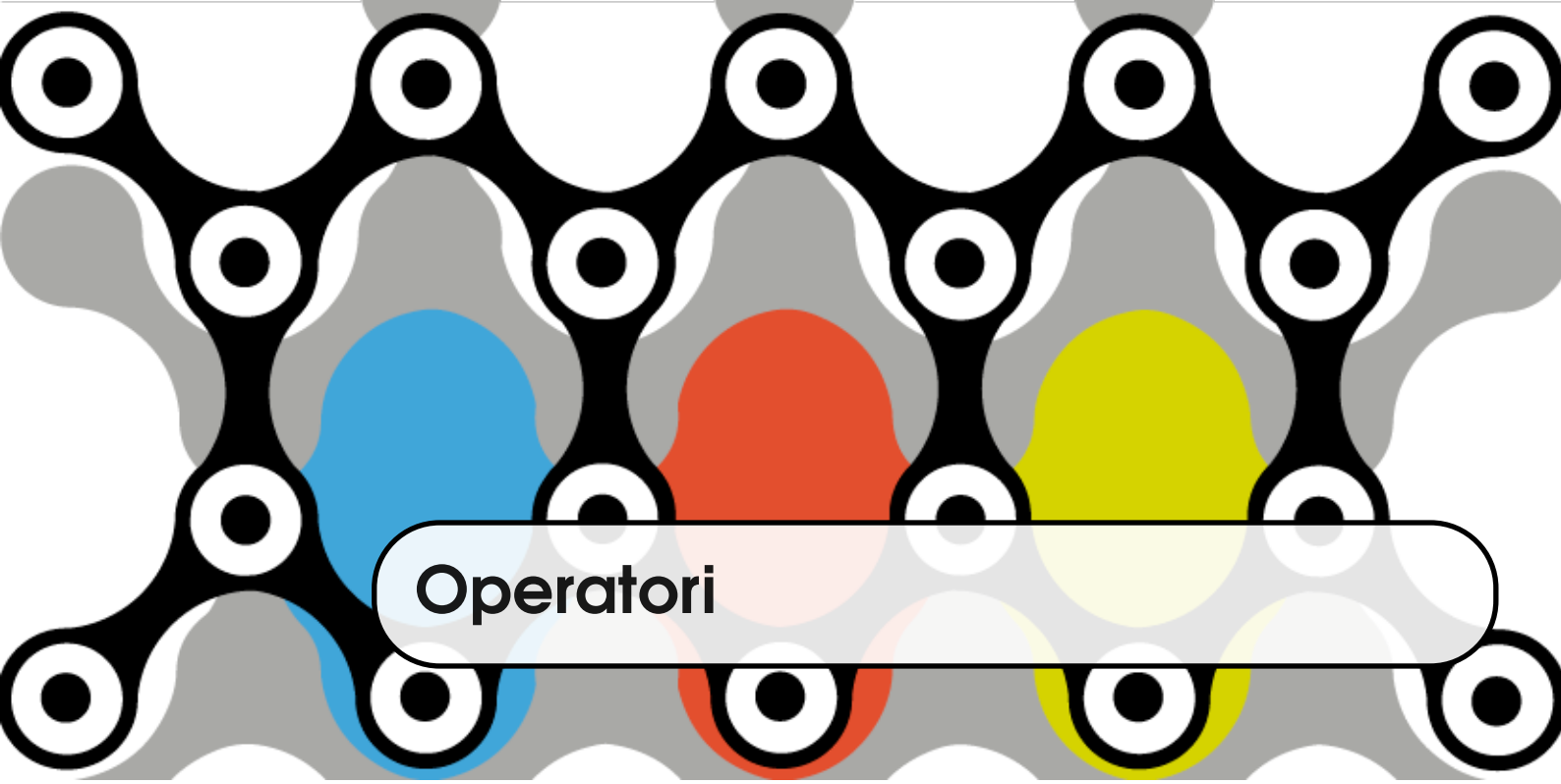
10. Napisati program koji sumira prvih n kvadrata prirodnih brojeva.
11. Napisati program koji množi prvih n kvadrata prirodnih brojeva.
12. Napisati funkciju koja za niz unesenih prirodnih brojeva vraća srednju vrijednost brojeva koji su djeljivi s 3. Brojevi se unose dok se ne unese nula.
13. Napisati funkciju koja za niz svih prirodnih troznamenkastih brojeva vraća broj koji ima najveću sumu kvadrata znamenki.
14. Napisati program koji za niz unesenih znakova `char` računa sumu pripadajućih ASCII vrijednosti. Korisnik unosi znakove dok ne unese `'q'`.
15. Napisati program koji za unesene brojeve ispituje jesu li **obilni**. **Obilni broj** je broj koji je manji od sume svojih djelitelja (izuzevši samog sebe). Brojevi se unose dok korisnik ne unese 0.
16. Napisati program koji za unesene prirodne brojeve ispituje jesu li **oskudni**. **Oskudan broj** je broj koji je veći od sume svojih djelitelja (izuzevši samog sebe). Brojevi se unose dok korisnik ne unese 0.
17. Napisati program koji za unesene brojeve ispituje jesu li svi njihovi djelitelji neparni. Brojevi se unose dok korisnik ne unese 0.

18. Napisati funkciju koja za uneseni broj `n` provjerava je li **automorfan**, tj. završava li njegov kvadrat istim znamenkama.
    Primjer:
    $\mathbf{76}^2 = 57\mathbf{76}$
    $\mathbf{625}^2 = 390\mathbf{625}$
19. Napisati program koji za unesene prirodne brojeve ispituje imaju li u heksadecimalnom zapisu sve jedinice (**repunit**). Brojevi se unose dok korisnik ne unese 0.
20. Napisati funkciju koja za parametre `a` i `b` vraća zbroj svih cijelih brojeva u intervalu `<a, b>` koji su djeljivi sa 2, a nisu djeljivi ni sa 3 ni sa 5.
21. Napisati funkciju koja računa zbroj parnih brojeva manjih od `n`.
22. Napisati program koji za uneseni prirodni broj `n` ispisuje svaku znamenku u svom retku. Ne zna se unaprijed broj znamenki u danom broju.
    Primjer: Za broj `2342` ispisuje
    `2`
    `3`
    `4`
    `2`
23. Napisati funkciju koja ispituje jesu li svi djelitelji prirodnog broja (osim jedinice) parni brojevi. Ukoliko jesu, funkcija vraća njihovu sumu.
24. Napisati funkciju koja ispituje je li broj izbalansiran tj. ima li u binarnom zapisu jednak broj nula i jedinica.
25. Napisati funkciju koja za nazivnike dvaju razlomaka vraća zajednički nazivnik (najmanji zajednički višekratnik).
26. Napisati funkciju koja vraća sumu prirodnih brojeva do `n` čiji su djelitelji parni brojevi.
27. Napisati funkciju koja vraća sumu prirodnih brojeva do `n` koji imaju manje od `m` djelitelja. Parametri funkcije su `m` i `n`.
28. **Repunit** je broj čije su sve znamenke 1. Napisati funkciju koja za prirodni broj ispituje je li on repunit.
29. **Valoviti** brojevi su brojevi koji imaju formu *ababa*. Napisati program koji

ispisuje sve peteroznamenkaste valovite brojeve djeljive sa 5.
Primjer: `12121, 13131, 43434` su valoviti brojevi.

30. Napisati funkciju koja za prirodni broj `n` vraća sumu svih prostih djelitelje broja `n`.
31. Napisati funkciju koja vraća broj unesenih znakova koji predstavljaju slova. Korisnik unosi znak po znak sve dok ne unese `0`.
32. Napisati funkciju koja za unesene prirodne brojeve vraća produkt onih koji su veći od 10 i manji od 15. Korisnik unosi brojeve. Kraj unosa je kada unese `0`.
33. Napisati program u kojem korisnik unosi znakove dok ne unese znak `'0'`. Program ispisuje koliko je uneseno samoglasnika.
34. Napisati funkciju koja ispituje je li prirodni broj `n` prost broj. Napisati program u kojem korisnik unosi broj $N \geq 3$ i koji ispisuje sumu prvih N prostih brojeva.
35. Napisati funkciju koja računa `k`-tu potenciju broja `n`.
36. Napisati funkciju koja vraća sumu apsolutnih vrijednosti cijelih brojeva između `a` i `b`.
37. Napisati funkciju koja za cjelobrojni parametar `n` računa imaju li brojevi `n` i `n + 1` jednak broj djelitelja.
Primjer: Broj `14` ima `4` djelitelja (`1, 2, 7, 14`) i broj `15` ima `4` djelitelja (`1, 3, 5, 15`).
38. Napisati program koji ispisuje sve brojeve do 100 koji su djeljivi sa sumom svojih znamenaka.
Primjer: 12 je djeljiv sa sumom svojih znamenaka tj. sa brojem 3.
39. Napisati funkciju koja računa broj pojavljivanja znamenke `k` u oktalnom zapisu broja `n`. U slučaju da znamenka `k` nije broj između 0 i 7, funkcija vraća `-1`.
40. Napisati funkciju koja vraća broj znamenki u broju `n` koje su jednake nekoj znamenki `k` ($0 \leq k \leq 9$).
41. Napisati funkciju koja, za dani `k`, ispisuje sve brojeve do `1000` koji imaju `k` djelitelja.

42. Napisati program koji računa sumu kvadrata parnih unesenih brojeva i sumu kubova neparnih unesenih brojeva. Korisnik unosi brojeve dok ne unese `0`.
43. Napisati funkciju koja ispituje da li je prirodan broj `n` repunit u nekoj od baza od 2 do 16.
    Primjer: Broj $15_{(10)} = 1111_{(2)}$ je repunit u bazi 2, a broj $273_{(10)} = 111_{(16)}$ je repunit u bazi 16. Broj $7_{(10)} = 111_{(2)} = 11_{(6)}$ je repunit u bazi 2 i bazi 6.
44. Napisati program u kojem korisnik unosi znakove sve dok ne unese znak `'!'`. Ako su svi uneseni znakovi bili znakovi engleskog alfabeta, program ispisuje poruku "engleski alfabet". Ako su svi uneseni znakovi bili znamenke, program ispisuje poruku "znamenke". U svim ostalim slučajevima program ispisuje "kombinacija".
45. Napisati funkciju koja za parametre `a`, `b` i `k` vraća broj svih cijelih brojeva u intervalu `<a, b>` koji su djeljivi sa brojem `k`.
46. Napisati funkciju koja računa n-ti Lucasov broj. Lukasov niz je niz brojeva u kojem je svaki član zbroj prethodna dva broja, i koji počinje brojevima `1` i `3` (niz: `1, 3, 4, 7, 11, 18, ...`)
    Primjer: za `n = 5`, funkcija vraća `11`.
47. Napisati program u kojem korisnik unosi znakove sve dok ne unese znak `'!'`. Program ispisuje koliko je uneseno znakova koji nisu znamenke.
48. Napisati funkciju u kojoj korisnik unosi znakove dok ne unese znak `'!'`. Funkcija vraća znak koji ima najveću ASCII vrijednost.
49. Napisati funkciju koja ispisuje *k*-tu znamenku prirodnog broja *n*. Prototip funkcije je
    ```
    unsigned znamenka(unsigned n, unsigned k);
    ```
50. Napisati funkciju koja računa koji broj u rasponu od `0` do `n` ima najviše djelitelja. `n` je parametar funkcije.
51. Napisati funkciju u kojoj korisnik upisuje `n` prirodnih brojeva. Funkcija vraća prvi broj koji je veći od prosječne vrijednosti brojeva.
52. Napisati funkciju koja za `n` unesenih brojeva vraća broj koliko puta se pojavio najveći uneseni broj.



Operatori

Slično pojmu operacije iz matematike, **operatori** označavaju operaciju nad jednim ili više operanada. U trenutku izvođenja operacije, operandi moraju imati neku brojčanu vrijednost. Operandi mogu biti varijable, povratne vrijednosti funkcije (ako funkcija nije `void` povratnog tipa), literali te kombinacija operanada i operatora.

OPERATORI

Kombinacija operatora i operanada čini **izraz**.

IZRAZ

Primjer 4.1 `a == b * (c + 3)`

Kroz gornji primjer možemo uočiti nekoliko stvari vezanih uz izraze i operatore. Poredak izračunavanja izraza je određen sa tri stvari:

- zagrada
- prioritet operatora
- smjer asocijativnosti.

Kao i u matematičkim izrazima, **zagrade** mogu promijeniti poredak izvođenja operacija. **Prioritet operatora** određuje u kojem će se redoslijedu izvršavati

PRIORITET OPERATORA

operatori. U izrazu `b * c + 3`, operator množenja ima prioritet pred operatorom zbrajanja, te se prvo množe `b` i `c`, a zatim se umnošku doda `3`. U gornjem primjeru `b * (c + 3)` zagrada je promijenila redoslijed izvođenja.

SMJER ASOCIJATIVNOSTI

Smjer asocijativnosti određuje da li će, u slučaju da u izrazu sudjeluju operatori jednakog prioriteta, redoslijed izvođenja biti s lijeva na desno ili obrnuto.

Primjer 4.2 Vrijednost izraza `24 / 3 * 8` je `64`.

Smjer asocijativnosti multiplikativnih operatora (*, /, %) je s lijeva na desno, pa se prvo obavlja dijeljenje broja `24` sa `3`, a zatim se rezultat te operacije pomnoži sa `8`. Rezultat izraza je `64`.

Poredak operatora, tj. grupe operatora prema prioritetu dan je u tablici 4.1.

Tablica 4.1: Prioritet operatora

kategorija	operatori	smjer asocijativnosti
postfiksni	`[] () . -> ++ --`	s lijeva na desno
prefiksni unarni	`--++ sizeof & * + - ~`	s desna na lijevo
cast operatori	`(type)`	s desna na lijevo
multiplikativni	binarni `* / %`	s lijeva na desno
aditivni binarni	`+ -`	s lijeva na desno
bitovni pomak	`>> <<`	s lijeva na desno
relacijski	`< > <= >=`	s lijeva na desno
operatori jednakosti	`== !=`	s lijeva na desno
bitovni "i"	`&`	s lijeva na desno
bitovni "ekskluzivno ili"	`^`	s lijeva na desno
bitovni "inkluzivno ili"	`|`	s lijeva na desno
logički "i"	`&&`	s lijeva na desno
logički "ili"	`||`	s lijeva na desno
uvjetni operator	`?:`	s desna na lijevo
operatori dodjeljivanja	`= += -= *= /= %= >>= >>= &= |= ^=`	s desna na lijevo
operator zarez	`,`	s lijeva na desno

4.1 Vrste operatora

UNARNI OPERATORI
BINARNI OPERATORI
TERNARNI OPERATORI

Operatori u C-u se dijele na **unarne**, **binarne** i **ternarne**. **Unarni operatori** su operatori koji djeluju na jedan operand, **binarni** na dva i **ternarni** na tri operanda. U C-u je jedini primjer ternarnog operatora uvjetni operator `?:`.

Ovisno o poziciji u izrazu neki operatori mogu biti unarni ili binarni.

- U izrazu `-n` operator - je unarni operator.
- U izrazu `m - n`, operator - je binarni operator.

4.2 Izrazi i tipovi

Ako u izrazu sudjeluju varijable istog tipa, tada će rezultat biti tog istog tipa.

Ako u izrazu sudjeluju varijable različitog tipa, onda će rezultat biti najvećeg tipa koji sudjeluje u izrazu. Pod većim tipom podrazumijeva se tip u kojem se može zapisati broj veće duljine. Općenito pravilo bi bilo `short -> int -> unsigned -> long -> unsigned long -> float -> double -> long double`.

4.3 Opis i svojstva operatora

Iako su u tablici 4.1 navedeni operatori prema prioritetu, u nastavku ćemo ih grupirati prema značenju.

4.3.1 Aritmetički operatori

Aritmetički operatori nabrojani su u tablici 4.2.

Tablica 4.2: Aritmetički operatori

operator	opis
unarni +	operand je aritmetičkog tipa, rezultat je vrijednost operanda
unarni -	operand je aritmetičkog tipa, rezultat je negativna vrijednost operanda
binarni +	rezultat je suma operanada
binarni -	rezultat je razlika operanada
binarni *	operandi su aritmetičkog tipa, rezultat je produkt operanada
/	operandi su aritmetičkog tipa, rezultat je kvocijent operanada
%	operandi su cjelobrojnog tipa, rezultat je ostatak pri dijeljenju
prefiksni ++	operand je *lvalue*, rezultat je vrijednost operanda uvećana za 1
prefiksni --	operand je *lvalue*, rezultat je vrijednost operanda umanjena za 1
postfiksni ++	operand je *lvalue*, rezultat je vrijednost operanda (prije uvećanja), nakon izraza vrijednost operanda je uvećana za 1
postfiksni --	operand je *lvalue*, rezultat je vrijednost operanda, nakon izraza vrijednost operanda je umanjena za 1

Unarni plus i unarni minus Unarni plus operator djeluje na aritmetičke operande tako da ne mijenja vrijednost operanda, ali obavlja promociju na `int` tip ako je operand manjeg cjelobrojnog tipa.

Primjer 4.3 Promocija na `int` operatorom `+`.

```
int main(void)
{
    short ch = 40;
    printf("velicina %lu bajta\n", sizeof(ch));
    printf("promocija %d na %lu bajta\n", +ch, sizeof(+ch)
        );
    printf("promocija %d na %lu bajta\n", -ch, sizeof(-ch)
        );
    return 0;
```

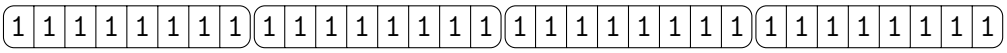
```
}
```

Unarni minus mijenja vrijednost operanda u negativnu. Ako operator unarni minus djeluje na `unsigned` izraz, vrijednost rezultata bit će maksimalna vrijednost za taj tip umanjena za vrijednost operanda, te uvećana za 1. Pogledajmo zašto.

Primjer 4.4 Djelovanje unarnog operatora -

```
int main(void)
{
    unsigned short n = 1;
    unsigned short m = -n;
    short ms = -n;
    unsigned mu = -n;
    printf("%hu\n", m);
    printf("%hd\n", ms);
    printf("%u\n", mu);
    return 0;
}
```

Bez obzira na to u koju varijablu (tj. koji tip) će se spremiti, vrijednost `-n` je u memoriji zapisan kao broj -1.



Ako se takav broj sprema u varijablu tipa `unsigned short`, dva viša bajta se odbacuju, a rezultat interpretira kao `unsigned short`, tj. broj `65535`.

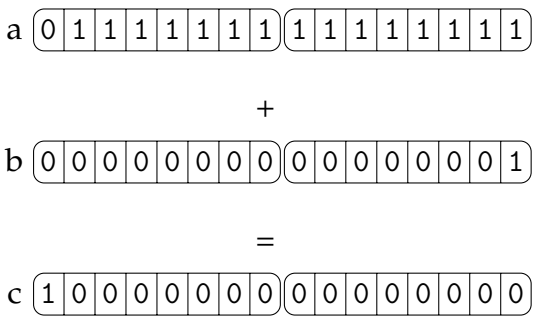
Ako se takav broj sprema u varijablu tipa `short`, dva viša bajta se odbacuju, a rezultat interpretira kao `short`, tj. broj `-1`.

ADITIVNI OPERATORI

Aditivni operatori Binarno zbrajanje i binarno oduzimanje izvodi se slično analognim matematičkim operacijama. Ipak treba voditi računa nad kojim tipovima se izvodi operator. Npr. promatrajmo izraz `a + b`. Ako su vrijednosti varijabli `a` i `b` takve da njihov zbroj izlazi iz opsega njihovog tipa, rezultat izraza neće odgovarati rezultatu koje bi dalo klasično matematičko zbrajanje dva broja.

```
short a = 32767, b = 1;
short c = a + b;
```

Uz pretpostavku da je veličina `short` tipa 2 byte, vrijednost izraza `a + b` bit će `-32768`. Razlog tome je što je najveći broj koji se može zapisati kao `short` `32767` (tj. $2^{15} - 1$). To je broj koji u binarnom zapisu ima sve jedinice (osim vodećeg bita predznaka). Kad takvom broju (`a`) dodamo jedinicu (`b`), dobit ćemo zapis u kojem su svi bitovi nula (`c`), tj.



a to je broj `-32768`.

MULTIPLIKATIVNI OPERATORI

Multiplikativni operatori Kod upotrebe multiplikativnih operatora treba obratiti pozornost na nekoliko detalja. Prisjetimo se implicitne konverzije tipa i pogledajmo primjer:

```
float b = 3 / 4;
```

Očekivani rezultat bio bi da je vrijednost varijable `b` 0.75. Međutim, prilikom izračunavanja desnog izraza operator `/` nalazi se između dvije cjelobrojne konstante, pa je i rezultat desnog izraza cjelobrojan, tj. `0`. Zatim se cjelobrojna vrijednost `0` pridružuje `float` varijabli `b`, pa je vrijednost varijable `b` opet `0`, ali sada konvertirana u `float` vrijednost.

Važno je uočiti da kod primjene operatora `/` i `%` drugi operand ne smije biti nula. Rezultat operacije `a / b`, ako je vrijednost varijable `b` nula, je `inf` tj. beskonačno. Isto tako, postoje vrijednosti `-inf` ($-\infty$) i `Nan` (not a number).

Primjer 4.5 Nedozvoljene operacije.

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    float a = 5, b = 0;

    printf("a / b = %f\n", a / b);
    printf("log(b) = %f\n", log(b));
    printf("log(-a) = %f\n", log(-a));
    return 0;
}
```

Prisjetimo se matematike i logaritamske funkcije. Ona nije definirana za negativne brojeve, pa je izraz `log(-5)` za rezultat daje `Nan`. Jednako tako funkcija je definirana za strogo pozitivne brojeve i ima desnu vertikalnu asimptotu u `0`. Vrijednost funkcije kad se približava nuli s desne strane je $-\infty$.

Što se tiče operatora `%`, operandi nad kojima operator djeluje moraju biti cjelobrojni. Ostatak pri djeljenju ima smisla samo u cjelobrojnoj aritmetici.

Napomena: Ako želite ispisati znak % pomoću `printf` funkcije, napravite to ovako: `printf("a %% b");`.

PREFIKSNI INKREMENT I DEKREMENT

Prefiksni i postfiksni inkrement i dekrement Operator prefiksni inkrement (dekrement) je operator koji uvećava (umanjuje) vrijednost operanda za **1**. Rezultat izraza `++a` (`--a`) je **nova** uvećana (umanjena) vrijednost od `a`. Operator prefiksni inkrement (dekrement) je operator koji uvećava (umanjuje) vrijednost operanda za **1**. Rezultat izraza `++a` (`--a`) je **nova** uvećana (umanjena) vrijednost od `a`.

Primjer 4.6 Prefiksni inkrement.

```
int main(void)
{
    int a = 3, b = 5;
    int c = ++a + b;
    printf("a = %d, b = %d, c = %d\n", a, b, c);
    return 0;
}
```

```
a = 4, b = 5, c = 9
```

Uočite da izraz `++(a + b)` nije legalan. Naime, kao što ne možemo napisati `a + b = 3`, jer nemamo čemu pridružiti vrijednost 3 (izraz `a + b` nije l-value), tako ni u slučaju `++(a + b)` ne možemo upotrebljavati operator da bi uvećao vrijednost izrazu koji nije l-value. Postfiksni `++` i postfiksni `--` su operatori koji uvećavaju tj. umanjuju vrijednost operanda za jedan. Rezultat izraza `a++` je **stara, neuvećana** vrijednost varijable `a`. Treba uočiti tu razliku između prefiksnih i postfiksnih operatora: u izrazu čiji je dio prefiksni operator sudjeluje uvećana vrijednost operanda, dok u izrazu čiji je dio postfiksni operator sudjeluje neuvećana vrijednost operanda. Pokažimo to primjerima.

Primjer 4.7 Postfiksni inkrement

```
int main(void)
{
    int a = 3;
    int b = a++;
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

```
a = 4, b = 3
```

Primjer 4.8 Prefiksni inkrement

```
int main(void)
{
    int a = 3;
    int b = ++a;
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

```
a = 4, b = 4
```

4.3.2 Logički i relacijski operatori

Logički i relacijski operatori nabrojani su u tablici 4.3.

Tablica 4.3: Logički i relacijski operatori

operator	opis
`==`	operator jednakosti, rezultat izraza je `0` ili `1`
`!=`	operator nejednakosti, rezultat izraza je `0` ili `1`
`< > <= >=`	relacijski operatori, rezultat izraza je `0` ili `1`
`&&`	logički operator "i", rezultat izraza je `0` ako je barem jedan operand `0`
`||`	logički operator "ili", rezultat izraza je `1` ako je barem jedan operand različit od nule
`!`	logička negacija, rezultat izraza je `0` ako je operand različit od nule, a `1` ako je operand `0`

Operatori jednakosti Vrijednosti izraza oblika `a == b`, `a != b` mogu biti `0` ili `1`. Vrijednost `0` odgovara logičkoj vrijednosti `false`, a vrijednost `1` odgovara logičkoj vrijednosti `true`.

Primjeri upotrebe operatora jednakosti su razni:

- U izrazima u kojima se ispituje je li vrijednost izraza različita od nule, često se umjesto `if (a != 0) ...` piše `if (a) ...`
- U izrazima u kojima se ispituje je li vrijednost izraza jednaka nuli, često se umjesto `if (a == 0) ...` piše `if (!a) ...` Uočite da naredba `if` provjerava je li u zagradi izraz koji je različit od nula. Ako je `a` jednak nuli, onda izraz `a == 0` daje vrijednost `1` (`true`), a to je različito od `0`. Analogno, izraz `!a` je u tom slučaju različit od nule, pa je uobičajeno pisati `if (!a) ...`

Primjer 4.9 Sljedeća dva primjera su jednaka. U njima se izračunava suma neparnih prirodnih brojeva do `n`.

```
unsigned i, s = 0, n = 100;
for (i = 0; i < n; i++)
```

```
    if (i % 2 != 0) s = s + i;
printf("s = %u\n", s);
```

```
unsigned i, s = 0, n = 100;
for (i = 0; i < n; i++)
    if (i % 2) s = s + i;
printf("s = %u\n", s);
```

Jednako tako sljedeći su primjeri jednaki.

Primjer 4.10 Suma parnih brojeva do `n`.

```
unsigned i, s = 0, n = 100;
for (i = 0; i < n; i++)
    if (i % 2 == 0) s = s + i;
printf("s = %u\n", s);
```

```
unsigned i, s = 0, n = 100;
for (i = 0; i < n; i++)
    if (!(i % 2)) s = s + i;
printf("s = %u\n", s);
```

- Vrijednost izraza `a != b` ili `a == b` je nula ili jedan. Takav izraz može sudjelovati u nekom složenijem izrazu. Npr.

```
unsigned a = 2, b = 3, c;
c = a == b;
```

S obzirom na to da operator == ima veći prioritet u odnosu na operator pridruživanja =, on će se prvi izvršiti. Vrijednost izraza `a == b` je `0` (`a` i `b` su različiti), pa će `c` dobiti vrijednost `0`.

Primjer 4.11 Promotrimo sljedeći primjer:

```
unsigned a = 3, b = 3, c = 3;
if (a == b == c)
    printf("jednaki su");
else
    printf("hm..razliciti su");
```

Iako su vrijednosti od `a`, `b` i `c` jednake, rezultat izraza `a == b == c` nije `1`. Rezultat izraza `a == b` je `0`, i ta nula se uspoređuje s `c`, tj. `0 == c`. Rezultat toga izraza je `0`.

Primjer 4.12 Što će se ispisati u sljedećem primjeru?

```
int main(void)
{
    unsigned a = 3, b = 4, c = 1, d;
    d = a != b == c;
        printf("%u\n", d);
}
```

RELACIJSKI OPERATORI

Relacijski operatori Rezultat izraza u kojem sudjeluju relacijski operatori `<`, `>`, `<=`, `>=` su `0`, ako relacija nije istinita (*false*) ili `1` ako je relacija istinita (*true*). Izraz `a<b<c` se parsira kao `(a<b)<c`, a `a<b` kao rezultat mogu dati `0` ili `1`. Ako želimo npr. ispitati da li je neki broj u intervalu između 10 i 20, nećemo napisati `if (10 <= a <= 20)`, kao što bi matematički zapisali, već ćemo napisati `if (10 <= a && a <= 20)`.

LOGIČKI OPERATORI

Logički operatori Logički "i" operator (`&&`) vraća `1` ako su oba operanda različita od nule. Inače, vraća `0`. Pritom se prvo izračunava lijevi operand i ako je lijevi operand kao rezultat dao nulu, desni operand se **ne izračunava**.

Primjer 4.13 U izrazu u kojem sudjeluje logički "i" operator desni operand se ne izračunava ako je lijevi jednak nula.

```
int main(void)
{
    int a = 5, b = 3, c;
    if (a == b && (c = b))
        printf("%d", c);
}
```

Izraz `c = b` neće se izvršiti, jer je rezultat lijevog operanda nula.

Logički "ili" operator (`||`) vraća `1` ako je barem jedan operand različit od nule. Inače, vraća `0`. Pritom se prvo izračunava lijevi operand i ako je lijevi operand kao rezultat dao jedinicu, desni operand se **ne izračunava**.

Primjer 4.14 U izrazu i kojem sudjeluje logički "i" operator desni operand se ne izračunava ako je lijevi jednak nula.

```
int main(void)
{
    int a = 5, b = 5, c;
    if (a == b || (c = b))
        printf("%d", c);
}
```

Izraz `c = b` neće se izvršiti, jer je rezultat lijevog operanda jedan.

Operator **logičke negacije** `!` je operator koji nenula vrijednost operanda pretvori u nula, a nulu pretvori u jedinicu. Sam operator ne mijenja operand.

Primjer 4.15 Operator logičke negacije.

```
int main(void)
{
    int a = 5, b = 0, c = -5;
    int d = !a, e = !b, f = !c;
    float fa = 5, fb = 0;
    printf("a = %d, !a = %d, !!a = %d\n", a, d, !!a);
    printf("b = %d, !b = %d, !!b = %d\n", b, e, !!b);
    printf("c = %d, !c = %d, !!c = %d\n", c, f, !!c);
    printf("fa = %f, !fa = %d, fb = %f, !fb = %d\n", fa, !
       fa, fb, !fb);
    return 0;
}
```

U primjeru 4.15 smo pokazali nekoliko detalja. Djelovanje unarnog operatora na operand neće promijeniti samu vrijednost operanda. Možemo primijetiti da je vrijednost varijable `a` nakon linije `d = !a` i dalje `5`.

Nadalje, u izrazu `!!a` zbog smjera asocijativnosti s desna na lijevo, prvo se izvrši desni operator, koji vrijednost varijabe `a` pretvori u `0`, a zatim se izvrši lijevi operator koji tu nulu pretvori u jedinicu. Dakle, `!!` će nenula vrijednost pretvoriti u jedinicu, a vrijednost nula pretvoriti opet u nulu.

Treća stvar koju treba uočiti je da je vrijednost koja se dobije djelovanjem operatora `!` cjelobrojna, bez obzira što je operand u primjeru bio `float`.

4.3.3 Bitovni operatori

BITOVNI OPERATORI

Bitovni operatori nabrojani su u tablici 4.4.

Tablica 4.4: Bitovni operatori

operator	opis
`&`	bitovno "i", uspoređuje bitove prvog operanda sa odgovarajućim bitovima drugog operanda; ako su oba bita `1`, bit rezultata je `1`, inače je `0`
`|`	bitovno "ili", uspoređuje bitove prvog operanda sa odgovarajućim bitovima drugog operanda; ako su oba bita `0`, bit rezultata je `0`, inače je `1`
`^`	bitovno ekskluzivno "ili, uspoređuje bitove prvog operanda sa odgovarajućim bitovima drugog operanda; ako su oba bita različiti, bit rezultata je `1`, inače je `0`
`~`	bitovna negacija, rezultat je binarni komplement broja
`<<`	pomak ulijevo
`>>`	pomak udesno

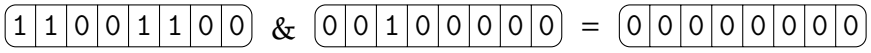
Bitovni "i", "inkluzivno ili" i "ekskluzivno ili" su binarni operatori tj. zahtijevaju dva operanda. Sama operacija realizira se tako da se uspoređuju bitovi (bit po bit) oba operanda.

Bitovno "i" Bitovni operator `&` primjenjuje operaciju `'i'` bit po bit na lijevi i desni operand.

Primjer 4.16 1 1 0 0 1 1 0 0 & 1 1 1 1 0 0 0 0 = 1 1 0 0 0 0 0 0

Najčešća upotreba bitovnog "i" operatora je ispitivanje vrijednosti pojedinog bita. Npr. zanima nas je li peti bit (gledano s desna na lijevo i počevši od nultog) u binarnom zapisu broja setiran ili ne. Sjetimo se da operacija `&` bita sa jedinicom odražava stanje bita, tj. `0 & 1 == 0`, `1 & 1 == 1`. Dakle, da bi ispitali peti bit zanima nas rezultat izraza u kojem je jedan operand broj koji ispitujemo, a drugi

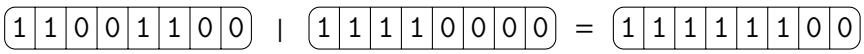
`00100000` tj. `1<<5`.



Ako je rezultat izraza `0`, peti bit je bio nula, inače je bio `1`.

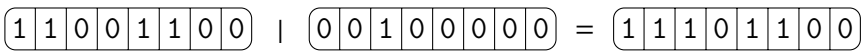
Bitovno inkluzivno "ili" Bitovni operator `|` primjenjuje operaciju `'ili'` bit po bit na lijevi i desni operand.

Primjer 4.17



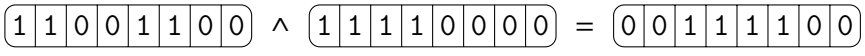
Najčešća upotreba bitovnog "inkluzivno ili" operatora je postavljanje (setiranje) pojedinog bita u nekoj varijabli. Npr. želimo setirati peti bit u binarnom zapisu broja. Sjetimo se da je operacija "inkluzivno ili" takva da ako je jedan od bitova koje uspoređujemo `1` tada će i rezultat biti `1`, tj. `0 | 1 == 1`, `1 | 1 == 1`. Dakle, da bismo ispitali peti bit zanima nas rezultat izraza u kojem je jedan operand broj koji ispitujemo, a drugi `00100000` tj. `1<<5`.

Primjer 4.18 Ispitivanje petog bita.



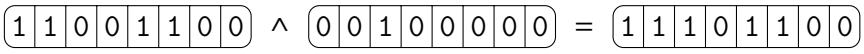
Dakle, ako je `n` bio $204_{10} = 11001100_2$, onda će rezultat izraza `n = n | 1<<5` biti $236_{10} = 11101100_2$.

Bitovno ekskluzivno "ili" Bitovni operator `^` primjenjuje operaciju `'ekskluzivno ili'` bit po bit na lijevi i desni operand prema kojoj je rezultat jedan ako su bitovi različiti, a nula ako su isti.



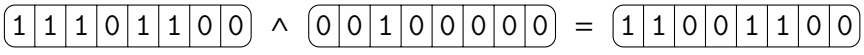
Rezultat operacije `^` između dva različita bita je `1`, inače je `0`.

Najčešća upotreba bitovnog "ekskluzivno ili" operatora je mijenjanje (engl. *toggle*) pojedinog bita u nekom broju. Npr. želimo promijeniti peti bit u binarnom zapisu broja. Sjetimo se da je operacija "ekskluzivno ili" takva da ako je jedan od bitova koje uspoređujemo `1` tada će i rezultat biti `1`, tj. `0 \^{} 1 == 1`, `1 ^ 1 == 0`. Dakle, da bismo promijenili peti bit zanima nas rezultat izraza u kojem je jedan operand broj koji ispitujemo, a drugi `00100000` tj. `1<<5`.

Primjer 4.19 

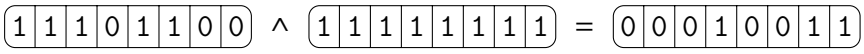
Peti bit u binarnom zapisu broja `n` je `0`, nakon djelovanja operatora `^` s jedinicom na petom mjestu, promijenio se u `1`.

Možemo pogledati što bi se dogodilo da je peti bit bio `1`.

Primjer 4.20 

Dakle, peti bit bi se promijenio iz `1` u `0`.

Poznavajući ovo svojstvo ekskluzivnog ili operatora, možemo lako okrenuti pojedine bitove (a i sve) u binarnom zapisu nekog broja tako da na napravimo operaciju ekskluzivno ili s brojem predstavljenim svim jedinicama (u primjeru `UCHAR_MAX`).



Primjer 4.21 Funkcija koja za uneseni broj sve njegove bitove preokreće, tj. nule pretvara u jedinice i obratno.

```
#include <limits.h>
unsigned toggle(unsigned n)
```

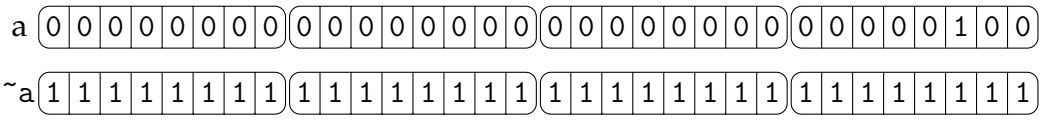
```
{
    return n^UINT_MAX;
}
```

U ovom primjeru `UINT_MAX` je maksimalna vrijednost koju `unsigned` varijabla može postići. Odgovara binarnoj reprezentaciji broja kojemu su svi bitovi jedinice.

OPERATOR BITOVNE NEGACIJE

Operator bitovne negacije Operator bitovne negacije ~ je operator koji sve jedinice u bitovnom zapisu broja pretvori u nule i obratno.

Primjer 4.22 Neka je a tj.



Operator bitovne negacije vrši promociju u `int`.

Primjer 4.23 Promocija u `int`.

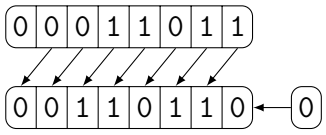
```
int main()
{
    unsigned char n = 1;
    printf("%u %hhu\n", sizeof(~n), ~n);
    return 0;
}
```

```
4 254
```

OPERATORI BITOVNOG POMAKA

Operatori bitovnog pomaka Operatori bitovnog pomaka << i >> imaju za pos-

ljedicu pomak bitova u izrazu za određeni broj mjesta lijevo ili desno (slika 4.1). Naprimjer `x << 3` je izraz čija vrijednost odgovara vrijednosti binarnog zapisa broja `x` za 3 mjesta ulijevo.



(a) Pomak ulijevo **(b)** Pomak udesno

Slika 4.1: Operatori bitovnog pomaka

Primjer 4.24 Pomak za tri mjesta ulijevo.

```
unsigned x = 15, y;
y = x << 3;
```

Vrijednost varijable `y` bit će `120`. Naime, promotrimo bitovni zapis broja `15`.

$15_{10} = 1111_2$ Ako tu binarnu reprezentaciju pomaknemo ulijevo za tri mjesta dobit ćemo broj `1111000`, a to je u stvari broj `120`.

Dakle, pri pomaku broja ulijevo za `k` mjesta, s desne strane bitovne reprezentacije dodaje se `k` nula. S obzirom na to da je broj zapisan u određenom broju bitova (npr. za `unsigned` u 32 bita), `k` bitova s lijeve strane otpada.

Primjer 4.25 Kod bitovnog pomaka moramo voditi računa o veličini tipa.

```
unsigned char x = 214, y;
y = x << 3;
```

Vrijednost varijable y bit će 176. Naime, bitovni zapis broja 214 je 11010110. Kad broj pomaknemo za tri mjesta ulijevo s desne strane će se dopisati tri nule, a prva tri bita (s lijeve strane) će "otpasti". Novi broj je 10110000, odnosno 176.

U ograničenom smislu pomak ulijevo možemo promatrati kao množenje broja s potencijom broja 2. Slično kao što u dekadskom zapisu dodavanje nule na kraj odgovara množenju broja sa 10, tako u binarnom sustavu množenje s 2 odgovara dodavanju nule na kraj.

Primjer 4.26 Operator bitovnog pomaka << kao kraće množenje sa potencijom broja 2.

```
unsigned x = 7, y;
y = x << 1;
```

Vrijednost varijable y je 14.

```
unsigned x = 7, y;
y = x << 2;
```

Vrijednost varijable y je 28.

Općenito možemo reći da je vrijednost `n << k` jednaka umnošku $n \cdot 2^k$.

Ipak, to ne vrijedi u svakom slučaju. Vratimo se na primjer 4.25.

```
unsigned char x = 214, y;
y = x << 3;
```

Vrijednost varijable y je, kao što smo rekli, 176. Očito je da gornja relacija `n << k` $\equiv n \cdot 2^k$ ne vrijedi, odnosno 176 nije jednako `214 * 8` zbog vodećih bitova koji su "ispali". Sjetimo se da je zapis broja $214_{(10)} = 11010110_{(2)}$. Pri pomaku ulijevo za tri mjesta, a s obzirom da `unsigned char` zauzima 8 bitova, vodeća tri bita 110 su "ispala". Upravo činjenica da ti bitovi nisu bili nula, čini razliku u odnosu na slučajeve za koje relacija vrijedi.

Potrebno je zapaziti još jednu primjenu operatora <<. Kada zatrebamo k-tu potenciju broja 2, možemo pozvati funkciju `pow(2, k)` (i uključiti pripadajuću `math.h` datoteku). Međutim, brže je i jednostavnije napisati `1<<k`. Značenje izraza `1<<k` je $1 \cdot 2^k$, a to je ono što smo tražili. Razmislimo malo o tome. Binarni zapis nekog broja n je u stvari niz potencija od 2.

$$24 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$$

tj.

$$24_{10} = 11000_2$$

Ako je sam broj n potencija od 2, to znači da je jedinica samo uz odgovarajuću potenciju broja dva, a sve ostale potencije su nula.

$$16 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$$

tj.

$$16_{10} = 10000_2$$

Dakle, broj 10000_2 možemo napisati kao `1<<4`.

S druge strane, bitovni pomak na desno, realizira se tako da se s lijeve strane dodaju nule, a bitovi s desne strane "ispadaju".

Primjer 4.27 `214 >> 3 = 26` tj. `11010110 >> 3 = 00011010`.

Prilikom pomaka udesno za tipove sa predznakom standard ne definira što se mora dogoditi s vodećim bitom. Ipak, većina kompajlera implementira desni

shift operator tako da zadrži vodeći bit za predznak (shiftaju se svi ostali bitovi). Pomak kod kojeg se pomiče i vodeći bit naziva se **zero-fill right shift**.

ZERO-FILL RIGHT SHIFT

Primjer 4.28 Napisati program koji uneseni `unsigned` broj rastavlja na četiri `unsigned char` broja.

```
#include <stdio.h>
unsigned char izvuci_bajt(unsigned i, unsigned n)
{
    unsigned char m = (1 << 8) - 1; //pow(2, 8)-1
    n >>= 8*i;
    return (unsigned char)n & m;
}

int main()
{
    unsigned n, i;
    unsigned char a;
    scanf("%u", &n);
    for (i = 0; i < sizeof(unsigned); i++)
    {
        a = izvuci_bajt(i, n);
        printf("%hhu\n", a);
    }
    return 0;
}
```

4.3.4 Operatori dodjeljivanja

OPERATOR DODJELJIVANJA

Operator jednostavnog dodjeljivanja je operator =. Prisjetimo se da operatori dodjeljivanja imaju niski prioritet (samo operator zarez ima niži prioritet). Zbog toga lako možemo pridružiti varijabli vrijednost nekog složenog izraza npr. `a = b + c << 2`. Sigurni smo da će se operand s desne strane operatora = prvi izračunati, a onda će se njegova vrijednost pridružiti varijabli `a`. Ostali operatori `izraz_1 op= izraz_2` dodjeljivanja spadaju u **složene** operatore dodjeljivanja.

Oni su, u stvari samo pokrata za `izraz_1 = izraz_1 op izraz_2`.

Tablica 4.5: Operatori dodjeljivanja

operator	opis
`=`	`a = b`, vrijednost varijable `b` pridružuje varijabli `a`
`+=`	`a += b`, pokrata za `a = a + b`
`-=`	`a -= b`, pokrata za `a = a - b`
`*=`	`a *= b`, pokrata za `a = a * b`
`/=`	`a /= b`, pokrata za `a = a / b`
`%=`	`a %= b`, pokrata za `a = a % b`
`<<=`	`a <<= b`, pokrata za `a = a << b`
`>>=`	`a >>= b`, pokrata za `a = a >> b`
`&=`	`a &= b`, pokrata za `a = a & b`
`^=`	`a ^= b`, pokrata za `a = a ^ b`
`|=`	`a |= b`, pokrata za `a = a | b`

4.3.5 Operatori eksplicitne konverzije (type cast)

Sintaksa: `(ime_tipa)izraz`

- asocijativnost s desna na lijevo
- izračunava se `izraz`
- vrijednost izraza pretvara se iz svojeg prirodnog tipa u tip određen sa `ime_tipa`.

Postoje dvije vrste konverzije: implicitna i eksplicitna. Implicitnu smo već spominjali kada smo govorili o složenim izrazima u kojima sudjeluju operandi različitog tipa. Eksplicitna konverzija realizira se navođenjem imena tipa u zagradama ispred operanda.

Prilikom eksplicitne konverzije iz `float` ili `double` u cjelobrojni tip, zaokruživanje se vrši tako da se "odrežu" decimale.

Primjer 4.29 Konverzija u cjelobrojnu vrijednost.

```c
int main(void)
{
    double a = 3.14, b = 4.84;
    int m = (int)a;
    int n = (int)b;
    printf("%.2lf %.2lf\n", a, b);
    printf("%d %d\n", m, n);
    return 0;
}
```

```
3.14 4.84
3 4
```

4.3.6 Ostali operatori

Tablica 4.6: Ostali operatori

operator	opis
`sizeof`	
unarni `&`	adresa varijable
unarni `*`	indirekcija
`[]`	indeksiranje
`()`	poziv funkcije
`.`	izbor člana strukture ili unije
`->`	izbor člana strukture ili unije (preko pokazivača)
`,`	operator zarez
`?:`	uvjetni

Operator `sizeof` je operator koji računa veličinu operanda u bajtovima. Može se pisati na dva načina: `sizeof a` veličinu varijable `a` u bajtovima, a `sizeof(int)` će dati veličinu tipa u bajtovima. U drugom primjeru treba uočiti zagradu.

Operatori unarni `*` i `&` su dio pokazivačke aritmetike, pa ćemo ih obraditi u poglavlju 5, kao i operator indeksiranja. Izbor člana strukture i unije objašnjen je u poglavlju 8. Operator `()` je operator funkcijskog poziva.

Uvjetni operator jedini je ternarni operator (djeluje na tri operanda) u C-u.

Sintaksa:

```
expr_1 ? expr_2 : expr_3
```

- `expr_1` kao rezultat mora dati broj
- ako je taj broj različit od nule, izračunava se `expr_2` i njegova vrijednost je vrijednost cijelog izraza
- u suprotnom (ako je broj jednak nuli) izračunava se `expr_3` i njegova vrijednost je vrijednost cijelog izraza
- asocijativnost s desna na lijevo.

Primjer 4.30 `a = b > 0 ? b : -b .`

Ako je vrijednost varijable `b` pozitivna, varijabla `a` će poprimiti vrijednost koju ima varijabla `b`, inače će poprimiti vrijednost `-b`.

Gornji izraz mogao se napisati i pomoću naredbe `if`:

```
if (b > 0)
    a = b;
else
    a = -b;
```

Razlika između naredbe `if` i uvjetnog operatora `?:` je u tome što je izraz u kojem sudjeluje operator `?:` izraz koji poprima vrijednost, pa može sudjelovati u složenijim izrazima. Za razliku od toga, naredba ne poprima vrijednost. Npr. možemo napisati `if (b != a ? b : a) b = 0;`.

Operator sekvencijalnog izračunavanja `expr_1, expr_2` je operator s najnižim prioritetom. Operator izračunava sekvencijalno prvi pa drugi izraz, a rezultat operatora je vrijednost drugog izraza. Primjena tog operatora je u situacijama kada C očekuje jedan izraz, a mi imamo potrebu "reći" dvije stvari.

Primjer 4.31 Upotreba operatora , u naredbi `for`.

```
for (i = 0, j = 10; i < j; i++, j--)
    naredba
```

Prioritet operatora je nizak, pa u izrazu oblika `x = y + 1, y - 1`, vrijednost izraza će biti vrijednost izraza `x = y + 1`. Iako je vrijednost izraza u kojem sudjeluje operator zarez vrijednost drugog izraza. Npr. `if (x = 0, y = 3)` vrijednost izraza je vrijednost drugog izraza tj. varijable `y`, a to je `3`.

Primjer 4.32 Upotreba operatora ,.

```
int main(void)
{
  int i = 0;
  i = (i + 2,3);
  while(i <= 5)
  {
    i = i + 2,5;
    printf("%d\n", i);
  }
  return 0;
}
```

```
5 7
```

U gornjem primjeru u liniji 4 pojavljuje se operator sekvencijalnog izračunavanja. Vrijednost izraza `(i + 2, 3)` je drugi dio izraza, dakle `3`. U `while` petlji,

pak nalazi se naredba `i = i + 2, 5`. Ovaj put je izraz bez zagrade koja određuje prioritet, pa se moramo prisjetiti pravila: operator zarez ima manji prioritet u odnosu na operator dodjeljivanja. Dakle, gornji izraz mogli bismo zapisati kao `(i = i + 2), 5`. Dakle, vrijednost varijable `i` bit će uvećana za 2.

4.4 Riješeni zadaci

1. Napisati funkciju koja nalazi najveći mogući broj za `unsigned int` (upotrebljavaj operator bitovne negacije).

```
#include <stdio.h>
unsigned max_uint()
{
    return ~0;
}
int main(void)
{
    printf("%u\n", max_uint());
    return 0;
}
```

2. Napisati funkciju koja vraća poziciju vodeće jedinice u binarnom zapisu nenegativnog broja `n`.

```
unsigned msb_position(unsigned n)
{
    unsigned i;
    for (i = 8 * sizeof (unsigned); i > 0; i--)
        if ((1 << i-1) & n)
            return i - 1;
    return -1; /* nema vodeceg bita - tj. broj je 0
       */
}
```

3. Napisati funkciju koja dijeli broj na dijelove od po 4 bita. Svaki dio ispisuje u svom retku.

```
void print_quadruples(unsigned n)
{
    unsigned k;
    while(n)
    {
        printf("%u\n", n & 15);
        n >>= 4;
```

```
}
```

4. Napisati funkciju koja dijeli broj na dijelove od po 3 bita. Svaki dio ispisuje u svom retku.

```
void print_triples(unsigned n)
{
    unsigned k;
    while(n)
    {
        printf("%u\n", n & 7);
        n >>= 3;
    }
}
```

5. Napisati funkciju koja dijeli broj na dijelove od po `k` bitova. Svaki dio ispisuje u svom retku.

```
void print_separate(unsigned n, unsigned k)
{
    unsigned mask = (1 << k) - 1;
    /* broj pomicemo za k mjesta */
    while (n)
    {
        printf("%u\n", n & mask);
        n >>= k;
    }
}
```

6. Napisati funkciju koja dijeli broj na dijelove od po `k` bitova. Svaki dio ispisuje u svom retku u pravilnom poretku.

```
void print_separate_right(unsigned n, unsigned k)
{
    unsigned mask, size = sizeof n * 8;
    unsigned num = size / k, i, tmp;
    mask = (1 << k) - 1;

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

```
            tmp = n >> (num - i -1) * k;
            printf("%u ", tmp & mask);

        }
        printf("\n");
    }
```

Rekurzivno rješenje:

```
void print_separate_rec(unsigned n, unsigned k)
{
    unsigned mask = (1 << k) - 1, i, tmp;
    if (!n) return;
    /* broj pomicemo za k mjesta */
    print_separate_rec(n >> k, k);
    printf("%u ", n & mask);
}
```

7. Napisati funkciju koja za broj n kreira i vraća novi broj tako da mu između svake dvije jedinice u binarnom zapisu ubaci nulu.
   Primjer: Za broj $n = 7_{10} = 111_2$, novi broj je $10101_2 = 21_{10}$

```
int br_znamenki(int n)
{
    return (int)log10(n) + 1;
}
int ubaci(int n)
{
    int k = 0;
    int i = br_znamenki(n) - 1;
    int zn1, zn2;
    while (i >= 0)
    {
        /* prvo izdvajamo i-tu i znamenku pored nje*/
        zn1 = (n & 1<<i) ? 1 : 0;     /* i-ta znamenka
            */
        zn2 = (n & 1<<(i-1)) ? 1 : 0; /* i-1 znamenka
           */
        /* ako znamenku 1 slijedi znamenka 1*/
        if (zn1 && zn2)
            k = (k<<2) + (1<<1);
```

```
            else
                k = (k<<1) + zn1;
            i--;
        }
        return k;
    }
```

8. Napisati funkciju koja ispituje je li prirodni broj Armstrongov broj. Armstrongov broj je n-teroznamenkasti broj koji je jednak sumi n-tih potencija svojih znamenaka.
   Primjer Armstongovih brojeva:

$$153 = 1^3 + 5^3 + 3^3$$

$$1634 = 1^4 + 6^4 + 3^4 + 4^4$$

```
  unsigned count_digits(unsigned n, unsigned k)
{
    if (!n) return 0;
    return 1 + count_digits(n/k, k);
}
int armstrong(unsigned n)
{
  unsigned k = count_digits(n, 10), m = n, sum = 0;
  printf("k = %u\n", k);
  while (m)
  {
      sum += pow(m%10, k);
      printf("sum = %u\n", sum);
      m /= 10;
  }
  return n == sum ? 1 : 0;

}
int armstrong_rec(unsigned n, unsigned k)
{
   if (!n)
     return 0;
   return (int)pow(n%10, k) + armstrong_rec(n/10, k);
}
```

9. Napisati funkciju koja za dva broja ispituje da li je u zapisu jedan završetak drugog.
   Primjer: Za brojeve 126 i 32126, funkcija vraca 1, a za brojeve 123 i 32125 vraca 0.

```
#include <math.h>
unsigned count(unsigned m)
/* broji znamenke u broju*/
{
    return (unsigned)log10(m) + 1;
}

unsigned zavrsetak(unsigned m, unsigned n)
{
    unsigned k, t;
    if (m < n) n ^= m ^= n ^= m; // zamjena
       vrijednosti
    k = (unsigned)pow(10, count(n));
    if (m % k == n) return 1;
    else return 0;

}
```

4.5 Zadaci za vježbanje

1. Koristeći se operatorom ?: napisati funkciju koja za neparne vrijednosti argumenta `n` vraća vrijednost `3 * n + 1`, a za parne `n/2`.
2. Napisati funkciju koja za broj `n` ispisuje niz brojeva tako da je svaki sljedeći broj dobiven kako je opisano u zadatku 1.
3. Napisati funkciju koja ispisuje broj u binarnom sustavu.
4. Napisati program u kojem korisnik unosi brojeve između `0` i `31`. Za unesene brojeve program setira pripadajuće bitove te ispisuje broj.
   Primjer: za unesene brojeve `2`, `3` i `5`, setira se drugi, treći i peti bit. Rezultat je broj $101100_{(2)} = 44_{(10)}$.
5. Napisati funkciju koja za godinu vraća je li prestupna ili ne. Godina je prestupna ako je djeljiva sa `4` i nije djeljiva sa `100` ili ako je djeljiva sa `400`.
6. Napisati fukciju koja služeći se bitovnim operatorima ispituje je li dani `n` potencija broja dva.
7. Napisati funkciju koja broji broj jedinica u binarnom zapisu nekog prirodnog broja.
8. Napisati funkciju koja računa koliko najviše susjednih jedinica ima u bitovnom zapisu prirodnog broja `n`.
   Primjer: za $\texttt{n} = 7055_{(10)} = 1101110001111_{(2)}$ funkcija bi trebala vratiti broj 4.
9. Napisati funkciju koja kriptira (i dekriptira) broj koristeći se operatorom ekskluzivno ili. Funkcija ima parametre `key` i `value`. Iz funkcije `main` pozvati funkciju za kriptiranje broja `value`, te za dekriptiranje dobivenog kriptiranog broja, te provjeriti je li dekriptiran broj jednak originalnom broju.
10. Napisati program koji detektira veličinu podatka koji daje `sizeof` operator.
11. Napisati funkciju za oduzimanje dva broja, ne koristeći se operatorom `-`.
12. Koristeći uvjetni operator (?:) napisati funkciju koja vraća najmanji od tri broja.
13. Napisati funkciju koje će ispitati da li uneseni prirodni broj `n` ima simetričan binarni zapis.

Primjer:
za `n = 17`
10001 je simetričan zapis

14. Napisati funkciju koja ispituje je li broj izbalansiran tj. ima li u binarnom zapisu ima jednak broj nula i jedinica. Upotrebljavati bitovne operatore. Primjer: broj 9 (1001) ima jednak broj nula i jedinica.
15. Napisati program za zamjenu dviju varijabli (swap) ne koristeći se dodatnom varijablom i služeći se bitovnim operatorima (ekskluzivno ili).
16. Napisati funkciju koja provjerava ima li uneseni prirodni broj u binarnom zapisu sve jedinice.
17. Napisati funkciju koja zadani unsigned short konvertira tako da sve jedinice u binarnom zapisu gurne desno do kraja.
18. Napisati funkciju koja u binarnom zapisu broja pretvara sve jedinice na parnim pozicijama u nule, a sve nule (iz originalnog zapisa) pretvara u jednice.
    Napomena: Parnim mjestima smatramo `0, 2, 4, ...` bit gledano od bita najmanje težine.
    Primjer: $n = 23_{(10)} = 10111_{(2)}$ pretvori u $00010_{(2)} = 2_{(10)}$
19. Napisati funkciju koja setira bit za slučajno odabrani broj između 0 i 31.
20. Napisati program koji za svaki uneseni broj između 0 i 31 postavlja (setira) bit u nekom broju `k`. Unos brojeva je gotov kada korisnik unese broj koji nije između 0 i 31. Program dalje ispisuje koliko je različitih brojeva unešeno.
21. Napisati funkciju koja za uneseni prirodni broj `n` vraća broj koji ima `n` znamenki u binarnom zapisu i kojem su na neparnim mjestima u binarnom zapisu jedinice. Upotrebljavati bitovne operatore.
    Primjer: za `n = 4`, izračunati broj je `1010` binarno tj. broj `10`.
22. Napisati funkciju koja broju `n` (tipa `unsigned char`) u binarnom zapisu na neparnim mjestima pretvori nule u jedinice i vrati novi broj. Znamenke na parnim mjestima ne mijenja.
    Primjer: Za $19_{(10)} = 000010011_{(2)}$ vraća broj $187_{(10)} = 010111011_{(2)}$
23. Napisati funkciju koja za prirodni broj `n` vraća broj koliko u bitovnom

zapisu broja `n` ima nula.
Primjer: za broj `n = 8`, funkcija vraća broj `3`.

24. Napisati funkciju koja vraća broj nula na neparnim mjestima u bitovom zapisu broja `n`.
25. Prirodni broj je **zao** (engl. *evil*) ako u svojoj binarnoj reprezentaciji ima neparan broj jedinica. Napisati program koji broji koliko ima takvih prirodnih brojeva do broja `m`. Broj `m` unosi korisnik.
    Napomena 1: Za ispitivanje svojstva "je zao" napisati funkciju

    ```
    int is_evil(unsigned n).
    ```

    Napomena 2: Koristiti bitovne operatore.
26. Napisati funkciju koja vraća najmanju potenciju broja dva veću od danog broja. Koristiti se bitovnima operatorima.
    Primjer: za broj 27 ($27_{(10)} = 11011_{(2)}$) funkcija vraća broj 32 ($32_{(10)} = 100000_{(2)}$).
27. Napisati funkciju koja provjerava ima li unsigned broj `n` neki bajt jednak danom broju `k`.
28. Napisati funkciju koja vraća 1 ako broj `n` ima paran broj jedinica u bitovnom zapisu, inače vraća 0. Koristiti se bitovnim operatorima.
29. Napisati funkciju koja računa (i vraća) na koliko mjesta u bitovnom zapisu dva broja imaju različitu vrijednost bitova.
30. Korisnik unosi znamenke, znak po znak, odvojene razmakom. Funkcija iz znamenki konstruira i vraća broj (int). Ako nije unesena znamenka, unos se nastavlja. Unos završava kad korisnik unese znak 'q'.
    Primjer: za unos
    2
    3
    4
    r
    5
    funkcija vraća broj 2345.
31. Napisati funkciju koja vraća rezultat "logičkog ekskluzivnog ili" operatora,

tj. vraća 1 ako je jedan operand različit od nule, a drugi jednak nuli ili obrnuto. U svim ostalim slučajevima funkcija vraća 0.

32. Napisati funkciju koja provjerava je li prirodni broj jednak sumi faktorijela svojih znamenki (strong number). *Primjer*:

$$145 = 1! + 4! + 5!$$

33. Napisati funkciju koja ispisuje Floydov trokut od n redaka. 1
    2 3
    4 5 6
    7 8 9 10
    11 12 13 14 15
34. Napisati funkciju koja broj dijeli na dva dijela, te vraća razliku nova dva broja. Ako je broj znamenaka parni, broj se dijeli na dva jednaka dijela, a ako je neparni prvi broj rastava ima znamenku više.
    Primjer:
    1234 će se rastaviti kao 12 i 34
    12345 će se rastaviti kao 123 i 45
35. Napisati funkciju koja računa sumu znamenki broja sve dok ne dođe do broja od jedne znamenke.
    Primjer: za 3456: $3 + 4 + 5 + 6 = 18$, $1 + 8 = 9$
36. Napisati funkciju koja iz broja n kreira novi broj tako da zbraja susjedne znamenke.
    Primjer: za broj 3456 novi broj je 7911.
37. Napisati funkciju koja za prirodni broj n vraća novi broj koji se dobije tako da se broj podijeli na dva dijela te se zamijene prvi i zadnji dio broja. Ukoliko je broj znamenaka negativan, srednja znamenka ostaje na mjestu.
    Primjer: 1234 novi broj je 3412
    12345 novi broj je 45312.
38. Koristeći operator binarni minus(-) napisati funkciju koja vraća najmanji od tri broja.
39. Napisati funkciju koja provjerava je li binarni zapis nekog broja takav da su sve jedinice ispred nula.

Primjer: U broju $248_{(10)} = 11111000_{(2)}$ su sve jedinice ispred nule.
Primjer: Za broj $n = 7_{(10)} = 111_{(2)}$, novi broj je $10101_{(2)} = 21_{(10)}$

40. Napisati funkciju koja vraća broj svih prostih brojeva do 1000000 koji ima sve jednake znamenke.
41. Napisati funkciju koja ispituje je li neki broj u hexadecimalnom obliku palindrom.

4.6 Ponavljanje

Što ispisuju sljedeći programi i zašto?

1. ——

```
int main(void)
{
    int i = 0;
    i = (i + 1,5);
    while(i <= 5)
    {
        i = i + 1,5;
        printf("%d\n", i);
    }
    return 0;
}
```

2. ——

```
int main(void)
{
    unsigned n = -1;
    if (n < 0)
        printf("%d\n", n);
    else
        printf("%d\n", n - 1);
    return 0;

}
```

3. ——

```
int main(void)
{
    unsigned n = -1;
    printf("%d\n", n < 0 ? n : (n - 1,5));
    return 0;
}
```

4. ——

```
int main(void)
{
    char ch = 'a';
    switch(ch)
```

```
    {
        case 'a': printf("ja sam a\n");
        case 'b': printf("ja sam b\n");break;
        default: printf("nisam ni a ni b\n");
    }
    return 0;
}
```

5. ———

```
int is_odd(unsigned n)
{
    if (!(n % 2))
    {
        printf("ja sam paran\n");
        return 0;
    }
    printf("ja sam neparan\n");
    return 1;
}

int main(void)
{
    unsigned n = -1;
    if (n < 0 && is_odd(n))
        printf("neparan i negativan\n");
    else
        printf("pozitivan\n");
    return 0;
}
```

6. ———

```
int main(void)
{
  int i;
  i = -1,1;
  printf("i: %d\n",i ? -i : i);
  return 0;
}
```

7. ———

```
int main(void)
{
  int i = 0;
  i = (i + 4,3);
```

```
  while(i <= 5)
  {
    i = i + 4,3;
    printf("%d\n", i);
  }
  return 0;
}
```

8. ———

```
int main(void)
{
  int i;
  i = -1,1;
  printf("i: %d\n",i ? 1 - i : i);
  return 0;
}
```

9. ———

```
int main(void)
{
  int i;
  i = (1, 3, 2);
  switch(i++)
  {
    case 1: printf("1\n");
    case 2: printf("2\n");
    case 3: printf("3\n"); break;
    default: printf("nijedan");
  }
  return 0;
}
```

10. ———

```
int main(void)
{
  unsigned n = -1;
  printf("%d\n", n < 0 ? n : (n - 1,5));
  return 0;
}
```

11. ———

```
int main(void)
{
```

```
    unsigned i, b = -10;
    for (i = b; i < 0; i+=2)
        printf("%u \t", i * i);
    printf("%d \n", i);
    return 0;

}
```

12. ———

```
int main(void)
{
  unsigned i = 32;
  while (i)
  {
     printf("%u\t", i >> 1);
     i >>= 2;
  }
  return 0;
}
```

13. ———

```
int main(void)
{
    unsigned n = -1, k = 6;
    while(n++ > 0)
        printf("%u\n", k >> 2);
    return 0;
}
```

14. ———

```
int main(void)
{
    int i = (1, 2, 3), j = (i, 2);
    if (i > j)
        printf("%d, %d\n", i++, ++j);
    else
        printf("%d, %d\n", i--, --j);
    return 0;
}
```

15. ———

```
#include<stdio.h>
int main(void)
```

```
{
  char c = 'A';
  int d = 65;
  printf("%d %d %d\n", d, d += c > 'A'&& c <= 'Z', c++);
  return 0;
}
```

16. ———

```
#include<stdio.h>
int main(void)
{
  int x = 10, y = 4;
  printf("x/y: %d\n", x/y);
  printf("x<<y: %d\n", x<<y);
  printf("x>>y: %d\n", x>>y);
  return 0;
}
```

17. ———

```
#include <stdio.h>
int main(void)
{
  int i = 1;
  do
  {
    printf("%d\n",i++);
    if(++i < 3)
      continue;
    printf("%d\n",i++);
  }while(0);
  return 0;
}
```

18. ———

```
#include<stdio.h>
int main(void)
{
  int i = 2222;
  printf("%d\n", printf("%d\t", printf("%d\t", i)));
  return 0;
}
```

19. ———

```
#include <stdio.h>
int main(void)
{
  int a = 5, b;
  b = a-- >= 5 ? 0,20 : 0,10;
  printf("%d\n",b);
  return 0;
}
```

20. ———

```
#include <stdio.h>
int main(void)
{
  char ch = 'A';
  while(ch <='F')
  {
    switch(ch)
    {
      case 'A': case 'B':
      case 'C': case 'D': ch++; continue;
      case 'E': case 'F': ch++;
    }
    printf("%c\n", ch);
  }
  return 0;
}
```

21. ———

```
#include <stdio.h>
int main(void)
{
  int a = 8, b = 5, c = 2, d = 1;
  if ((a < b) && (c = ++d))
    printf(" %d %d ", c, d);
  else
    printf(" %d %d ", c, d);
  return 0;
}
```

22. ———

```
#include <stdio.h>
int main(void)
{
  int p;
```

```
  for(p = 1; p <= 5, --p; p++)
    printf("Hello \n");
  printf("%d\n", p);
  return 0;
}
```

23. ———

```
#include<stdio.h>
int main(void)
{
  int a = 3,b = 1;
  a = b << a + b >> 2;
  b = b << a + b >> 2;
  printf("%d %d\n", a, b);
  return 0;
}
```

24. ———

```
#include<stdio.h>
int main(void)
{
  int a=1,b=3,c,d;
  c=(a,b);
  d=a,b,c;
  printf("%d %d\n",c,d);
  return 0;
}
```

25. ———

```
int main(void)
{
    int n = 1, k = 2;
    if (n & k == n + k)
        printf("%d\n", n << k);
    else
        printf("%d\n", k << n);
    return 0;
}
```

26. ———

```
int main(void)
{
    int n, k;
```

```
    n, k = 2, 3;
    for (n = 0; n < k--; n++)
        printf("%d %d\n", n, k);
    return 0;
}
```

27. ———

```
int main(void)
{
    unsigned n = -1, i, j;
    for (i = 0, j = n; i < j; j++)
        printf("%d %d\n", i, j);
    return 0;
}
```

28. ———

```
int main(void)
{
    int n = 4, k = 0;
    while(n--)
        k = n ? !k : ++k;
    printf("n = %d, k = %d\n", n, k);
    return 0;
}
```

29. ———

```
int main(void)
{
    int n = 2;
    n = 5,2 * n;
    while (n % 8)
        n <<= 1;
    printf("%d\n", n);
    return 0;
}
```

30. ———

```
int main(void)
{
    unsigned m = -1, n = 1;
    n ^= m^= n^= m;
    printf("%d %d\n", m, n);
    return 0;
}
```

31. ———

```
int main(void)
{
  int i;
  i = 10;
  printf("i : %d\n",i);
  printf("i : %d\n", i++);
  printf("i : %d\n", ++i);
  return 0;
}
```

32. ———

```
int main(void)
{
  unsigned n = -1;
  printf("%d\n", n < 0 ? n : (n - 1,5));
  return 0;
}
```

33. ———

```
int main(void)
{
  int i;
  i = (1, 3, 2);
  switch(i++)
  {
    case 1: printf("1\n");
    case 2: printf("2\n");
    case 3: printf("3\n"); break;
    default: printf("nijedan");
  }
  return 0;
}
```

34. ———

```
int main()
{
    int i, n = 10;
    for(i = 0; i < n; i++)
    {
        if(i < 5)
            continue;
```

```
        printf("%d\n",i);
    }
    printf("i = %d\n", i);
    return 0;
}

int main(void)
{
    int n = 1, k = 2;
    if (n * k == k & n)
        printf("%d\n", n << k);
    else
        printf("%d\n", k << n);
    return 0;
}
```

35. ———

```
int main()
{
    unsigned x = 11;
    int count = 0;
    while(x)
    {
        count++;
        x = x&(x-1);
    }
    printf("%d\n", count);
    return 0;
}
```

36. ———

```
int main(void)
{
    int n = 5, m = 2;
    int x = n ^ m && n << m;
    if (x & 2)
        printf("%d\n", x >> 2);
    printf("%d\n", x);
    return 0;
}
```

37. ———

```
int main(void)
{
```

```
    int n = 4, m = 2;
    int x = n ^ m && n >> m;
    if (x & 1)
        printf("%d\n", x << 2);
    printf("%d\n", x);
    return 0;
}
```

38. ———

```
int main(void)
{
   unsigned n = -1;
   printf("%d\n",n<0 ? n<<2 : (n + 1,1));
   return 0;
}
```

39. ———-

```
int main(void)
{
  char k = 7;
  unsigned sz = 8 * sizeof k;
  int i;
  for (i = 0; i < sz; i++)
     printf("%d",(k<<i & 1<<sz-1)? 1:0);
  return 0;
}
```

40. ———

```
int func(char p, int n)
{
    int br = 0;
    while((p&(1<<n--)) == 0)
        br++;
    return br;
}
int main()
{
    char c = 1;
    int n = sizeof(char) * 8 - 1;
    printf("%d\n", func(c, n));
    return 0;
}
```

41. ———

```
int main(void)
{
    int i = 5;
    i = ++i/i++;
    printf("%d\n", i);
    return 0;
}
```

42. ———

```
void fja(int n)
{
    static int m = 5;
    if (m-- >= --n)
        printf("%d\t",--m);

}
int main(void)
{
    int k = 4, i;
    for (i = 0; i < k; i++)
        fja(i);
    return 0;
}
```

43. ———

```
int main()
{
  int i=-1;
  -i;
  printf("i=%d, -i=%d\n",i,-i);
  return 0;
}
```

44. ———

```
#include <stdio.h>
int fun(int a)
{
     return a << 2 + a;
}

int main()
{
     int a = 1, b = 2;
     printf("%d\n", fun(a));
```

```
    printf("%d\n", fun(b));
    printf("%d\n", fun(a&b));
    return 0;
}
```

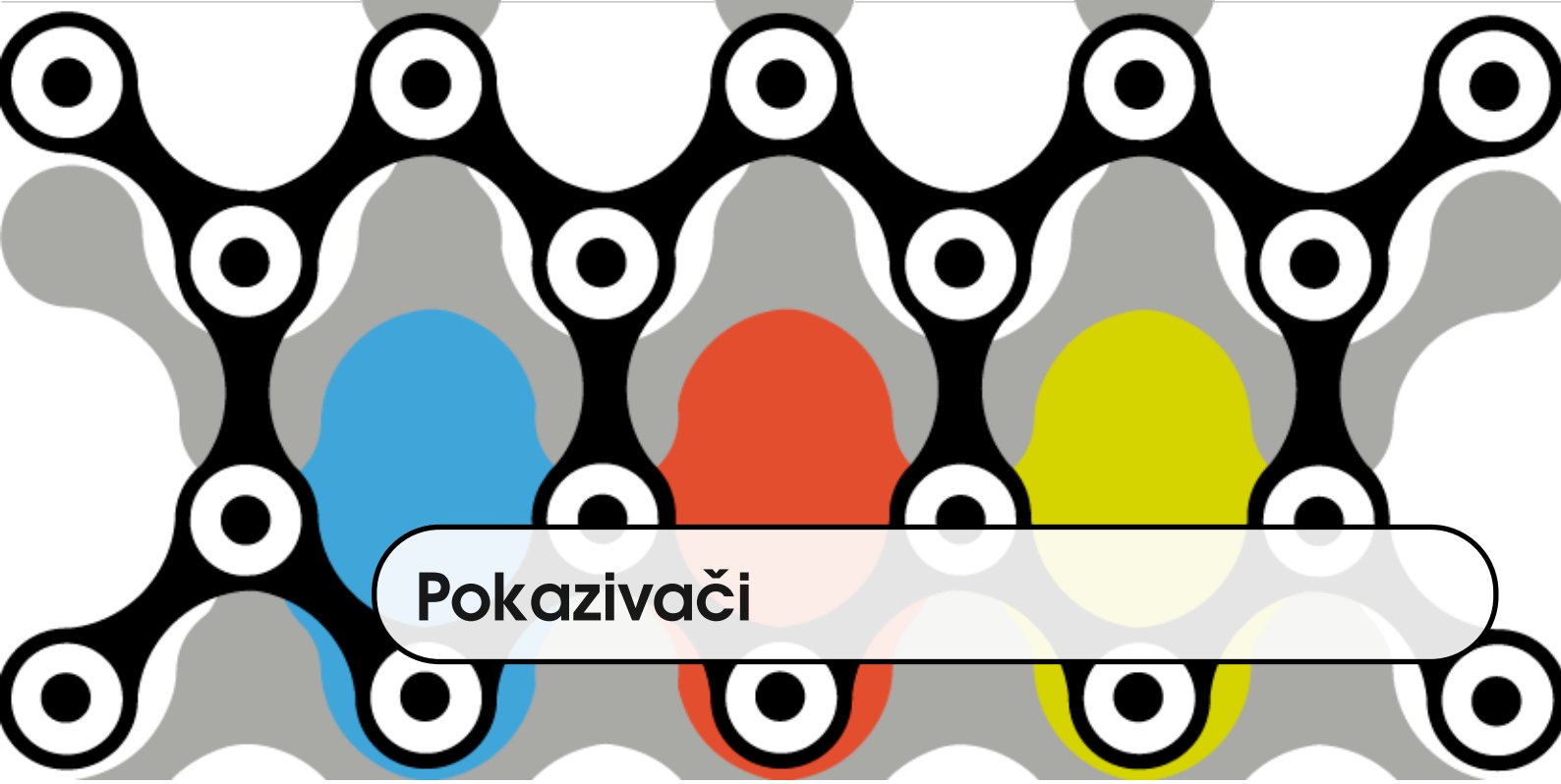
45. ———

```
#include <stdio.h>
int fun(int a, int b)
{
    static int x = 6;
    return a + b > x-- ? ++a: ++b;
}
int main()
{
    int x = 1, y = 4;
    while(fun(x, y) < y--);
    printf("%d %d\n", x, y);
    return 0;

}
```

46. ———

```
int main()
{
    int i, n = 10;
    for(i = 0; i < n; i++)
    {
        if(i < 5)
            continue;
        printf("%d\n",i%2);
    }
    printf("i = %d\n", i-n);
    return 0;
}
```



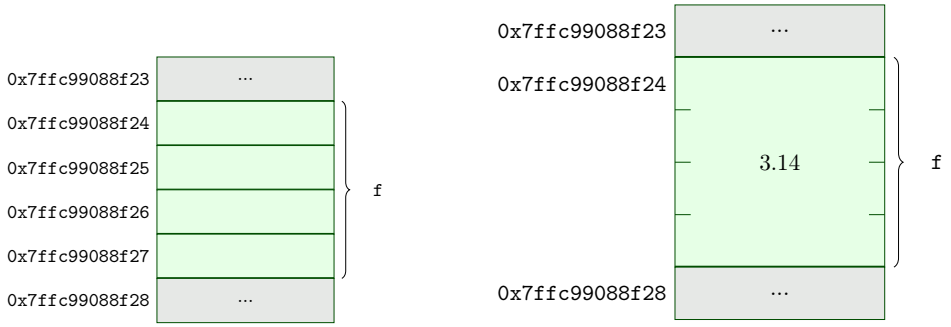
Pokazivači

5.1 Pokazivačke varijable

Da bi mogli pričati o pokazivačima (engl. *pointer*) i adresama, potrebno se vratiti na temu što je varijabla i što smo o varijablama do sada naučili.

Varijabla je simboličko ime za neku lokaciju u memoriji i njena vrijednost je sadržaj lokacije u memoriji. Varijabla se **deklarira** npr. `float a;`. Značenje deklaracije je zauzimanje određene količine lokacija u memoriji, ali i dodjeljivanje određenog tipa. C je jezik **statičkih tipova**, što znači da kad se jednom varijabla deklarira kao varijabla određenoga tipa, ona do kraja svog života ostaje varijabla tog istog tipa. Veličina nekog tipa, tj. količina memorije koju će joj kompajler dodjeliti ovisi o arhitekturi za koju se program kompajlira. Npr. `float` varijabla zauzima `sizeof(float)` bajtova. Na većini 32-bitnih platformi, `float` će biti velik 4 bajta.

Kada se varijabla deklarira, njena vrijednost nije određena. Pridruživanje vrijednosti varijabli može se izvršiti **inicijalizacijom**. Primjer je `a = 0.5;`. Ako se korisniku želi prepustiti dodjeljivanje vrijednosti varijabli, to se može realizirati npr. pozivom funkcije `scanf`.



(a) 4 bajta rezervirana za `f` **(b)** Vrijednost `3.14` pridružena varijabli `f`

Slika 5.1: Memorija

Svaka varijabla prilikom deklaracije dobije svoje "mjesto" u memoriji, odnosno dobije nekoliko memorijskih lokacija. S druge strane, svaka memorijska lokacija ima **adresu**. Na slici 5.1a vidimo da je za varijablu `f` rezervirano 4 bajta na adresi `0x7ffc99088f24`. Prilikom inicijalizacije `f = 3.14;` toj se varijabli dodjelila vrijednost `3.14` (slika 5.1b). Adresu varijable možemo dohvatiti **adresnim operatorom** `&` (ampersand).

ADRESU

ADRESNI OPERATOR

Primjer 5.1 Isprobajte sljedeći kôd.

```
#include <stdio.h>
int main(void)
{
    float f = 3.14;
    printf("f = %f, na adresi %p\n", f, &f);
    return 0;
}
```

Možemo primjetiti da se adresa varijable dohvaćena operatorom `&` ispisuje kao heksadecimalni broj, koristeći se oznakom formata `%p` u `printf` funkciji.

Tako će se prikazivati i u debuggeru.

5.2 Pokazivači i adrese

Sada kada znamo da svaka memorijska lokacija ima adresu i kada znamo čak i kako dohvatiti tu adresu, postavlja se pitanje: što s tom adresom možemo učiniti? Kojeg tipa treba biti varijabla u koju bismo mogli spremiti vrijednost adrese.

Tu na scenu upadaju pokazivači. Pokazivači su varijable čija vrijednost može biti adresa neke memorijske lokacije. Kako adresa memorijske lokacije ne kaže ništa o tome koji će tip varijable biti zapisan u nju, ne može se deklarirati generički pokazivač kojemu će vrijednost biti adresa bilo koje varijable, nego se deklarira pokazivač na određeni tip varijable.

Deklaracija pokazivača: `float* pf;` kaže da je pf pokazivač čija vrijednost može biti adresa neke `float` varijable. Kako je pf varijabla, za nju vrijedi isto što i za bilo koju varijablu: u trenutku deklaracije kompajler joj dodijeli memoriju. Koliko memorije zauzima pokazivač ovisi o platformi: na 32-bitnim platformama obično zauzimaju 4 bajta, a na 64-bitnim 8 bajtova. Međutim, veličinu je opet najlakše odrediti pomoću `sizeof` operatora.

Dakle, deklaracija pokazivača (kao i bilo koje druge varijable) određuje dvije stvari: dodjeljuje memoriju varijabli i određuje kakve vrijednosti varijabla može poprimiti. U ovom trenutku, nakon same deklaracije, varijabla još nije poprimila vrijednost.

5.3 Inicijalizacija pokazivača

INICIJALIZACIJA POKAZIVAČA

Vrijednost koju pokazivač može poprimiti je adresa neke varijable određenog tipa. Ako je pokazivač bio deklariran kao `float*`, on može kao vrijednost poprimiti adresu neke `float` varijable.

Primjer 5.2 Ispis varijable i njene adrese.

```
#include <stdio.h>
int main(void)
{
    float f = 3.14;
    float* pf = &f;
    printf("float varijabla f = %f, na adresi %p\n",
                                f, &f);
    printf("pointer na float pf = %p, na adresi %p\n",
                                  pf, &pf);
    return 0;
}
```

Možemo primjetiti da i sam pokazivač ima svoju adresu koju smo ispisali sa `&pf`. Takve vrijednosti možemo spremati u varijable tipa pokazivač na pokazivač.

5.4 Dereferenciranje pokazivača

DEREFERENCIRANJE POKAZIVAČA

Kada smo pokazivaču pridijelili vrijednost adrese neke varijable, do same vrijednosti varijable možemo pristupiti **dereferenciranjem pokazivača**. To se izvodi pomoću **operatora dereferenciranja ili indirekcije** *.

Primjer 5.3 Pristup vrijednostima varijabli preko pokazivača.

```
int main(void)
{
    int a = 5, b = 4;
    int* p = &a;
    printf("a je %d, preko pointera %d\n", a, *p);

    /* preko pointera p mozemo promijeniti *
     * vrijednost varijable a           */
    *p = 10;
```

```
        printf("a je %d, preko pointera %d\n", a, *p);

        /* pointer je varijabla pa moze promijeniti vrijednost
           */
        p = &b;
        printf("p = %p, a to je adresa varijable b %p\", p, &b
           );
        printf("b je %d, preko pointera %d\n", b, *p);

        /* preko pointera p mozemo promijeniti *
         * vrijednost varijable b          */
        *p = 3;
        printf("b je %d, preko pointera %d\n", b, *p);
    }
```

5.5 Pokazivači i funkcije

Jedna od najčešćih primjena pokazivača je omogućavanje funkcijama da promijene varijable izvan svoje vidljivosti. Naime, svaka funkcija (pa i `main` funkcija, naravno) vidi svoje lokalne varijable i globalne varijable (o kojima će biti riječi u poglavlju 10). Međutim, funkcija ne vidi varijable koje su lokalne za neku drugu funkciju.

Primjer 5.4 Pokrenite program i proučite što se događa (koristeći se *debuggerom*).

```
void swap(int a, int b)
{
    int t = a;
    a = b;
    b = t;
}

int main(void)
{
    int x = 5, y = 3;
```

```
    printf("x = %d, y = %d\n", x, y);
    swap(x, y);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Varijable `a` i `b` su lokalne varijable funkcije `swap`. Kao takve, vidljive su samo iz funkcije `swap`, a životni vijek im je od ulaska u funkciju `swap` do izlaska iz funkcije `swap`.

Varijable `x` i `y` su lokalne varijable funkcije `main`. Vidljive su iz funkcije `main`, a životni vijek im je od ulaska u funkciju `main` do izlaska iz funkcije `main`.

Prilikom poziva funkcije `swap` u liniji 12, **kopiraju se vrijednosti** varijabli `x` i `y` u varijable `a` i `b`. Dakle, vrijednosti 5 i 3 se dohvaćaju i šalju funkciji `swap`. Parametri funkcije `swap` prime te vrijednosti, te parametar `a` dobije vrijednost `5`, a parametar `b` vrijednost 3. Nakon toga u funkciji `swap` varijable `a` i `b` zamjenjuju svoje vrijednosti, ali to nema nikakvog utjecaja na varijable `x` i `y` funkcije `main`. Nakon izlaska iz funkcije `swap`, `x` i `y` su u funkciji `main` ostali isti.

Da bismo zaobišli ovo ograničenje upotrebljavamo pokazivače. Funkciji `swap` nećemo poslati kopije vrijednosti varijabli, nego kopije vrijednosti adresa varijabli, dakle pokazivače.

Primjer 5.5 Ispravna funkcija za zamjenu vrijednosti parametara.

```
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

int main(void)
{
```

```
    int x = 5, y = 3;
    printf("x = %d, y = %d\n", x, y);
    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Sada pri pozivu funkcije `swap`, vrijednosti adresa varijabli `x` i `y` se kopiraju i dodjeljuju parametrima funkcije `swap`. Parametri `a` i `b` funkcije `swap` su pokazivači. Preko njih, upotrebljavajući operator dereferenciranja *, funkcija `swap` mijenja (indirektno) vrijednosti varijabli `x` i `y`.

Funkcija preko naredbe `return` može vratiti samo jednu vrijednost. Kada želimo da nam funkcija izračunava više od jednog podatka, rezultat se ne može na jednostavan način vratiti preko naredbe `return`. U tu svrhu koristimo se pokazivačima. Adrese lokacija na koje želimo zapisati rezultate proslijeđuju se funkciji, a funkcija operatorom dereferenciranja pristupa tim lokacijama.

Primjer 5.6 Napisati funkciju koja računa sumu i produkt prvih `n` prirodnih brojeva.

```
void sum_product(unsigned n, unsigned* ps, unsigned* pp)
{
    unsigned i;
    *ps = 0;
    *pp = 1;
    for (i = 1; i <= n; i++)
    {
        *ps += i;
        *pp *= i;
    }
}

int main(void)
{
```

```
    unsigned n, suma, produkt;
    printf("unesi n\n");
    scanf("%u", &n);
    sum_product(n, &suma, &produkt);
    printf("suma je %u, a produkt %u\n", suma, produkt);
    return 0;
}
```

5.6 Pokazivačka aritmetika

ODUZIMANJE POKAZIVAČA

Zbrajanje dva pokazivača nema puno smisla. Vrijednost pokazivača je memorijska adresa, pa dvije zbrojene memorijske adrese ne moraju biti opet memorijska adresa. S druge strane, **oduzimanje pokazivača** ima smisla. Rezultat oduzimanja pokazivača je udaljenost između varijabli na koje ti pokazivači pokazuju.

Primjer 5.7 Možemo li predvidjeti što će biti ispis nakon izvršavanja ovog programa?

```
#include <stdio.h>
int main()
{
    int a, b, r;
    int* p, *q;
    p = &a;
    q = &b;
    r = p - q;
    printf("r = %d\n", r);
    return 0;
}
```

Množenje i djeljenje kako pokazivača s pokazivačom, tako i pokazivača sa brojem, također nemaju smisla.

Nadalje, pokazivaču možemo dodati vrijednost i oduzeti neku vrijednost. Naprimjer, ako je `p` pokazivač na varijablu `x`, a `n` cijeli broj, onda je `p + n` memorijska lokacija koja je `n` memorijskih lokacija daleko od varijable `x`. Ovo će nam biti važno kod rada s nizovima.

Slično će i `p - n` biti memorijska lokacija koja se nalazi `n` memorijskih lokacija ispred varijable `x`. Pri tome treba paziti da ne dereferenciramo pokazivač koji pokazuje na dio memorije koja nam ne pripada.

Još jedna dozvoljena operacija s pokazivačima je operacija uspoređivanja pokazivača. Za dva pokazivača možemo reći jesu li jednaki ili je jedan ispred drugog, tj. vrijednost jedne pokazivačke varijable je manja od vrijednosti druge pokazivačke varijable.

USPOREDJIVANJE POKAZIVAČA

Kod rada s pokazivačima važno je obratiti pažnju na tipične pogreške. Jedna od njih je dereferenciranje nul-pokazivača:

```
int a = 5, b;
int* p = 0;
b = *p + a;
```

U drugoj liniji deklariran je pokazivač na integer `p` i inicijaliziran je na vrijednost `0`, tj. on je nul-pokazivač. **Nul-pokazivač ne smijemo dereferencirati**.

Sličan problem vidimo i u sljedećem kodu:

```
int a = 5, b;
int* p;
b = *p + a
```

U liniji 3 pokušavamo dereferencirati pokazivač koji nema nikakvu vrijednost. Analogno tome kako ni brojčane varijable ne možemo upotrebljavati u izrazima ako nemaju neku vrijednosti, tako se ni pokazivačkim varijablama ne bi trebali koristiti ako su neinicijalizirane.

5.7 Pokazivači i nizovi

NIZ

Situacije modeliramo nizovima kada imamo određen (u slučaju statičkog niza) broj varijabli istog tipa. Naprimjer, niz ocjena studenata. Ili niz unešenih realnih brojeva.

Deklaracija niza Niz se deklarira tako da se navede tip elemenata te broj elemenata u nizu. Naprimjer, za niz od 10 integera deklaracija je: `int a[10];` Kao i prilikom deklaracije varijabli jednostavnih tipova, deklaracijom niza se također samo alocira prostor za niz. Vrijednosti elemenata niza time nisu određene. Važno je uočiti da navedeni broj mora biti dostatan za planirani niz; naknadno se ne može promijeniti veličina niza.

DEKLARACIJA NIZA

INICIJALIZACIJA NIZA

Inicijalizacija niza Vrijednosti elemenata niza mogu se pridijeliti prilikom inicijalizacije. Ukoliko se niz inicijalizira prilikom deklaracije, nije potrebno navesti veličinu niza u uglatim zagradama.

Primjer 5.8 `int a[] = {1, 2, 0, -5, 3, 8};`

Ako je veličina navedena, a prilikom inicijalizacije je pobrojano manje elemenata od broja deklariranih elemenata, niz će se nadopuniti nulama.

Primjer 5.9 int a[10] = `{1, 2, 0, -5, 3, 8};` U ovom primjeru elementi `a[6]` do `a[9]` će biti nula.

Prilikom alokacije memorije za niz, alocirat će se **uzastopne** memorijske lokacije. Na taj način smo sigurni da je element `a[i]` sigurno u memoriji odmah iza `a[i - 1]`.

Samo ime niza može se identificirati s **pokazivačkom konstantom koja po-**

kazuje na prvi član niza. Dakle za niz a vrijedi

$$a \equiv \&a[0]$$

S obzirom na to da se elementi niza nalaze na uzastopnim memorijskim lokacijama, slijedi da je

$$a + 1 \equiv \&a[1]$$

i općenito

$$a + i \equiv \&a[i]$$

Kad na taj izraz primjenimo operator dereferenciranja dobivamo

$$*(a + i) \equiv a[i]$$

Primjer 5.10 Napisati funkcije za unos i ispis članova niza cijelih nenegativnih brojeva.

```
void input_arr(unsigned arr[], unsigned n)
{
    unsigned i;
    for (i = 0; i < n; i++)
        scanf("%u", &arr[i]);
}
void print_arr(unsigned arr[], unsigned n)
{
    unsigned i;
    for (i = 0; i < n; i++)
        printf("%u\t", arr[i]);
}
```

Pokazivačkom notacijom iste bi funkcije izgledale ovako:

```
void input_arr(unsigned* arr, unsigned n)
{
    unsigned i;
    for (i = 0; i < n; i++)
```

```
        scanf("%u", arr + i);
}
void print_arr(unsigned* arr, unsigned n)
{
    unsigned i;
    for (i = 0; i < n; i++)
        printf("%u\t", *(arr + i));
}
int main(void)
{
    int arr[10];
    input_arr(arr, n);
    print_arr(arr, n);

    return 0;
}
```

Posljedica je činjenice da ime niza možemo identificirati sa konstantnim pokazivačem na početak niza da vrijednost te varijable ne možemo promijeniti. To je ilustriramo na sljedećem primjeru:

Primjer 5.11 Greška!

```
#include <stdio.h>
int main(void)
{
    int a[] = {1, 2, 3, 4, 5};
    a += 2;           // ERROR
    return 0;
}
```

Međutim, sljedeći kôd će se uredno kompajlirati.

```
#include <stdio.h>
int main(void)
{
```

```
    int a[] = {1, 2, 3, 4, 5};
    int* p = a;
    p += 2;
    return 0;
}
```

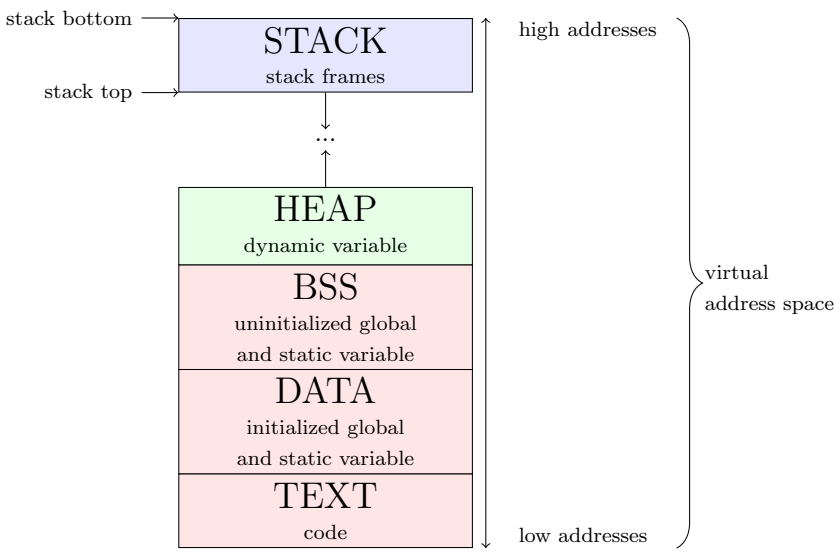
5.8 Stog i hrpa

Da bismo shvatili dinamičku alokaciju memorije, treba razumjeti dodjeljivanje memorije procesu. **Proces** je program u izvršavanju. Prilikom punjenja u memoriju, proces dobiva određeni dio memorije koji zovemo **virtualni adresni prostor** (slika 5.2). Dio virtualnog adresnog prostora zauzima sam kôd programa (text segment), dio globalne i statičke varijable, a ostatak zauzima stog (engl. *stack*) i hrpa (engl. *heap*) procesa. Stack se sastoji od stack frameova koji se alociraju za parametre, adrese povratnih vrijednosti i lokalne varijable funkcija. Heap je dio memorije koje proces dobiva za svoje dinamičke varijable koje se u C-u alociraju pomoću `malloc()`, `calloc()` i `realloc()` funkcija. Svojstvo takvih varijabli je da ne pripadaju određenoj funkciji, već cijelom procesu, te se mogu alocirati unutar jedne, a dealocirati unutar druge funkcije. Međutim, za razliku od automatskih lokalnih varijabli, čija se memorija dealocira prilikom izlaska iz funkcije, pri dinamičkoj alokaciji nema automatske dealokacije dok izvršavanje programa ne završi. Programer je dužan brinuti se o dealokaciji. Dealokacija se izvršava pozivom funkcije `free()`. Pokazivač koji se prosljeđuje funkciji `free()` mora biti točno onaj pokazivač koji je dobiven pomoću funkcija `malloc`, `realloc()` ili `calloc()`.

5.9 Dinamička alokacija memorije

DINAMIČKA ALOKACIJA

U predhodnim primjerima razmatrali smo statičke nizove, tj. nizove kojima je unaprijed dana veličina koja se ne može mijenjati za vrijeme izvršavanja pro-



Slika 5.2: Stog i hrpa

grama. Često je potrebno mijenjati veličinu niza tijekom izvršavanja programa (npr. ako želimo da korisnik unosi veličinu niza), pa te situacije modeliramo služeći se dinamičkom alokacijom. Za dinamičku alokaciju u C-u se koriste funkcije standardne biblioteke. To su `malloc`, `realloc`, `calloc` i funkcija `free` za oslobađanje memorije. Razlika između statičke i dinamičke alokacije je u tome što se memorija za statičku alokaciju dodjeljuje prilikom kompajliranja, dok se kod dinamičke alokacije dodjeljuje prilikom pokretanja programa (*run-time*). Memorija, koja je dodjeljena nizu prilikom dinamičke alokacije, je na heapu. Heapu se može pristupiti samo preko pokazivača.

`malloc`
`realloc`
`calloc`
`free`

Primjer 5.12 Dinamička alokacija niza.

```
int main(void)
{
    int* arr;
    int n = 10;
```

```
    arr = (int*) malloc(n * sizeof(int));
    unesi_niz(arr, n);
    ispisi_niz(arr, n);
    free(arr);
    return 0;
}
```

Usporedite ovaj primjer s primjerom 5.10.

5.10 Pokazivači kao povratna vrijednost funkcije

Pokazivač može biti povratna vrijednost iz funkcije. U tom slučaju treba obratiti pozornost na ono što predstavlja njegova vrijednost. Funkcija nikada ne bi trebala vratiti vrijednost adrese varijable alocirane na stacku, jer se memorija alocirana na stacku oslobađa prilikom izlaska iz funkcije. Funkcija može vratiti adresu varijable alocirane na heapu. Pritom je za oslobađanje memorije zadužen onaj tko funkciju poziva, te je dobro to naglasiti u dokumentaciji.

Sljedeći će se program kompajlirati uz eventualno upozorenje (*warning*) kompajlera, a možda će čak i ispisati točne vrijednosti. Međutim, takve poruke kompajlera ne treba zanemariti, jer su točne vrijednosti u ispisu posljedica slučaja. U kompleksnijim programima brzo će se dogoditi da je memorija koja je oslobođena nakon izlaska iz funkcije `test_ret` dodijeljena nekoj drugoj funkciji, pri čemu će i vrijednost na adresi koju je funkcija `test_ret` vratila biti pregažena.

Primjer 5.13 Greška: adresa lokalne varijable kao povratna vrijednost funkcije.

```
#include <stdio.h>
int* test_ret(void)
{
    int a = 10;
    return &a; //not OK
}
```

```
int main(void)
{
    int* p = test_ret();
    printf("na adresi %p je %d\n", p, *p);
    return 0;
}
```

Često želimo u funkciji kreirati i vratiti niz. Tada također treba paziti da niz nije alociran na stacku.

Primjer 5.14 Greška: povratna vrijednost je lokalna varijabla.

```
int* arr_ret(void)
{
    int a[5] = {1, 2, 3, 4, 5};
    return a; // not OK
}

int main(void)
{
    int* p = arr_ret();
    int i;
    for (i = 0; i < 5; ++i)
        printf("adresa %p, vrijednost %d\n", p + i, p[i]);
    return 0;
}
```

Ispravan bi način bio alocirati niz na heapu.

Primjer 5.15 Povratna vrijednost je adresa varijable alocirane na heapu.

```c
int* arr_ret_heap(int n)
{
    int i;
    int* a = (int*) malloc(n * sizeof(int));
    if (!a) return NULL;
    for (i = 0; i < n; ++i)
    {
        scanf("%d", &a[i]);
    }
    return a;
}

int main(void)
{
    int i;
    int* p = arr_ret_heap(5);
    if (!p) return 1;
    for (i = 0; i < 5; ++i)
        printf("adresa %p, vrijednost %d\n", p + i, p[i]);
    free(p); /* obavezna dealokacija */
    return 0;
}
```

5.11 Riješeni zadaci

1. Napisati funkciju koja za uneseni pozitivni cijeli broj računa sumu i produkt kvadrata njegovih znamenki.

```
void sp_square_digits(unsigned n, unsigned* psum,
    unsigned* pprod)
{
  unsigned zn;
  *psum = 0;
  *pprod = 1;
  while(n)
  {
    zn = n % 10;
    *psum += zn * zn;
    *pprod *= zn * zn;
    n /= 10;
  }
}

int main(void)
{
    unsigned n = 12345;
    unsigned s, p;
    sp_square_digits(n, &s, &p);
    printf("%u %u\n", s, p);
    return 0;
}
```

2. Napisati funkciju koja računa sumu i umnožak članova niza cijelih brojeva koji su djeljivi sa 3.

```
void calc(int* arr, int n, int* ps, int* pu)
{
  int i;
  *ps = 0, *pu = 1;
  for (i = 0; i < n; i++)
    if (!(arr[i] % 3))
    {
        *ps += arr[i];
```

```
            *pu *= arr[i];
        }
    }

    int main (void)
    {
      int s, u;
      int niz[] = {2, 4, 3, 6, 9, 1, 7};
      calc(niz, sizeof niz/sizeof niz[0], &s, &u);
      printf("suma je %d, umnozak %d\n", s, u);
      return 0;
    }
```

3. Napisati funkciju koja presloži niz brojeva tako da sve nule budu na kraju niza.

```
    void swap(int* a, int* b)
    {
      int t = *a; *a = *b; *b = t;
    }

    void preslozi(int* arr, int n)
    {
      int i, j = n - 1;
      for (i = 0; i < j; i++)
      {
        if (!arr[i])
        {
          while(!arr[j] && j > 0) j--;
          swap (&arr[i], &arr [j]);
          j--;
        }
      }
    }
```

4. Napisati funkciju koja računa sumu recipročnih vrijednosti danog niza.

```
    double sum_rec(double* arr, unsigned n)
    {
      unsigned i;
      double s = 0.0;
```

```
  for (i = 0; i < n; i++)
    s += 1 / arr[i];
  return s;
}

int main(void)
{
  double a[] = {1, 2, 3, 4, 5};
  printf("suma reciprocnih vrijednosti je %lf\n",
      sum_rec(a, sizeof a / sizeof a[0]));
  return 0;
}
```

5. Napisati funkciju koja za niz cijelih brojeva vraća adresu najvećeg neparnog broja u nizu.

```
int* adr_odd(int* arr, int n)
{
  int* p = arr;
  int* end = arr + n;
  int* max_odd;
  while (*p % 2 == 0 && p != end)
      p++; /* preskoci parne na pocetku*/
  max_odd = p;
  while (p != end)
  {
    if (*p % 2 && *p > *max_odd) max_odd = p;
    p++;
  }
  if (max_odd == end) return NULL;
  return max_odd;
}
```

6. Napisati funkciju koja presloži niz cijelih brojeva tako da svi neparni brojevi budu ispred parnih.

```
void rearrange(int* arr, int n)
{
  int *p = arr, *q = arr + n - 1;
  int t;
  while(p < q)
```

```
    {
      while(*p % 2 != 0 && p < q) p++;
      while(*q % 2 == 0 && p < q) q--;
      t = *p; *p = *q; *q = t;
      p++; q--;
    }

}
void print_arr(int* arr, int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d\t", *(arr + i));
}
int main(void)
{
  int a[] = {1, 3, 2, 6, 8};
  rearrange(a, sizeof a / sizeof a[0]);
  print_arr(a, sizeof a / sizeof a[0]);
  return 0;
}
```

7. Napisati funkciju koja računa koordinate točke sjecišta dva pravca koji su zadani sa `k` i `l`.
   - pravac $p_1..y = k_1x + l_1$
   - pravac $p_2..y = k_2x + l_2$

```
void intersect(float k1, float l1, float k2, float l2
    , float* x, float* y)
{
  *x = (l2 - l1) / (k1 - k2);
  *y = k1 * *x + l1;
}
```

8. Napisati funkciju koja kružno zamijeni vrijednosti triju varijabli. Napisati `main` funkciju iz koje se poziva funkcija za zamjenu vrijednosti.
   Primjer:
   `a = 1, b = 2, c = 3`
   Nakon izmjene:

`a = 3, b = 1, c = 2`

```
void swap_three(int* a, int* b, int* c)
{
  int tmp = *a; // tmp = 1
  *a = *c;      // a = 1 => a = 3
  *c = *b;      // b = 2 => b = 1
  *b = tmp;     // c = 3 => c = 2
}
int main(void)
{
  int a, b, c;
  puts("unesi 3 broja");
  scanf("%d %d %d", &a, &b, &c);
  swap_three(&a, &b, &c);
  printf("%d %d %d", a, b, c);
  return 0;
}
```

9. Napisati funkciju koja za niz cijelih brojeva vraća dvije sume: sumu svih elemenata pozitivnog i sumu svih elemenata negativnog predznaka.

```
void sums(int* arr, int n, int* ppos, int* pneg)
{
  int i;
  *ppos = *pneg = 0;
  for (i = 0; i < n; i++)
  {
    if (arr[i] > 0) *ppos += arr[i];
    if (arr[i] < 0) *pneg += arr[i];
  }
}
```

10. Napisati funkciju koja nekom nizu brojeva invertira članove na parnim pozicijama.
    Primjer: Za niz `{2, 1, 3, 5, 7, 9, 4}` promijenjeni niz će biti `{4, 1, 7, 5, 3, 9, 2}`

```
void swap(int*a, int* b)
{
```

```
  int t = *a;
  *a = *b;
  *b = t;
}
void swap_even(int* arr, int n)
{
  int i;
  int k = n&1 ? n - 1 : n - 2; // zadnji parni index
  for (i = 0; i < k/2; i+=2)
    swap(&arr[i], &arr[k - i]);
}
```

11. Napisati funkciju koja računa najveći i najmanji broj po apsolutnoj vrijednosti u nizu realnih brojeva.

```
#include <math.h>
void minmax (float* arr, unsigned n, float* min,
                                     float* max)
{
  int i;
  *min = *max = fabs(arr[0]);
  for (i = 1; i < n; i++)
  {
    if (arr[i] < *min) *min = fabs(arr[i]);
    if (arr[i] > *max) *max = fabs(arr[i]);
  }
}
int main(void)
{
  float min, max;
  float a[] = {-34, 25, -12, -11, 2, 0, 3};
  minmax(a, sizeof a/sizeof a[0], &min, &max);
  printf("%f %f\n", min, max);
  return 0;
}
```

12. Napisati funkciju za unos pozitivnih cijelih brojeva. Korisnik unosi brojeve sve dok ne unese nulu. Funkcija alocira u startu memoriju za 10 elemenata. Svaki put kad veličina niza dosegne alociranu veličinu, alocira se duplo.

```
void print_niz(unsigned* arr, unsigned n)
{
  unsigned i;
  for(i = 0; i < n; i++)
    printf("%u\t",arr[i]);
  puts("");
}

unsigned input_elems(unsigned** arr, unsigned n)
{
  unsigned i = 0, a;
  *arr = (unsigned*) malloc(n * sizeof (int));
  while(1)
  {
    scanf("%d", &a);
    if(!a) break;
    if (i == n)
    {
      n *= 2;
      *arr = (unsigned*)realloc(*arr, n * sizeof (int)
          );
    }
    (*arr)[i++] = a;

  };
  n = i;
  return i;

}
int main(void)
{
  unsigned n = 3, k;
  unsigned* a;
  k = input_elems(&a, n);
  print_niz(a, k);
  free(a);
}
```

13. Napisati funkciju koja računa realna rješenja kvadratne jednadžbe. (Nije dovoljno ispisati rješenja!!).

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

int calculate(double a, double b, double c, double*
    x1, double* x2)
{
  double d = b * b - 4 * a * c;
  if (d < 0) return 0;
  else if (d == 0)
    *x1 = *x2 = -b / (2 * a);
  else
  {
    *x1 = (-b + sqrt(d)) / (2 * a);
    *x2 = (-b - sqrt(d)) / (2 * a);
  }
  return 1;
}
int main(void)
{
  double a, b, c, x1, x2;
  printf("Unesi a, b i c\n");
  scanf("%lf %lf %lf", &a, &b, &c);
  if(calculate(a, b, c, &x1, &x2))
  {
    if (x1 != x2)
      printf("Rjesenja su: %lf i %lf\n", x1, x2);
    else
      printf("Samo jedno rjesenje %lf\n", x1);
  }
  else
    printf("Nema rjesenja u skupu realnih brojeva");
  return 0;
}
```

5.12 Zadaci za vježbanje

1. Napisati funkciju koja vraća sumu i produkt znamenki cijelog broja.
2. Napisati funkciju koja vraća sumu i produkt kvadrata brojeva do `n`. Broj `n` je parametar funkcije.
   Primjer: Za `n = 4`, funkcija računa sumu $1^2 + 2^2 + 3^2 + 4^2$, te produkt $1^2 \cdot 2^2 \cdot 3^2 \cdot 4^2$
3. Napisati funkciju koja za niz brojeva vraća najveći i najmanji broj u nizu.
4. Napisati funkciju koja za niz brojeva vraća vrijednost najvećeg člana i njegov indeks.
5. Napisati funkciju koja vraća prosječne vrijednosti pozitivnih i prosječne vrijednosti negativnih elemenata u nizu.
6. Napisati funkciju koja za dva niza cijelih brojeva vraća onaj koji ima veću prosječnu vrijednost svojih elemenata. Nizovi ne moraju biti jednake duljine.
7. Napisati funkciju koja modificira niz cijelih brojeva tako da svaki parni broj u njemu prepolovi, a neparni udvostruči.
8. Napisati funkciju koja za niz cijelih brojeva vraća adresu najvećeg neparnog broja u nizu.
9. Napisati funkciju koja za broj `n` vraća broj nula i broj jedinica u bitovnom zapisu broja `n`.
10. Napisati funkciju koja za neki broj `n` vraća sumu i produkt prve i zadnje znamenke u broju `n`.
11. Napisati funkciju koja iz niza brojeva vraća dva najmanja elementa.
12. Napisati funkciju koja za dva broja `a` i `b` vraća njihovu sumu kvadrata $a^2 + b^2$ i kvadrat sume $(a + b)^2$.
13. Napisati funkciju koja vrijednosti svojih parametara udvostruči. Funkcija ima dva parametra.
14. Napisati funkciju koja za stranicu kocke `a` računa oplošje i volumen kocke.
15. Napisati funkciju koja izgenerira n brojeva između `a` i `b`. Funkcija vraća broj koji je najviše puta izgeneriran, te informaciju koliko je puta bio izgeneriran.

Broj `n` mora biti veći od `100`, a interval ne veći od 10. Ako ti uvjeti nisu zadovoljeni, funkcija vraća `0`.

Primjer: `a = 2, b = 7` je ispravan interval, dok `a = 10, b = 60` nije.

16. Napisati funkciju koja vraća sumu i produkt recipročnih vrijednosti brojeva do `n`. Broj `n` je parametar funkcije.

    Primjer: Za `n = 4`, funkcija računa sumu $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4}$, te produkt $1 \cdot \frac{1}{2} \cdot \frac{1}{3} \cdot \frac{1}{4}$
17. Napisati funkciju koja za niz brojeva vraća najveći i najmanji prosti broj (ako postoji). Ako ne postoji ni jedan prosti broj u nizu, funkcija vraća nule.
18. Napisati funkciju koja računa sumu i produkt recipročnih vrijednosti elemenata niza cijelih brojeva.
19. Napisati funkciju koja preuredi niz brojeva tako da svi brojevi koji nisu djeljivi sa 6 dođu na kraj.
20. Napisati funkciju koja za niz znakova vraća poziciju prvog malog slova u nizu i prvog velikog slova u nizu.
21. Napisati funkciju koja za niz brojeva vraća sumu onih koji su djeljivi sa 3 i produkt onih koji su djeljivi sa 6.
22. Napisati funkciju koja za niz brojeva vraća broj koji se načešće ponavlja u nizu i broj ponavljanja. Ako se više brojeva ponavlja jednaki broj puta, vratiti vrijednosti za jedan od njih.
23. Napisati funkciju koja presloži niz brojeva tako da sve nule budu na kraju niza.
24. Napisati funkciju koja presloži niz brojeva tako da sve nule budu na početku niza.
25. Napisati funkciju koja će vratiti broj elemenata većih i broj elemenata manjih od danog broja u nekom nizu.
26. Napisati funkciju koja vraća prosječne vrijednosti pozitivnih i prosječne vrijednosti negativnih elemenata u nizu.
27. Napisati funkciju koja u nizu cijelih brojeva računa (ne ispisuje) koliko je brojeva djeljivo sa 3 te koliko je brojeva djeljivo sa 5.
28. Napisati funkciju koja za niz cijelih brojeva vraća adresu najvećeg negativnog broja u nizu.

29. Napisati funkciju koja za niz cijelih brojeva vraća redni broj najvećeg neparnog i najmanjeg parnog broja u nizu.
30. Napisati funkciju koja rotira niz brojeva udesno tako da zadnjih `n` brojeva prebaci na prvo mjesto, a sve ostale pomakne `n` mjesta udesno.
31. Napisati funkciju koja za niz cijelih brojeva vraća adresu predzadnjeg člana u nizu.
32. Napisati funkciju koja iz niza cijelih brojeva kreira drugi u kojem su samo oni brojevi koji se u originalnom nizu pojavljuju više od jednog puta.
    Primjer: za dani niz `{1, 2, 3, 2, 3, 5, 4 }` novi niz je `{2, 3}`.
33. Napisati funkciju koja za niz realnih brojeva vraća adresu najvećeg po apsolutnoj vrijednosti broja u nizu.
34. Napisati funkciju koja za niz brojeva vraća najveći i najmanji element.
35. Napisati funkciju koja računa sumu i produkt kubova negativnih brojeva u nizu cijelih brojeva.
36. Napisati funkciju koja vraća indeks i adresu najmanjeg elementa u nizu brojeva.
37. Napisati funkciju koja kreira i vraća niz slučajno izgeneriranih cijelih brojeva između `a` i `b`. Brojevi `a` i `b` su parametri funkcije, a veličinu niza definira korisnik.
    Napomena: Funkcija treba vratiti niz i njegovu veličinu.
38. Napisati funkciju koja za niz brojeva vraća drugi najveći i drugi najmanji.
    Primjer: Za niz `{2, 5, 4, 7, 6, 9}` funkcija vraća `7` i `4`.
39. Napisati funkciju koja presloži niz pozitivnih cijelih brojeva tako da na početu budu svi parni, a na kraju svi neparni.
    Primjer: Niz `{2, 4, 5, 7, 4, 8, 3}` treba presložiti u `{2, 4, 4, 8, 5, 7, 3}`.
40. Napisati funkciju koja izgenerira niz od `n` brojeva između `a` i `b`, najveći i najmanji među njima pretvori u stringove, te vrati broj znamenki u njima.
    Napomena: `n`, `a` i `b` su parametri funkcije.
41. Napisati funkciju koja za dva niza brojeva jednake veličine kreira treći u kojem je svaki element kvocijent odgovarajućih elemenata iz prva dva niza.

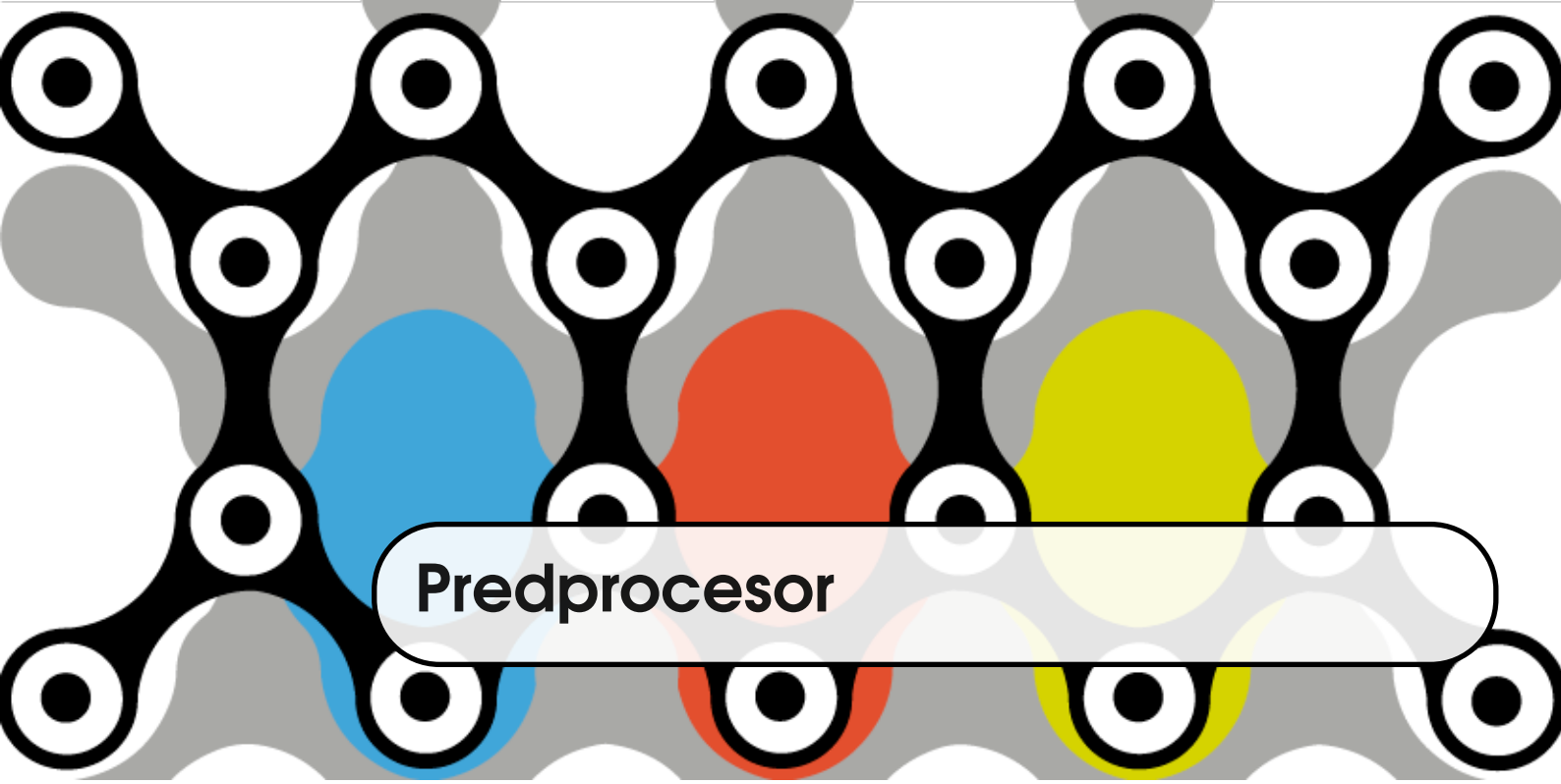
Primjer: Za nizove `{1, 1, 2, 3, 4}` i `{2, 4, 1, 1, 5}` rezultirajući niz je `{0.5, 0.25, 2, 3, 0.8}`.

42. Napisati funkciju koja spaja dva niza brojeva u jedan niz u kojemu su svi neparni ispred parnih. Funkcija vraća novonastali niz.
    Primjer: Za niz `A = {1, 2, 3, 4, 5, 6}` i niz `B = {10, 9, 8, 7}`, rezultantski niz je `C = {1, 3, 5, 9, 7, 2, 4, 6, 10, 8}`.
43. Napisati funkciju koja iz dva niza brojeva kreira treći tako da je i-ti element u novom nizu aritmetička sredina i-tih članova početnih nizova. Ako je jedan niz dulji, novi niz je nadopunjen elementima duljeg niza.
44. Napisati funkciju koja će iz dva niza brojeva `a` i `b` kreirati treći tako da je i-ti član novog niza veći broj odabran između dva i-ta člana originalnih nizova `a` i `b`. Memoriju za novi niz alocirati dinamički. Duljina novonastalog niza treba biti jednaka duljini kraćeg niza.
    Primjer: za niz `a = {2.1, 2.2, 3.5, 1, 2, 3}` i niz `b = {1.9, 2.3, 1.8}` novi niz je `{2.1, 2.3, 3.5}`.
45. Napisati funkciju koja od dva niza brojeva kreira treći tako da su u rezultirajućem nizu naizmjenice elementi jednog pa drugog niza. Ako su nizovi različite duljine, kraj novog niza su svi preostali elementi iz duljeg niza. Memoriju za novi niz alocirati dinamički.
    Napomena: Funkcija treba vratiti novi niz, te njegovu duljinu.
    Primjer: Za nizove `{1, 2, 3, 4, 5}` i `{11, 22, 33, 44, 55, 66, 77, 88, 99}` rezultirajući niz je `{1, 11, 2, 22, 3, 33, 4, 44, 5, 55, 66, 77, 88, 99}`.
46. Napisati funkciju koja za niz cijelih brojeva vraća dvije sume: sumu svih elemenata pozitivnog i sumu svih elemenata negativnog predznaka.
47. Napisati funkciju koja za niz cijelih brojeva vraća zbroj i produkt svih elemenata niza koji su djeljivi sa 3 i manji od 30.
48. Napisati funkciju koja za niz `float` brojeva računa (vraća) sumu i produkt brojeva čiji je drugi korijen manji od `1.5`.
49. Napisati funkciju koja za niz cijelih brojeva računa (vraća) zbroj apsolutnih vrijednosti i zbroj brojeva u nizu manjih od 25.

50. Napisati funkciju koja, za niz brojeva i broj `k`, kreira i vraća novi niz u kojem su samo oni elementi koji su manji od `k`.
    Napomena: Funkcija treba vratiti i broj elemenata u novom nizu.
51. Napisati funkciju koja iz jednog niza kreira i vraća drugi u kojem je svaki element drugi korijen pripadajućeg elementa iz prvog niza.
    Primjer: Za niz `{4, 2, 16, 4}` povratni niz je `{2, 1.41, 4, 2}`.
    Napomena: Funkcija treba vratiti i broj elemenata novonastalog niza.
52. Napisati funkciju koja za dva niza cijelih brojeva vraća niz u kojem je svaki element suma kvadrata pripadajućih elemenata nizova.
    Primjer: Za nizove `{1, 2, 3, 7, 9}` i `{1.1, 3, 4, 1, 1}` rezultat je niz `{2.21, 13, 25, 50, 82}`.
    Napomena: Funkcija treba vratiti i broj elemenata novonastalog niza.
53. Napisati funkciju koja za niz pozitivnih cijelih brojeva vraća sumu svih neparnih elemenata koji su manji od nekog pozitivnog cijelog broja *k*.
54. Napisati funkciju koja će za niz decimalnih brojeva izračunati (i vratiti) broj brojeva većih od 3 i broj brojeva manjih od 5.
55. Napisati funkciju koja za dva niza brojeva ispituje je li jedan podniz drugoga.
    Primjer: Niz `{4, 6, 7}` je podniz niza `{2, 3, 4, 6, 7, 9}`.
56. Napisati funkciju koja nekom nizu brojeva invertira članove na parnim pozicijama.
    Primjer: Za niz `{2, 1, 3, 5, 7, 9, 4}` promijenjeni niz će biti `{4, 1, 7, 5, 3, 9, 2}`
57. Napisati funkciju koja za dva niza cijelih brojeva vraća treći koji nastane spajanjem ta dva niza.
    Napomena: Funkcija mora vratiti i duljinu niza.
58. Napisati funkciju koja vraća niz *emirp* brojeva, tj. prostih brojeva koji su i čitani naopačke prosti brojevi.
    Primjer: 37 i 73.
59. Napisati funkciju koja za dva niza brojeva, ne nužno jednake duljine, vraća treći niz u kojem su svi brojevi iz prva dva niza, ali tako da se nijedan broj

ne ponavlja. Funkcija vraća novonastali niz i njegovu veličinu.

60. Napisati funkciju koja za niz brojeva vraća prosječnu vrijednost brojeva u nizu koji su manji od 42 i maximalnu vrijednost brojeva koji su veći od 42.
61. Napisati funkciju koja iz niza `char` vrijednosti izbaci sve znakove koji nisu znakovi alfabeta, te promijeni vrijednost parametra koji označava duljinu niza.
62. Napisati funkciju koja za dani `n` vraća niz cijelih brojeva veličine `n` čije članove unosi korisnik. Broj `n` je parametar funkcije. Memoriju za novi niz alocirati dinamički.
63. Napisati funkciju koja iz dva niza `int` vrijednosti `a` i `b` kreira i vraća niz u kojem je svaki član niza suma članova na odgovarajućim mjestima u nizovima `a` i `b`. Nizovi `a` i `b` su jednake veličine. Memoriju za novi niz alocirati dinamički.
64. Napisati funkciju koja vraća adrese najmanjeg i najvećeg elementa u nizu brojeva.
65. Napisati funkciju koja mijenja niz brojeva tako da na odgovarajućem mjestu u nizu umetne `n` jedinica. Odgovarajuće mjesto u nizu i broj jedinica su parametri funkcije. Memoriju za niz alocirati dinamički.
66. Napisati funkciju koja iz dva niza brojeva kreira treći u kojem su svi članovi prvog i drugog niza bez duplikata. Memoriju za niz alocirati dinamički.
67. Napisati funkciju koja alocira niz brojeva duljine `n` u kojem su svi članovi 0 (koristiti se `calloc()` funkcijom). Funkcija napuni parna mjesta u nizu brojem nula do tog mjesta, te vrati novi niz.
    Primjer: za `n = 10` novi niz je `{0, 0, 2, 0, 3, 0, 4, 0, 5, 0}`.
68. Napisati funkciju koja niz brojeva poveća za `k` mjesta, gdje je `k` parametar funkcije, te tih novih `k` mjesta napuni nulama. Koristiti se funkcijom `realloc()`.
    Napomena: Funkcija `realloc()` može promijeniti lokaciju (adresu) niza.



Predprocesor

Predprocesor je program koji se izvodi prije faze kompajliranja, a transformira izvorni kôd [15]. Naziva se i *macro* procesor, jer omogućuje definiranje makroa tj. pravila zamjene. Pravila zamjene određena su predprocesorskim direktivama koje počinju znakom `'#'` iza kojeg slijedi ime predprocesorske direktive.

PREDPROCESOR

MACRO

U ovom ćemo poglavlju dati kratki pregled predprocesorskih direktiva koje služe za umetanje header datoteka, definiranje pravila zamjene i uvjetno kompajliranje.

6.1 Datoteke zaglavlja

Datoteke zaglavlja (engl. *header*) su datoteke u kojima se nalaze prototipovi funkcija (vidi 2.5). Obično imaju ekstenziju `'.h'`. Koriste se kada u kôdu pozivamo funkciju koja u trenutku kompajliranja još nije definirana, tj. kompajler još nije došao do njene definicije. Tada prototipom funkcije omogućavamo kompajleru da provjeri ispravnost poziva funkcije. U header datoteci ne moraju se nalaziti samo prototipovi funkcija, već i predprocesorske direktive, ali i bilo kakav C kôd. Međutim, pisanje kôda unutar header datoteka nije uobičajeno.

DATOTEKE ZAGLAVLJA

Predprocesorska direktiva za umetanje header datoteke u izvorni kôd je `#include`. Koristi se za umetanje sistemskih datoteka, ali i za umetanje prototipa funkcija čije se definicije nalaze u drugoj izvornoj datoteci. Na taj način možemo pisati složenije programe koji se sastoje od više datoteka i koji pozivaju funkcije napisane i zapakirane unutar neke tuđe biblioteke funkcija.

`#include`

Umetanje header datoteke omogućeno je kroz dva načina upotrebe predprocesorske direktive `#include`:

- `#include <ime_datoteke>`
- `#include "ime_datoteke"`

Ako je ime datoteke dano unutar špicastih zagrada, datoteka je prvo traži u posebno zadanim direktorijima. To su direktoriji zadani unutar varijable okoline (engl. *environment variable*) `PATH` ili dani opcijom `'-I'` kompajlera.

Ako je ime datoteke navedeno unutar dvostrukih navodnika, datoteka se traži u tekućem direktoriju (onom u kojem se nalazi sama datoteka izvornog kôda) te ako nije pronađena traži se unutar posebno zadanih direktorija. Samo ime datoteke može sadržavati i relativnu ili apsolutnu putanju do datoteka s tim da se za razdvajanje dijelova putanje upotrebljava znak `'/'`.

Primjer 6.1 Organizacija izvornog kôda pomoću header datoteka i predprocesorskih direktiva.

array.h

```
#include <stdio.h>

void input_arr(int* arr, int n);
void print_arr(int* arr, int n);
```

array.c

```
#include "array.h"

void input_arr(int* arr, int n)
{
    int i;
```

```
    for (i = 0; i < n; ++i)
        scanf("%d", arr + i);
}
void print_arr(int* arr, int n)
{
    int i;
    for (i = 0; i < n; ++i)
        printf("%d\t", arr[i]);
}
```

main.c

```
#include <stdlib.h>
#include "array.h"

int main(void)
{
    int n, *a;
    printf("n = ");
    scanf("%d", &n);
    a = (int*) malloc(n * sizeof(int));
    if (!a)
        return 1;
    input_arr(a, n);
    print_arr(a, n);
    free(a);
    return 0;
}
```

Rezultat predprocesiranja datoteke `array.c` je datoteka koja na početku ima kompletan izvorni kôd datoteke `stdio.h`, potom kôd datoteke `array.h` i na samom kraju sâm kôd datoteke `array.c`.

6.2 Pravila zamjene

Pravila zamjene definiraju se predprocesorskom direktivom `#define`. Dvije su varijante: jednostavna zamjena (jednostavni makro) i zamjena s argumentima (makro s argumentima).

PRAVILA ZAMJENE
`#define`
MAKRO

- `#define IME zamjena`
- `#define IME(p1,p2,...) zamjena`

Pravila zamjene se mogu poništiti direktivom `#undef IME`. Nakon toga se za `IME` može definirati novo pravilo zamjene.

Jednostavna zamjena Kod jednostavne zamjene predprocesor zamjenjuje ime makroa sa znakovima navedenim nakon imena makroa. Pritom se zamjena ne vrši unutar znakovnih konstanti u izvornom kôdu. Iako ne postoji nikakvo ograničenje na to što se može pisati unutar makroa, najčešće se koristi za definiranje konstanti, pogotovo ako se one nalaze na nekoliko mjesta u kôdu. Nadalje, običaj je da se imena makroa pišu velikim tiskanim slovima, što omogućava lakše snalaženje u kôdu.

Primjer 6.2 Upotreba predprocesorske direktive `#define` za definiranje broja π.

```
#define PI 3.141592

double area(double r)
{
    return r * r * PI;
}

double circumference(double r)
{
    return 2 * r * PI;
}
```

Predprocesirani kôd izgleda ovako:

```


double area(double r)
{
    return r * r * 3.141592;
}
```

```
double circumference(double r)
{
    return 2 * r * 3.141592;
}
```

Tijelo makroa završava krajem linije u kojoj se nalazi `#define`, a ako je potrebno može se napisati u više redova upotrebljavajući oznaku `'\'` za novi red.

U jednostavnoj zamjeni mogu se koristiti i zagrade. U sljedećem primjeru uočimo da između imena makroa i zagrada postoji razmak.

Primjer 6.3 Jednostavni višelinijski makro.

```
#define PI 3.141592
#define PI3 (PI \
          * PI \
          * PI)
int main(void)
{
    printf("%lf\n", PI3);
    return 0;
}
```

Rezultat predprocesora je:

```
int main()
{
    printf("%lf\n", (3.141592 * 3.141592 * 3.141592));
    return 0;
}
```

Zamjena s argumentima Jednostavna zamjena se koristi kada zamjenjujemo svaki put isti dio kôda. Drugi način upotrebe makroa je zamjena s argumentima,

koji su dijelovi kôda koje navodimo prilikom upotrebe makroa. Ovakva zamjena s argumentima naziva se *function-like macro* zbog sintakse koja podsjeća na poziv funkcije.

Važno je uočiti da prilikom definiranja makroa s argumentima, otvorena zagrada mora biti odmah iza imena makroa bez praznina.

Primjer 6.4 Makro s argumentima koji računa duljinu niza koristeći `sizeof` operator. Datoteka `array.h` je iz primjera 6.1.

```
#include "array.h"
#define CNT(x) (sizeof (x) / sizeof (x[0]))
int main(void)
{
    int arr[] = {10, 20, 30, 40};
    int i;

    print_arr(arr, CNT(arr));
    return 0;
}
```

Predprocesor pretvara ovaj kôd u sljedeći kôd (zanemarit ćemo `stdio.h` datoteku):

```
void input_arr(int* arr, int n);
void print_arr(int* arr, int n);

int main(void)
{
    int arr[] = {10, 20, 30, 40};
    int i;

    print_arr(arr, (sizeof (arr) / sizeof (arr[0])));
    return 0;
}
```

Prilikom rada s predprocesorskim pravilima zamjene treba posebno voditi računa o tome da predprocesor doslovno zamjenjuje ime makroa i eventualne argumente sa znakovima navedenim nakon imena, zamjenjujući samo argumente navedene u zagradama s argumentima danim u "pozivu" makroa. To je najčešći izvor grešaka pri radu s makroima, jer pri definiranju makroa ne znamo s kojim argumentima će makro biti pozvan. U sljedećem primjeru ćemo vidjeti da neopreznom upotrebom pravila zamjene možemo previdjeti prioritet operatora.

Primjer 6.5 Neupotrebom zagrade prioretet operatora se mijenja.

```
#define PRODUCT(a, b, c) a * b * c
int main(void)
{
    int x = 2;
    printf("%d\n", PRODUCT(x, x + 1, x + 2));
    return 0;
}
```

Očekivani ispisani rezultat (možda) bi bio 24, međutim predprocesor kao izlaz daje sljedeći kôd:

```
int main(void)
{
    int x = 2;
    printf("%d\n", x * x + 1 * x + 2);
    return 0;
}
```

Rezultat ispisa je 8 zbog prioriteta operatora množenja nad operatorom zbrajanja.

Dakle, dobra je praksa staviti svaki pojedini član makro izraza u zagrade, te i sam cijeli izraz omeđiti zagradama.

6.3 Uvjetno uključivanje

Uvjetno uključivanje dijelova kôda je direktiva koja omogućuje predprocesoru da uključi ili isključi dijelove kôda iz postupka kompajliranja. Predprocesor može testirati konstantne cjelobrojne aritmetičke izraze i/ili ispitati je li neki makro definiran. Direktive koje razrješuju takve dijelove su `#if`, `#ifdef` i `#ifndef` koje možemo upotrebljavati u kombinaciji sa `#elif` i `#else` i koje završavaju sa `#endif`.

`#if`
`#ifdef`
`#ifndef`
`#elif`
`#else`
`#endif`

Primjer 6.6 Uključivanje dijela kôda ovisno o vrijednosti predprocesorskog makroa `ZAD`.

```
#include <stdio.h>
#define ZAD 2

int main(void)
{
#if ZAD == 1
    printf("1.\n");
#elif ZAD == 2
    printf("2.\n");
#else
    printf("...\n");
#endif
    return 0;
}
```

Često se predprocesorska direktiva `#if` koristi kako bi se zakomentirao ili odkomentirao dio kôda.

Primjer 6.7 Isključivanje dijela kôda.

```
#if 0
int main(void)
{
  printf("test");
  return 0;
}
#endif
int main(void)
{
  printf("work");
  return 0;
}
```

Predprocesorske direktive `#ifdef` i `#ifndef` ispituju je li neki predprocesorski makro definiran ili ne.

Primjer 6.8 Onemogućavanje višestrukog uključivanja header datoteke upotrebljavajući *wrapper* `#ifndef`.

```
/* file array.h */
#ifndef ARRAY_H
#define ARRAY_H

void input_arr(int* arr, int n);
void print_arr(int* arr, int n);
#endif  /* !ARRAY_H*/
```

6.4 Zadaci za vježbanje

1. U kakav izraz će se razviti predprocesorski makro? Uočititi problem s višestrukim postfiksnim inkrementom.

```
#define ABS(x) ((x) > 0 ? (x) : -(x))
int main()
{
    unsigned x = -1, y;
    y = ABS(x++);
    printf("y = %u, x = %u\n", y, x);
    return 0;
}
```

2. Na koji način prepraviti predprocesorski makro kako bi vrijednost varijable y bila očekivana tj. 2.

```
#define ABS(x) (x > 0 ? x : -x)
int main()
{
    unsigned x = -1, y = -1;
    y = ABS(x + y);
    printf("y = %d, x = %d\n", y, x);
    return 0;

}
```

3. Što će se ispisati i zašto?

```
#define SUM(a,b) a + b
int main()
{
    int x = 3;
    char y = 48;
    y = SUM(x|y, sizeof x / sizeof y);
    printf("y = %d, x = %d\n", y, x);
    return 0;
}
```

4. Što će se ispisati i zašto?

```
#define MAX(a, b) a > b ? a : b
#include <stdio.h>
int main(void)
{
```

```
    float a = 2., b = 1., max1 = MAX(a, b), max2 = MAX(++a, b
        ++);
    printf("Razlika maxova %lf i %lf je %lf\n", max1, max2,
        max1 - max2);
    return 0;
}
```

5. Što će se ispisati i zašto?

```
#define OSTATAK(a, b) a % b
int main()
{
    int a = 8, b = 3, ost1 = OSTATAK(a,b), ost2 = OSTATAK(++a,
        b++);
    printf("%d %d\n", ost1, ost2);
    printf("%d\n", OSTATAK(a + b, a + b));
    return 0;
}
```

6. Što će se ispisati i zašto? Prepravite predprocesorski makro kako bi se ispisao očekivani rezultat.

```
#define fraq(x, y) x / y

int main(void)
{
    float x = 4, y = 2, z = 5;
    printf("%f\n", fraq((x+y), y+z));
    return 0;
}
```

7. Što će se ispisati i zašto? Prepravite predprocesorski makro kako bi se ispisao očekivani rezultat.

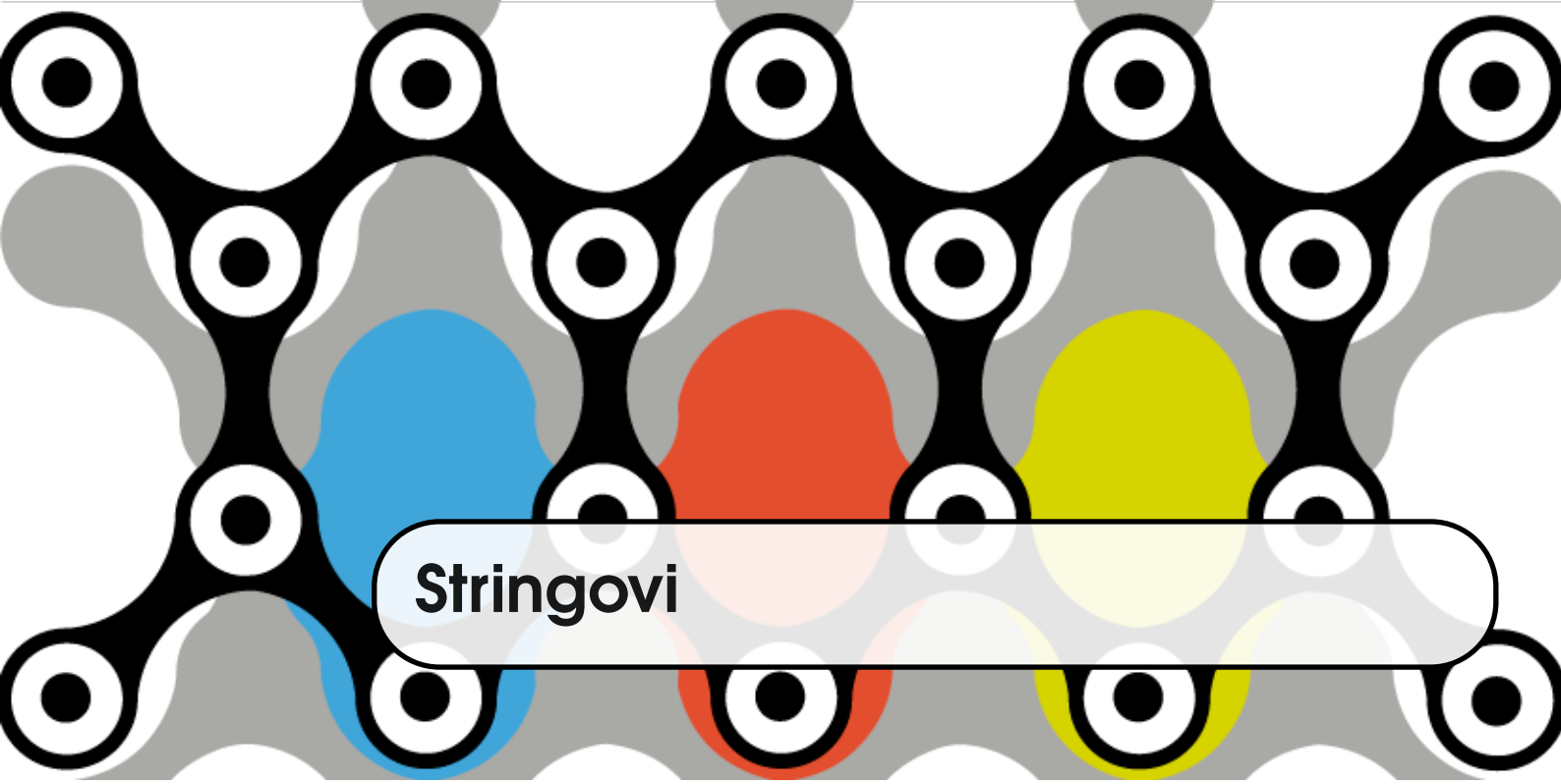
```
#define KUB(a) a * a * a
int zad_2()
{
    int x = 2, y = 3;
    printf("%d\n", KUB(x) + KUB(y));
    printf("%d\n", KUB(x + y));
    return 0;
}
```

8. Što ispisuje sljedeći program?

```
#define f(a,b) a+b
```

```
#define g(a)  -a

int main()
{
    printf("%d\n",g(f(1,2)));
    return 0;
}
```



Stringovi

U programskom jeziku C ne postoji ugrađeni (engl. *built-in*) tip string. String se sastoji od niza znakova (`char`) koji završava nul-terminatorom. Nul-terminator je znak '\0' čija je brojčana vrijednost `0`. Prilikom rada sa stringovima koriste se svojstva nizova i pokazivača, ali i standardna biblioteka koja pomoću funkcija čije se zaglavlje nalazi u `string.h` olakšava manipulaciju stringom. Ipak, programer mora voditi računa o memoriji te o načinu na koji funkcije standardne biblioteke obavljaju se operacije sa stringom.

NUL-TERMINATOR

Primjer 7.1 Deklaracija stringa.

```
int main(void)
{
  char str[4];
  str[0] = 'P';
  str[1] = 'M';
  str[2] = 'A';
  str[3] = '\0';
  return 0;
}
```

STRING LITERAL

Naravno, za očekivati je da stringu nećemo pridjeljivati vrijednosti kao u primjeru 7.1. String literal u C-u je niz znakova koji završavaju nul terminatorom i omeđeni su dvostrukim navodnicima. String literal ima životni vijek jednak životnom vijeku programa (*static storage duration*). Primjer 7.1 ljepše bi mogli napisati na sljedeći način.

Primjer 7.2 Deklaracija i inicijalizacija stringa.

```
int main(void)
{
    char str[] = "PMA";
    str[1] = 'D';       /* OK */
    return 0;
}
```

Prilikom inicijalizacije string literal kopiran je na uzastopne lokacije u memoriji koje počinju na adresi danoj sa `str`. Uočimo u primjeru 7.2 da kod istovremene deklaracije i inicijalizacije string nije potrebno navesti veličinu niza. Za niz `str` se alocira 4 bajta: za 3 znaka i nul terminator. Ako pretpostavljamo da će string rasti, možemo alocirati veći prostor za njega. Nadalje, ako ne inicijaliziramo string prilikom deklaracije, ne možemo mu naknadno pridružiti vrijednost služeći se operatorom =.

Primjer 7.3 Neispravno pridruživanje vrijednosti.

```
int main(void)
{
    char str[4];
    str = "PMA";       /* not OK */
    return 0;
}
```

Sjetimo se da ime niza možemo poistovjetiti s pokazivačkom konstantom na početak niza. Pokazivačkoj konstanti ne možemo promijeniti vrijednost. Nadalje, promotrimo sljedeći primjer.

Primjer 7.4 Promjena vrijednosti string literala.

```
int main(void)
{
    char* str = "PMA";
    str[1] = 'D';       /* not OK */
    return 0;
}
```

Iako se čini da su primjeri 7.2 i 7.4 slični, u drugom slučaju se pokazivaču `str` pridružuje adresa string literala `"PMA"`. Kako se literali ne mogu mijenjati, ovakav kôd će izazvati grešku. Dakle, string se može deklarirati i inicijalizirati pomoću pokazivača kao u primjeru 7.4, ali se tada ne mogu mijenjati njegove vrijednosti.

Samo ime stringa, kao i ime bilo kojeg niza, predstavlja konstantni pokazivač na prvi član niza. Samim tim vrijednost mu se ne može mijenjati.

Primjer 7.5 Ime stringa kao `const` pokazivač.

```
int main(void)
{
    char str[] = "PMA";
    str++;       /* not OK */
    return 0;
}
```

S druge strane, pokazivač pomoću kojeg je string deklariran i inicijaliziran u

primjeru 7.4 može se mijenjati.

Primjer 7.6 Pokazivač na string može promijeniti vrijednost.

```
int main(void)
{
    char* str = "PMA";
    str++;       /* OK */
    return 0;
}
```

Kako je string ustvari niz, alokacija se može napraviti na stacku i na heapu kao i kod bilo kojeg niza. Alokacija stringa na heapu izvodi se funkcijama `malloc`, `calloc` i `realloc`.

7.1 Funkcije standardne biblioteke za rad sa stringovima

Za rad sa stringovima koriste se funkcije standardne biblioteke, čiji se prototipovi nalaze u `string.h` datoteci. Za unos/ispis koriste se funkcije iz `stdio.h`.

Duljina stringa

```
size_t strlen(const char* str);
```

`strlen` Funkcija `strlen` računa duljinu stringa. Tip `size_t` je ustvari `unsigned` tip. Ključnom riječi `const` funkcija jamči da se poslani string neće mijenjati.

Funkcija vraća duljinu stringa, ne računajući nul-terminator. Npr. za string `str = "PMA"` funkcija će vratiti 3.

Primjer 7.7 Ispis stringa - duži način.

```
#include <string.h>
int main(void)
{
    char str[] = "PMA";
    size_t i;
    for (i = 0; i < strlen(str); ++i)
        printf("%c", str[i]);
    return 0;
}
```

Unos i ispis

```
puts(const char* str);
char* gets(char* str);
```

Funkcija `puts` koristi se za ispis stringa. Ispis počinje od znaka na adresi `str` i završava nailaskom na nul-terminator. Funkcija `puts` dodaje oznaku za novi red na kraj ispisa.

`puts`

Primjer 7.8 Ispis stringa.

```
#include <string.h>
int main(void)
{
    char str[] = "PMA";
    puts(str);
    return 0;
}
```

gets

Funkcija gets čita znakove sa standardnog ulaza i zapisuje ih na adresu str. Oznaku za novi red pretvara u nul-terminator.

Ova funkcija se smatra nesigurnom jer ne vrši provjeru je li alocirano dovoljno mjesta u memoriji za string, pa se može dogoditi da korisnik unosi više znakova nego što je alocirano. Npr. u programu je deklariran niz od 15 znakova, a korisnik unosi 20. U tom slučaju nul-terminator neće biti zapisan kao 15-ti znak, pa string neće biti ispravan. Štoviše, vrijednosti na memorijskim lokacijama koje se nalaze neposredno nakon alociranog niza na steku bit će "pregažene" unesenim vrijednostima. To svojstvo zove se prekoračenje bafera (*engl. buffer overflow*). To može biti potencijalno opasno za nizove alocirane na steku gdje nove vrijednosti mogu pregaziti vrijednosti povratne adrese neke funkcije ili funkcijski pokazivač [14]. Napadi povezani s tim svojstvom zovu se *stack smashing*.

BUFFER OVERFLOW

Kopiranje stringa

```
char* strcpy(char* strDestination, const char* strSource);
```

strcpy

Funkcija strcpy kopira stringove. Kopiranje stringova ne može se realizirati operatorom =, jer bi takvo kopiranje bilo kopiranje pokazivača.

strDestination je pokazivač koji pokazuje na dio memorije alociran za kopiju, a strSource je string koji se kopira. Povratna vrijednost funkcije je strDestination.

Primjer 7.9 Kopiranje stringa.

```
#include <string.h>
#include <stdio.h>
int main(void)
{
    char a[] = "Prvi string";
    char b[1024];
    strcpy(b, a);
```

```
    puts(b);
    return 0;
}
```

Važno je alocirati dovoljno memorijskog prostora za string koji će se kopirati na `strDestination`.

Dodavanje jednog stringa na kraj drugog stringa (konkatenacija stringova)

```
char* strcat(char* strDestination, const char* strSource);
```

Funkcija `strcat` kopira string `strSource` na kraj stringa `strDestination`. `strcat`
Moguća implementacija funkcije dana je u sljedećem primjeru.

Primjer 7.10 Konkatenacija stringova.

```
char* string_cat(char* dest, const char* src)
{
  char* p = dest;
  char const* s = src;

  while(*p) p++;
  /* p sada pokazuje na kraj prvog stringa */

  while (*s)
    *p++ = *s++;

  *p = '\0';
  return dest;
}
```

Uspoređivanje stringova

```
int strcmp(const char* string1, const char* string2);
```

strcmp

Funkcija `strcmp` uspoređuje leksikografski stringove koji počinju na memorijskim lokacijama `string1` i `string2`. Ako je `string1` leksikografski ispred `string2` funkcija vraća negativan broj, ako je iza vraća pozitivan broj, a ako su stringovi jednaki funkcija vraća `0`.

7.2 Riješeni zadaci

1. Napisati funkciju koja provjerava je li uneseni string broj i ako je, izračunati sumu znamenaka.

```
int sum_digits(char* str)
{
    int n = 0, s = 0, i;
    sscanf(str, "%d", &n);
    if (!n) return 0;
    for (i = 0; i < strlen(str); i++)
        s += (str[i] - '0');
    return s;
}
```

2. Napisati funkciju koja presloži string tako da su sva mala slova ispred velikih. Možete pretpostaviti da se string sastoji samo od slova.

```
void rearrange(char* str)
{
    char *p = str, *q = str + strlen(str) - 1;
    char t;
    while(p < q)
    {
        while(*p >= 'a' && *p <= 'z' && p < q) p++;
        while(*q >= 'A' && *q <= 'Z' && p < q) q--;
        t = *p; *p = *q; *q = t;
        p++; q--;
    }
}
```

3. Napisati funkciju koja vraća broj znamenki i broj slova u stringu.

```
#include <ctype.h>
void count(char* arr, unsigned* pbrz, unsigned* pbrs)
{
    char* p = arr;
    *pbrz = 0;
    *pbrs = 0;
```

```
    while (*p)
    {
        if (isalpha(*p)) (*pbrs)++;
        if (isdigit(*p++)) (*pbrz)++;
    }
}

int main(void)
{
    char a[] = "Ovo je 3tr1ng";
    unsigned br_z, br_s;
    count(a, &br_z, &br_s);
    printf("broj znamenki je %u, broj slova %u\n",
                                br_z, br_s);
    return 0;
}
```

4. Napisati funkciju koja napuni string duljine `n` slučajno generiranim malim slovima alfabeta i prebroji koliko puta se izgeneriralo slovo 'z'. Duljinu stringa unosi korisnik, a memorija za string se alocira dinamički.

```
int fill_count(char* str, int n)
{
    int i, sz = 0;
    srand((unsigned)time(NULL));
    for (i = 0; i < n; i++)
    {
        str[i] = rand() % ('z' - 'a' + 1) + 'a';
        if (str[i] == 'z') sz++;
    }
    return sz;
}

int main (void)
{
   int n, rez;
   char* s;
   puts("unesi n");
   scanf("%d", &n);
   s = (char*) malloc((n + 1) * sizeof(char));
   rez = fill_count(s, n);
   printf("u stringu %s ima %d pojavljivanja
```

```
            znaka 'z'", s, rez);
    free(s);
    return 0;
}
```

5. Napisati funkciju koja će vratiti broj ponavljanja određenog znaka u stringu i poziciju zadnjeg pojavljivanja tog znaka.
   Primjer: za string `s = "tralalalalaaaa"` i znak `c = 'l'`, broj pojavljivanja znaka `'l'` je 4, a pozicija zadnjeg znaka `'l'` je 10.

```
void fun(char const* str, char znak, unsigned* p_brpon,
    unsigned* p_poz)
{
    const char* p = str;
    *p_brpon = 0;
    *p_poz = 0; /* pretp. da nema znaka u stringu*/
    while(*p)
    {
        if (*p == znak)
        {
            (*p_brpon)++;
            *p_poz = p - str + 1;
        }
        p++;
    }
}
```

6. Napisati funkciju koja modificira string tako da više uzastopnih razmaka u stringu zamjeni jednim.
   Primjer: za string `"Ovo je string."` modificirani string je `"Ovo je string."`

```
void fun(char* str)
{
    char* tmpstr = (char*) malloc(strlen(str) + 1);
    char* p = str;
    char* q = tmpstr;
    while(*p)
    {
        *q = *p;
        while (*p == ' ' && *(p + 1) == ' ') p++;
```

```
            p++;
            q++;
        }
        *q = '\0';
        strcpy(str, tmpstr);
        free(tmpstr);
    }

    void fun_1(char* str)
    {
        char* p = str;
        char* end = str + strlen(str); /*end pokazuje na \0*/

        while(*p)
        {
            while (*p == ' ' && *(p + 1) == ' ')
                memmove(p, p + 1, end - p);
            p++;
        }

    }
```

7. Napisati funkciju koja u novi string stavlja sve znakove iz stringa koji nisu ni slova ni brojevi. Memoriju za novi niz dinamiči alocirati.

```
int alphanum(char ch)
{
    if ((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= '
       z')
        || (ch >= '0' && ch <= '9'))
        return 1;
    return 0;
}

char* novi_str(char* str)
{
    char* novi = (char*) malloc((strlen(str) + 1) *
       sizeof (char));
    char* p = str, *q = novi;
    while(*p)
    {
        if (!alphanum(*p))
```

```
            *q++ = *p;
        p++;
    }
    *q = '\0';
    return novi;
}
```

8. Napisati funkciju koja provjerava je li rečenica *pangram*. *Pangram* je recenica u kojoj se svako slovo alfabeta (abecede) pojavljuje najmanje jednom. *Primjer*: String "The quick brown fox jumps over the lazy dog" je *pangram*. Ili, na hrvatskom "Gojazni đačić s biciklom drži hmelj i finu vatu u džepu nošnje."

```
int pangram(char* str)
{
    int ch;
    for (ch = 'A'; ch <= 'Z'; ch++)
        if (!strchr(str, ch) && ! strchr(str, tolower(ch)
            ))
            return 0;
    return 1;
}
int main(void)
{
    char str1[]="The quick brown fox jumps over the lazy
        dog";
    char str2[]="Ovo nije pangram";
    pangram(str1) ? printf("%s je pangram\n", str1) :
        printf("%s nije pangram\n", str1);
    pangram(str2) ? printf("%s je pangram\n", str2) :
        printf("%s nije pangram\n", str2);
    return 0;
}
```

9. Napisati funkciju koja za neki string vraća poziciju drugog pojavljivanja nekog slova. U slučaju da se slovo ne pojavljuje dva puta, funkcija vraća -1. *Primjer:* Za string "ONoffON" pozicija drugog pojavljivanja slova 'O' je 6

```c
int position(char* str, char letter)
{
    char* pos = strchr(strchr(str, letter) + 1, letter);
    return pos ? pos - str + 1 : -1;
}
int position_2(char* str, char letter)
/* drugi nacin*/
{
    int br = 0, i;
    for (i = 0; i < strlen(str); i++)
        if (str[i] == letter && ++br == 2) return i + 1;
    return -1;

}
int main(void)
/* test */
{
    char str[] = "ONoffON";
    char let = 'f';
    printf("Druga pozicija slova %c je %d\n", let,
       position_2(str, let));
    printf("Druga pozicija slova %c je %d\n", let,
       position(str, let));
    return 0;
}
```

10. Napisati funkciju koja spaja tri stringa. Memoriju za novi niz dinamički alocirati.

```c
char* join(char* a, char* b, char* c)
{
    char* novi = malloc((strlen(a) + strlen(b) + strlen(
       c) + 1) * sizeof(char));
    strcpy(novi, a);
    strcat(novi, b);
    strcat(novi, c);
    return novi;
}
```

11. Napisati funkciju koja iz stringa izbacuje svako pojavljivanje podstringa. *Primjer:* Za string `"Oh, la la"`! izbacivanjem podstringa `"la"` bi se dobilo

```
"Oh, "!.
```

```
void remove_substr(char* str, char* substr)
{
    char* p = str;
    char* q = substr;
    char* tmpstr = (char*)malloc(strlen(p) + 1);
    char* res = tmpstr;
    while(*p)
    {
        if (*p == *q)
        {
            char* poc = p; /* zapamti pocetak   *
                           * podstringa u stringu */
            while(*q)
            {
                if (*p++ != *q++)
                {
                    p = poc; /* substr nije u stringu*/
                    break;
                }
            }
        }
        *res++ = *p++;
    }
    *res = '\0';
    strcpy(str,tmpstr);
    free(res);
}

int main(void)
{
    char str[] = "Ovo je originalni string iz kojeg
       izbacujemo string";
    char substr[] = "str";
    remove_substr(str, substr);
    puts(str);
    return 0;
}
```

12. Napisati funkciju koja kao argument prima string te ga izvrne iznutra prema vani (okrene prvu i drugu polovicu nezavisno). Ako string ima

neparan broj slova, srednje ostaje netaknuto. *Primjer:* `ABCDE -> BACED`

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

void swap_sequence(char *beg, char *end)
{
    char t;
    int i, n = (int)(end - beg);
    for(i = 0; i < (n / 2); ++i)
    {
        t = beg[i],
        beg[i] = beg[n - i - 1],
        beg[n - i - 1] = t;
    }
}
void fun(char *str)
{
    size_t l = strlen(str);
    swap_sequence(&str[0], &str[l / 2]);
    swap_sequence(&str[l -(l / 2)], &str[l]);
}

int main()
{
    char str[256];

    printf("Unesite string: ");
    scanf("%s", str);
        fun(str);
    printf("novi string = %s\n", str);

    return 0;
}
```

7.3 Zadaci za vježbanje

1. Napisati funkciju koja vraća broj znamenki i broj slova u stringu.
2. Napisati funkciju koja ispituje jesu li dvije riječi anagrami (jedna riječ je permutirana druga riječ).
   Primjer: riječi "vatra", "trava" i "vrata" su anagrami.
3. Napisati funkciju koja modificira string tako da izbaci sve samoglasnike iz njega.
   Primjer: "Popokatepetl" postaje "Ppktptl".
4. Napisati funkciju koja će iz stringa koji predstavlja url izvući (i vratiti) url *roota* (bez oznake protokola).
   Primjer: za url `http://www.hackcanada.com/ice3/2600/2600_15-3_p10.txt` adresa root-a je `www.hackcanada.com`.
5. Napisati funkciju koja provjerava je li uneseni string broj i ako je, izračuna prosječnu vrijednost parnih i neparnih znamenaka.
6. Napisati funkciju koja prima tri stringa, te vraća novi string koji je rezultat spajanja dva dulja stringa.
7. Napisati funkciju koja provjerava da li string završava sa nekim podstingom.
   Primjer: za podstring `".csv"` i string `"datoteka.csv"` funkcija vraća pozitivan odgovor, a za string `"datoteka.cvss"` negativan.
8. Napisati funkciju koja preuređuje string tako da svako veliko slovo udvostruči, a mala slova izbaci. Sve druge znakove ostavi.
   Primjer: za string `"ONoffON"` promijenjeni string je `"OONNOONN"`.
9. Napisati funkciju koja alocira prostor za string, te generira niz znakova duljine *n* tako da se isti znak nikada ne ponavlja. Znakovi mogu biti velika slova alfabeta. Funkcija vraća novonastali string.
10. Napisati funkciju koja okreće svaku riječ u rečenici.
    Primjer: string `"Morski pas je gladan."` postaje `"iksoM sap ej nadalg."`
11. Napisati funkciju koja preuređuje rečenicu tako da nakon svakog samoglasnika doda slovo 'p' i sam samoglasnik. Realokaciju za dodatne znakove

realizirati dinamički.
Primjer: rečenica `"Ja sam dobro."` postaje `"Japa sapam dopo bropo."`

12. Napisati funkciju koja zamjenjuje po dvije susjedne riječi u rečenici.
Primjer: string `"Morski pas je gladan."` postaje `"pas Morski gladan je."`
13. Napisati funkciju koja spaja riječi iz rečenice u novi string od najduže do najkraće riječi. Ako su riječi jednako duge, sortiraju se leksikografski.
Primjer: za rečenicu `"Da mi je biti morski pas."` novi string je `"morski biti pas Da je mi."`
14. Napisati funkciju koja string, koji predstavlja bitovni zapis nekog broja, pretvara u broj. Funkcija vraća dobijeni broj.
15. Napisati funkciju koja u stringu zamijeni svaku prazninu sa drugim stringom.
16. Napisati funkciju koja modificira string tako da udvostučuje sve znakove koji nisu ni slova ni brojevi, a ostale znakove ne.
Primjer: za string `s = "24/07/gggg"` novi string je `s = "24//07//gggg"`.
17. Napisati program u kojem korisnik unosi datum u obliku dd-mm-yyyy i koji računa da li je uneseni datum u prijestupnoj godini. Godina je **prijestupna** ako je djeljiva s 4 i nije djeljiva sa 100 ili ako je djeljiva sa 400.
18. Napisati funkciju koja napuni string duljine `n` slučajno generiranim velikim slovima alfabeta i prebroji koliko puta su se izgenerirala slovo 'X' i 'Y' (odvojeno). Duljinu stringa unosi korisnik, a memorija za string se alocira dinamički.
19. Napisati funkciju koja u zadanom stringu broji veličinu najvećeg niza uzastopnih istih znakova.
20. Napisati funkciju koja ubacuje uneseni string između dva stringa. Prostor za rezultirajući string alocirati dinamički.
Napomena: Parametri funkcije su dva stringa. Korisnik unosi treći string u funkciji.
21. Napisati funkciju koja za dani niz znamenki i danu veličinu niza provjerava da li je nul-terminiran i ako nije, dodaje nul-terminator na kraj.
22. Napisati funkciju koja vraća broj riječi u stringu koje počinju velikim slo-

vom, broj riječi koje počinju malim slovom i broj riječi koje počinju sa znamenkom.

23. Napisati funkciju koja za zadani string vraća poziciju prvog i zadnjeg pojavljivanja danog znaka. Ukoliko danog znaka nema u stringu vraća -1.
24. Napisati funkciju koja će vratiti broj ponavljanja određenog znaka u stringu i poziciju zadnjeg pojavljivanja tog znaka.
    Primjer: za string s = "tralalalalaaaa" i znak c = 'l', broj pojavljivanja znaka 'l' je 4, a pozicija zadnjeg znaka 'l' je 10.
25. Napisati funkciju koja u neki string nakon točke ili zareza ubaci razmak ako ne postoji na tom mjestu.
26. Napisati funkciju koja modificira string tako da nakon svake riječi ubaci oznaku za novi red.
    Primjer: za string `s = "Ovo je recenica"`, modificirani string je `s = "Ovo \nje\nrecenica."`.
27. Napisati funkciju koja prima tri stringa, te vraća novi string koji je rezultat spajanja dva kraća stringa. Memoriju za novi string alocirati dinamički.
28. Napisati funkciju koja će vratiti broj ponavljanja određenog znaka u stringu i poziciju prvog pojavljivanja tog znaka. Ukoliko znak nije pronađen, funkcija vraća 0.
29. Napisati funkciju koja za string oblika `"xxx-xxx-xxx` vrati informaciju, može li string predstavljati telefonski broj. Telefonski broj ima ukupno 9 ili 10 znamenki i počinje sa nulom.
    Primjer: string "021-333-444" je telefonski broj, a "02a-aa1-11a" nije telefonski broj.
30. Napisati funkciju koja iz dva stringa koji predstavljaju ime izvođača i naziv pjesme, kreira treći string koji ima oblik "izvođač-pjesma.mp3" tako da prvo slovo imena izvođača i imena pjesme bude zapisano velikim slovom, a svi drugi znakovi malim.
    Napomena: Ime izvođača i ime pjesme su parametri funkcije. Funkcija vraća novi string.
31. Napisati funkciju koja od dva stringa kreira treći tako da oba stringa okrene

naopako i spoji. Memoriju za novi string alocirati dinamički.
Primjer: za stringove `"Hello"` i `"world"` rezultirajući string je "olleHdlrow".

32. Napisati funkciju koja iz stringa koji je oblika `"broj-broj-broj"` pročita i vrati sumu i prosječnu vrijednost brojeva.
Primjer: za string `"021-220-2002"` funkcija vraća `2243` i `2243/3` tj. `747,67`.
33. Napisati funkciju koja generira string proizvoljne duljine (duljinu stringa unosi korisnik). Funkcija vraća novi string. String se može sastojati od slova i znamenki.
34. Napisati funkciju koja za dani `n` zapisuje u string prvih `n` Fibonaccijevih brojeva. Funkcija vraća novokreirani string.
Primjer: za `n = 10` funkcija vraća string `"11235813213456"`.
35. Napisati funkciju koja za string u kojem su brojevi odvojeni znakom '-' vraća koliko ima brojeva u stringu, te najveći broj među njima.
Primjer: za string `"225-11-23-2456-2"` funkcija vraća broj brojeva `5`, te najveći broj `2456`.
36. Napisati funkciju koja vraća ukupan broj pojavljivanja samoglasnika u stringu, te broj pojavljivanja najčešćeg samoglasnika u stringu.
37. Napisati funkciju koja provjerava da li je rečenica heterogram. Heterogram je rečenica u kojoj se nijedno slovo u rečenici ne pojavljuje više od jednom.
Primjer: string "The big dwarf only jumps." je heterogram.
38. Napisati funkciju koja provjerava je li rečenica izogram. Izogram je rečenica u kojoj se svako slovo u rečenici pojavljuje jednak broj puta.
Primjer: stringovi "Caucasus" ili "Abba" su izogrami.
39. Napisati funkciju koja vraća ukupan broj slova u stringu koji se ne pojavljuju i kao malo i kao veliko slovo alfabeta.
Primjer: u stringu "Alpha, Beta, Theta", slova 'l', 'p', 'h', 'b', 'e', 'g' i 'm' se ne pojavljuju kao malo i kao veliko slovo.
40. Napisati funkciju koja za dani niz znakova broji koliko se slova pojavljuje i kao malo i kao veliko slovo alfabeta.
Primjer: u stringu "Abba Theta Terra" slova 'a' i 't' se pojavljuju kao mala i velika slova, pa funkcija treba vratiti broj `2`.

41. Napisati funkciju koja iz stringa koji se sastoji od nekoliko riječi kreira akronim (početna velika slova svake od riječi u stringu).
    Primjer: za string "Asynchronous JavaScript and XML" akronim je "AJAX".
42. Napisati funkciju koja za string provjerava da li je njegov akronim (početna slova svake od riječi u stringu) rekurzivni akronim (prva riječ stringa je jednaka njegovom akronimu).
    Primjer: za string "SPARQL Protocol And RDF Query Language" akronim je "SPARQL", dakle rekurzivan je.
43. Napisati funkciju koja iz dva stringa kreira i vraća treći string tako da u njemu prvo bude kraći, pa duži string, odvojeni zarezom i razmakom.
44. Napisati funkciju koja ispituje jesu li dva stringa anagrami.
    Napomena: dvije riječi su anagrami ako se permutiranjem slova jedne riječi može dobiti druga riječ. Npr. riječi "vatra" i "trava" su anagrami.
45. Napisati funkciju koja će između dva stringu umetnuti treći, ali tako da između riječi doda jedan razmak.
    Primjer: za stringove "Veni" i "vici", i treći "vidi", rezultirajući string treba biti "Veni vidi vici".
46. Napisati funkciju koja provjerava je li rečenica pangram. Pangram je recenica u kojoj se svako slovo alfabeta (abecede) pojavljuje najmanje jednom.
    Primjer: string "The quick brown fox jumps over the lazy dog" je pangram. Ili, na hrvatskom "Gojazni đačić s biciklom drži hmelj i finu vatu u džepu nošnje."
47. Napisati funkciju koja za dva stringa provjerava jesu li zadnja dva slova jednoga stringa prva dva slova drugoga stinga.
48. Napisati funkciju koja iz stringa čita znakove i ako je znak znamenka sprema u niz brojeva. Funkcija vraća novonastali niz. *Primjer*: "Kaladont" i "jabuka" imaju takvo svojstvo.
49. Napisati funkciju koja za neki string vraća informaciju da li je string palindrom i je li string "dupli string".
    Napomena: Primjer palindroma: "Ana nabra par banana." Primjer duplog stringa: "Jalta, Jalta".

50. Napisati funkciju koja za neki string broji sva pojavljivanja podstringa u njemu.
    Primjer: za podstring "Homer" u stringu "The author of the Iliad is either Homer or, if not Homer, somebody else of the same name." funkcija treba vratiti broj dva.
51. Napisati funkciju koja iz stringa izbaci sve znakove koji nisu znamenka, te od preostalih znakova formira i vraća broj.
    Primjer: za string "6 puta 9 je 42.", funkcija vraća broj 6942.
52. Napisati funkciju koja za string oblika `"DD.MM.YYYY"` računa koliko osoba s tim datumom rođenja ima godina.
53. Napisati funkciju koja modificira string tako da udvostručuje sve brojeve, a ostale znakove ne.
    Primjer: s = "24/07/gggg" Novi string: s = "2244/0077/gggg"
54. Napisati funkciju računa sumu i umnožak kubova članova niza. Za kubiranje napisati predprocesorsku direktivu.
55. Napisati funkciju računa najveće i najmanje oplošje niza kocaka (kocka je zadana jednom stranicom, oplošje je $6a^2$). Za računanje oplošja napisati predprocesorsku direktivu.
56. Napisati funkciju koja postojeći niz proširuje tako da između svaka dva elementa ubaci njihovu aritmetičku sredinu. Funkcija vraća novi niz.
    Primjer: za niz brojeva `A = 1, 2, 3, 4, 6` novi niz je `B = 1, 1.5, 2, 2.5, 3, 3.5, 4, 5, 6`.
57. Napisati funkciju koja popravlja string tako da iza svakog znaka interpunkcije ubacuje, ako ne postoji i ako to nije zadnji znak, prazninu.
    Primjer: `"Tri,dva,jedan, nula!"` ispravi u `"Tri, dva, jedan, nula!"`
58. Napisati funkciju koja na mjestu pojavljivanja znamenke u stringu ubacuje tekstualni naziv znamenke.
    Primjer: string `"3 2 1 kreni!"` zamjenom bi postao `"tri dva jedan kreni!"`.
59. Napisati funkciju koja će iz slučajno generiranih brojeva iz intervala [10, 1000] kreirati niz i to tako da, ako je novi broj veći od prethodno izgeneriranog broja, novi broj stavlja na kraj niza, a ako je manji na početak niza.

Funkcija vraća novi niz.

7.4 Ponavljanje

Što ispisuju sljedeći programi i zašto?

1. ———

```
int main(void)
{
    char *ptr = "ABCD";
    *ptr++;
    puts(ptr);
    ptr++;
    puts(ptr);
    return 0;
}
```

2. ———

```
int main(void)
{
    char str[] = "BDCB";
    char* p = str;
    char* q = str + strlen(str) - 1;
    while (p < q)
    {
        (*p++)--;
        (*q--)++;
    }
    puts(str);
    return 0;
}
```

3. ———

```
int main(void)
{
    char str[] = "Q1E3";
    char* q = str + strlen(str) - 1;
    char* p = str;
    while (p < q)
    {
        (*p++)++;
        (*q--)--;
    }
    puts(str);
```

```
    return 0;
}
```

4. ———

```
int main(void)
{
    unsigned i;
    char str[] = {'B', '-', '0', '0', '\0'};
    do{
        if (!(i%2)) str[i]++;
    }while (str[i++]);
    puts(str);
    return 0;
}
```

5. ———

```
#define WRAP(x, y, cnt) y = (x + 1) % cnt
int main(void)
{
    int n = 5, k = 6;
    WRAP(n|k, n, k);
    printf("n = %d, k = %d\n", n, k);
    return 0;
}
```

6. ———

```
void print_k(unsigned n, unsigned k)
/* zapis broja n u bazi k */
{
    if (!n) return;
    print_k(n/k, k);
    printf("%d", n%k);
}
#define K 13
int main(void)
{
    int a[] = {6, 9};
    int* p = a;
    int k = *p;
    print_k(*p++, K);
    printf("*");
    k *= *p;
    print_k(*p++, K);
    printf("=");
```

```
    print_k(k, K);

    return 0;
}
```

7. ———

```
#define ASSIGN(a,b) a=(char)b
int main(void)
{
    int x = 1 << 10;
    char y;
    ASSIGN(y,x >> 8);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

8. ———

```
int main(void)
{
    char a[] = "4,03048";
    char* p = a;
    char* q = a + strlen(a) -1;
    while (*p != ',')
        (*p++)--;
    while (*q != ',')
        (*q--)++;
    puts(a);
    return 0;

}
```

9. ———

```
int main(void)
{
    int m = 5;
    int* p = &m;
    *p *= ++m;
    printf("%d\n", m);

}
```

10. ———

```
#define cnt(x) (sizeof(x)/sizeof(x[0]))
```

```
int main(void)
{
    int a[] = {2, -3, 5, 1, 8, 0};
    int* p = a;
    int* q = a + cnt(a) - 1;
    while (*p++ && *q--)
        q = p;
    printf("*p = %d\n", *p);
    printf("*q = %d\n", *q);
    return 0;
}
```

11. ———

```
#define fraq(x, y) x / y
#define max(x, y) x > y ? x : y

int main(void)
{
    int a[] = {2, 3, 5, 1};
    printf("%d\n",max(fraq(a[0],a[1]), fraq(a[2],a[3])));
    return 0;
}
```

12. ———

```
#define cnt(x) (sizeof(x)/sizeof(x[0]))

int main(void)
{
    int a[] = {2, -3, 5, 1, 8, 0, 2};
    int* p = a;
    int* q = a + cnt(a) - 1;
    while (++*p && *q--)
        ;
    printf("*p = %d\n", *p);
    printf("*q = %d\n", *q);
    return 0;
}
```

13. ———

```
#define fraq(x, y) x / y
#define cnt(x) sizeof(x)/sizeof(x[0])

int main(void)
{
```

```
  int i;
  double a[] = {2, 5, 1};
  for (i = 0; i < cnt(a) - 1; )
    printf("%lf\n",fraq(a[i++],a[i]));
  return 0;
}
```

14. ———

```
#define cnt(x) sizeof(x)/sizeof(x[0])

int main(void)
{
    int a[] = {0, 2, -1, -1, 0}, i;
    int* p = a, *q = a + cnt(a) - 1;
    do{
        p++;
    }while (++*p && p != q);

    for (i = 0; i < cnt(a); i++)
        printf("%d, ", a[i]);
    puts("");
    return 0;
}
```

15. ———

```
#define max(x, y) x > y ? x : y

int main(void)
{
    int a[] = {2, 3, 5, 1};
    printf("%d\n", max(a[0]+a[1], a[2]+a[3]));
    return 0;
}
```

16. ———

```
#define cnt(x) sizeof(x)/sizeof(x[0])

int main(void)
{
    int a[] = {0, 2, -1, -1, 0}, i;
    int* p = a;
    int* q = a + cnt(a) - 1;
    while (++*p && p != q)
        p++;
```

```
    for (i = 0; i < cnt(a); i++)
        printf("%d, ", a[i]);
    puts("");
    return 0;
}
```

17. ———

```
int main(void)
{
    char a[] = "Hello_world!\n";
    char* p = a;
    char* q = a + strlen(a)-1;
    while(p++ != q-- && *p);
    printf("*p je %c\n", *p);
    printf("*q je %c\n", *q);
    return 0;
}
```

18. ———

```
int main(void)
{
    char str1[] = "Prvi";
    char str2[] = "Drugi";
    char *p = str1, *q = str2;
    while(*p++ != *q++ && *p);
    printf("*p je %c\n", *p);
    printf("*q je %c\n", *q);
    return 0;
}
```

19. ———

```
#define KVADAR(a, b, c) a * b * c
int main(void)
{
    int x = 2;
    printf("%d\n", KVADAR(x, x + 1, x + 2));
    printf("%d\n", KVADAR(x, x - 1, x + 1));
    return 0;
}
```

20. ———

```
int main(void)
{
```

```
    char str[] = "popokatepetl";
    char* p = str;
    while(p < str + strlen(str) - 2)
    {
        if (*p == *(p + 2))
            printf("%c%c\n",*p,*(p+2));
        p++;
    }
    return 0;
}
```

21. ———

```
#define VOL(a, b, c) a * b * c
int main(void)
{
    int x = 2;
    printf("%d\n", VOL(x, x - 1, x + 1));
    printf("%d\n", VOL(x - 2, x - 1, x));
    return 0;
}
```

22. ———

```
int main(void)
{
    char a[] = "Hello_world!\n";
    char* p = a;
    char* q = a + strlen(a)-1;
    while(++p != --q && *p);
    printf("*p je %c\n", *p);
    printf("*q je %c\n", *q);
    return 0;
}
```

23. ———

```
#define PRODUKT(a, b) a * b
int main(void)
{
    int x = 3, y = 3;
    printf("%d\n", PRODUKT(x - 1, y - 1));
    printf("%d\n", PRODUKT(x - 1, y + 1));
    return 0;
}
```

24. ———

```
int main(void)
{
    char str[] = "Traalala";
    char* p = str;
    do{
        if (!*p) break;
        if (*p == *(p - 1))
            printf("slovo %c\n", *p);
        ++p;
    } while(1);
    return 0;
}
```

25. ———

```
#define VECI(a, b) a > b ? a : b
int main(void)
{
    int x = 3, y = 3;
    printf("%d\n", VECI(++x, --y));
    printf("%d\n", VECI(y, --x));
    return 0;
}
```

26. ———

```
int main(void)
{
    int a[] = {1, 2, 3, 2, 3, 5};
    int* p = a;
    int k = *p++;
    while (p < a+sizeof a/sizeof a[0])
        k += *p++;
    printf("k = %d\n", k);
    return 0;
}
```

27. ———

```
#define MANJI(a, b) a < b ? a : b
int main(void)
{
    int x = 3, y = 3;
    printf("%d\n", MANJI(++x, --y));
    printf("%d\n", MANJI(y, --x));
    return 0;
}
```

28. ———

```
int main(void)
{
    char str[20] = "Hello";
    char* const p = str;
    *p = 'B';
    printf("%s\n", str);
    return 0;
}
```

29. ———

```
#define PREV(i) arr[i--]
int main(void)
{
    int arr[] = {6, 2, 0, 0, 7, 0, 2, 5};
    int k = sizeof(arr)/sizeof(arr[0]) - 1;
    int s = 0;
    while(PREV(k)) s += PREV(k);
    printf("%d\n", s);
    return 0;
}
```

30. ———

```
int main(void)
{
    char str1[] = "r2";
    char str2[] = "d2";
    char *s1 = str1, *s2=str2;
    while(*s1++ = *s2++)
        printf("%s", str1);

    printf("\n");
    return 0;
}
```

31. ———

```
#define SQR(x) ((x)*(x))

int main(void)
{
    int a, b = 3;
    a = SQR(b + 2);
    printf("%d\n", a);
```

```
    return 0;
}
```

32. ———

```
int main(void)
{
    int i, n;
    char str[] = "PMA";
    char* x = str;
    n = strlen(x);
    *x = x[n - 1];
    printf("%s\n", x);
    x++;
    printf("%s\n", x);
    return 0;
}
```

33. ———

```
#define z x + y
#define zz x - y
int main(void)
{
    int x = 1;
    int y = 2;
    x = y * z * zz;

    printf("%d %d %d %d\n", x, y, z, zz);
    return 0;
}
```

34. ———

```
#define CNT(arr) sizeof arr / sizeof arr[0]
void change(int *b, int n)
{
    int i;
    for(i = 0; i < n; i++)
        *(b + 1) = *(b + i) + n;
}
int main(void)
{
    int i, a[] = {1, 2, 3, 4, 5};
    change(a, CNT(a));
    for(i = 0; i < CNT(a); i++)
        printf("%d, ", a[i]);
```

```
    return 0;
}
```

35. ———

```
void fun(int*, int*);
int main(void)
{
    int i = 5, j = 2;
    fun(&i, &j);
    printf("%d, %d", i, j);
    return 0;
}
void fun(int *i, int *j)
{
    *i = *i**i;
    *j = *j**j;
}
```

36. ———

```
int main(void)
{
    int arr[] = {12, 13, 14, 15, 16};
    printf("%d, %d, %d\n", sizeof(arr), sizeof(*arr), sizeof(
        arr[0]));
    return 0;
}
```

37. ———

```
#define NEXT(i) arr[i++]
int main(void)
{
    int arr[] = {6, 2, 3, 5, 7, 0, 0, 5};
    int k = 0, s = 0;
    while(NEXT(k)) s += NEXT(k);
    printf("%d\n", s);
    return 0;
}
```

38. ———

```
int main(void)
{
    char str[] = "ispit";
    char* p = str;
```

```
    ***++p;
    puts(str);
    puts(p);
    return 0;
}
```

39. ———

```
#define SQUARE(x) x*x
int main(void)
{
    int s = 32/SQUARE(2);
    printf("%d\n", s);
    return 0;
}
```

40. ———

```
#define cnt(x) sizeof(x)/sizeof(x[0])
int main(void)
{
    int a[] = {2, 2, 0, 1, 8, 0};
    int* p = a;
    int* q = a + cnt(a) - 1;
    while (*p++ && *q--)
        q = p;
    printf("*p = %d\n", *p);
    printf("*q = %d\n", *q);
    return 0;
}
```

41. ———

```
#define int char
int main(void)
{
    int i=65;
    printf("sizeof(i)=%d",sizeof(i));
    return 0;
}
```

42. ———-

```
int main(void)
{
   char str[] = "R2D2";
   int i = 2, c;
```

```
   while (str[i])
   {
      switch(str[i])
      {
         case 'R': printf("Y");
            continue;
         case 'D':
            printf("C-%c", 1+str[i+1]);
         case  1:  while (str[++i]) printf("P");
         case  2:  printf("O\n"); break;
         case '2': printf("X\t"); break;
      }
   }
   return 0;
}
```

43. ———

```
int main(void)
{
    int n = 4, m = 5;
    switch(n % 2)
    {
        case 0: do{ n = m--;
        case 1: n = m--;
        case 2: n = --m;
             }while( --n > 0);
    }
    printf("n=%d, m=%d\n", n, m);
}
```

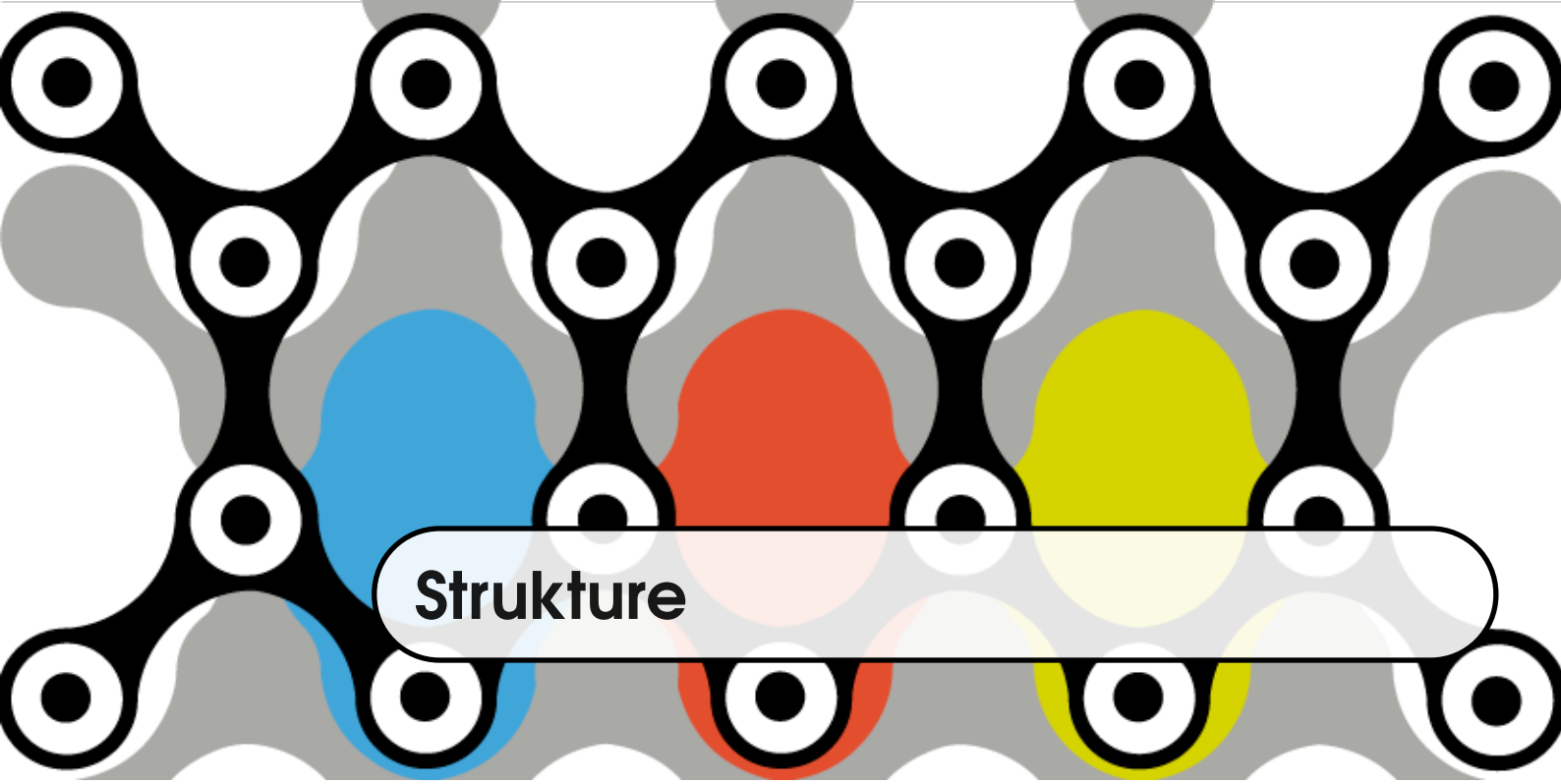
44. ———

```
#define DO(a,b) b=(a+b)-(a=b)

int main(void)
{
    int x = 10;
    int y = 15;
    printf("%d %d\n",x,y);
    DO(x, y);
    printf("%d %d\n",x,y);
    return 0;
}
```

45. ———

```
#include <stdio.h>
#define f(a,b) a+b
#define g(a) -a
int main(void)
{
      printf("%d\n",g(f(4,2)));
      printf("%d\n",f(g(4), 2));
      return 0;
}
```



Strukture

U programerskim zadacima često se upravlja kompleksnim objektima koji imaju složenu strukturu. Tipovi podataka obrađeni do sada ne omogućuju nam da elegantno opišemo takve objekte. Primjerice, ako želimo opisati točku u trodimenzionalnom prostoru, koje osim samog položaja u prostoru imaju i boju, jednostavni tipovi i nizovi neće biti adekvatni. Točka je opisana sa tri `float` ili `double` koordinate i jednom varijablom (npr. tipa `int`) koja opisuje boju. Kako se opis točke sastoji od varijabli različitih tipova, točka se ne može opisati nizom, jer je niz kolekcija podataka istog tipa.

```
double x;
double y;
double z;
int color;
```

Nadalje, kako biste modelirali problem koji uključuje niz točaka? S dosadašnjim znanjima iz ovog kolegija, mogli biste napraviti četiri niza, od kojih tri predstavljaju koordinate, a jedan boju.

```
double x[100];
double y[100];
double z[100];
int color[100];
```

Međutim, na ovaj se način podaci o jednoj točki nalaze u četiri različita niza. Dohvaćanje ili promjena jedne koordinate i boje podrazumijeva pristup različitim nizovima.

Dakle, postoji potreba da se točka opiše nekim tipom koji će na okupu držati podatke o točki, tj. potreba da jedan zapis predstavlja jednu točku. Takav zapis tvori složeni tip podataka i sastoji se od varijabli ne nužno istih tipova.

8.1 Definicija strukture

DEFINICIJA STRUKTURE

Definicija novog tipa sastoji se od dodjeljivanja imena novom tipu i deklaracije i imena varijabli koje čine novodefinirani tip.

```
struct point
{
    double x, y, z;
    int color;
};
```

U navedenom primjeru dana je definicija novog tipa `struct point`. Ime strukture je `point`, a ime tipa `struct point`. Nadalje, svaki objekt novog tipa `struct point` sastojat će se od četiri polja: `x`, `y`, `z` i `color`.

8.2 Instance strukture

Instanca strukture može se kreirati deklariranjem varijable tipa strukture.

Sama deklaracija varijable se piše uobičajeno: `ime_tipa ime_varijable`. Naprimjer, ako se želi deklarirati varijabla koja će predstavljati točku, piše se:

```
struct point tocka;
```

`operator .`

Do svakog pojedinog zapisa unutar strukture dolazi se operatorom točka. Tako `tocka.x` predstavlja `x` koordinatu točke `tocka` i tipa je `double`, a `tocka.color` predstavlja boju točke i tipa je `int`.

Prilikom deklaracije varijable, alocira se memorija za varijablu. Koja količina memorije će biti alocirana, ovisi o tipu deklaracije. Za gornji primjer `struct point`, bit će alocirana memorija za tri `double` vrijednosti i jednu `int` vrijednost. Međutim, ukupna veličina alociranog prostora ne mora biti jednaka sumi pojedinačnih članova strukture, jer ovisi o poravnanju (engl. *alignment*). Naime način na koji varijable zauzimaju memoriju je takav da se omogući najbrži pristup podacima [8]. Tako će se tipovi koji su veličine 2 bajta najčešće smještati na parnim adresama, tipovi veličine 4 na adresama koje su djeljive s 4, a tipovi veličine 8 bajta na adresama djeljivim sa 8. To se zove samoporavnanje (engl. *self-alignment*).

Instanca neke strukture će imati poravnanje svog najšireg člana. Pritom treba uzeti u obzira da je adresa prvog člana strukture jednaka adresi same strukture.

Primjer 8.1 Veličina instance ovisi o poredku tipova unutar strukture. U ovom primjeru možemo vidjeti da se veličina strukture mijenja kada promijenimo redoslijed članova unutar strukture.

```
#include <stdio.h>
struct test
{
    char c;
    int n;
    double d;
};

struct test2
{
    int n;
    double d;
    char c;
};


int main(void)
{
    printf("velicina %u\n", sizeof(struct test));
```

```
    printf("velicina %u\n", sizeof(struct test2));
    return 0;
}
```

```
velicina 16
velicina 24
```

Za bolje razumijevanje možemo se poslužiti standardnim makroom `offsetof` koji vraća udaljenost adrese člana strukture od početka strukture.

Primjer 8.2 Upotrebom `offsetof` doznat ćemo udaljenost pojedinog člana strukture od početka strukture. Na različitim konfiguracijama rezultati mogu biti različiti.

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

struct test
{
    char c;
    int n;
    double d;
};

struct test2
{
    int n;
    double d;
    char c;
};

int main(void)
{
    puts("struct test");
    printf("c je udaljen %lu bajta.\n", offsetof(struct
        test, c));
```

```
        printf("n je udaljen %lu bajta.\n", offsetof(struct
            test, n));
        printf("d je udaljen %lu bajta.\n", offsetof(struct
            test, d));

        puts("struct test2");
        printf("n je udaljen %lu bajta.\n", offsetof(struct
            test2, n));
        printf("d je udaljen %lu bajta.\n", offsetof(struct
            test2, d));
        printf("c je udaljen %lu bajta.\n", offsetof(struct
            test2, c));
        return 0;
    }
```

```
c je udaljen 0 bajta.
n je udaljen 4 bajta.
d je udaljen 8 bajta.
struct test2
n je udaljen 0 bajta.
d je udaljen 8 bajta.
c je udaljen 16 bajta.
```

8.3 Dodjeljivanje vrijednosti varijablama tipa strukture

Vrijednost varijabli tipa `struct point` možemo dodijeliti na više načina:

- pridruživanjem vrijednosti pojedinim poljima unutar strukture

  ```
  tocka.x = 15.;
  tocka.y = -1.1;
  tocka.z = 0.1;
  tocka.color = 5;
  ```

- inicijalizacijom tako da se pridruže vrijednosti svim poljima odjednom u poredku u kojem su navedeni u definiciji

  ```
  struct point tocka = {15.0, -1.1, 0.1, 5};
  ```

Vrijednost varijabli može se postaviti koristeći se funkcijom za unos podataka.

```
scanf("%lf %lf %lf %d", &tocka.x, &tocka.y, &tocka.y, &tocka.color);
```

8.4 Upotreba `typedef` u definiciji strukture

Definiranjem strukture tvori se novi tip čije je ime `struct ime_strukture` . Ako je tip na taj način definiran, prilikom deklaracije varijable i prenošenja varijable tog tipa u funkciju potrebno je pisati cijelo ime tipa.

`typedef`

Često se, zbog jednostavnosti pisanja upotrebljava ključna riječ `typedef` kada se nekom tipu želi dati alternativno ime. Naprimjer,

```
typedef int cijeli;
```

daje alternativno ime `cijeli` tipu `int`, a

```
typedef struct point tpoint;
```

daje alternativno ime `tpoint` tipu `struct point`. U drugom primjeru se vidi da je ime tipa `struct point` koje se sastoji od dvije riječi, zamijenjeno jednom riječju `tpoint`.

Deklaracija varijable se sada može pisati `tpoint tocka;`.

Primjer 8.3 Definirati strukturu student koja opisuje studenta (ili studenticu) i njegove/njene ocjene iz pet predmeta. Napisati funkciju koja za nekog studenta/icu računa prosjek ocjena.

```
#include <stdio.h>

struct student
{
    char name[80];
```

```
    int grades[5];
};

float average(struct student st, int num_grades)
{
    int i, s = 0;
    for(i = 0; i < num_grades; ++i)
        s += st.grades[i];
    return (float)s/num_grades;
}

int main(void)
{
    struct student st = {"Ante Antic", 5, 4, 5, 5, 4};
    printf("average grade %f\n", average(st, 5));
    return 0;
}
```

```
average grade 4.600000
```

8.5 Pointer na strukturu

U primjeru 8.3 prilikom prenošenja argumenata u funkciju dogodilo se kopiranje vrijednosti. Pritom se kopiralo 100 bajtova, 80 bajtova za ime i još pet puta po `sizeof(int)` bajtova za ocjene, što je, recimo, još 20 bajtova. To nije velika količina podataka, ali možemo zamisliti da funkciju `average` pozivamo za svakog od 10000 studenata. Dakle, 10000 puta će se izvršiti kopiranje od po 100 bajtova. Tu ima smisla razmišljati o tome kako smanjiti količinu podataka koji se kopiraju. Najjednostavniji je način za to da ne kopiramo cijelu instancu strukture, već samo adresu početka strukture, a onda preko pokazivača pristupamo članovima strukture. U tom slučaju kopiramo 4 ili 8 bajtova (za 32-bitne tj. 64-bitne arhitekture) umjesto 100 bajtova. To je ušteda od 25 ili 12.5 puta, što na 10000 poziva nije zanemarivo.

POKAZIVAČ NA STRUKTURU

operator ->

Članovima strukture tada pristupamo pomoću operatora ->. Primjer 8.3 sada možemo malo drukčije napisati.

Primjer 8.4 Definirati strukturu student koja opisuje studenta (ili studenticu) i njegove/njene ocjene iz pet predmeta. Napisati funkciju koja za nekog studenta/icu računa prosjek ocjena.

```
#include <stdio.h>

struct student
{
    char name[80];
    int grades[5];
};

float average(struct student* st, int num_grades)
{
    int i, s = 0;
    for(i = 0; i < num_grades; ++i)
        s += st->grades[i];
    return (float)s/num_grades;
}

int main(void)
{
    struct student st = {"Ante Antic", 5, 4, 5, 5, 4};
    printf("average grade %f\n", average(&st, 5));
    return 0;
}
```

```
average grade 4.600000
```

Preporuka je da se struktura uvijek šalje u funkciju putem pokazivača.

8.6 Niz struktura

Kao i za bilo koji drugi tip, ako imamo neku količinu podataka istog tipa, u ovom slučaju tipa neke strukture, možemo deklarirati niz struktura. Takav niz također može biti statički ili dinamički i za njega vrijede ista pravila kao za nizove podataka bilo kojeg jednostavnog tipa.

Primjer 8.5 Napisati strukturu koja opisuje pravokutnik. Napisati funkciju koja će za niz pravokutnika naći onaj čija je dijagonala najveća.

```
#include <math.h>

typedef struct
{
    float a, b;
}rect;

float diag(const rect* r)
{
    return sqrt(r->a * r->a + r->b * r->b);
}

rect* max(const rect* r_arr, int n)
{
    int i;
    int imax = 0;
    float max_dia = diag(&r_arr[0]);
    for (i = 1; i < n; i++)
    {
        float diagonal = diag(&r_arr[i]);
        if(diagonal > max_dia)
        {
            max_dia = diagonal;
            imax = i;
        }
    }
    return r_arr + imax;
}
int main(void)
```

```
{
    rect r_arr[] = {{2, 3}, {2, 1}, {3, 4}, {3, 3}, {2,
        2}};
    rect* max_rect = max(r_arr, sizeof r_arr/sizeof r_arr
        [0]);
    printf("najvecu dijagonalu ima pravokutnik sa
        stranicama %f i %f\n", max_rect->a, max_rect->b);
    return 0;
}
```

8.7 Riješeni zadaci

1. Definirati strukturu pravokutnik, te napisati funkciju koja će provjeravati sječe li se pravokutnik s jediničnim kvadratom.

```
typedef struct
{
    float x, y;
}point;
typedef struct
{
    point a, b;
}rectangle;
int intersect(const rectangle* r)
{
    point ptr[] = {{0, 0}, {0, 1}, {1, 1}, {0, 1}};
    int i = 0;
    for (; i < 4; i++)
    {
        if (ptr[i].x >= r->a.x && ptr[i].x <= r->b.x
        &&
         ptr[i].y >= r->a.y && ptr[i].y <= r->b.y)
         return 1;
    }
    return 0;
}

int main(void)
{
    rectangle r = {{-0.2, -0.5}, {-2, -3}};
    intersect(&r) ? puts("Da") : puts("Ne");
    return 0;
}
```

2. Definirati strukturu `olimpic` u kojoj su podaci o imenu države, broj stanovnika i broj osvojenih medalja na Olimpijadi. Napisati funkciju koja vraća koja država ima najviše medalja u odnosu na broj stanovnika.

```
olimpic calculate_best(olimpic* ol, int n);
```

```
typedef struct
{
        char country[50];
        int population;
        int gold;
        int silver;
        int bronze;
} olimpic;

float calculate_score(const olimpic* ol)
{
        return (float)(3 * ol->gold + 2 * ol->silver +
            ol->bronze) / ol->population;
}
olimpic calculate_best(const olimpic* ol, int n)
{
        int i, imax = 0;
        float q;

        float max = calculate_score(&ol[0]);
        for (i = 1; i < n; i++)
        {
                q = calculate_score(&ol[i]);
                if (q > max)
                {
                        imax = i;
                        max = q;
                }

        }
        return ol[imax];
}

void print_olimpic(olimpic best_olimpic)
{
        printf("name: %s, population: %d. medal: %d %d
            %d\n", best_olimpic.country, best_olimpic.
           population,best_olimpic.gold, \
               best_olimpic.silver, best_olimpic.
                   bronze);
}

int main(void)
```

```
{
        int n, i;
        olimpic best_olimpic;
        olimpic ol[100];
        puts("number of countries?");
        scanf("%d", &n);
        for (i = 0; i < n; i++)
        {
                printf("name: ");
                scanf("%s", &ol[i].country);
                printf("population: ");
                scanf("%d", &ol[i].population);
                printf("medals - gold: ");
                scanf("%d", &ol[i].gold);
                printf("medals - silver: ");
                scanf("%d", &ol[i].silver);
                printf("medals - bronze: ");
                scanf("%d", &ol[i].bronze);
        }
        best_olimpic = calculate_best(ol, n);
        print_olimpic(best_olimpic);
        return 0;
}
```

3. Definirati strukturu kvadrat koji je zadan sa jednom točkom (donji lijevi vrh kvadrata) i duljinom stranice. Napisati funkciju koja će računati sijeku li se dva kvadrata.

```
#define COUNTOF(x) (sizeof(x)/sizeof((x)[0]))
typedef struct
{
    float x, y;
} point;

typedef struct
{
    point p;
    float a;
} square;

int point_in_square(const square *s1, const square *
   s2)
```

```
/* vraca da li su tocke kvadrata s2 unutar kvadrata
   s1 */
{
    static int current = 0;
    static point ptr = {0, 0};
    if (current == 0) ptr = s2->p;
    if (current == 1) ptr.x = ptr.x + s2->a;
    if (current == 2) ptr.y = ptr.y + s2->a;
    if (current == 3) ptr.x = ptr.x - s2->a;
    printf("point %f %f, current %d\n", ptr.x, ptr.y,
        current);
    current++;

    if (ptr.x >= s1->p.x && ptr.x <= s1->p.x + s1->a
       &&
        ptr.y >= s1->p.y && ptr.y <= s1->p.y + s1->a )
        return 1;
    return 0;
}

int intersect (const square *s1, const square *s2)
{
    int i;
    for(i = 0; i < 4; ++i)
    {
        if (point_in_square(s1, s2)) return 1;
    }
    return 0;
}

int main(void)
{
    square a = {{0, 0}, 1};
    square b = {{2, 0}, 2};
    intersect(&a, &b) ? puts("Da") : puts("Ne");
    return 0;
}
```

4. Definirati strukturu za kompleksni broj. Napisati funkciju koja sortira niz takvih struktura po modulu kompleksnog broja.

```
typedef struct
{
    float Re, Im;
}complex;

int complexcmp(const void* a, const void* b)
{
    complex k1 = *(complex*)a;
    complex k2 = *(complex*)b;
    float mod1 = k1.Re * k1.Re + k1.Im * k1.Im;
    float mod2 = k2.Re * k2.Re + k2.Im * k2.Im;
    if ( mod1 < mod2 ) return -1;
    if ( mod1 == mod2 ) return 0;
    return 1;
}

void sort(complex* arr, unsigned n)
{
    qsort(arr, n, sizeof (complex), complexcmp);
}

#define COUNTOF(x) (sizeof(x)/sizeof((x)[0]))
int main(void)
{
    int i;
    complex arr[] = {{1, 2}, {5, 2}, {1, 6}, {-2,
       3}};
    sort(arr, COUNTOF(arr));
    for (i = 0; i < COUNTOF(arr); i++)
    {
        float mod1 = sqrt(arr[i].Re * arr[i].Re + arr[
           i].Im * arr[i].Im);
        printf("%f %f %f\n", arr[i].Re, arr[i].Im,
           mod1);
    }
    return 0;
}
```

5. Napisati program u kojem je definirana struktura koja sadrži id (identifikacijski broj), naziv izvođača i naziv CD-a.

```
typedef struct cd_s cd;
```

U `main`-u se korisnika pita koliko bi htio upisati CD-ova, a zatim se dinamički alocira potrebna memorija za upis tolikog broja CD-ova. Nakon toga korisnik upisuje podatke. Napisati posebnu funkciju za upis pojedinog diska:

```
void unesi_cd(cd* k );
```

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

struct cds
{
    char naziv[100];
    char izvodjac[100];
    int id;
};

typedef struct cds cd;

void unesi_cd(cd* k)
{
    scanf("%s %s %d", k->naziv, k->izvodjac, &k->id);
}

int main(void)
{
    int n, i;
    cd* arr;
    puts("koliko cdova");
    scanf("%d", &n);
    arr = (cd* )malloc(n * sizeof(cd));
    for (i = 0; i < n; i++)
        unesi_cd(&arr[i]);
    for (i = 0; i < n; i++)
        printf("%s %s %d\n", arr[i].naziv, arr[i].
            izvodjac, arr[i].id);
    free(arr);
    return 0;
}
```

6. Predložiti strukturu peterokut, te funkciju koja provjerava jesu li svi vrhovi

peterokuta u istom kvadrantu koordinatnog sustava.

```
#define N_VERTEX 5
typedef struct{double x,y;}point;
typedef struct{point lik[N_VERTEX]}polygon;
void input_polygon(polygon* p, int n)
{
    int i;
    for (i = 0; i < n; i++)
        scanf("%lf %lf", &p->lik[i].x, &p->lik[i].y);
}
void print_polygon(polygon* p, int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%lf %lf\t", p->lik[i].x, p->lik[i].y);
}
int sign(double k)
{
    if (k < 0) return -1;
    if (k > 0) return 1;
    return 0;
}

int same_quadrant(polygon* p, int n)
/*ako su vrhovi u istom kvadrantu, to znaci da sve x
   koordinate i sve y koordinate imaju isti predznak
   */
{
    int i;
    for (i = 0; i < n - 1; i++)
        if ((sign(p->lik[i].x) != sign(p->lik[i + 1].x
           )) || (sign(p->lik[i].y) != sign(p->lik[i +
            1].y)) )
            return 0;
    return 1;
}
int main(void)
{
    polygon nt;
    puts("unesi peteroku");
    input_polygon(&nt, N_VERTEX);
    print_polygon(&nt, N_VERTEX);
```

```
        same_quadrant(&nt, N_VERTEX) ? puts("jesu") :
            puts("nisu");
        return 0;
    }
```

7. Predložiti strukturu kugla, te napisati funkciju koja računa oplošje i volumen kugle.

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

typedef struct{double r;} sphere;

#define PI 3.141592
void calculate(sphere* s, double* v, double* o)
{
    *v = 4./3*pow(s->r, 3)*PI;
    *o = 4.*pow(s->r, 2)*PI;
}
int main(void)
{
    sphere s;
    double v, o;
    puts("unesi radijus");
    scanf("%lf", &s.r);
    calculate(&s, &v, &o);
    printf("volumen %lf, oplosje %lf\n", v, o);
    return 0;
}
```

8. Napišite strukturu koja predstavlja geometrijsko tijelo valjak. Napisati funkciju koja za niz valjaka vraća valjak koji ima najveći volumen.

```
#include <stdio.h>
#define PI 3.141592
typedef struct
{
    double r, h;
}cylinder;

double volume_cyl(cylinder* cyl)
```

```
{
    return cyl->r * cyl->r * PI * cyl->h;
}

void find_max_vol(cylinder* carr, int size, cylinder*
     cmax)
{
    int i;
    double max_vol = volume_cyl(&carr[0]);
    *cmax = carr[0];
    for (i = 1; i < size; i++)
        if (volume_cyl(&carr[i]) > max_vol)
            *cmax = carr[i];
}

int main(void)
{
    cylinder carr[] =
       {{1.4,5},{1.2,4},{1.7,3},{1,6}};
    cylinder cmax;
    find_max_vol (carr, sizeof(carr)/sizeof(carr[0]),
        &cmax);
    printf("najveci valjak je valjak radijusa %4.2f i
        visine %4.2f\n", cmax.r, cmax.h);
    return 0;
}
```

9. Predložiti strukturu *stack* i implementirati dodavanje na *stack* i skidanje sa stacka (push i pop metode). Koristeći se *stackom* napisati funkciju za ispis binarnog zapisa broja.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_STACK_SIZE 32
typedef struct
{
    unsigned item[MAX_STACK_SIZE], current_size;
}stack;

void push(stack* s, unsigned x)
{
    if (s->current_size == MAX_STACK_SIZE)
```

```
        {
            puts("stack overflow");
            return;
        }
        s->item[s->current_size++] = x;
    }

    unsigned pop(stack* s)
    {
        s->current_size--;
        if (s->current_size < 0)
        {
            puts("stack underflow");
            return 0;
        }
        return s->item[s->current_size];
    }

    void print_bin(unsigned n)
    {
        stack s;
        while (n)
        {
            push(&s, n%2);
            printf("pusham %u, velicina steka %u\n", n%2,
                s.current_size);
            n /= 2;
        }
        puts("ispis:");
        while(s.current_size)
        {
            unsigned k = pop(&s);
            printf("popam %u, velicina steka %u\n", k, s.
                current_size);
        }
    }
    int main()
    {
        unsigned n;
        puts("unesi pozitivan cijeli broj");
        scanf("%u", &n);
        print_bin(n);
        return 0;
    }
```

8.8 Zadaci za vježbanje

1. Napisati funkciju koja učitava binarnu datoteku. Argumenti funkcije su ime datoteke i žanr knjige. Datoteka sadrži podatke tipa:

```
struct osoba{
        char naslov[1024]:
        char ime_autora[60]:
        char prezime_autora[60];
        char zanr_knjige[60];
};
```

   Funkcija ispisuje naslov, ime i prezime autora svih knjiga koje spadaju u zadani žanr.
2. Definirajte strukturu razlomak koja sadrži dvije varijable brojnik i nazivnik.
   - Napisati funkciju koja za zadani razlomak obavlja njegovo skraćivanje. Uputa:
     Razlomak se skraćuje tako da se i brojnik i nazivnik podijele sa njihovim najvećim zajedničkim djeliteljem. Napisati i funkciju nzd koja pronalazi najveći zajednički djelitelj dvaju brojeva.
   - Napisati funkciju koja omogućuje unos razlomaka s tipkovnice:
     – korisnik unosi podatak koliko razlomaka namjerava unijeti,
     – dinamički alocirajte memoriju za unos zadanog broja razlomaka,
     – omogućite korisniku unos zadanog broja podataka.

     Funkcija vraća pokazivač na novo-alocirani i inicijalizirani niz.
3. Predložite strukturu pravac koja opisuje pravac u dvije dimenzije i napišite funkciju koja provjerava leći li točka na pravcu.
4. Defirajte strukturu tocka koja sadrži dvije koordinate x i y i strukturu duzina koja se sastoji od dviju točki.
   - Napisati funkciju koja izračunava duljinu dužine.
   - Napisati funkciju koja će za zadani broj elemenata u nizu i zadanu minimalnu duljinu dužine alocirati i inicijalizirati niz dužina čije su duljine veće od zadane.

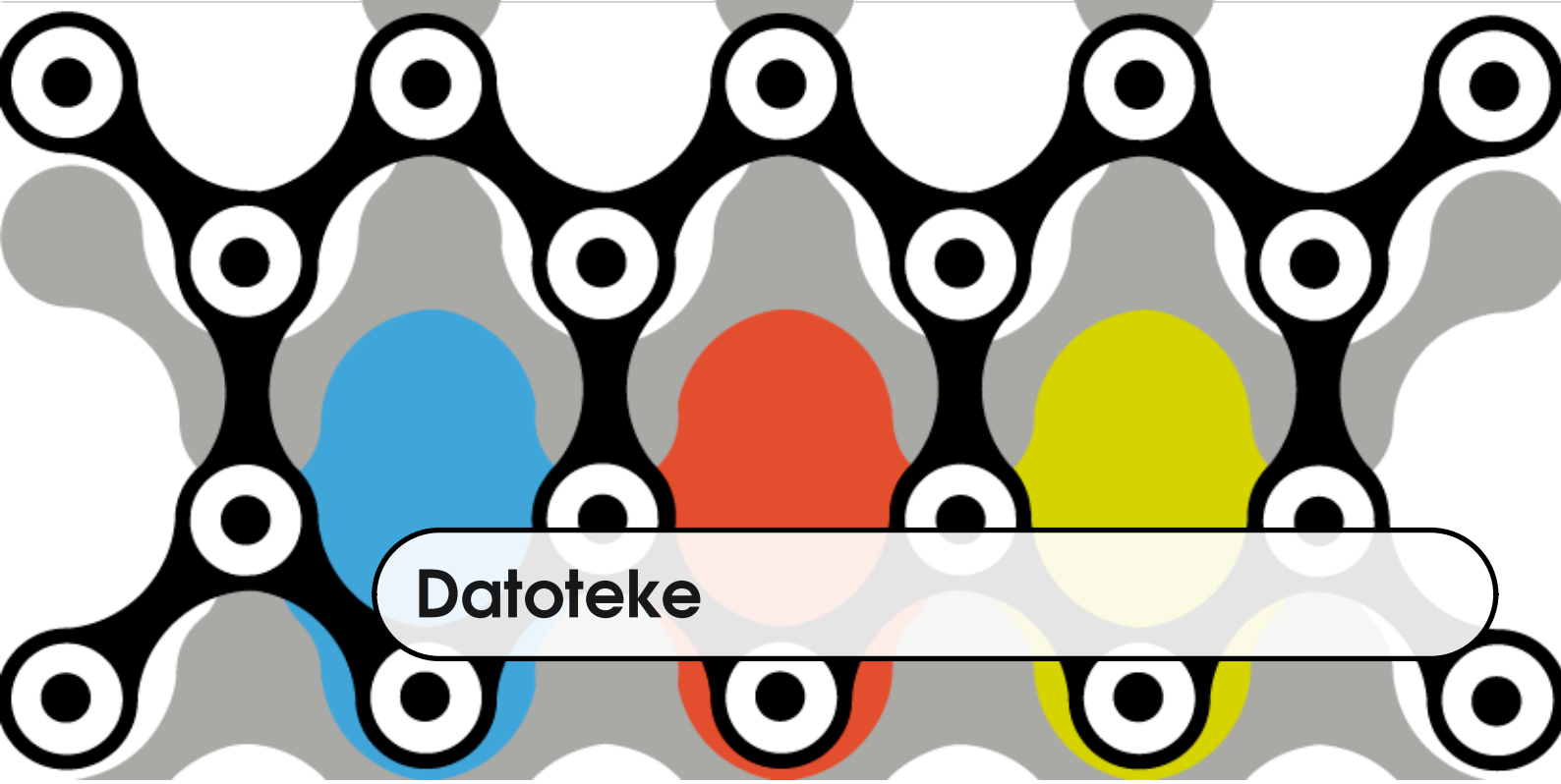
   Uputa:

- elementi niza su tipa točke, memoriju alocirajte dinamički;
- pomoću rand funkcije inicijalizirajte koordinate točaka jedne dužine (deklarirajte i popunite pomoćnu varijablu temp tipa točke);
- ako je duljina dužine zadane temp varijablom veća od minimalne veličine dužine, spremite je u niz, ako ne odbacite;
- ponavljajte postupak sve dok ne popunite cijeli niz zadovoljavajućim vrijednostima;
- vratite pokazivač na početak novoalocirane memorije.

5. Napisati strukturu kvadar. Napisati funkciju koja računa oplošje i volumen kvadra.
6. Predložiti strukturu za vrijeme. Napisati funkciju koja vraća razliku vremena. Ispisati tu razliku u formatu hh:mm:ss.
7. Napisati funkciju koja za niz pravokutnika zadanih sa dvije stranice, vraća one koji imaju najmanju i najveću površinu.
8. Definirati strukturu `slika` koja opisuje sliku sa imenom i prezimenom autora, imenom slike, dimenzijom i cijenom. Napisati funkciju koja za niz slika pročitanih iz datoteke, vraća ime i prezime najzastupljenijeg autora u kolekciji.
9. Definirati strukturu seminar. Seminar se sastoji od imena autora, imena seminara, godine izdavanja, ocjena i datoteke u kojoj se nalazi sam rad. Napisati funkciju koja za niz seminara nalazi (i vraća) one čije u čijem je naslovu određena riječ (riječ je parametar funkcije).
   Napomena: Funkcija može vratiti više rezultata.
10. Definirati strukturu završni rad. Završni rad se sastoji od imena autora, imena završnog rada, godine izdavanja, ocjena i datoteke u kojoj se nalazi sam rad. Napisati funkciju koja za niz završnih radova nalazi (i vraća) one koji su dobili određenu ocjenu (ocjena je parametar funkcije).
    Napomena: Funkcija može vratiti više rezultata.
11. Definirati strukturu trokut. Napisati funkciju koja za niz trokuta računa i vraća trokut najvećeg opsega.
12. Definirati strukturu trokut. Napisati funkciju koja za niz trokuta računa i

vraća trokut najveće površine i trokut najvećeg opsega.

13. Definirati strukturu kvadrat. Napisati funkciju koja za niz kvadrata računa i vraća kvadrat najveće površine i kvadrat najvećeg opsega.
14. Definirati strukturu trokut. Napisati funkciju koja za niz trokuta vraća najveću stranicu među svim trokutima.
15. Definirati strukturu datum koja predstavlja datum. Napisati funkciju za unošenje niza datuma, koja provjerava da li je datum korektno unesen, te vraća broj korektno unesenih, te broj pojavljivanja datuma iz ljetnih mjeseci (lipanj, srpanj i kolovoz).
16. Napisati strukturu koja opisuje kvadar. Napisati funkciju koja za niz kvadara vraća one koji imaju najmanji volumen.
    Napomena: Dva ili više kvadara mogu imati jednaki volumen.
17. Napisati funkciju koja za radijus kružnice računa i vraća površinu i opseg kružnice.
18. Definirati strukturu razlomak, te napisati funkcije za 4 osnovne operacije sa razlomcima.
    Napomena: razlomak koji funkcija vraća mora biti skraćen.
19. Definirati strukturu stol. Stol je određen sa svojom duljinom, širinom i visinom. Napisati funkciju koja za niz stolova vraća one stolove koji imaju najveću pisaću površinu.
    Napomena: Funkcija treba, osim niza stolova, vratiti i duljinu novonastalog niza stolova.



Datoteke

9.1 Ulazno izlazni tok

Programi koje smo do sada izvodili omogućavali su korisniku unošenje vrijednosti pomoću tipkovnice, te prikazivanje rezultata na zaslonu. Takav ulaz-izlaz zovemo konzolnim. Jasno da, kada imamo veliki broj ulaznih podataka, konzolni unos nije praktičan. U takvim slučajevima želimo ulazne podatke dohvatiti iz datoteke, a isto tako možemo i izlazne podatke zapisati u datoteku ili datoteke. Programski jezik C tretira tipkovnicu, zaslon i datoteke na isti način i za rad s njima možemo upotrebljavati iste funkcije standardne biblioteke čiji su prototipovi dani u datoteci `stdio.h`.

Prije nego što se iz datoteke može nešto pročitati ili u nju zapisati, potrebno je s njom povezati strukturu podataka koju nazivamo ulazno-izlazni tok ili *stream*. STREAM *Stream* je neka vrsta međuspremnika između ulazno izlaznih uređaja ili uređaja za pohranu podataka i programa u memoriji. Naime, dohvaćanje ili zapisivanje podataka sa ili na disk je puno sporije od dohvaćanja i zapisivanja podataka u memoriju. Stoga bi bilo nezgodno kada bi za zapis svakog znaka trebalo pristupiti disku. Zato se znakovi zapisuju u međuspremnik, a onda se veća

količina podataka prenosi odjednom na sporije uređaje.

Datoteke se povezuju sa streamom pomoću funkcije `fopen`. Tipkovnica i zaslon povezuju se pomoću tri *streama*, `stdin, stdout, stderr`, koji se automatski kreiraju prilikom pokretanja programa.

STDIN

Standardni ulaz, `stdin`, povezan je sa tipkovnicom. Funkcije `scanf`, `gets`, `getchar` upotrebljava `stdin` za unos podataka.

STDOUT

Standardni izlaz, `stdout`, povezan je s zaslonom. Funkcije `printf`, `puts` i sl. upotrebljava `stdout` za konzolni ispis.

STDERR

Standardna greška, `stderr` povezana je također s konzolnim ispisom. Funkcija `perror` upotrebljava `stderr` za ispisivanje poruke o grešci. Iako su i `stdout` i `stderr` povezani sa istim uređajem, ako postoji potreba za razdvajanjem izlaza, jedan od izlaza se može preusmjeriti u datoteku. Program koji bi pokrenili iz komandne linije sa

```
./program
```

možemo pokreniti preusmjeravajući izlaz u neku datoteku.

```
./program > izlaz.txt
```

Ili možemo preusmjeriti standardnu grešku koristeći se redirekcijom izlaza za greške `2>`.

```
./program 2> err.txt
```

9.2 Otvaranje datoteke

Otvaranje datoteke znači povezivanje datoteke sa *streamom*. To se radi koristeći se standardnom funkcijom `fopen` čije je zaglavlje oblika

```
FILE* fopen(const char* name, const char* mode);
```

Povratna vrijednosti funkcije `fopen`

Povratna vrijednost funkcije je tipa `FILE*`, što je pokazivač na strukturu `FILE`. Struktura FILE predstavlja *stream* objekt i sadrži informacije o pridruženoj datoteci, primjerice trenutnu poziciju u datoteci i informacije o međuspremniku. Međutim, mi o sadržaju strukture FILE ne moramo znati ništa. Za čitanje i pisanje u datoteku dovoljno je da pokazivač, koji je povratna vrijednost funkcije `fopen` proslijedimo drugim funkcijama za rad s datotekama.

FILE

Svaki *stream* ima i indikatore statusa greške i kraja datoteke. Oznaka kraja datoteke je `EOF` čija je brojčana vrijednost negativan broj, koji u GNU biblioteci `glibc`, a i u najvećem broju standardnih biblioteka, iznosi `-1`.

U slučaju neuspjeha prilikom otvaranja datoteke, funkcija `fopen` vraća `NULL`. Prije daljnje manipulacije datotekom, bitno je provjeriti je li datoteka uopće uspješno otvorena. Ako je došlo do greške, recimo uslijed nepostojanja datoteke ili navođenja krive putanje do datoteke koju želimo pročitati, možemo provjeriti zbog čega koristeći se funkcijom `perror`. Funkcija `perror` ispisuje poruku na `stderr` standardni izlaz za grešku opisujući posljednju grešku do koje je došlo prilikom korištenja standardnih funkcija.

perror

Parametri funkcije `fopen`

Parametri funkcije `fopen` predstavljaju ime datoteke i mod u kojem se datoteka otvara.

Prvi parametar funkcije `fopen` je string u kojem je zapisano ime datoteke. Ime datoteke može se pisati koristeći se relativnom ili apsolutnom putanjom do datoteke. Relativna putanja je putanja do datoteke krenuvši od tekućeg direktorija. Međutim, tekući direktorij razlikuje se ovisno o tome pokreće li se program iz

komandne linije ili iz nekog IDE programa. Ako program pokrećemo iz komandne linije, onda je tekući direktorij direktorij iz kojeg se program pokreće. Ako se program pokreće iz IDE programa, onda je tekući direktorij direktorij u kojem se nalazi `main` funkcija.

Linux/Unix operativni sustav Unixoidi upotrebljavaju znak / kao delimiter dva direktorija ili direktorija i datoteka u putanji. Primjer apsolutne putanje do neke datoteke je `/home/student/data/name/zad.txt`.

Windows operativni sustav koristi se znakom `'\'` kao delimiterom dva direktorija ili direktorija i datoteka u putanji. Naprimjer putanja do neke datoteke poprima oblik `c:\data\name\zad.txt`. Međutim, kako je znak `'\'` *escape* sekvenca (vidi tablicu 1.1), odnosno znak koji se tumači zajedno sa znakom koji dolazi nakon njega, to će se kombinacija znakova `'\n'` u navedenoj putanji tumačiti kao oznaka za novi red. Posljedica toga je da se pomoću poziva funkcije

```
fopen("c:\data\name\zad.txt", "r")
```

neće uspješno otvoriti datoteka s danim imenom. Problem se može riješiti na dva načina. Prvi, koji se i preporuča, je da se u putanji kao delimiter koristi znak `'\'`. Drugi je način da se umjesto jednog znaka `'\'` koristi dvostruki znak `'\\'`. Tada će poziv funkcije iz gornjeg primjera izgledati ovako:

```
fopen("c:\\data\\name\\zad.txt", "r")
```

MOD OTVARANJA DATOTEKE

Drugi parametar funkcije `fopen` je mod u kojem se datoteka otvara. Datoteka se može otvoriti za čitanje (`"r"`), pisanje (`"w"`) i dodavanje na kraju (`"a"`). Datoteka koja se otvara samo za čitanje mora postojati prije otvaranja. Datoteka koja se otvara samo za pisanje ili samo za dodavanje na kraj će se, ako ne postoji, kreirati prilikom otvaranja. Postojeći sadržaj datoteke otvorene samo za pisanje će se uništiti prilikom otvaranja datoteke.

Osim navedenih načina datoteke se mogu otvoriti sa oznakama `"r+"`, `"w+"` i `"a+"`. U ova sva tri slučaja omogućeno je i čitanje i pisanje uz određene razlike.

Mod `"r+"` znači da se datoteka može čitati i da se u nju može pisati, ali kad se piše ne uništava se prethodni sadržaj datoteke, već se znakovi ispisuju preko postojećih. Datoteka treba postojati prije otvaranja.

Mod `"w+"` znači da se datoteka može čitati i da se u nju može pisati, ali ako se piše u nju postojeći sadržaj datoteke će se izbrisati. Ako datoteka ne postoji, kreirat će se prilikom otvaranja. I, na kraju, mod `"a+"` znači da se datoteka može čitati i da joj se može dodavati sadržaj na kraj. Ako datoteka ne postoji, kreirat će se prilikom otvaranja.

Sve ovo možemo sažeti u tablici 9.1.

Tablica 9.1: Mod otvaranja datoteke

mod	opis
`"r"`	Čitanje. Datoteka mora postojati
`"w"`	Pisanje. Ako datoteka ne postoji, kreira se. Ako postoji, obriše joj se sadržaj.
`"a"`	Dodavanje na kraj. Ako datoteka ne postoji kreira se. Ako postoji, sadržaj se dodaje na kraj datoteke.
`"r+"`	Čitanje i pisanje. Datoteka mora postojati. Sadržaj se ne uništava prilikom otvaranja.
`"w+"`	Čitanje i pisanje. Ako datoteka ne postoji, kreira se. Ako postoji, njen sadržaj se uništava.
`"a+"`	Čitanje i dodavanje na kraj. Ako datoteka ne postoji, kreira se. Ako postoji, sadržaj se dodaje na kraj.

Primjer 9.1 Otvaranje datoteke `ime.txt` za čitanje. U slučaju neuspjeha ispisuje se poruka.

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

```c
int main(void)
{
    FILE* fp = fopen("ime.txt", "r");
    if (!fp)
    {
        perror("error: ");
        return 1;
    }
    fclose(fp);
    return 0;
}
```

9.2.1 Tekstualne i binarne datoteke

Osim znakova koji određuju da li će se datoteke čitati ili mijenjati, u stringu `mode` može se još napisati i znak `'t'` ili `'b'` koji označava hoće li se datoteke čitati/pisati u tekstualnom ili binarnom modu. Po ANSI standardu [5] postoje dvije vrste datoteka tj. dvije vrste *streamova*, tekstualni (ASCII) i binarni .

TEKSTUALNE DATOTEKE
BINARNE DATOTEKE

Tekstualna datoteka je datoteka u koju se sprema ASCII kôd znakova. Primjerice, niz znakova `"PMA"` pohranjen je kao niz brojeva 80, 77, 65, tj. kao niz bajtova `01010000`, `01001101` i `01000001`. Tekstualne datoteke mogu se editirati nekim od tekstualnih editora tipa Notepad, Kate, Gedit. U binarnu datoteku se zapisuju bilo kakve vrijednosti i to u svom binarnom obliku. Primjer takvih datoteka su izvršne datoteke, slike, zvučni zapisi i sl. Ako njih pokušamo otvoriti pomoću tekstualnih editora, dobit ćemo najčešće neke nerazumljive znakove. Iako tekstualne datoteke možemo smatrati vrstom binarnih datoteka, i to takvih datoteka koje sadrže samo vrijednosti ASCII kôda (`0` do `127`).

Binarne datoteke su kompaktnije. Ako želimo zapisati broj `1000000000`, zapisujući taj broj kao niz znakova u tekstualnu datoteku potrošit ćemo `10` bajtova. Zapisujući taj broj u binarnu datoteku, potrošit ćemo onoliko prostora koliko taj broj zauzima u memoriji, recimo `4` bajta za `int` vrijednost.

Tipično su tekstualne datoteke organizirane u linije. Svaka linija se sastoji od

nula ili više znakova i oznake za kraj reda. Oznaka za kraj reda se drukčije implementira na Linux i Windows operacijskim sustavima. Na Windows operacijskim sustavima prilikom zapisivanja podataka na disk, oznaka za kraj retka `'\n'` se kovertira u dva znaka `'\r\n'`, dok se na Linux/Unix operacijskim sustavima ta konverzija ne događakonverzija oznake za novi red. Ta dva znaka imaju naziv *carriage return* ili CR i *line feed* ili LF i njihove ASCII vrijednosti su `13` (ili `0x0D` u `heksadecimalnom zapisu`) i `10`(`0x0A`).

KONVERZIJA OZNAKE ZA NOVI RED

Carriage return i *line feed* kombinacija nastali su u vrijeme teleprintera koji su, po uzoru na pisaće mašine, imali mogućnost vraćanja na početak reda (*carriage return*) kako bi se pisalo preko već napisanih znakova. Osim toga, pozicioniranje na početak reda u sljedećem retku zahtijevalo je dvije operacije kako bi se glava printera imala vremena pozicionirati na početak sljedećeg reda. MS-DOS je preuzeo ovaj način prikaza prelaska u novi red, a Windows operacijski sutavi su naslijedili MS-DOS format [2].

Linux/Unix za oznaku za kraj retka nema konverziju, odnosno koristi se samo jednim znakom, *line feed* (LF). Dakle, datoteke zapisuje na jedinstven način, znak po znak, tj. byte po byte, pa ustvari ne radi razliku između načina zapisivanja tekstualne i binarne datoteke.

Bez obzira na to o kojem se operacijskom sustavu radi, funkcije za rad s datotekama prepoznaju oznaku za kraj retka u datoteci, bilo da je ona `"\r\n"` ili samo `'\n'` i pretvore ga u jedan znak `'\n'`.

9.3 Funkcije za pisanje/čitanje znak po znak

Standardne funkcije za pisanje/čitanje znak po znak su `fputc` i `fgetc`. Njihovi prototipovi su:

```
#include <stdio.h>
int fputc(int c, FILE *stream);
int fgetc(FILE *stream);
```

fputc

Funkcija `fputc` zapisuje znak `c` na izlazni *stream*. Iako je prvi parametar deklariran kao `int`, on se konvertira u `unsigned char`, te se kao znak zapisuje na *stream*. Povratna vrijednost funkcije `fputc` bit će vrijednost koja je zapisana, u slučaju uspješnog zapisa, ili `EOF` u slučaju neuspjeha.

fgetc

S druge strane, funkcija `fgetc` uzima sljedeći znak (`unsigned char`) iz ulaznog *streama* i pretvara ga u `int` vrijednost. U slučaju neuspjeha vraća `EOF`.

EOF

`EOF` je predefinirana vrijednost koja označava kraj datoteke, tj. *streama*. Sama vrijednost ovisi o implementaciji, ali nije jednaka nijednoj znakovnoj vrijednosti, jer ne smije biti jednaka nijednom validnom znaku u datoteci. Definirana je u datoteci `stdio.h` i najčešće iznosi `-1`. Kako je vrijednost EOF tipa `int`, tako i povratna vrijednost funkcija `fgetc` i `fputc` mora biti `int`, te se zato vrši konverzija iz `unsigned char` vrijednosti u `int` za uspješno pročitane/zapisane znakove.

Čitanje ili pisanje znak po znak odvija se sekvencijalno. Prvi poziv funkcije `fgetc` pročitat će prvi znak u datoteci, sljedeći poziv pročitat će drugi znak i tako dalje sve dok funkcija ne pročita znak oznaku za kraj datoteke `EOF`.

Primjer 9.2 Primjer čitanja iz datoteke znak po znak i ispisivanje znakova na `stdout`. Treba zapaziti da je izraz `ch = getc(fp)` u zagradama zbog višeg prioriteta operatora `!=` u odnosu na operator `=`.

```c
#include <stdio.h>

int main(void)
{
    char ch;
    FILE* fp = fopen("ime.txt", "r");
    if (!fp)
        return 1;

    while ((ch = fgetc(fp)) != EOF)
        fputc(ch, stdout);
    fclose(fp);
    return 0;
}
```

9.4 Funkcije za pisanje/čitanje red po red

Standardne funkcije za pisanje/čitanje red po red iz i u datoteku su `fputs` i `fgets`. Njihovi prototipovi su:

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
char *fgets(char *s, int n, FILE *stream);
```

Funkcija `fputs` zapisuje string na izlazni *stream*. Pritom oznaku za kraj stringa, tj. nul terminator **ne** zamijenjuje oznakom za novi red kao što radi funkcija `puts`. Povratna vrijednost funkcije je nenegativna vrijednost, u slučaju uspjeha, ili `EOF` ako dođe do greške.

Funkcija `fgets` čita znakove iz ulaznog `streama` do i uključujući oznaku za kraj reda. Drugi parametar označava najveći broj znakova koji će funkcija pročitati uvećan za `1`, jer će jedno mjesto pripasti nul terminatoru stringa. Pročitani znakovi smještaju se na adresu danu sa `s`, počevši od koje treba biti dovoljno mjesta za pročitane znakove, tj. prethodno treba alocirati prostor.

Povratna vrijednost funkcije `fgets` je pokazivač `s` ako je funkcija uspješno obavila posao ili NULL pokazivač ako se došlo do kraja datoteke ili u slučaju greške.

Primjer 9.3 Ispisivanje sadržaja datoteke na standardni izlaz.

```
#define N 256
#include <stdio.h>

int main(void)
{
    char str[N];
    FILE* fp = fopen("ime.txt", "r");
    if (!fp)
        return 1;
```

```
        while ((fgets(str, N, fp)) != NULL)
            fputs(str, stdout);
        fclose(fp);
        return 0;
    }
```

Funkcija `fgets`, za razliku od funkcije `gets`, zapisuje oznaku za novi red koji predstavlja kraj linije u string, te nakon nje zapisuje nul-terminator.

BUFFER OVERFLOW

Nadalje, kako funkcija ima ograničenje koliko znakova može pročitati, lakše je spriječiti prekoračenje veličine bafera (engl. *buffer overflow*). C baferom zovemo dio memorije koji je dodijeljen podacima istog tipa i alociran koristeći se C jezikom [3]. Primjerice, C bafer je `str` iz primjera 9.3. Taj je bafer velik 256 bajtova i predstavlja 256 uzastopnih memorijskih lokacija.

Kada bi u taj bafer zapisivali string pomoću funkcije `gets`, tj. očekujući od korisnika unos stringa, postoji mogućnost da korisnik unese niz znakova koji je duži od 256. U tom slučaju niz znakova neće imati oznaku nul-terminatora na kraju, a znakovi koji korisnik dodatno unese, zapisat će se u memoriju koja se nalazi nakon bafera i eventualno pregaziti vrijednosti drugih varijabli. To predstavlja slabost našeg programa i sigurnosnu rupu [7].

Upotrebljavajući `fgets`, ako proslijedimo ispravnu veličinu drugog parametra (koja ne smije biti veća od veličine alociranog bafera) osiguravamo da će se zadnji znak u stringu biti nul-terminator. Pritom treba voditi računa o tome da će pročitani string imati oznaku za kraj retka prije nul-terminatora.

Primjer 9.4 Ime datoteke se unosi sa standardnom ulaza pomoću funkcije `fgets`. Kako `fgets` u string smješta i oznaku za novi red, taj red se zamjenjuje znakom nul-terminatora.

```c
#define N 256
#include <stdio.h>

int main(void)
{
    char str[N], ime[N];
    FILE* fp;
    size_t len;

    /* cita se ime sa stand. ulaza i
     * izbacuje se oznaka za kraj retka '\n'
     */
    fgets(ime, N, stdin);
    len = strlen(ime);
    if (len > 0 && ime[len-1] == '\n')
        ime[len-1] = '\0';


    fp = fopen(ime, "r");
    if (!fp)
        return 1;

    while ((fgets(str, N, fp)) != NULL)
        fputs(str, stdout);
    fclose(fp);
    return 0;
}
```

9.5 Funkcije za formatirano čitanje i pisanje

Slično funkcijama `scanf` i `printf`, funkcije `fscanf` i `fprintf` čitaju sa *streama* i pišu na *stream* u formatiranom obliku. Funkcije `scanf` i `printf` pritom se uvijek koriste standardnim ulazom i izlazom, a funkcije `fscanf` i `fprintf` mogu upotrebljavati bilo kakve datoteke.

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format,...);
int fprintf(FILE *stream, const char *format,...);
```

Funkcije `fscanf` i `fprintf` su funkcije sa varijabilnim brojem parametara. Broj parametara ovisi o broju pojavljivanja znakova `%-tag` u format stringu. Format string je string u kojem je naveden tekst koji će se pohraniti ili čitati iz *streama* kombiniran sa `%-tag` tagovima. Funkcija `fprintf` vraća broj zapisanih znakova, a u slučaju greške vraća negativan broj. Funkcija `fscanf` vraća broj pročitanih znakova, a u slučaju greške vraća `EOF`.

Funkcije za formatirano čitanje i pisanje u datoteku koriste se kada nam je poznat format u kojem se podaci trebaju zapisati u datoteku ili pročitati iz datoteke.

Primjer 9.5 Dana je *comma separated value* datoteka `test.csv` u kojoj su u svakom retku zapisani podaci o knjizi. Redak datoteke čine podaci: redni broj, prezime autora, naslov knjige i kolegij u kojem se knjiga koristi. Podaci o knjizi odvojeni su zarezom, a svaka knjiga je u svom retku.

test.csv

```
1, Kernighan-Ritchie, The C Programming Language, PMA
2, Lippman; Lajoije; Moo, C++ Primer, OOP
3, Roberts, Programming Abstractions in C, PMA
4, Tanenbaum, Modern Operating Systems, OS
```

Program čita redove iz datoteke i zapisuje ih u niz struktura koje opisuju knjigu u funkciji `read_books`. Nakon toga zapisuje knjige u datoteku čije ime unosi korisnik i to tako da na prvo mjesto u retku stavi kolegij, prezime autora, naslov knjige i to sortirano po kolegijima.

```
#define N 256
#include <stdio.h>
#include <stdlib.h>

typedef struct
{
    unsigned no;
    char author_name[N];
    char book_name[N];
    char subject[N];
}book;

book* read_books(FILE* fp, int sz, int* num_books)
{
    int i = 0;
    book* arr = (book*) malloc(sz * sizeof(book));
    if (!arr) return NULL;

    while (fscanf(fp, "%d, %[^,],%[^,],%[^,\n]", \
        &arr[i].no, arr[i].author_name, arr[i].book_name, \

        arr[i].subject ) != EOF)
    {
        if (++i > sz)
        {
            sz *= 2;
            arr = (book*) realloc(arr, sz * sizeof(book));
            if (!arr) return NULL;
        }
    }
    *num_books = i;
    return arr;
}

void write_books(FILE* fp, book* arr, int n)
{
    int i;
    for(i = 0; i < n; ++i)
    {
        fprintf(fp, "%s, %s, %s\n", arr[i].subject, \
            arr[i].author_name, arr[i].book_name);
    }
}
```

```
int book_cmp(const void* p, const void* q)
{
    book* b1 = (book*)p;
    book* b2 = (book*)q;
    printf("%s - %s\n", b1->subject, b2->author_name);
    return strcmp(b1->subject, b2->subject);
}

int main(void)
{
    FILE* fp1, *fp2;
    int n, len;
    book* arr = NULL;
    char filename[N];

    fp1 = fopen("test.csv", "r");
    if (!fp1)
        return 1;
    puts("filename?");
    fgets(filename, N, stdin);
    len = strlen(filename);
    if (len > 0 && filename[len-1] == '\n')
        filename[len-1] = '\0';
    fp2 = fopen(filename, "w");
    if (!fp2)
    {
        fclose(fp1);
        return 1;
    }

    arr = read_books(fp1, 5, &n);
    qsort(arr, n, sizeof(book), book_cmp);
    write_books(fp2, arr, n);
    fclose(fp1);
    fclose(fp2);
    free(arr);
    return 0;
}
```

9.6 Funkcije za ispitivanje i mijenjanje pozicije u datoteci

Do sada su svi naši primjeri radili tako da se datoteka čita ili piše slijedno. No, nekada nas zanima podatak na točno određenom mjestu u datoteci. Recimo želimo provjeriti nalazi li se u datoteci određeni znak na određenoj poziciji. To nam omogućavaju funkcije za mijenjanje pozicije u datoteci.

Naime, prilikom otvaranja datoteke i pridruživanja datoteke *streamu* jedan od podataka koji se vezuju uz datoteku je i podatak o trenutnoj poziciji (engl. *current position*) u datoteci. Taj podatak predstavlja broj bajtova od početka datoteke. Kad otvorimo datoteku trenutna pozicija je `0`, jer nije ni pročitan ni zapisan nijedan podatak u nju putem *streama*. Kada zapišemo ili pročitamo jedan znak, trenutna pozicija postaje `1`. Ako pročitamo još 5 znakova trenutna pozicija bit će 6. Ako pročitamo ili zapišemo oznaku za novi red trenutnoj poziciji dodat će se `1` (Linux/Unix) ili `2` (Windows). Kad dođemo do kraja datoteke trenutna pozicija bit će upravo veličina datoteke.

CURRENT POSITION

Funkcija koja u svakom trenutku može vratiti trenutnu poziciju je `ftell`. Njen prototip je sljedeći:

`ftell`

```
#include <stdio.h>
long ftell(FILE *stream);
```

Primjer 9.6 Program čita datoteku. Funkcijom `ftell` treba ispitati ukupni broj zapisanih znakova. Testirajte program na maloj datoteci koja se sastoji od dva retka i provjerite odgovara li broj pročitanih bajtova broju znakova koji su zapisani u datoteku.

```
#include <stdio.h>

int main(void)
```

```
{
    FILE* fp;
    int ch;


    fp = fopen("mala.txt", "r");
    if (!fp)
        return 1;

    while((ch=fgetc(fp)) != EOF)
        ;
    printf("%u\n", ftell(fp));

    fclose(fp);
    return 0;
}
```

U primjeru 9.6 smo, da bi došli do kraja datoteke, čitali datoteku znak po znak. Čitanje datoteke je dodatni posao koji smo mogli izbjeći koristeći se funkcijom `fseek` koja mijenja tekuću poziciju u datoteci. Njen prototip je:

fseek

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int origin);
```

Parametar `origin` mora biti jedan od sljedeće tri vrijednosti definirane u `stdio.h`

- `SEEK_SET` - početak datoteke
- `SEEK_CUR` - trenutna pozicija u datoteci
- `SEEK_END` - kraj datoteke.

U odnosu na `origin`, `offset` predstavlja pomak. Očito je da pomak mora biti nenegativan ako je `origin` jednak `SEEK_SET`, te da mora biti negativan ako je `origin` jednak `SEEK_END`. Inače, važno je da se pomakom ne pređe kraj datoteke ili se pokuša pozicionirati ispred početka datoteke. Od nove pozicije nastavit će se čitanje ili pisanje u datoteku. U tekstualnim datotekama `offset` treba biti nula

ili vrijednost koju je vratila funkcija `ftell` ako je `origin` jednak `SEEK_SET` [12].

Funkcija `fseek` vraća nulu u slučaju uspješnog pozicioniranja, a ne-nulu ako je došlo do greške.

9.7 Funkcije za testiranje grešaka

Kada se upotrebom funkcije za rad s datotekama dođe do greške, postavlja se *error indicator*. Pomoću funkcije `ferror` možemo doznati o kojem se tipu greške radi. Funkcija će vratiti broj greške, a opis greške možemo dobiti koristeći se funkcijom `perror`. Prototipovi funkcija su sljedeći:

ERROR INDICATOR

```
int ferror(FILE *stream);
void perror(const char *s);
```

Ako je došlo do greške funkcija `ferror` će vratiti nenegativan broj koji označava vrstu greške. Funkcija `perror` ispisat će string dan parametrom `s` i poruku o greški.

Osim toga postoji i funkcija koja će detektirati da je trenutna pozicija kraj datoteke. To je funkcija `feof` opisana sljedećim prototipom:

```
int feof(FILE *stream);
```

Funkcija `feof` vraća nenegativnu vrijednost ako je trenutna pozicija `EOF`. Funkciju `feof` ne bi trebali q u petljama kao indikator izlaska iz petlje. Naime, povratna vrijednost funkcije `feof` je nenegativna tek nakon što je pročitana `EOF` vrijednost.

Primjer 9.7 Funkcija `feof` kao indikator izlaska iz petlje. Neka je dana datoteka `mala.txt` sljedećeg izgleda:

mala.txt

```
abc
def
```

```c
#include <stdio.h>
#define N 256
int main(void)
{
    FILE* fp;
    char str[N];

    fp = fopen("mala.txt", "r");
    if (!fp)
        return 1;

    while(!feof(fp))
    {
        fgets(str, N, fp);
        puts(str);
    }

    fclose(fp);
    return 0;
}
```

Rezultat izvođenja ovog programa je neočekivan. Naime, zadnji redak je pročitan dva puta.

```
abc

def

def
```

9.8 Funkcije za čitanje/pisanje blokova

Do sada smo razmatrali funkcije koje čitaju i pišu podatke u formatiranom ASCII obliku. *Stream* je bio niz znakova koje su interpretirale funkcije za čitanje i pisanje. Takav način je dobar za datoteke koje predstavljaju nekakav ASCII tekst, međutim za složenije strukture podataka koje želimo pohraniti u datoteku dosad spomenute funkcije nisu odgovarajuće.

Datoteke čiji zapis točno odgovara zapisu u memoriji su binarne datoteke i ne mogu se čitati običnim tekst editorima, već specijalnim programima koji znaju otvoriti baš taj određeni tip datoteke. Ako želimo pogledati sadržaj datoteke, možemo upotrebljavati programe za prikazivanje u heksadecimalnom obliku, primjerice `hexdump` ili `xxd` program. Primjerice, datoteka koja predstavlja sliku može se otvoriti programom za editiranje ili pregledavanje slika, a programom `hexdump` dobit ćemo zapis u sljedećem obliku (prikazan je samo dio podataka):

```
hexdump -C image.bmp

00000000 42 4d 3e 20 00 00 00 00 00 00 3e 00 00 00 28 00
00000010 00 00 00 01 00 00 00 01 00 00 01 00 01 00 00 00
00000020 00 00 00 20 00 00 00 00 00 00 00 00 00 00 00 00
00000030 00 00 00 00 00 00 00 00 00 00 ff ff ff 00 ff ff
00000040 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
*
000007c0 ff ff ff ff ff ff ff ff ff ff ff ff ff ff fe 07
000007d0 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
000007e0 ff ff ff ff ff ff ff ff ff ff ff ff ff ff f8 07
000007f0 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
00000800 ff ff ff ff ff ff ff ff ff ff ff ff ff ff f0 03
```

Gornji primjer je zapis monokromatske `bmp` datoteke. Da bi mogli nešto s njom napraviti moramo znati format u kojem su podaci u njoj zapisani. Na stranicama [1] opisan je format takvih datoteka. Prvih 62 bajta predstavlja zaglavlje (engl. *header*) datoteke, a nakon toga počinje sam zapis slike. Pogledajmo malo gornje zaglavlje.

Prva dva bajta `42` i `4d` su ASCII vrijednosti slova `'B'` i `'M'`. To su vrijednosti koje se nalaze na početku svake `.bmp` datoteke. Nakon toga je zapisana veličina slike u sljedeća četiri bajta i ona iznosi `3e20` tj. `8912`. Prvih `14` bajtova je zaglavlje datoteke, a nakon toga kreće zaglavlje same slike. U tom dijelu prva četiri bajta su veličina zaglavlja, sljedeća četiri su širina slike, a nakon toga sljedeća četiri visina slike. U gornjem primjeru vidimo da je veličina zaglavlja `0x00000028`, a širina i visina slike `0x00000100` tj. `256`.

Napomena Format zapisa podataka koji se sastoje od više bajtova je little-endian. To znači da je na najnižoj adresi najmanje značajni bajt (engl. *least significant byte*), a na najvišoj najznačajniji bajt (engl. *most significant byte*). Zato je broj `0x00000028` zapisan kao `28 00 00 00`. Suprotno od toga, zapis u kojem se bajtovi nižu od najznačajnijeg prema najmanje značajnom zove se big-endian. Način na koji će se podaci zapisivati u memoriju ovisi o procesoru. Primjerice, Intel procesori koriste se little-endian načinom zapisivanja podataka.

LITTLE-ENDIAN

BIG-ENDIAN

Funkcije koje omogućavaju zapis blokova iz memorije u datoteku i obratno su `fwrite` i `fread`. Njihovi prototipovi dani su sa:

```
size_t fread(void* buffer, size_t size,
             size_t count, FILE* stream);
size_t fwrite(const void* buffer, size_t size,
             size_t count, FILE* stream);
```

`fread` Funkcija `fread` čita `count` podataka, svaki veličine `size` bajta sa *streama* počevši od trenutne pozicije u datoteci i smješta ih na određenu adresu u memoriji. Povratna vrijednost funkcije `fread` je broj pročiatnih podataka. Ako je broj pročitanih podataka manji od `count`, čitanje je bilo neuspješno i indikator greške je postavljen.

`fwrite` S druge strane, funkcija `fwrite` zapisuje blokove iz memorije u datoteku. Blokovi počinju na adresi danoj sa `buffer`, ima ih `count`, a svaki je veličine *size* bajta. Blokovi će se zapisati u datoteku na trenutnu poziciju. Funkcija vraća broj uspješno zapisanih blokova, a broj manji od `count` u slučaju neuspjeha.

Primjer 9.8 Dana je monokromatska `bmp` slika. Program provjerava da li su na prva dva bajta zapisane potrebne vrijednosti, te ispisuje veličinu slike u pixelima.

```
#include <stdio.h>

int get_dimension(FILE* f, unsigned* pl, unsigned* ph)
{
    char buf[2], bmp[] = "BM";
    fread(buf, 1, 2, f);
    if (memcmp(buf, bmp, 2)) return 1;

    if(fseek(f, 18, SEEK_SET))
        return 2;
    /* podaci o duljini i sirini pocinju na 18 bajtu */
    if(fread(pl, 4, 1, f) != 1)
        return 3;

    if(fread(ph, 4, 1, f) != 1)
        return 3;

    return 0;
}

int main(void)
{
    FILE *fp;
    unsigned length, height;
    int ret;

    fp = fopen("slika.bmp", "rb");
    if(!fp)
    {
        perror("");
        return 1;
    }
    ret = get_dimension(fp, &length, &height);
    if (ret)
        puts("not bmp");
    else
        printf("%u x %u\n", length, height);
    fclose(fp);
    return 0;
```

```
    }
```

9.9 Riješeni zadaci

1. Napisati funkciju koja za datoteku i niz znakova (string), na ekran ispiše one retke iz datoteke u kojima se nalazi taj niz znakova.
   Napomena: String je bilo kakav niz znakova.

```
void print_lines(FILE* fp, char* str)
{
    char red[1024];
    while( !feof(fp))
    {
        fgets(red, 1024, fp);
        if (strstr(red, str))
            printf("%s",red);
    }
}
```

2. Napisati funkciju koja za datoteku u kojoj je u svakom retku jedan broj i jedna riječ, ispisuje na ekran riječ onoliko puta koliki je broj.
   Primjer: Ako u datoteci piše:
   `4 psa`
   `2 tigra`
   `1 deva`
   ispisat će se
   `psapsapsapsa`
   `tigratigra`
   `deva`

```
void print(FILE* fp)
{
    int n, i;
    char str[1024];
    while (fscanf(fp, "%d %s", &n, str) != EOF)
    {
        for (i = 0; i < n; i++)
            printf("%s ", str);
        puts("");
```

```
        }
    }
```

3. Napisati funkciju za kopiranje datoteka. Imena datoteka su parametri funkcije, a funkcija vraća broj kopiranih znakova.

```
int ffcopy(const char* inname, const char* outname)
{
    FILE *in, *out;
    int zn, cnt = 0;
    int ret = 0;
    in = fopen(inname, "r");
    if(in == NULL)
        return 1;
    out = fopen(outname, "w");
    if(out == NULL)
    {
        fclose(in);
        return 0;
    }

    while ((zn = fgetc(in)) != EOF)
    {
        cnt++;
        fputc(zn, out);
    }

    if (ferror(in) || ferror(out))
        return 0;

    fclose(in);
    fclose(out);
    return cnt;
}
```

4. Napisati funkciju koja u datoteci svako pojavljivanje riječi "glava" zamijeni sa "trava". Ime datoteke je parametar funkcije.

```
void substitution(const char* filename)
{
    FILE* fp;
```

```
    char rijec[80];
    if (!(fp = fopen(filename, "r+t")))
    {
        perror("");
        return;
    }
    while (!feof(fp))
    {
        fscanf(fp, "%s", rijec);
        printf("%s", rijec);
        if (!strcmp(rijec, "glava"))
        {
            fseek(fp, -5, SEEK_CUR);
            fprintf(fp, "trava");
        }
    }

    fclose(fp);
}

int main(void)
{
    char name[] = "datoteka";
    substitution(name);
    return 0;

}
```

5. Napisati program koji iz jedne datoteke u drugu kopira sve redove u kojima se barem jednom pojavljuje slovo 'A'. Imena datoteka upisuje korisnik.

```
#include <stdio.h>
#include <string.h>
void copy_rows(FILE* fp1, FILE* fp2)
{
    char buff[80];
    while (fgets(buff, 80, fp))
    {
        if (strchr(buff, 'A'))
            fputs(buff, fp2);
    }
}
```

```
int main(void)
{
    FILE* fp1, fp2;
    char filename[100], filename2[100];
    scanf("%s", filename);
    scanf("%s", filename2);
    fp1 = fopen(filename, "r");
    fp2 = fopen(filename2, "w");
    if (!fp1) return;
    if (!fp2) { fclose(fp); return;}
    copy_rows(fp1, fp2);
    fclose(fp1);
    fclose(fp2);
    return 0;
}
```

6. Napisati program koji iz datoteke briše sve redove u kojima se barem jednom pojavljuje slovo 'A'. Ime datoteke upisuje korisnik.

```
#include <stdio.h>
#include <string.h>
void del_rows(const char* filename)
{
    char buff[80];
    FILE* fp = fopen(filename, "r");
    FILE* fp_temp = fopen("temp", "w");
    if (!fp) return;
    if (!fp_temp) { fclose(fp); return;}
    while (fgets(buff, 80, fp))
    {
        if (!strchr(buff, 'A'))
            fputs(buff, fp_temp);
    }
    remove(filename);
    rename("temp", filename);
    fclose(fp_temp);
}

int main(void)
{
    char filename[100];
```

```
    scanf("%s", filename);
    del_rows(filename);
    return 0;
}
```

7. Napisati funkciju koja za svaka dva broja u datoteci njihovu sumu zapisuje u drugu datoteku. Imena datoteka su parametri funkcije.

```
void sums(const char* namein, const char* nameout)
{
    FILE* fp1, *fp2;
    int m, n;
    fp1 = fopen(namein, "r");
    if (!fp1) return;
    fp2 = fopen(nameout, "w");
    if (!fp2) {fclose(fp1); return;}
    while(!feof(fp1))
    {
        fscanf(fp1, "%d %d", &m, &n);
        fprintf(fp2, "%d ", m + n);
        m = n = 0;
    }
}
int main(void)
{
    char n1[256], n2[256];
    puts("unesi imena datoteka");
    gets(n1);
    gets(n2);
    sums(n1, n2);
    return 0;

}
```

8. Zadana je datoteka koordinate.txt u kojoj se u svakom redu nalazi x, y i z koordinata točke. Napisati funkciju koja točke zapisuje u niz struktura. Napisati i strukturu.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct{ float x, y, z;} point;
```

```
point* read_points(point* parr, int* n)
{
    FILE* fp = fopen("koordinate.txt", "r");
    int i = 0;
    if (!fp) return;
    while (!feof(fp))
    {
        fscanf(fp, "%f %f %f", &parr[i].x, &parr[i].y, &
            parr[i].z);
        if (i == *n - 1)
        /* *n je maximalni broj elemenata niza*/
        {
            *n *= 2;
            parr = (point*)realloc(parr, *n * sizeof(
                point));
        }
        i++;
    }
    *n = i-1;
    return parr;
}
void write_points(const point* parr, int n)
/*test*/
{
    int i;
    for(i = 0; i < n; i++)
        printf("%f %f %f\n", parr[i].x, parr[i].y, parr[i
            ].z);
}
int main(void)
{
    int n = 5;
    point* parr = (point*) malloc(n * sizeof (point));
    parr = read_points(parr, &n);
    write_points(parr, n);
    free(parr);
    return 0;
}
```

9. Napisati funkciju koja za datoteku i niz znakova (string), na ekran ispise one retke iz datoteke u kojima se nalazi taj niz znakova.

```c
void print_lines(FILE* fp, char* str)
{
    char red[1024];
    while( !feof(fp))
    {
        fgets(red, 1024, fp);
        if (strstr(red, str))
            printf("%s",red);
    }
}
int main(void)
{
    char str[]="printf";
    FILE* fp = fopen("rok3.c","r");
    if (!fp)
        return 1;
    print_lines(fp, str);
    return 0;
}
```

10. Zadana je datoteka osobe.txt u kojoj je svaki redak oblika ime, prezime, visina, težina. Napisati program koji podatke iz datoteke zapisuje u niz struktura, te ih sortira po indeksu tjelesne težine koji izosi težina/kvadrat_visine i zapisuje ih sortirane u drugu datoteku. Korisnik određuje da li će se sortirati uzlazno ili silazno.

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

#define COUNTOF(x) (sizeof (x) / sizeof (x[0]))
#define N 20
typedef struct
{
    char name[80], surname[80];
    unsigned height, weight;
}person;

int osobe_cmp_asc(const void* a, const void* b)
{
    person os1, os2;
```

```
            float raz;
        os1 = *(person*)a;
        os2 = *(person*)b;
            raz = os1.weight / pow(os1.height, 2) - os2.
               weight / pow(os2.height, 2);
            if (raz > 0) return 1;
            else if (raz < 0) return -1;
            return 0;
    }

    int osobe_cmp_desc(const void* a, const void* b)
    {
        return -1 * osobe_cmp_asc(a, b);
    }

    unsigned read_persons(FILE* fp1, person* o_arr, unsigned
         n)
    {
        unsigned i = 0;
        while (!feof(fp1))
        {
            if (i < n)
            {
                fscanf(fp1, "%s %s %u %u", o_arr[i].name,
                   o_arr[i].surname, &o_arr[i].height, &o_arr
                   [i].weight);
                i++;
            }
            else
            {
                n = n * 2;
                o_arr = realloc(o_arr, n * sizeof(person));
            }
        }
        return i - 1;

    }

    void write_persons(person* o_arr, unsigned n)
    {
        unsigned i = 0;
        for(i = 0; i < n; i++)
                printf( "%s %s %u %u", o_arr[i].name, o_arr[i
                   ].surname, o_arr[i].height, o_arr[i].
```

```
                    weight);
}
void sort_persons(FILE* fp, person* o_arr, unsigned n)
{
    int k;
    unsigned i;
    printf("uzlazno - 1\nsilazno - 2?\n");
    scanf("%d", &k);
    if (k == 1)
        qsort(o_arr, n, sizeof (o_arr[0]), osobe_cmp_asc)
            ;
    else
        qsort(o_arr, n, sizeof (o_arr[0]), osobe_cmp_desc
            );
    for (i = 0; i < n; i++)
    {
          fprintf(fp, "%s %s %u %u\n", o_arr[i].name,
             o_arr[i].surname, o_arr[i].height, o_arr[i].
             weight);
    }

}

int main(void)
{
    FILE *fp1, *fp2;
        person* osobe_arr;
        unsigned n;
    if ((fp1 = fopen("osobe.txt", "r")) == NULL)
    {
        perror("");
        return 1;
    }
    if ((fp2 = fopen("osobe_sort.txt", "w")) == NULL)
    {
        perror("");
        fclose(fp1);
        return 1;
    }
    osobe_arr = (person*) malloc(N * sizeof (person));
    n = read_persons(fp1, osobe_arr, N);
    sort_persons(fp2, osobe_arr, n);
    write_persons(osobe_arr, n);
    free(osobe_arr);
```

```
    fclose(fp1);
    fclose(fp2);
    return 0;


}
```

11. Dana je datoteka `persons.txt` u kojoj se u svakom retku nalazi ime, prezime, JMBG i OIB neke osobe. Napisati funkciju koja u drugu datoteku zapisuje samo maloljetne osobe. Možemo pretpostaviti da su maloljetne osobe osobe koje će u tekućoj godini imati manje 18 godina. Korisnik unosi tekuću godinu.

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

typedef struct
{
    char name[20], surname[20], jmbg[14];
    unsigned oib;

}person;

unsigned get_year_from_jmbg(char* jmbg)
{
    unsigned d, y;
    sscanf(jmbg, "%4u%3u%6u", &d, &y, &d);
    if (y/100 == 9)
        y += 1000;
    else
        y += 2000;
    return y;
}

int juvenile(FILE* fp1, FILE* fp2, unsigned curr_year)
{
    person p;
    unsigned y;

    while(fscanf(fp1, "%s %s %s %d", p.name, p.surname,
       p.jmbg, &p.oib) != EOF)
```

```
        {
            y  = get_year_from_jmbg(p.jmbg);
            if (curr_year - y < 18)
                fprintf(fp2, "%s %s %s %d\n", p.name, p.
                    surname, p.jmbg, p.oib);
        }

        return 0;
    }

    int main(void)
    {
        FILE* fp1, *fp2;
        unsigned y;
        fp1 = fopen("persons.txt", "r");
        if (!fp1)
        {
            perror("");
            return 1;
        }
        fp2 = fopen("persons_juv.txt", "w");
        if (!fp2)
        {
            perror("");
            fclose(fp1);
            return 1;
        }
        puts("unesi tekucu godinu");
        scanf("%u", &y);
        juvenile(fp1, fp2, y);
        fclose(fp1);
        fclose(fp2);
        return 0;
    }
```

12. Definirati strukturu `person` koja se sastoji od imena, prezimena i ocjena iz tri ispita. Napisati funkciju koja će sortirati niz osoba od osobe s najboljim prosjekom do najgore.

```
    typedef struct
    {
        char name[20], surname[20];
```

```
    int o1, o2, o3;
}person;

float avg(person* a)
{
    return (a->o1 + a->o2 + a->o3)/3.;
}

int personcmp(const void* a, const void* b)
{
    person* os1 = (person*)a;
    person* os2 = (person*)b;
    if (avg(os1) < avg(os2)) return -1;
    if (avg(os1) > avg(os2)) return 1;
    return 0;
}

void sort(person* arr, unsigned n)
{
    qsort(arr, n, sizeof (person), personcmp);
}
#define COUNTOF(x) (sizeof(x)/sizeof((x)[0]))

int main(void)
{
    int i;
    person a[] = {{"Mate", "Matic", 2, 3, 4},
                  {"Frane", "Franic", 2, 3, 5},
                  {"Frane", "Antic", 5, 3, 5},
                  {"Ante", "Antic", 5, 3, 2}};
    sort(a, COUNTOF(a));
    for (i = 0; i < COUNTOF(a); i++)
    {
           printf("%s %s %d %d %d\n", a[i].ime, a[i].
              prezime, a[i].o1, a[i].o2, a[i].o3);
    }
    return 0;

}
```

13. U datoteci, čije ime zadaje korisnik, nalazi se tekst podijeljen na paragrafe, gdje su paragrafi odvojeni oznakom za novi red. Napisati funkciju koja čita datoteku, te vraća paragraf u kojem ima najviše rečenica.

```c
#include <stdio.h>

unsigned cnt_sentences(char* str)
{
    unsigned cnt = 0;
    char* p = str;
    while (*p)
    {
        if (*p == '.' || *p == '!' || *p == '?')
            ++cnt;
        ++p;
    }
    return cnt;
}

void find_max_cnt(FILE* fp, char* maxline)
{
    char line[1024];
    int cnt, maxcnt = 0;
    while (fgets(line, 1024, fp) != NULL)
    {
        puts(line);
        cnt = cnt_sentences(line);
        if (cnt > maxcnt)
        {
            maxcnt = cnt;
            strcpy(maxline, line);
        }

    }
}

int main(void)
{
    char str[1024]="";
    char filename[1024];
    gets(filename);
    FILE* fp = fopen(filename, "r");
    if (!fp)
        return 1;

    find_max_cnt(fp, str);
    puts(str);
```

```
        fclose(fp);
    return 0;
    }
```

9.10 Zadaci za vježbanje

1. Napisati funkciju koja će otvoriti postojeću tekstualnu datoteku i izračunati njezinu duljinu u bajtovima. Duljina i naziv datoteke argumenti su funkcije. Funkcija vraća jedan u slučaju neuspjelog otvaranja datoteke, odnosno nulu u slučaju uspješnog izvršavanja zadatka.
2. Napisati funkciju koja će otvoriti zadanu tekstualnu datoteku i odrediti koliko ukupno ima pojavljivanja zadanog znaka u datoteci (ime datoteke i znak su argumenti funkcije).
3. Napisati funkciju koja će otvoriti zadanu tekstualnu datoteku i prekopirati podatke u drugu zadanu datoteku (imena datoteka su argumenti funkcije). U drugu datoteku se svaka riječ iz prve datoteke pohranjuje u novi red. Riječ je svaki niz znakova omeđen sa razmacima (space i tab).
4. Napisati funkciju koja će iz zadane tekstualne datoteke učitati podatke i kopirati ih u novu datoteku. Imena datoteka su argumenti funkcije. Kopiranje izvršite tako da se preskoči prvi znak svakog učitanog reda.
5. Napisati program koji čita binarnu datoteku unazad znak po znak i zapisuje je u drugu datoteku.
6. Napisati program koji čita red po red u datoteci, izbaci sve znakove interpunkcije i praznine, te zapiše natrag preuređeni red.
7. Napisati funkciju koja će otvoriti tekstualnu datoteku i u njoj izbrisati svako pojavljivanje zadane riječi. Funkcija vraća jedan u slučaju neuspjelog otvaranja datoteke, odnosno nulu u slučaju uspješnog izvršavanja zadatka. Ime datoteke i riječ koju treba izbrisati ulazni su argumenti funkcije:
   ```
   int izbrisi_rijec(const char* ime_dat, char* rijec);
   ```
8. Napisati program koji će monokromatsku .bmp sliku dimenzija 256x256 zrcalno promjeniti po osi koja ide sredinom slike (ako je na slici naprimjer lice, desno oko postat ce lijevo, a lijevo desno).
9. Napisati program koji će monokromatsku .bmp sliku dimenzija 256x256 zrcalno promjeniti po osi koja ide sporednom dijagonalom slike (ono što je na slici u donjem desnom kutu, bit će nakon promjene u gornjem lijevom).

10. Napisati funkciju koja za neku datoteku vraća broj redaka koji su prazni ili počinju sa znakom `'#'`.
11. Definirati strukturu `slika` koja opisuje sliku sa imenom i prezimenom autora, imenom slike, dimenzijom i cijenom. Napisati funkciju koja za niz slika pročitanih iz datoteke, vraća ime i prezime najzastupljenijeg autora u kolekciji.
12. Napisati funkciju koja iz datoteke vraća broj redaka u kojem se nalazi točno dva pojavljivanja određenog slova (slovo je parametar funkcije).
13. Napisati funkciju koja iz datoteke vraća broj redaka u kojem se nalazi točno dvije znamenke.
14. Napisati funkciju koja čita datoteku u kojoj svaki red predstavlja ime studenata, matični broj i broj ostvarenih ECTS bodova, te vraća ukupan broj ostvarenih ETCS bodova svih studenata.
15. Napisati funkciju koja kreira datoteku u kojoj je svaki red niz znakova upisan sa tastature. Funkcija vraća ukupan broj zapisanih znakova u datoteci.
16. Napisati funkciju koja kopira datoteku, ali tako da kopira samo one znakove koji se pojavljuju više od jednom u retku.
17. Napisati funkciju koja čita datoteku i u drugu zapisuje sve znakove koji su mala slova. Sve druge preskače.
18. Napisati funkciju u kojoj se čitaju znakovi iz datoteke, te računa zbroj vrijednosti znakova koji su znamenke.
19. Dana je datoteka `test.txt`. Napisati funkciju koja čita datoteku red po red, te vraća duljinu najdužeg reda.
20. Napisati funkciju koja za datoteku i niz znakova, na ekran ispiše one retke iz datoteke u kojima se nalazi taj niz znakova.
21. Dana je datoteka `dictionary.txt` u kojoj je u svakom retku jedna engleska riječ i prijevod na hrvatski. Napisati program koji za neku riječ vraća sve odgovarajuće riječi drugog jezika.
22. Zadana je datoteka u kojoj se u svakom redu nalazi šest brojeva koji predstavljaju koordinate trokuta u dvodimenzionalnom prostoru. Napisati

strukturu trokut i funkciju koja brojeve iz datoteke zapisuje u niz struktura trokut, te sortira niz po opsegu (od najvećeg prema najmanjem). Funkcija vraća sortirani niz i njegovu duljinu.

23. Napisati funkciju koja čita datoteku u kojoj su brojevi, te iz njih kreira niz brojeva tako da se nijedan broj u njemu ne ponavlja. Funkcija vraća novonastali niz i njegovu duljinu.
24. Napisati funkciju koja čita csv (comma separated value) datoteku u kojoj je u svakom retku ime izvođača, naziv albuma, godina izdavanja i žanr. Ako se neko od polja u datoteci sastoji od više riječi, riječi su napisane unutar navodnika. Funkcija vraća niz zapisa koji odgovara određenom žanru i izdan je u određenoj godini (žanr i godina su parametri funkcije).
    Primjer retka u datoteci: "Red Hot Chili Peppers", "The Uplift Mofo Party Plan", 1987, "funk rock"
25. Napisati funkciju koja kopira datoteku tako da iz nje izbaci sve znakove koji nisu brojevi.
26. Napisati funkciju koja kopira tekstualnu datoteku, ali tako da kopira samo one znakove koji se pojavljuju više od jednom u retku.
27. Napisati funkciju koja za datoteku u kojoj je u svakom retku jedan broj i jedna riječ, ispisuje na ekran riječ onoliko puta koliki je broj.
    Primjer: Ako u datoteci piše:
    ```
    4 psa
    2 tigra
    1 deva
    ```
    ispisat će se
    ```
    psapsapsapsa
    tigratigra
    deva
    ```
28. Napisati funkciju koja čita neku datoteku te ispisuje na ekran sve riječi iz datoteke koje završavaju na "ska".
29. Napisati funkciju koja u neku datoteku zapisuje sve znakove koje korisnik unosi koji nisu veliko slovo.

30. Napisati funkciju koja za datoteku i niz znakova (string), na ekran ispiše one retke iz datoteke u kojima se nalazi taj niz znakova.
    Napomena: String je bilo kakav niz znakova.

```
void ispisi_retke(FILE* fp, char* str)
{
    char red[1024];
    while( !feof(fp))
    {
        fgets(red, 1024, fp);
        if (strstr(red, str))
            printf("%s",red);
    }
}
```

31. Napisati funkciju koja čita datoteku u kojoj je u svakom redu matični broj studenta i prosjek ocjena studenta, te vraća matični broj studenta sa najvećim prosjekom ocjena.
32. Napisati program koji čita binarnu datoteku unazad znak po znak i zapisuje je u drugu datoteku.
33. Napisati program koji će monokromatsku .bmp sliku dimenzija 256x256 zrcalno promjeniti po osi koja ide sredinom slike (ako je na slici naprimjer lice, desno oko postat će lijevo, a lijevo desno).
34. U svakom redu datoteke, čije ime zadaje korisnik, nalaze se koordinate točke u 3D koordinatnom sustavu koje opisuju neko tijelo. Napisati funkciju koja čita točke iz datoteke i vraća onu točku koja je najbliža promatraču. Promatrač se nalazi u smjeru x-osi.
35. U datoteci, čije ime zadaje korisnik, nalazi se tekst podijeljen na paragrafe, gdje su paragrafi odvojeni oznakom za novi red. Napisati funkciju koja čita datoteku, te vraća paragraf u kojem ima najviše rečenica. Rečenica završava jednim od znakova `'.'`, `'!'` ili `'?'`.
36. Zadana je datoteka u kojoj svaki red predstavlja ime i prezime studenata te matični broj. Napisati funkciju koja čita podatke iz datoteke te u drugu datoteku zapisuje e-mail studenta. Studentov e-mail se kreira od prvog slova imena i prvog slova prezimena i matičnog broja, a nakon znaka `@` se dopisuje `oss.unist.hr`.

37. Dana je struktura ucenik:

```
struct reprezentacija
{
  char ime_drzave[30];
  unsigned br_pobjeda, br_poraza, br_utakmica;
};
```

Napisati funkciju koja niz struktura reprezentacija zapisuje sortirano u datoteku po broju bodova. Bodovi se računaju tako da pobjeda nosi 2 boda, poraz 0, a neriješeno 1. Za sortiranje upotrebljavati standardnu funkciju `qsort`.

38. Zadana je datoteka u kojoj su u svakom redu 4 cijela broja. Napisati funkciju koja čita brojeve iz datoteke te vraća informaciju u kojem redu se nalaze brojevi sa najvećom prosječnom vrijednošću.
39. Napisati funkciju koja iz jedna datoteku čita riječi te ih zapisuje u drugu ako pročitana riječ ima manje od 5 slova.
40. Dana je struktura ucenik:

```
struct ucenik
{
  char ime[30], prezime[30];
  unsigned bodovi;
  char pravo_potpisa;
  char status;
};
```

Napisati funkciju koja niz struktura ucenik zapisuje sortirano u datoteku po broju bodova. Za sortiranje upotrebljavati standardnu funkciju `qsort`.

41. Zadana je datoteka u kojoj je u svakom redu zapis oblika

(x, y), (r, g, b)

gdje su `x` i `y` koordinate koje predstavljaju točku, a `r`, `g` i `b` vrijednosti između `0` i `255` koje pretstavljaju vrijednosti crvene, zelene i plave boje u boji točke. Napisati funkciju koja čita datoteku te vraća koliko ima crnih točaka (kombinacija 0, 0, 0) te koliko ima bijelih točaka (255, 255, 255).

42. Dane su dvije datoteke u kojima je u prvom redu broj članova niza brojeva, a u sljedećem se redu nalaze sami članovi niza. Napisati funkciju koja

pročita oba niza i ako oba niza imaju jednak broj članova, zbrojiti elemente na odgovarajućim mjestima te zapisati rezultate u treću datoteku tako da u prvom redu bude broj članova niza, a u drugom sami članovi niza. Ime treće datoteke unosi korisnik.

43. Napisati funkciju koja čita neku datoteku te ispisuje na ekran sva velika slova iz datoteke.
44. Dana je datoteka `FIFA_WC.txt` u kojoj su u svakom redu zapisani ime reprezentacije, broj odigranih utakmica, gol razlika i broj osvojenih bodova. Primjer jednog reda datoteke je

    ```
    Mexico 2 3 4
    ```

    Napisati funkciju koja čita datoteku `FIFA_WC.txt`, te u drugu datoteku `FIFA_WC_2.txt` zapisuje one momčadi, i njihove rezultate, koje nisu osvojile niti jedan bod na prvenstvu.
45. Napisati strukturu pravokutnik, gdje je pravokutnik zadan sa koordinatama dviju nasuprotnih točaka. Napisati funkciju koja prima strukturu pravokutnik i računa opseg i površinu pravokutnika.

    Neka su u datoteci zapisane koordinate pravokutnika tako da su u svakom retku dane dvije točke, prvi red datoteke je naslov, a drugi imena stupaca. Ime datoteke unosi korisnik. Napisati funkciju koja čita koordinate, zapisuje ih u niz struktura pravokutnik, te drugu koja iz niza struktura pravokutnik računa pravokutnik najveće površine.

x_1	y_1	x_2	y_2
2.0	3.5	4.5	4.5
1.0	1.0	3.0	3.0
0.0	0.0	5.5	3.0

Vidljivost i životni vijek varijabli

Opseg djelovanja neke varijable (engl. *scope*) je dio kôda u kojem je varijabla vidljiva i u kojem se može koristiti. Postoje tri mjesta na kojima varijable mogu biti deklarirane.

SCOPE
OPSEG DJELOVANJA

- Varijable deklarirane unutar tijela funkcije ili bloka (varijable deklarirane između para vitičasti zagrada). Takve varijable se obično zovu **lokalne varijable**.
- Varijable deklarirane izvan svih funkcija. Takve varijable se zovu **globalne varijable**.
- Varijable deklarirane u definiciji funkcije kao formalni parametri. [1]

LOKALNE VARIJABLE

GLOBALNE VARIJABLE

10.1 Lokalne varijable

Lokalne varijable vidljive su samo unutar bloka u kojem su deklarirane. Nastaju kada su deklarirane/definirane, a nestaju kada program u izvođenju izađe iz bloka u kojem su definirane. Ako je deklaracija oblika `type name;` takve

[1] http://www.tutorialspoint.com/cprogramming/c_scope_rules.htm

varijable su automatske. Ispred deklaracije se može napisati ključna riječ `auto`, ali ako ne piše, podrazumijeva se da su automatske.

`static`

Varijabla se može deklarirati i kao `static`. U tom slučaju njen životni vijek jednak je životnom vijeku programa. Nakon izlaska iz funkcije u kojoj se `static` varijabla nalazi, vrijednost varijable će se sačuvati, iako neće biti vidljiva van funkcije. Prilikom ponovnog ulaska u funkciju varijabla će imati sačuvanu vrijednost.

Ako `static` varijabla nije inicijalizirana, inicijalizirat će se na nulu ili nul-pokazivač, ako se radi o pokazivačkoj varijabli.

Primjer 10.1 Primjer `static` varijable. Vrijednost varijable `n` bit će `0` prilikom prvog izvođenja funkcije, a zatim će se uvećati za `2` unutar funkcije. Pri sljedećem pozivu `n` će imati vrijednost dva itd.

```c
#include <stdio.h>

void fun(void)
{
    static int n;
    int m = 2;
    printf("%d\t", n);
    n = n + m;
    printf("%d\n", n);
}

int main(void)
{
    function();
    function();
    function();
    return 0;
}
```

```
0       2
2       4
4       6
```

Primjer 10.2 Varijabla `n` broji pozive funkcije u kojoj se nalazi. Varijabla je inicijalizirana na vrijednost `1`.

```
#include <stdio.h>
void fun(void)
{
    static int n = 1;
    printf("%d. poziv\t", n);
    ++n;
}

int main(void)
{
    function();
    function();
    function();
    return 0;
}
```

```
1. poziv    2. poziv    3. poziv
```

10.2 Globalne varijable

Varijable koje su deklarirane izvan tijela funkcija su globalne varijable. Vidljive su unutar datoteke u kojoj se nalaze i to nakon mjesta na kojem su definirane. Dodavanjem ključne riječi `static` ispred deklaracije onemogućavamo vidljivost izvan datoteke u kojoj je deklarirana. S druge strane, ako je varijabla deklarirana bez ključne riječi `static`, može se učiniti vidljivom u drugoj datoteci i to tako da se u drugoj datoteci ponovno deklarira varijabla koristeći se ključnom riječi `extern`.

GLOBALNE VARIJABLE

EXTERN

Životni vijek globalnih varijabli je jednak životnom vijeku programa. Inicijalna vrijednost globalne varijable je nula ili nul-pokazivač.

Nekoliko je nedostataka globalnih varijabli i svakako je preporuka programerima da pročitaju članak "Global variables considered harmful" [16]. Naime, projekti znaju biti poprilično veliki i na njima često radi velik broj programera. Definirajući globalnu varijablu mi omogućavamo raznim funkcijama pristup istom memorijskom prostoru. Da bismo se sigurno koristili globalnom varijablom, trebali bismo znati što sve druge funkcije rade s njom. Nadalje, može se dogoditi da programer definira lokalnu varijablu s istim imenom i tako prekrije vrijednost globalne varijable. Stoga treba biti oprezan s globalnim varijablama i izbjegavati njihovu upotrebu.

Primjer 10.3 Funkcija `changle_str` mijenja vrijednost stringa koji je definiran kao globalna varijabla.

```
#include <stdio.h>

/* globalna varijabla */
char str[] = "What a beautiful day";

void change_str()
{
    str[4] = '\0';
}

void change_param_str(char* str)
{
    /* parametar funkcije sakriva globalnu varijablu */
    str[0] = 'Y';
}

int main(void)
{
    char s[] = "Beautiful world";
    change_param_str(s);
    change_str();

    printf("%s\n", s);
    printf("%s\n", str);
    return 0;
}
```

```
Yeautiful world
What
```

Primjer 10.4 Dvije funkcije pristupaju istoj globalnoj varijabli, jedna je i mijenja.

```c
#include <stdio.h>

/* globalna varijabla */
int m = 10;

void f()
{
    for (; m > 0; m -= 10)
        printf("%d\t", m);
}

void g()
{
    int i;
    for (i = 0; i < m; ++i)
        f();
}

int main(void)
{
    g();
    return 0;
}
```

```
10
```

10.3 Deklaracija vs. definicija

DEKLARACIJA

DEFINICIJA

Do sada smo se koristili terminom deklaracija za uvođenje varijable, tj. opisivanje njenog tipa i uvođenje imena, što će biti od koristi kompajleru, ali i za zauzimanje mjesta u memoriji za tu varijablu. Međutim, alokacija memorije za varijablu je definicija, a deklaracija je informacija kompajleru o tipu i imenu varijable. Kod automatskih varijabli nema razlike između deklaracije i definicije. Naredba `int x;` daje informaciju o tipu i imenu, ali i istodobno alocira memoriju za varijablu. Međutim, kod globalnih varijabli to ne mora biti tako. Naime, ako je globalna varijabla deklarirana pomoću ključne riječi `extern`, to znači da je varijabla definirana negdje drugje. Kako se alokacija memorije može dogoditi samo jednom, dogodit će se prilikom definicije, a deklarirana `extern` varijabla će se povezati sa tom definicijom, tj. tim memorijskim prostorom prilikom linkanja.

10.4 Varijable kao parametri funkcije

Primjer 10.5 Parametri funkcije skrivaju vrijednost globalnih varijabli s istim imenom. Primjer je dan kroz funkciju koja računa najveći zajednički nazivnik.

```
#include <stdio.h>

int a = 100, b = 50;

unsigned gcd(unsigned a, unsigned b)
{
    if (!b)
       return a;
    else
       return gcd(b, a%b);
}

int main(void)
{
    unsigned a = 10, b = 15;
```

```
        printf("gcd(%u, %u) = %u", a, b, gcd(a, b));
        return 0;
    }
```

```
gcd(10, 15) = 5
```

10.5 Riješeni zadaci

1. Napisati funkciju koja ispisuje poruku "opet sam pozvana" svaki četvrti put kada je pozvana.

```
void funkcija(void)
{
    static int count = 0;
    if(!(count++ % 4)) printf("%d opet sam pozvana\n",
        count);
}

int main(void)
{
    int i;
    for (i = 0; i < 20; i++)
        funkcija();
    return 0;
}
```

10.6 Zadaci za vježbanje

1. Dana je struktura:

   ```
   struct point
   {
       double x, y;
   };
   ```

   Napisati funkciju koja generira slučajne vrijednosti koordinata u rasponu od `-10.0` do `10.0` (decimalno), te pri svakom pozivu uveća varijablu `counter` za jedan ako se generirana točka nalazi unutar kružnice radijusa `10`.

   U `main` funkciji se poziva funkcija sve dok se ne izgenerira i ispiše deset točaka unutar kružnice radijusa `10`.

   Napomena: Generirani brojevi trebaju biti različiti (npr. upotrebljavajući `srand(time(NULL) + pomak)`, gdje se varijabla `pomak` mijenja prije generiranja novog broja).

2. Što ispisuju sljedeći programi?

   (a) ——

   ```
   #define COUNT(x) sizeof x / sizeof x[0]
   int next()
   {
     static int nums[] = { 8, 7, 6, 3, 3, 5, 6};
     static int i = -1;
     do{
         i++;
     }while(nums[i]%3 && i < COUNT(nums));

     return nums[i];
   }

   int main()
   {
     printf("%d\n", next() );
     printf("%d\n", next() );
     printf("%d\n", next() );
     return 0;
   }
   ```

(b) ——

```
int x = 1;
int func1()
{
    return (x += 10);
}
int func2(int x)
{
    return(x += 1);
}
int main(void)
{
    int x = 10;
    func1(x);
    printf("%d\n",x);
    func2(x);
    printf("%d\n",x);
    return 0;
}
```

(c) ——

```
void fja(int n)
{
    static int m = 5;
    if (m-- >= --n)
        printf("%d\t",--m);

}
int main(void)
{
    int k = 4, i;
    for (i = 0; i < k; i++)
        fja(i);
    return 0;
}
```

(d) ——

```
static int x = 10;

void fun(void)
{
    int j = 0, x = 0;
    {
```

```
        static int x = 100;
        printf("x: %d\t", x );
        x++ ;
    }
    printf("x: %d\n", x );
}

int main(void)
{
    for (; x < 12; x++ )
    {
        printf("x: %d\t", x );
        fun();
    }
    return 0;
}
```

10.7 Ponavljanje

Što ispisuju sljedeći programi i zašto?

1. ———

```
int fun1(int i, int j)
{
    i += j;
    j += i;
    return j;
}

int i = 10;
int main(void)
{
    int j = 5;
    printf("%d\n", fun1(i, j));
    printf("%d\n", i);
    printf("%d\n", j);
    return 0;
}
```

2. ———

```
#define COUNT(x) (sizeof (x) / sizeof (x[0]))
int next()
{
  static int nums[] = {1,2,3,4,5,6,7,8};
  static int i = -1;
  do{
      i++;
  }while(nums[i]%2 && i < COUNT(nums));

  return nums[i];
}

int main()
{
  printf("%d\n", next() );
  printf("%d\n", next() );
  printf("%d\n", next() );
  return 0;
}
```

```
int x = 1;
int func1()
{
    return (x += 10);
}
int func2(int x)
{
    return(x += 1);
}
int main(void)
{
    func1(); printf("%d\n",x);
    func2(5);printf("%d\n",x);
    return 0;
}
```

4. ———

```
int x = 5;
int main(void)
{
    while(x--)
    {
        int x = -2;
        if (++x + 2) continue;
        printf("%d\n", x);
    }
    printf("%d\n", x);
    return 0;
}
```

5. ———

```
unsigned x = -1;
int main(void)
{
    while(x++)
    {
        int x = -2;
        if (++x + 1) continue;
        printf("%d\n", x);
    }
    printf("%d\n", x);
    return 0;
}
```

6. ———

```
#include <stdio.h>
void func(int x)
{
    printf("%d\n", x);
}
int main(void)
{
    int k = 3;
    void (*n)(int);
    n = &func;
    (*n)(k++);
    n(k);
    return 0;
}
```

7. ———

```
int a = -1;
int fun()
{
    static int a = 4;
    while (a < 4) a += 2;
    return a--;
}
int main(void)
{
    int b = fun();
    printf("%d\n", b);
    a = a > b ? a : fun();
    printf("%d\n", a);
}
```

8. ———

```
int a = -1;
int fun()
{
    static int a = 6;
    while (a > 4) a -= 2;
    return a++;
}
int main(void)
{
    int b = fun();
```

```
    printf("%d\n", b);
    a = a > b ? a : fun();
    printf("%d\n", a);
}
```

9. ———

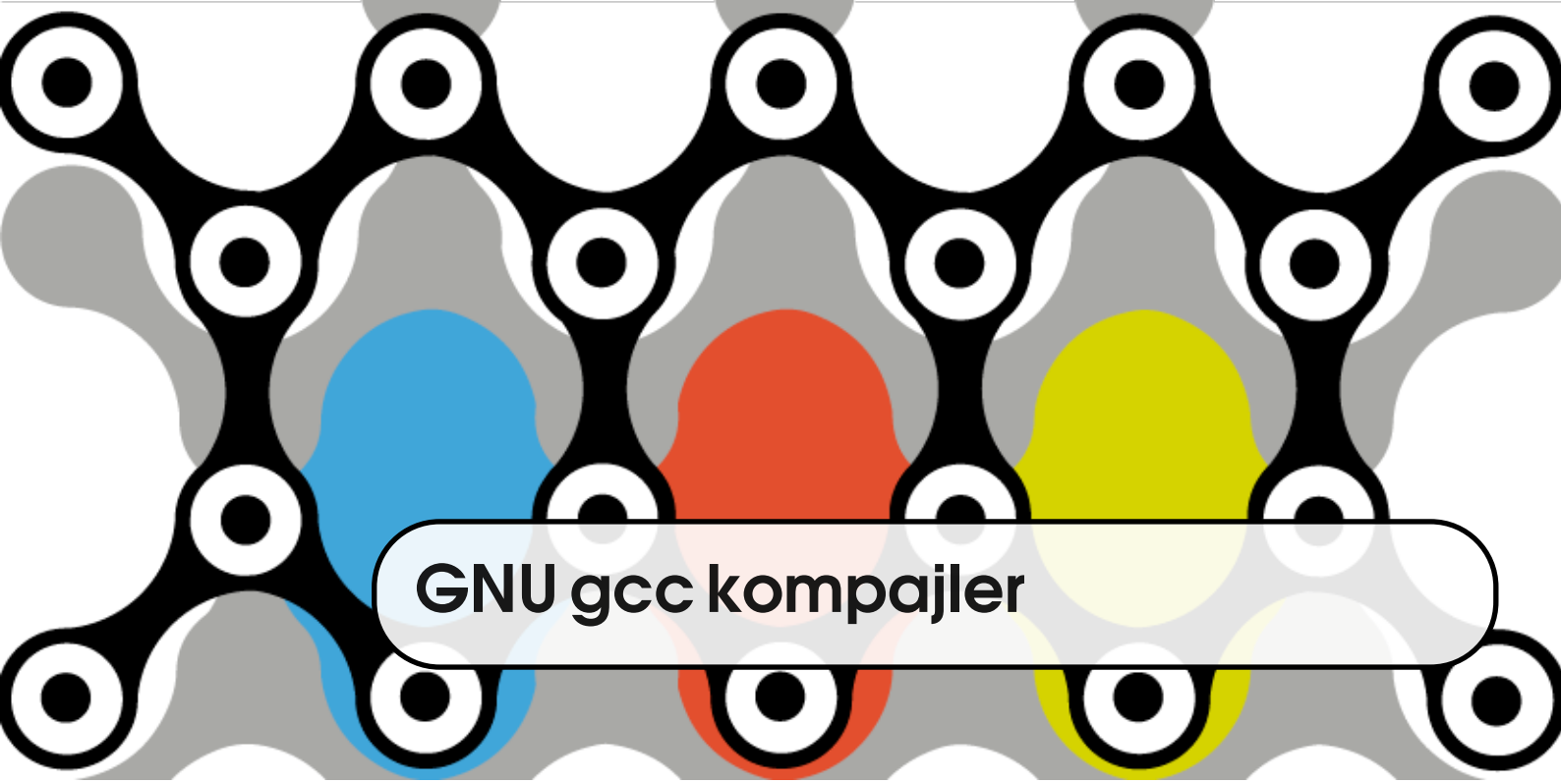
```
int fun1(int a, int b)
{
    return (a + b - 1) / 2;
}
int fun2(int a, int b)
{
    return (a + b) / 2;
}
int main()
{
    int k = 1, m = 4;
    int (*fun)(int, int);
    fun = fun1;
    while((k&1)^(1&m))
        printf("%d\n", fun(k, m++));
    fun = fun2;
    printf("%d\n", fun(k, m));
    return 0;

}
```

10. ———

```
#include<stdio.h>
char f(void)
{
    static char c='A';
    return c += 2;
}
int main(void)
{
    f();
    printf("%c\n",f());
}
```

Dodaci



GNU gcc kompajler

A.1 Kompajliranje gcc kompajlerom iz komandne linije

```
tty:  /bin/bash
    $ gcc primjer.c
```

Izvršna datoteka bit će `a.out`. Iz komandne linije (iz tekućeg direktorija) pokrenite je sa `./a.out` (točka označava tekući direktorij).

```
tty:  /bin/bash
    $ gcc primjer.c -o program
```

ili

```
tty:  /bin/bash
    $ gcc -o program primjer.c
```

Opcija `-o ime_izlaza` definira ime generirane datoteke. U gornjem primjeru

ime izvršne datoteke bit će `program`. Izvršni program pokreće se sa `./program`.

Primjer A.1 U `primjer.c` datoteku napišite sljedeći kôd

```
#include <stdio.h>
int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

Kompajlirajte program pomoću gornjih naredbi.

A.1.1 Opcije gcc kompajlera

Ukoliko se želi izvršiti samo dio procesa prevođenja, opcije `-c`,`-S` i `-E` zaustavit će proces na određenom mjestu.

Kompajliranje bez linkanja Opcija `-c` omogućuje kompajliranje bez linkanja. Rezultat je objektna datoteka. Ako ime objektne datoteke nije dano sa `-o` opcijom, bit će jednako imenu izvorne datoteke sa nastavkom `.o`.

Primjer A.2 Kompajliranje bez linkanja bez i sa zadanim imenom izlazne datoteke.

tty: /bin/bash

```
$ gcc -c primjer.c
$ gcc -c primjer.c -o prog.o
```

Kompajliranje bez asembliranja Opcija `-S` omogućuje kompajliranje bez asembliranja. Rezultat je asemblerska datoteka. Ako ime objektne datoteke nije dano sa `-o` opcijom, bit će jednako imenu izvorne datoteke sa nastavkom `.s`.

Primjer A.3 Kompajliranje bez asembliranja bez zadanog imena i sa zadanim imenom izlezne datoteke.

```
tty:  /bin/bash
    $ gcc -S primjer.c
    $ gcc -S primjer.c -o prog.s
```

Pogledajte kako izgleda `*.s` datoteka nekim text preglednikom.

Predprocesiranje Opcija `-E` omogućuje samo predprocesiranje. Rezultat je predprocesirana datoteka.

Primjer A.4 Predprocesiranje bez i sa zadanim imenom izlazne datoteke.

```
tty:  /bin/bash
    $ gcc -E primjer.c
    $ gcc -E primjer.c -o prog.i
```

Pogledajte kako izgleda `*.i` datoteka nekim text preglednikom.

A.1.2 Kompajliranje više izvornih datoteka

```
tty:  /bin/bash
    $ gcc primjer01.c primjer02.c
    $ gcc primjer01.c primjer02.c -o program
```

Točno jedna datoteka treba sadržavati `main` funkciju. Ukoliko nema `-o` opcije generirat će se `a.out` izvršna datoteka. Inače, generirat će se datoteka čije je ime dano uz opciju `-o`.

Primjer A.5 U datoteku `primjer01.c` napišite sljedeći kôd

primjer01.c

```
#include <stdio.h>
int main(void)
{
    float x, y;
    scanf("%f %f", &x, &y);
    printf("%f * %f = %f\n", x, y, x * y);
    return 0;
}
```

a u `primjer02.c` sljedeći kôd

primjer02.c

```
float funkcija(float a, float b)
{
    return a + b;
}
```

Ukoliko se iz jedne funkcije poziva druga, čija se definicija nalazi u drugoj izvornoj datoteci, prototip pozivane funkcije stavlja se u zasebnu izvornu datoteku koja ima nastavak `.h` i koja se uključuje predprocesorskom direktivom `#include` u izvornu datoteku iz koje se poziva funkcija.

Kompajler se koristi prototipom za provjeravanje ispravnosti poziva funkcije.

A.2 Biblioteke funkcija

Biblioteke su kolekcije predkompajliranih funkcija napravljene da bi se ponovno koristile u različitim programima. Standardna C biblioteka `libc.a` se automatski linka prilikom kompajliranja programa. U standardnoj C biblioteci nalaze se predkompajlirane funkcije `printf`, `scanf`, `strlen`, `strstr`, itd.

A.2.1 Kreiranje statičke biblioteke

Biblioteka je kolekcija predkompajliranih funkcija. Na linuxu statičke biblioteke obično imaju prefiks `lib` i završavaju sa nastavkom `.a`.

Da bi se iz izvornih datoteka kreirala statička biblioteka, potrebno je prvo prevesti izvorne datoteke u objektne datoteke. Opcija `-c` će zaustaviti proces prevođenja nakon što su se kreirale objektne datoteke.

```
tty: /bin/bash
    $ gcc -c primjer05.c primjer06.c
```

Ako želimo sve izvorne datoteke u tekućem direktoriju prevesti u objektni kôd možemo upotrebljavati specijalni znak `*`.

```
tty: /bin/bash
    $ gcc -c *.c
```

Sada se kolekcija objektnih datoteka pakira u statičku biblioteku programom za arhiviranje `ar`.

```
tty: /bin/bash
    $ ar -r libracun.a primjer05.o primjer06.o
```

Opcijom `-t` programa `ar` može se vidjeti što je sve spakirano u biblioteku.

```
tty: /bin/bash
    $ ar -t libracun.a
```

Neke od ostalih opcija su `c` (kreiraj novu biblioteku), `q` dodaj datoteku u arhiv, `r` zamijeni datoteku u arhivu/biblioteci navedenom datotekom.

Napomena U biblioteku se ne pakira funkcija `main`.

A.2.2 Kreiranje dinamičke biblioteke

Svaka dijeljena datoteka ima *soname* koje se sastoji od prefiksa `lib`, samog imena, nastavka `so` i verzije koja se pri svakoj promjeni sučelja uvećava za jedan. Svaka dijeljena datoteka ima i *real name*, što je ime datoteke koja sadrži kôd. *Real name* dodaje na *soname* točku, broj koji označava *minor number*, pa onda opet točku i `release number`. Uz sve to, još postoji i ime kojim se koristi kompajler prilikom povezivanja s bibliotekom (*linker name*), koje je jednako imenu *soname* bez ikakve oznake verzija.

Da bi se iz izvornih datoteka kreirala dinamička biblioteka, potrebno je prvo prevesti izvorne datoteke u objektne datoteke. Opcija `-c` će zaustaviti proces prevođenja nakon što su se kreirale objektne datoteke. Osim toga, prilikom prevođenja u objektne datoteke koristi se još i opcija `-fPIC`. PIC je kratica od *position independent code* (detaljno u tome http://people.redhat.com/drepper/dsohowto.pdf i http://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html)

```
tty:  /bin/bash
    $ gcc -c -fPIC primjer05.c primjer06.c
```

Kreiranje dijeljene biblioteke izvršava se sa:

```
tty:  /bin/bash
    $ gcc -shared -Wl,-soname,libracun.so.1 -o libracun.so.1.0.1
       primjer05.o primjer06.o
```

Opcija `-shared` producira kao rezultat dijeljenu bibiloteku (*shared object*).
Opcija `-Wl,opcije` je prosljeđivanje `opcije` linkeru. Ukoliko ima više opcija odvojenih zarezom (npr. `-Wl,opcija1,opcija2`) linkeru će se proslijediti u obliku `opcija1 opcija2`. U gornjem primjeru linkeru će se proslijediti opcija `-soname libracun.so.1`.
Opcija `-o ime` je kao i do sada ime izlazne datoteke, a nabrojane datoteke sa `.o` nastavkom su prekompajlirane datoteke koje će biti dio dijeljene biblioteke.

Nakon kreiranja dijeljene datoteke potrebno je setirati simboličke linkove: simbolički link sa *soname* na *real name* i link sa imena bez oznake verzija na *real name*. To se može napraviti pomoću `ldconfig`. Sa `ldconfig -p` može se pogledati koje su sve dijeljene datoteke prijavljene u sustav, a sa

tty: /bin/bash
```
$ ldconfig -n direktorij
```

mogu se dijeljene datoteke dodati na popis (`/etc/ld.so.conf.d/*.conf` je lokacija popisa).

A.2.3 Povezivanje sa bibliotekom

Kada u programu pozivamo funkcije definirane u biblioteci prvo prevodimo datoteke u objektni kôd:

tty: /bin/bash
```
$ gcc -c primjer07.c -o primjer07.o
```

Prisjetimo se da je opcija `-c` zaustavljanje procesa prevođenja nakon kreiranja objektnog kôda (neće se kreirati izvršni kôd), a `-o` zadavanje imena izlazne datoteke.

Nakon toga povezujemo naš objektni kôd s kôdom biblioteke:

tty: /bin/bash
```
$ gcc primjer07.o -L. -lracun -o program
```

Opcija `-L` kaže u kojem direktoriju je statička biblioteka (u našem slučaju je to tekući direktorij `.`), opcija `-l` zadaje ime statičke biblioteke i navodi se bez početnog prefixa `lib` i nastavka `.a` u slučaju statičke biblioteke, tj. `.so` u slučaju dinamičke biblioteke.

A.3 Pisanje `makefile` datoteke

Pretpostavimo da su dane sljedeće datoteke:

main.c

```
#include <stdio.h>
#include "circle.h"

int main(void)
{
    struct circle k;
    //printf("unesi koordinate sredista kruznice\n");
    //scanf("%f %f", &k.center.x, &k.center.y);
    printf("unesi radijus\n");
    scanf("%f", &k.radius);
    printf("povrsina kruznice je %f", circumference(&k));
    return 0;
}
```

point.c

```
#include "point.h"

float distance_x(struct point p1, struct point p2)
{
    return p1.x - p2.x;
}

float distance_y(struct point p1, struct point p2)
{
    return p1.y - p2.y;
}
```

point.h

```
struct point
{
    float x, y;
};

float distance_x(struct point p1, struct point p2);
float distance_y(struct point p1, struct point p2);
```

circle.c

```
#include "circle.h"

float circumference(struct circle* c)
{
    return 2 * c->radius * 3.14;
}
```

circle.h

```
#include "point.h"

struct circle
{
    struct point center;
    float radius;
};

float circumference(struct circle* c);
```

Datoteka `main.c` uključuje `circle.h`, datoteka `circle.c` uključuje `circle.h` i `point.h`, a datoteka `point.c` uključuje `point.h`.

Kreiranje izvršne datoteke može se odviti u dva koraka: kreiranje objektnih datoteka i linkanje.

- `gcc -c point.c circle.c main.c` - kriraju se objektne datoteke.
- `gcc -o main point.o circle.o main.o -lm` - kreira se izvršna datoteka `main` (opcija -lm je linkanje s math bibliotekom).

Skraćeno, to se može odviti u jednom koraku:

tty: /bin/bash

```
$ gcc -o main point.c circle.c main.c -lm
```

Možemo i spakirati `point.o` i `circle.o` u zasebnu biblioteku. U tom slučaju prvo kompajliramo kôd biblioteke, pa kôd `main.c`, pa linkamo `main.o` sa bibliotekom.

Kada promjenimo kôd u datoteci `main.c`, moramo prekompajlirati samo tu datoteku. Međutim, kada promjenimo kôd u jednoj od ostalih datoteka, trebamo prekompajlirati sve. Ako smo napravili zasebnu biblioteku, trebamo prekompajlirati biblioteku, pa `main.c`, pa linkati `main.c` s bibliotekom. S obzirom na to da su projekti mnogo veći nego što su dani u primjeru, i često se linkaju s velikim brojem biblioteka, posao prekompajliranja je zamoran.

Program `make` olakšava posao. On se koristi datotekom `makefile` ili `Makefile` u kojoj je definirano što će se kompajlirati, što o čemu ovisi i u kojem poretku. Program `make` provjerava koje su se datoteke promjenile (po vremenu promjene) i ponovo kompajlira one datoteke čije je vrijeme modifikacije starije od vremena modifikacije datoteka o kojima ovise. Kompajliranje se vrši naprosto pozivanjem programa `make`. Ako je ime `makefile` datoteke neko drugo, pokretanje kompajliranja se izvršava naredbom `make ime_makefilea`.

Datoteke `makefile` sastoji se od dva dijela:

- skup varijabli koje definiraju koje će se kompajler koristiti, koje opcije i slično,
- skup ciljeva (*target*), tj. datoteka koje će se generirati.

Komentari u datoteci `makefile` počinju sa znakom #.

A.3.1 Varijable

Varijable koje se prve definiraju su ime kompajlera i opcije kompajlera.

```
CXX = gcc                    # kompajler je gcc
CXXFLAGS = -Wall             # ukljucena opcija prikazivanja
    svih warninga
```

A.3.2 Targeti

Targeti su dateteke koje trebaju biti generirane. Za svaki target piše se obično dvije linije:

1. ovisnosti - nabrajaju se datoteke o kojima ovisi target,
2. naredbe za generiranje targeta.

Prvi target nabrojan je onaj koji želimo kao izlaznu datoteku. U našem primjeru to je izvršna datoteka `main`. Ona ovisi o `circle.o` datoteci (to znamo zbog `#include "circle.h"`) i naravno, o `main.o`. Naredba za generiranje targeta mora početi pritiskom tipke <TAB> (ne space!).

```
main: main.o circle.o
<TAB>$(CXX) $(CXXFLAGS) -o main.o circle.o
```

Druga linija mora početi pritiskom tipke <TAB>. Znak $ ispred imena varijable je dohvaćanje vrjednosti varijable. Zapravo se druga linija prevodi u

`gcc -Wall -o main.o circle.o`.

Sada idemo napisati kako dobiti objektne datoteke `main.o` i `circle.o`. Prvi target je `main.o`, a on ovisi o datotekama `main.c`, `circle.h`

```
main.o: main.c circle.h
        $(CXX) $(CXXFLAGS) -c main.c
```

Sljedeći target `circle.o` ovisan je o `circle.h` i `point.o`.

```
circle.o: circle.h point.o
<TAB>$(CXX) $(CXXFLAGS) -c circle.c
```

Na kraju target `point.o` ovisi o `point.h`.

```
point.o: point.h
<TAB>$(CXX) $(CXXFLAGS) -c point.c
```

Kompajliranje se izvršava naredbom `make`. Ako se želi generirati samo jedan target piše se `make ime_targeta`. Često se pišu još dva targeta `all` i `clean`.

```
all:
	$(CXX) $(CXXFLAGS) -o main main.c circle.c point.c
```

```
clean:
	rm *.o main
```

Kompajliranje programa izvodi se iz komandne linije na sljedeći način: `make target`. Ako korisnik napiše `make clean`, izvršit će se naredba `rm *.o main`, a ako napiše samo `make` kreirat će se izvršna datoteka `main`.

A.3.3 Povezivanje sa drugim bibliotekama

Datoteke circle.o i point.o možemo spakirati u biblioteku. Datoteka `makefile` može izgledati ovako:

```
CXX = gcc
CXXFLAGS = -Wall

all: libcurve.a

libcurve.a : point.o circle.o
<TAB>ar -rcs libcurve.a point.o circle.o

# circle.o ovisi o point.o i circle.h
circle.o : circle.h point.o
<TAB>$(CXX) $(CXXFLAGS) -c circle.c
```

```
point.o : point.h
<TAB>$(CXX) $(CXXFLAGS) -c point.c

clean:
<TAB>rm *.o libcurve.a
```

Program, primjerice `main.c` koji se koristi statičkom bibliotekom `libcurve.a`, može imati ovakav `makefile`

```
CXX = gcc
CXXFLAGS = -Wall

main : main.o
<TAB>$(CXX) $(CXXFLAGS) main.o -L. -lcurve -o main

main.o : main.c circle.h
<TAB>$(CXX) $(CXXFLAGS) -c main.c

clean:
<TAB>rm *.o main
```

Sve zajedno (kreiranje stičke biblioteke i povezivanje s njom) možemo izvesti u jednom `makefile`-u

```
CXX = gcc
CXXFLAGS = -Wall

all: libcurve.a main

main : main.o
       $(CXX) $(CXXFLAGS) main.o -L. -lcurve -o main
main.o : main.c circle.h
       $(CXX) $(CXXFLAGS) -c main.c

libcurve.a : point.o circle.o
       ar -rcs libcurve.a point.o circle.o
```

```
# circle.o ovisi o point.o i circle.h
circle.o : circle.h point.o
	$(CXX) $(CXXFLAGS) -c circle.c

point.o : point.h
	$(CXX) $(CXXFLAGS) -c point.c

clean:
	rm *.o
```

Slično se može napraviti i za dijeljene biblioteke.

```
CXX = gcc
CXXFLAGS = -Wall

all: libcurve.so main

# kompajliranje test programa
main : libcurve.so main.o
	$(CXX) $(CXXFLAGS) main.o -L. -lcurve -o main
main.o : main.c circle.h
	$(CXX) $(CXXFLAGS) -c main.c

#kompajliranje shared objekta (ln -sf su simbolicki
   linkovi)
libcurve.so : point.o circle.o
	$(CXX) -shared -Wl,-soname,libcurve.so.1 -o
	   libcurve.so.1.0.1 point.o circle.o
	ln -sf libcurve.so.1.0.1 libcurve.so.1
	ln -sf libcurve.so.1.0.1 libcurve.so

# circle.o ovisi o point.o i circle.h
circle.o : circle.h point.o
	$(CXX) $(CXXFLAGS) -fPIC -c circle.c

point.o : point.h
	$(CXX) $(CXXFLAGS) -fPIC -c point.c

clean:
	rm *.o *.so main lib*
```

Može se primijetiti da su nastala tri slične .so datoteke: libcurve.so, libcurve.so.1 i libcurve.so.1.0.1. Taj način povezivanja i stvaranja simboličkih linkova naziva se *library versioning*.

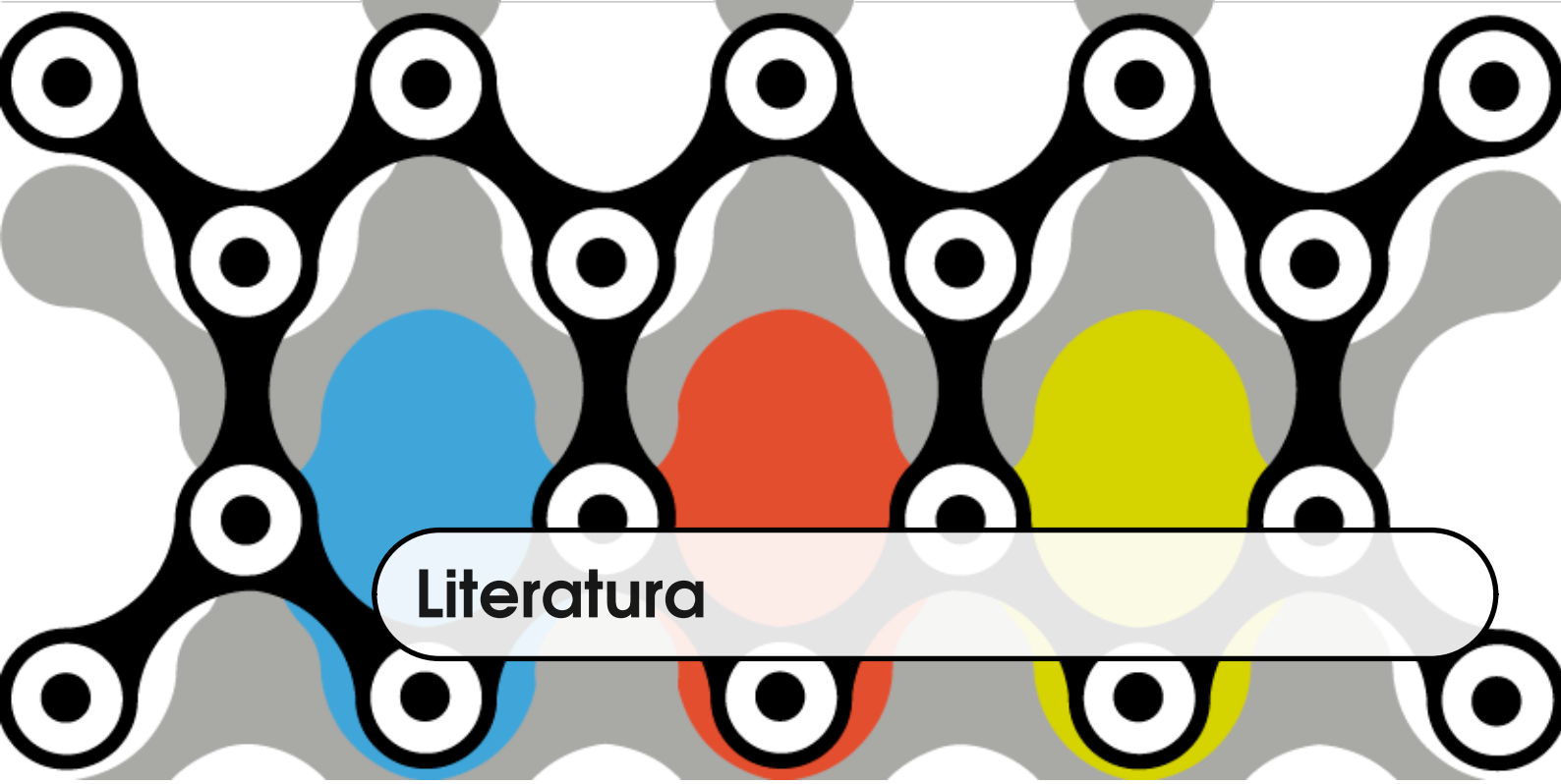
Ako pokušamo pokreniti main dobit ćemo poruku

```
./main: error while loading shared libraries: libcurve.so.1: cannot
open shared object file: No such file or directory
```

Razlog je u tome što dinamički linker ne traži .so datoteke u tekućem direktoriju. Zato se treba instalirati biblioteka u neki direktorij, te programom ldconfig dodati taj direktorij na listu direktorija koji sadrže dijeljene biblioteke. Nakon toga se taj direktorij doda u environment varijablu LD_LIBRARY_PATH.

Primjerice, ako dijeljenu biblioteku želimo staviti u mytilde/libs napravit ćemo sljedeće:

```
mkdir -p ~/libs
install -m 0755 libcurve.so libcurve.so.1 libcurve.so
   .1.0.1 ~/libs
ldconfig -n ~/libs
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:~/libs
```



Literatura

[1] Simplified windows bmp bitmap file format specification. http://www.dragonwins.com/domains/getteched/bmp/bmpfileformat.htm. [pristupljeno 13.11.2016.].

[2] ATWOOD, J. The great newline schism. http://blog.codinghorror.com/the-great-newline-schism/. [pristupljeno 13.11.2016.].

[3] BRAUER, C. Working with buffers.

[4] CARDELLI, L. Type systems. *ACM Computing Surveys 28*, 1 (1996), 263–264.

[5] ISO, I. Iec 9899: 1999: Programming languages c. *International Organization for Standardization* (1999).

[6] KAHANWAL, D., ET AL. Abstraction level taxonomy of programming language frameworks. *arXiv preprint arXiv:1311.3293* (2013).

[7] ONE, A. Smashing the stack for fun and profit. *Phrack magazine 7*, 49 (1996), 14–16.

[8] RAYMOND, E. S. The lost art of c structure packing. http://www.catb.org/esr/structure-packing/. [pristupljeno 13.11.2016.].

[9] RITCHIE, D. M. C reference manual. *Unpublished memorandum, Bell Telephone Laboratories* (1973).

[10] RITCHIE, D. M. The unix system: The evolution of the unix time-sharing system. *AT&T Bell Laboratories Technical Journal 63*, 8 (1984), 1577–1593.

[11] RITCHIE, D. M. The development of the c language. *ACM SIGPLAN Notices 28*, 3 (1993), 201–208.

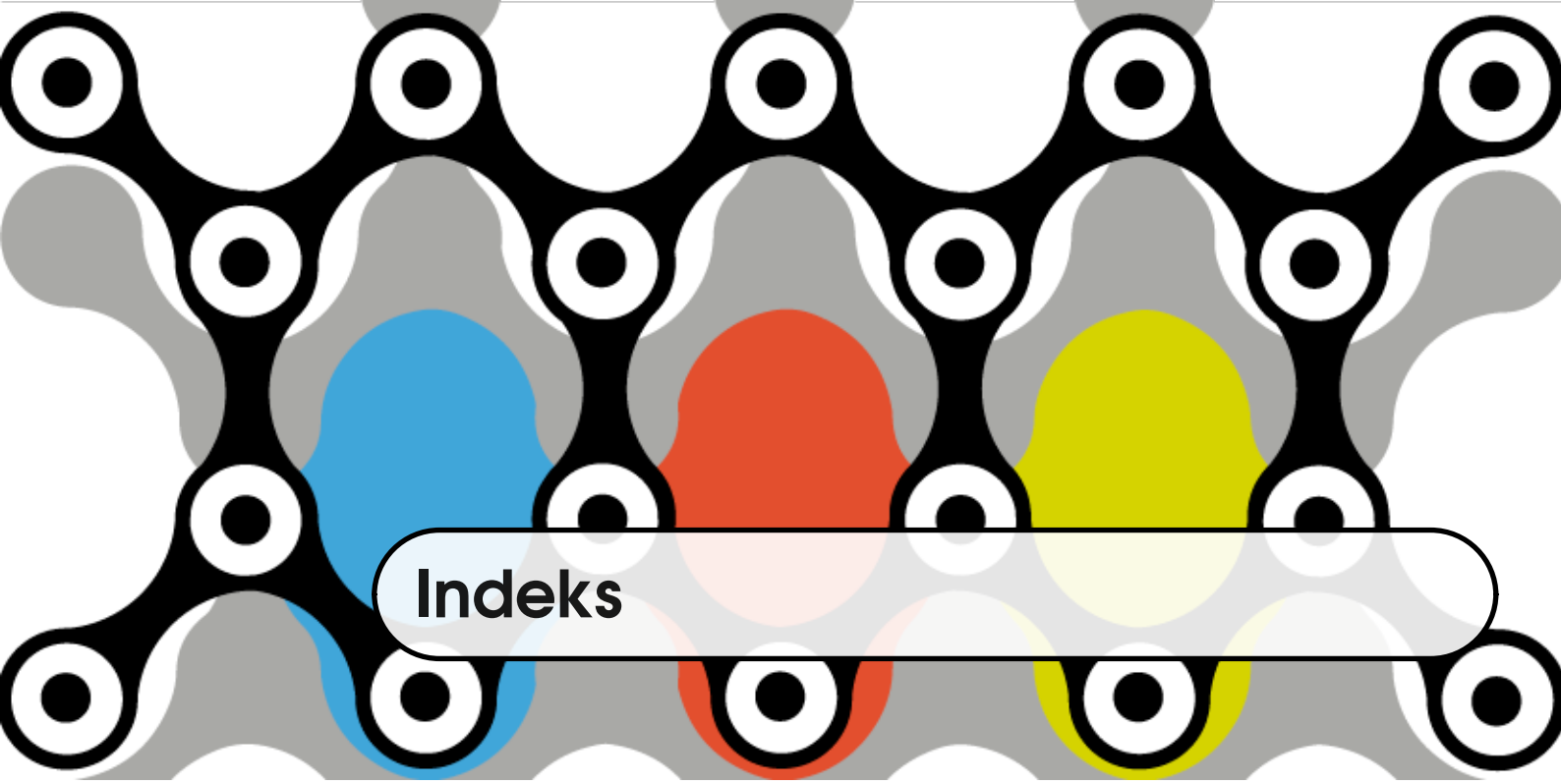
[12] RITCHIE, D. M., KERNIGHAN, B., AND LESK, M. *The C programming language*. Bell Laboratories, 1975.

[13] ROBERTS, E. *Programming Abstractions in C: A Second Course in Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[14] SEACORD, R. C. *Secure Coding in C and C++*. Pearson Education, 2005.

[15] STALLMAN, R. M., AND WEINBERG, Z. The c preprocessor. *Free Software Foundation* (1987).

[16] WULF, W., AND SHAW, M. Global variable considered harmful. *ACM Sigplan Notices 8*, 2 (1973), 28–34.



Indeks