Dennis M. Ritchie
Brian W. Kernighan

Programski jezik C

Drugo izdanje
Prijevod: Ante Denić

Predgovor

Od izdavanja "Programskog jezika C" 1978. godine, svijet računala doživio je veliki napredak. Veliki računalni sustavi postali su još snažniji, a osobna računala dobila su mogućnosti koje se do desetak godina nisu mogle nazrijeti. Za to vrijeme i sam C se mijenjao, mada neznatno, i razvijao sve dalje od svojih začetaka kao jezika UNIX operativnog sistema.

Rastuća popularnost C-a, promjene u jeziku tokom godina i kreiranje prevoditelja od strane onih kojima nije bitan izgled, kombinirano je s potrebom za preciznijom i suvremenijom definicijom jezika od one koja je bila prezentirana u prvom izdanju ove knjige. Godine 1983. American National Standard Institute (ANSI) zasniva udrugu čija je svrha bila napraviti "nedvosmislenu i od računala nezavisnu definiciju C jezika". Rezultat svega je ANSI standard za C.

Standard formalizira konstrukcije koje su bile najavljene, ali ne i opisane u prvom izdanju, kao što su dodjela strukture i nizovi dobiveni pobrojavanjem. Standard određuje novi način deklariranja funkcije koji omogućuje provjeru definicije u praksi. Određuje, također i standardnu biblioteku, s proširenim skupom funkcija za pripremu ulaza i izlaza, upravljanje memorijom, rad s nizovima i sl. Standard precizira vladanja atributa koji nisu bili u originalnoj definiciji i istovremeno jasno pokazuje koji su aspekti jezika ostali zavisni o računalu.

Drugo izdanje "Programskog jezika C" opisuje C onako kako ga definira ANSI standard (Za vrijeme pisanja ove knjige, standard je bio u završnom stadiju usvajanja; očekivalo se da bude usvojen krajem 1988.god. Razlike između onoga što piše ovdje i konačne forme standarda su minimalne.). Iako smo naznačili mjesta na kojima se jezik proširio i razvio, odlučili smo ga ekskluzivno predstaviti u novom obliku. U velikom dijelu razlike su neznatne; najuočljivija izmjena je novi način deklaracije i definicije funkcije. Moderni prevoditelji već podržavaju najveći dio standarda.

Pokušali smo zadržati suštinu prvog izdanja. C nije opširan jezik, pa ga nije potrebno opisivati opširnim knjigama. Doradili smo predstavljanje kritičnih faktora kao što su pokazivači, koji su suština C programiranja. Pročistili smo originalne primjere i dodali nove, u većini poglavlja. Na primjer, dio koji opisuje komplicirane deklaracije proširen je programima koji pretvaraju deklaracije u riječi i obratno. Kao i u prethodnom slučaju i ovdje su svi primjeri provjereni direktno iz teksta, prepoznatljivog računalu.

Dodatak A, uputa za rad, nije standard već samo pokušaj prikazivanja njegove suštine u kraćem obliku. Poradi lakšeg razumijevanja vrlo je značajno da posebna uloga pripadne samom standardu. Dodatak B predstavlja pregled karakteristika standardne biblioteke. Ovo je bitna napomena za programera, a ne za korisnika. Dodatak C pruža kratak pregled izmjena u odnosu na originalno izdanje.

Kao što smo rekli u predgovoru prvom izdanju, C je "korisniji što je veće iskustvo u radu s njim". Poslije desetogodišnjeg iskustva, to i dalje tvrdimo. Nadamo se da će vam ova knjiga pomoći da naučite i primijenite programski jezik C.

Jako su nas zadužili prijatelji koji su pomogli pri radu na drugom izdanju. Jon Bentley, Doug Gwyn, Doug McIlroy, Peter Nelson i Rob Pike komentirali suoriginalni rukopis. Zahvaljujemo Alu Ahou, Dennisu Allisonu, Joeu Campbellu, G. R. Emlinu, Karen Fortgang, Allenu Holubu, Andrewu Humeu, Daveu Kristolu, Johnu Lindermanu, Davidu Prosseru, Gene Spafford i Chrisu Van Wyku na pažljivoj kontroli napisanog teksta. Svojim primjedbama pomogli su nam i Bill Chestwick, Mark Kernighan, Andy Koenig, Robin Lake, Tom London, Jim Reeds, Clovis Tondo i Peter Weinberger. Dave Prosser je detaljno odgovorio na mnoga pitanja glede ANSI standarda. Koristili smo Bjarne Stroustrupov C++ translator za lokalno testiranje naših programa, a Dave Kristol nam je nabavio C prevoditelj kojim su obavljena završna testiranja. Rich Drechsler nam je mnogo pomogao pri pisanju teksta na računalu. Svima im iskreno zahvaljujemo.

Brian W. Kernighan
Dennis M. Ritchie

Predgovor prvom izdanju

C je programski jezik opće namjene koji karakterizira mali broj izraza, moderna kontrola tijeka i strukture podataka kao i veliki broj operatora. C nije "high level" jezik niti je opširan, a nije namijenjen nekoj posebnoj vrsti primjene. Međutim, općenitost i nepostojanje ograničenja čine ga prihvatljivijim i efikasnijim od drugih programskih jezika.

C je u originalu kreirao i primijenio Dennis Ritchie. Operativni sustav, C prevoditelj i sve UNIX-ove aplikacije (uključujući cjelokupan software korišten u pripremi ove knjige) su napisani u C-u. Napravljeni su prevoditelji koji se vrte na većem broju računala, uključujući IBM System/370, Honeywell 6000 i InterData 8/32. C nije neposredno povezan s nekim posebnim sklopovljem ili sustavom, ali je lako napisati programe koji se daju, bez ikakvih izmjena koristiti na bilo kojem računalu koje podržava C.

Ova knjiga je korisna, jer pomaže čitatelju naučiti programirati u C programskom jeziku. Ona daje osnovne napomene, koje omogućuju novim korisnicima početak rada u najkraćem mogućem roku, posebna poglavlja o svakoj važnijoj temi, te uputstvo za rad. Najveći dio operacija temelji se na čitanju, pisanju i razradi primjera, a ne na šturom predstavljanju postojećih pravila. Najčešće, primjeri su cjeloviti programi, a ne izolirani dijelovi programa. Svi primjeri su provjereni direktno iz teksta, koji je napisan u obliku prepoznatljivog računalu. Pored predstavljanja načina na koje se jezik dade najbolje upotrijebiti, trudili smo se, gdje god je to bilo moguće, prikazati korisne algoritme, dobro kreirane i pravilno koncipirane principe.

Knjiga ne predstavlja uvod u programiranje; ona čini bliskim osnovne čimbenike programiranja kao npr. varijable, petlje i funkcije. Iako je ovako početnicima omogućeno upoznavanje i korištenje jezika, više će mu koristiti konzultacije s kolegama koji bolje barataju s C-om.

Naše iskustvo pokazuje da je C pogodan za rad, sadržajan i raznovrstan jezik za najveći broj programa. Lak je za učenje, i sve korisniji što je veće iskustvo u radu s njim. Nadamo se da će vam ova knjiga pomoći da ga što bolje savladate.

Sadržajne kritike i primjedbe mnogih prijatelja i kolega ovoj knjizi doprinijele su da je sa zadovoljstvom napišemo. Naročito su Mike Bianchi, Jim Blue, Stu Feldman, Doug McIlroy, Bill Roome, Bob Rosin i Larry Rosler pažljivo pregledali mnoge inačice. Zahvaljujemo također Alu Ahou, Steveu Bourneu, Danu Dvoraku, Chucku Haleyu, Debbie Haleyu, Marion Harris, Ricku Holtu, Steveu Johnsonu, Johnu Masheyu, Bobu Mitzeu, Ralphu Muhau, Peteru Nelsonu, Elliotu Pinsonu, Billu Plaugeru, Jerryu Spivacku, Kenu Thompsonu i Peteru Weinebergeru na komentarima koji su bili od pomoći u različitim situacijama, te Mikeu Lesku i Joeu Ossannai na svesrdnoj pomoći pri obradi teksta.

Brian W. Kernighan
Dennis M. Ritchie

UVOD

C je programski jezik opće namjene. Tijesno je povezan s operativnim sistemom UNIX na kojemu je razvijen, jer su i sistem i većina programa koji rade na UNIX-u napisani baš u C-u. Jezik, ipak, nije vezan samo za jedan operativni sistem ili računalo; iako je nazvan "jezikom za sistemsko programiranje" zato što se koristi pri pisanju prevoditelja i operativnih sistema, podjednako se dobro koristi za programiranje u drugim područjima. Većina bitnih ideja C-a potječe od jezika BCPL koji je razvio Martin Richards. Utjecaj BCPL-a na C ostvaren je indirektno preko B jezika koji je 1970. napisao Ken Thompson za prvi UNIX sistem na DEC PDP-7 računalu.

BCPL i B su jezici bez "tipova podataka". Nasuprot njemu, C nudi mnoštvo različitih tipova podataka. Osnovni tipovi su znaci, cjelobrojne vrijednosti i vrijednosti iz područja realnih brojeva (vrijednosti s pomičnim zarezom) u više veličina. Uz to postoji hijerarhija izvedenih tipova podataka kreiranih pokazivačima, poljima, strukturama i unijama. Izrazi se sastoje od operatora i operanda; bilo koji izraz, uključujući i dodjelu vrijednosti ili pozivanje funkcije, može biti naredba. Pokazivači omogućuju nezavisnu adresnu aritmetiku.

C nudi osnovne konstrukcije za kontrolu toka koje traže dobro strukturirani programi: grupiranje naredbi, donošenje odluka (if-else), izbor (switch), petlje s uvjetima na početku (while) i na kraju (do), te izlaz iz petlje prije kraja (break).

Funkcije mogu vraćati vrijednosti osnovnih tipova, struktura, unija ili pokazivača. Bilo koja funkcija može se rekurzivno pozivati. Lokalne varijable su tipično "automatske" (gube vrijednost pri izlasku iz funkcije) ili se kreiraju svakim novim pozivanjem. Definicije funkcija ne moraju se umetati, a varijable se mogu deklarirati u blokovima. Funkcije C programa mogu se nalaziti u različitim izvornim datotekama koje se posebno prevode. Varijable mogu biti unutrašnje, vanjske (za koje se zna samo unutar jedne izvorne datoteke) ili dostupne cijelom programu (globalne).

Preprocesorska faza obavlja makrosupstitucije na izvornom tekstu programa, uključivanje ostalih izvornih datoteka i uvjetno prevođenje.

C je jezik relativno "niskog nivoa". Ovakav epitet nije nedostatak, već govori da C radi s istim vrstama objekata s kojima rade i sama računala, a to su znakovi, brojevi i adrese. Ovi objekti se mogu kombinirati i premještati pomoću aritmetičkih i logičkih operatora kojima su opremljena postojeća računala.

C ne radi direktno sa složenim objektima kao što su nizovi znakova, skupovi, liste ili matrice. Ne postoje operacije koje obrađuju cijelu matricu ili niz, iako strukture mogu biti kopirane kao jedinka. C ne definira ni jednu drugu mogućnost memoriranja lokacija osim statičke definicije i discipline stoga, koja je omogućena lokalnim varijablama funkcija; ovdje nema nagomilavanja ili skupljanja nebitnih elemenata. Na kraju, sam C ne nudi ulazno/izlazne olakšice; u njemu ne postoje READ ili WRITE stanja, te nema ugrađenih metoda za pristup datotekama. Svi ovi mehanizmi "višeg nivoa" moraju biti određeni funkcijama koje se zovu eksplicitno. Manje-više sve implementacije C-a imaju standardnu kolekciju takovih funkcija.

Shodno tomu, C zapravo nudi samo jednoznačni kontrolni tok: uvjeta, petlji, grupiranja i potprograma, ali ne i multiprogramiranje, paralelne operacije ili sinkronizaciju.

Iako nepostojanje neke od ovih karakteristika može izgledati kao ozbiljan nedostatak ("Znači da bih usporedio dva znakovna niza moram pozivati funkciju?"), održavanje jezika na umjerenoj razini ima svoju stvarnu korist. Pošto je C relativno mali jezik, dade se opisati na relativno malo prostora i naučiti brzo. Programer s punim pravom može očekivati lako učenje i razumijevanje korektne upotrebe cijelog jezika.

Dugi niz godina, jedina definicija C-a je bio referentni priručnik prvog izdanja ove knjige. American National Standards Institute (ANSI) je 1983.god. osnovao udrugu koja se skrbila za modernu i cjelovitu definiciju C-a. Očekuje se da će ANSI standard, ili ANSI C biti odobren u 1988.god. (odobren je op.prev). Sve karakteristike standarda već su podržane preko novih prevoditelja.

Standard se bazira na originalnom referentnom priručniku. Jezik je razmjerno malo mijenjan; jedan od ciljeva standarda je bio osigurati da većina postojećih programa ostane primjenjiva, ili, ako to ne uspije, prevoditelji moraju dati upozorenje o drugačijem načinu rada.

Za većinu programera, najbitnija promjena je u novoj sintaksi za deklariranje i definiranje funkcija. Deklaracija funkcije može sada imati opis argumenata funkcije; sintaksa definicije je na određen način izmijenjena. Ova dodatna informacija pomaže da prevoditelji mnogo lakše otkrivaju greške nastale neslaganjem argumenata; po našem iskustvu, to je vrlo koristan dodatak jeziku.

Postoje i još neke, manje izmjene. Dodjela struktura i nizova dobivenih pobrojavanjem, koji su se naširoko primjenjivali, postali su i zvanično dio jezika. Izračunavanje realnih brojeva može se obaviti i jednostrukom točnošću. Aritmetička svojstva, posebice za neoznačene tipove su razjašnjena. Preprocesor je savršeniji. Većina od ovih promjena ipak neće previše zanimati programere.

Drugi značajan doprinos standarda je definicija biblioteke koja prati C jezik. Ona određuje funkcije za pristup operativnom sistemu (npr. za čitanje i pisanje datoteka), formatira ulaz i izlaz, određuje položaj u memoriji, radi s nizovima i sl. Zbirka standardnih zaglavlja osigurava jednoznačan pristup deklaracijama

funkcija i tipovima podataka. Da bi mogli biti u vezi sa osnovnim sistemom, programi što koriste ovu biblioteku su zasigurno kompatibilni i portabilni. Mnoge biblioteke su vrlo slične modelu standardne ulaz/izlaz biblioteke UNIX sistema. Ova biblioteka je opisana u prvom izdanju i masovno je bila u upotrebi na drugim sistemima. Kažimo ipak, još jednom, da mnogi programeri neće uočiti bitnije izmjene.

Zbog toga što tipove podataka i kontrolnih struktura određenih C-om podržava veliki broj računala, radna biblioteka koja je snabdjevena vlastitim programima jest mala. Jedino se funkcije iz standardne biblioteke pozivaju eksplicitno, a i one se mogu zaobići. Te funkcije daju se napisati u C-u, te prenijeti s računala na računalo, osim onih koje neposredno oslikavaju konkretno računalo na kojemu se radi i njegov operativni sistem.

Iako C odgovara mogućnostima većine računala, on je nezavisan od konkretne arhitekture računalskog sustava. Sa malo pažnje lako je napisati prenosive programe, što znači programe koje možemo pokrenuti bez zahtjeva za sklopovskim promjenama. Standard čini sve aspekte prenosivosti eksplicitnim, a propisuje i skup konstanti koje karakteriziraju računalo na kojem se program vrti.

C nije strogo tipiziran, ali kako se razvijao, njegova kontrola tipova je jačala. Originalna definicija C-a ne odobrava, ali dopušta zamjenu pokazivača i cijelih brojeva; nakon dužeg razdoblja i to je riješeno, i standard zahtjeva točne deklaracije i jasne konverzije koje su činili dobri prevoditelji. Nove deklaracije funkcija su slijedeći korak u tom smjeru. Prevoditelji će upozoriti na većinu tipskih grešaka, mada ne postoji automatsko pretvaranje neusuglašenih tipova podataka. C ipak zadržava osnovnu filozofiju koju programeri poznaju; on samo zahtjeva jasno definiranje ciljeva.

C, kao i svaki drugi jezik ima svoje nedostatke. Neki operatori imaju pogrešan prioritet; neki dijelovi sintakse mogli bi biti bolji. Pored svega, C se dokazao kao jedan od najkorisnijih i najsadržajnijih za veliki broj različitih aplikacija.

Knjiga je organizirana na slijedeći način:

Poglavlje 1 je udžbenik osnovnih svojstava C-a. Prvotna namjera bila je pripremanje čitatelja za početak rada u najkraćem mogućem roku, jer vjerujemo da je najbolji način učenja jezika pisanje različitih programa u njemu. Baza podrazumijeva znanje stečeno radom s osnovnim elementima programiranja; ovdje nema pojašnjavanja u vezi s računalom, prevođenjem, ali ni značenja izraza kao što je n=n+1. Iako smo pokušali da, gdje god je takvo što bilo moguće, prikažemo korisne programske tehnike, knjiga nema namjeru biti udžbenik za strukture podataka i algoritme; kad god smo bili prinuđeni birati, usredotočili bismo se na jezik.

Poglavlja 2 i 6 razmatraju različite aspekte jezika C s više detalja, te mnogo formalnije negoli Pogavlje 1, iako je naglasak još uvijek na cjelovitim programskim primjerima, a ne na izoliranim fragmentima. Poglavlje 2 bavi se osnovnim tipovima podataka, operatorima i izrazima. Poglavlje 3 obrađuje kontrolu toka: if-else, switch, while, for itd. Poglavlje 4 pokriva funkcije i programske strukture - vanjske varijable, pravila područja, umnožene izvorne datoteke, itd., a također se dotiče i preprocesora. Poglavlje 5 bavi se pokazivačima i aritmetičkim adresama. Poglavlje 6 govori o strukturama i unijama. Poglavlje 7 opisuje standardnu biblioteku, koja oprema operativni sistem jednostavnim interface-om. Ova biblioteka je definirana ANSI standardom kojeg podržava C program na svakom računalu, pa se programi koji ga koriste za ulaz, izlaz i pristup drugom operativnom sistemu mogu prenijeti s sistema na sistem bez izmjena.

Poglavlje 8 opisuje vezu između C programa i operativnog sistema UNIX, usredotočivši se na ulaz/izlaz, sistem datoteka i raspodjelu memorije. Iako je dio ovog poglavlja specifičan za UNIX sisteme, programeri koji koriste ostale sisteme još uvijek mogu naći koristan materijal, uključujući i neke detaljnije uvide u to kako je jedna verzija standardne biblioteke opremljena, te primjedbe o prenosivosti.

Dodatak A sadrži referentni priručnik. Zvanični prikaz sintakse i semantike C-a je sam ANSI standard. Ovaj dokument je najprije namijenjen piscima prevoditelja. Referentni priručnik ovdje prikazuje definiciju jezika konciznije, a ne na uobičajeni, klasičan način. Dodatak B daje sadržaj standardne biblioteke, koji je potrebniji za korisnike programa nego za one koji ih prave. Dodatak C daje kratak pregled izmjena originalnog jezika. U slučaju sumnje, pak, standard i vlastiti prevoditelj ostaju najkompetentniji autoriteti za jezik.

POGLAVLJE 1: OSNOVNE NAPOMENE

Krenimo s brzim uvodom u C. Namjera nam je bila prikazati osnovne elemente jezika kroz realne programe, ali bez ulaženja u detalje, pravila i izuzetke. U ovom trenutku, ne pokušavamo biti općeniti kao ni precizni (podrazumijeva se da su primjeri koje namjeravamo prikazati karakteristični). Želimo vas samo na najbrži mogući način dovesti do razine kad sami možete pisati korisničke programe, a za to, trebamo se usredotočiti na osnove: varijable i konstante, aritmetiku, kontrolu toka, funkcije i osnove ulaza i izlaza. Namjerno smo izostavili iz ovog poglavlja karakteristike C-a koje su bitne za pisanje većih programa. Tu spadaju pokazivači, strukture, veliki broj operatora u C-u, nekoliko naredbi kontrole toka, te standardna biblioteka.

Ovaj pristup ima i nedostatke. Najuočljiviji je činjenica da se ovdje ne može naći kompletan prikaz neke određene karakteristike jezika, a baza ako nije potpuna, također nije efikasna. Kako prikazani primjeri ne koriste sasvim kapacitete C-a, oni nisu tako koncizni i elegantni kakvi bi mogli biti. Tek toliko da znate, takve smo dojmove pokušali umanjiti. Slijedeći nedostatak je što će se u kasnijim poglavljima ponavljati nešto iz ovog poglavlja. Smatramo da je ponavljanje majka znanja.

U svakom slučaju, iskusni programeri će biti u mogućnosti izvući najbitnije iz ovog poglavlja prema svojim potrebama. Početnici će to nadoknaditi pišući male programe, slične priloženima. Obje grupe moći će koristiti ovo poglavlje kao bazu na kojoj će se zasnivati detaljniji opisi s početkom u Poglavlju 2.

1.1 Puštanje u rad

Jedini način učenja novog programskog jezika jest pisanje programa u njemu. Prvi program koji ćemo napisati isti je za sve jezike:

Ispiši riječi

```
Hello, World
```

Pojavila se velika prepreka: da bi je savladali, morate moći kreirati izvorni tekst, uspješno ga prevesti, učitati, pokrenuti i pronaći gdje mu se pojavljuje izlaz. U usporedbi s ovim tehničkim detaljima, kad ih svladate, sve drugo je lako.

U C-u, program za ispis "Hello, World" jest

```
#include <stdio.h>

main(){
      printf("Hello, World\n");
}
```

Kako će se ovaj program pokrenuti zavisi o sistemu koji koristite. Kao specifičan primjer, na UNIX operativnom sistemu možete kreirati program u datoteci čije se ime završava sa ".c", kao što je hello.c, koji se zatim prevodi naredbom

```
cc hello.c
```

Ako niste nigdje pogriješili, bilo da ste izostavili neki znak ili neko slovo pogrešno napisali, prevođenje će se obaviti i stvorit će se izvršna datoteka s imenom a.out. Ako pokrenete a.out tipkajući naredbu

```
a.out
```

on će ispisati

```
Hello, World
```

Na drugim sistemima, vrijedit će druga pravila; to možete priupitati nekog s iskustvom.

Sad ćemo se pozabaviti samim programom. C program, bilo koje veličine, sastoji se od funkcija i varijabli. Funkcija se sastoji od naredbi koje određuju operacije koje treba izvršiti, a varijable imaju vrijednosti koje koristimo tijekom njihova izvršenja. C funkcije izgledaju kao potprogrami i funkcije Fortrana ili procedure i funkcije Pascala. U našem primjeru je funkcija s imenom main. Naravno, vi ste slobodni dati

funkcijama imena po želji, ali ime "main" ima specijalnu namjenu - program se izvršava od početka funkcije main. Ovo znači da svaki program mora imati "main" negdje.

Funkcija main će obično pozivati ostale funkcije da omoguće njeno odvijanje, i to neke koje ste vi napisali, a druge iz ponuđenih biblioteka. Prva linija programa,

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

govori računalu da uključi informacije o standardnoj ulazno/izlaznoj biblioteci; ova linija se pojavljuje na početku mnogih C izvornih datoteka. Standardna biblioteka je opisana u Poglavlju 7 i Dodatku B.

Jedan način razmjene podataka između funkcija jest određivanje funkcijske liste vrijednosti, koje se zovu argumentima pozivne funkcije. U ovom slučaju main je definirana kao funkcija koja ne očekuje nikakve argumente, što je predstavljeno praznom listom ().

Naredbe funkcije su ograđene velikim zagradama {}. Funkcija main ima samo jednu naredbu

```
printf("Hello, World\n");
```

Funkcija se poziva njenim imenom, a popraćena je listom argumenata u zagradi. Tako, pozivamo funkciju printf argumentom "Hello, World\n". Funkcija printf je iz biblioteke koja ispisuje izlaz, u ovom slučaju niz znakova između navodnika. Niz znakova između dvostrukih navodnika, kao što je "Hello, World\n", zove se znakovni niz. U početku ćemo koristiti znakovne nizove samo kao argumente za printf i ostale funkcije.

Dio \n u nizu je oznaka u C-u za znak novog reda, koji kada se ispisuje pomiče ispis do kraja ulijevo na slijedećoj liniji. Ako izostavite \n (potencijalno koristan eksperiment, a nikako štetan), uočit ćete da nakon ispisa izlaza ne postoji više nijedna linija. Morate upotrijebiti \n da bi se znak novog reda priključio printf argumentu; ako napišete nešto kao

```
printf("Hello, World
");
```

C prevoditelj će prijaviti grešku.

Nikada printf ne određuje novu liniju automatski, već se iz višestrukih pozivanja može postupno formirati izlazna linija. Naš prvi program mogao je biti ovako napisan

```
#include <stdio.h>
main(){
      printf("Hello, ");
      printf("World");
      printf("\n");
}
```

da bi načinio isti izlaz.

Primijetimo da \n predstavlja samo jedan znak. Niz kakav je \n osigurava opći mehanizam za predstavljanje znakova koje nazivamo specijalnima (zato što su nevidljivi ili se ne daju otipkati). Između ostalih nizova koje C definira su \t za tabulator, \b za povratnik (backspace), \" za dvostruke navodnike i \\ za obrnutu kosu crtu (slash). Potpuna lista postoji u dijelu pod naslovom 2.3.

Vježba 1-1. Pokrenite "Hello, World" program na vašem računalu. Igrajte se s izostavljanjem dijelova koda, samo radi poruka o greškama koje ćete dobivati.

Vježba 1-2. Što se događa kad niz argumenata za printf sadrži \c, gdje je c neki znak koji nije na listi navedenoj gore?

1.2 Varijable i aritmetički izrazi

Slijedeći program koristi formulu $^{o}C=(5/9)(^{o}F-32)$ da ispiše tablicu temperatura u Fahrenheitovim stupnjevima i njihove ekvivalente u Celsiusovim:

```
0     -17
20    -6
40    4
60    15
```

```
80    26
100   37
120   48
140   60
160   71
180   82
200   93
220   104
240   115
260   126
280   137
300   148
```

Program se sastoji od definicije funkcije pod imenom main. On je duži od prvog, koji ispisuje "Hello, World", ali ne i složeniji. Program uvodi dosta novih ideja, uključujući komentare, deklaracije, varijable, aritmetičke izraze, petlje i formatirani izlaz.

```
#include <stdio.h>

/* Ispiši Fahrenheit-Celsius tablicu za fahr=0, 20, 40, ..., 300 */
main(){
      int fahr, celsius;
      int lower, upper, step;
      lower=0;    /* donja granica tablice tempreratura */
      upper=300;  /* gornja granica */
      step=20;
      fahr=lower;
      while(fahr<=upper){
            celsius=5*(fahr-32)/9;
            printf("%d\t%d\n", fahr, celsius);
            fahr=fahr+step;
      }
}
```

Linija

```
/* Ispiši Fahrenheit-Celsius tablicu za fahr=0, 20, 40, ..., 300 */
```

predstavlja komentar, koji u prikazanom primjeru pojašnjava što program točno radi. Bilo koje znakove između /* i */ prevoditelj ne prepoznaje; oni se slobodno mogu upotrebljavati kako bi se program lakše razumijevao. Komentari može imati i razmak (pusti znak?!), tabulator ili znak nove linije.

U C-u sve varijable moraju biti prijavljene prije upotrebe, najčešće na početku funkcije, prije bilo koje izvršne naredbe. Deklaracija prijavljuje osobine varijabli; ona se sastoji iz imena tipa i liste varijabli kao npr.:

```
int fahr, celsius;
int lower, upper, step;
```

Tip int znači da su navedene varijable cijeli brojevi, za razliku od funkcije float, koja označava realne brojeve, tj. brojeve koji mogu imati decimalni dio. Opseg obaju tipova, i int i float, zavisi o mogućnostima računala na kojemu radite: 16-bitni int brojevi imaju opseg [-32768 - 32767], a jednostavni su kao i 32-bitni. Tipična dužina float broja je 32 bita, sa posljednjih šest značajnih znamenki i najčešćom vrijednošću između [10-38 - 10+38].

C nudi i druge osnovne tipove podataka pored int i float tipa, uključujući:

`char`	znakovni tip dužine jednog okteta
`short`	kraći oblik cijelog broja
`long`	duži oblik cijelog broja
`double`	realan broj s dvostrukom točnošću

Veličine ovih objekata isto tako variraju od računala do računala. Tu su i polja, strukture i unije ovih osnovnih tipova, pokazivači na njih, te funkcije koje ih vraćaju, a koje ćemo tek upoznati.

Izračunavanje programa konverzije temperature počinje s naredbama dodjele

```
lower=0;
upper=300;
step=20;
fahr=lower;
```

koji predstavljaju varijable na početne vrijednosti. Pojedinačne naredbe se završavaju točkom-zarezom.

Svaka linija u tablici izračunava se na isti način, pa zato koristimo petlju koja se ponavlja jednom po izlaznoj liniji; to je svrha while petlje.

```
while(fahr<=upper){
      ...
}
```

Petlja while radi na slijedeći način: Uvjet u zagradi se provjeri. Ako je ispunjen(fahr je manji ili jednak upper), tijelo petlje (tri naredbe unutar vitičastih zagrada) se izvršavaju. Zatim se uvjet nanovo testira, i ako je ispunjen, ponovno se izvršavaju. Kada uvjet ne bude istinit (fahr dosegne upper), petlja se završava, a izvršenje programa nastavlja prvom naredbom iza petlje. Kako u prikazanom programu nema više naredbi, on se završava. Tijelo while petlje može biti jedna ili više naredbi u zagradama (vitičastim), kao u programu za pretvaranje temperatura ili jedna naredba bez zagrada, kao

```
while(i<j)
      i=2*i;
```

U oba slučaja, naredbe kontrolirane pomoću while uvučene su tabulatorom (ma koliko razmaka on iznosio) tako da su lako uočljive. Ovakvo lociranje ističe logičku strukturu programa. Iako C prevoditelji ne vode računa o tome kako program izgleda, pravilno definiranje i dimenzioniranje bitno je radi čitljivosti programa. Mi preporučujemo pisanje samo jedne naredbe po liniji, te upotrebu razmaka oko operatora radi razdvajanja grupacija. Pozicija zagrada manje je bitna. Mi smo izabrali samo jedan od jednostavnih stilova. Vi odaberite stil koji vam odgovara, ali budite dosljedni.

Najveći dio posla obavi se u tijelu petlje. Celsiusova temperatura se izračunava i pridjeljuje varijabli celsius naredbom:

```
celsius=5*(fahr-32)/9;
```

Razlog množenja s 5 i dijeljenja s 9 umjesto samo množenja s 5/9 je da u se C-u, kao i u mnogim jezicima, kao rezultat dijeljenja dva cijela broja dobiva cijeli broj i ostatak koji se odbacuje. Kako su 5 i 9 cijeli brojevi, 5/9 bio bi jednak nuli, te bi sve Celsiusove temperature bile nula.

Ovaj princip također pokazuje malo bolje kako printf radi. Funkcija printf je izlazna formatirajuća funkcija opće namjene, koju ćemo detaljno opisati u Poglavlju 7. Njen prvi argument je niz znakova za ispis, koji svakim % pokazuje gdje su neki od slijedećih argumenata zamijenjeni i u kojoj će formi biti ispisana. Npr. %d određuje cjelobrojni argument, tako da naredba

```
#include <stdio.h>
printf("%d\t%d\n", fahr, celsius);
```

ispisuje vrijednosti dvaju cijelih brojeva fahr i celsius, sa tabulatorom (\t) između njih.

Svaka % konstrukcija u prvom argumentu printf-a je u skladu s argumentima iza navodnika (drugim, trećim, ...); oni moraju biti složeni točno po tipu i broju, inače ispis neće biti korektan.

Uzgred, printf nije dio C-a; ne postoji ulaz i izlaz definiran u samom C-u. Funkcija printf je korisna funkcija iz standardne biblioteke funkcija koje su dostupne C programima. Rad printf funkcije je definiran prema ANSI standardu, pa su njene osobine iste kod svih prevoditelja koji mu odgovaraju.

Da bismo se koncentrirali na sam jezik C, nećemo mnogo govoriti o ulazu i izlazu do Poglavlja 7. Detaljno formatiranje ulaza ćemo odložiti do tada. Ako treba unositi brojeve, pročitajte diskusiju o funkciji scanf u dijelu pod naslovom 7.4. Funkcija scanf liči na printf, osim što učitava s ulaza, a ne ispisuje na izlaz.

Postoji nekoliko problema u vezi s programom za pretvaranje temperature. Najjednostavniji je vezan uz izgled izlaznog dokumenta jer brojevi nisu udesno poravnati. To je lako riješiti; ako povećamo svaki %d u printf naredbi po širini, ispisani brojevi će biti poravnati s desne strane u svojim poljima. Npr. mogli smo kazati

```
printf("%3d %6d\n", fahr, celsius);
```

da bi se ispisao prvi broj svake linije u polju širokom tri znamenke, te drugi u polju širokom šest znamenki, kao:

```
  0   -17
 20   -6
 40    4
 60   15
 80   26
100   37
...
```

Ozbiljniji problem nalazi se u činjenici da koristimo aritmetiku cijelih brojeva, pa Celsiusove temperature nisu baš egzaktne; npr., 0^oF jest zapravo -17.8^oC, a ne -17. U cilju rješavanja ovakovih problema, potrebno je upotrebljavati aritmetiku realnih brojeva (brojevi s pokretnom decimalnom točkom) umjesto cijelih. To zahtjeva određene izmjene u programu. Postoji i druga verzija:

```
#include <stdio.h>
/* Ispiši Fahrenheit-Celsius tablicu za fahr=0, 20, 40, ..., 300 */
main(){
      float fahr, celsius;
      int lower, upper, step;
      lower=0;    /* donja granica tablice tempreratura */
      upper=300;  /* gornja granica */
      step=20;
      fahr=lower;
      while(fahr<=upper){
            celsius=(5.0/9.0)*(fahr-32);
            printf("%3.0f %6.1f\n", fahr, celsius);
            fahr=fahr+step;
      }
}
```

Ova verzija je jako slična prethodnoj, osim što su varijable fahr i celsius tipa float, a izraz za konverziju napisan prirodnije. Nismo bili u prilici koristiti 5/9 u ranijoj verziji, jer bi dijeljenje cijelih brojeva dalo vrijednost nula. Decimalna točka u konstanti kaže da je to realan broj, pa 5.0/9.0 ne daje nulu jer je to odnos dviju realnih veličina.

Ako aritmetički operator ima cjelobrojne operande, obavlja se operacija sa cijelim brojevima. Ako aritmetički operator ima jedan realan broj kao operand, a cijeli broj kao drugi, cijeli će se broj pretvoriti u realni da bi se operacija izvela kako treba. Ako napišemo fahr-32, 32 će biti pretvoreno u realan broj. Ipak, blaga je preporuka da se realni brojevi pišu s eksplicitnim decimalnim točkama čak i kad imaju vrijednosti kao cjelobrojni, jer tako naglašavamo njihovu realnu prirodu.

Detaljnija pravila o pretvaranju cjelobrojnih varijabli u one s pomičnim zarezom dani su u Poglavlju 2. Za sada primijetimo da naredba

```
fahr=lower;
```

i uvjet

```
while(fahr<=upper)
```

djeluje na isti način - int se pretvara u float prije negoli se obavi operacija.

Parametar printf pretvorbe %3.0f nam kaže da realni broj (u ovom slučaju fahr) treba biti ispisan sa zauzećem najviše tri znaka, bez decimalne točke i decimalnih znamenki. Parametar %6.1 opisuje drugi broj (celsius) koji treba biti ispisan u najviše šest znakova, s jednom znamenkom iza decimalne točke. Izlaz izgleda ovako:

```
  0   -17.8
 20   -6.7
```

```
 40     4.4
 ...
```

Širina i točnost ne moraju biti navedene: %6f kaže da broj mora biti širok najviše šest znakova; %.2f određuje dva znaka iza decimalne točke, ali širina nije naglašena; %f jedino kazuje kako se radi o broju s pomičnom decimalnom točkom.

%d	ispiši kao decimalni cijeli broj
%6d	ispiši kao decimalni cijeli broj, širok najviše šest znakova
%f	ispiši kao realan broj
%6f	ispiši kao realan broj, širok najviše šest znakova
%.2f	ispiši kao realan broj, sa dva znaka iza decimalne točke
%6.2f	ispiši kao realan broj, širok najviše šest znakova, sa dva znaka iza decimalne točke

Između ostalog, printf prepoznaje %0 kao oktalni broj, %x kao heksadecimalni, %c kao znak, %s kao niz znakova i %% kao %.

Vježba 1-3. Izmijenite program pretvorbe temperature da ispisuje zaglavlje iznad tablice.

Vježba 1-4. Napišite odgovarajući program za ispis tablice pretvorbe Celsiusovih u Fahrenheitove stupnjeve.

1.3 Programska petlja for

Mnogo je različitih načina na koje se dade napisati program određene namjene. Pokušat ćemo napraviti promjenu na pretvaraču temperature

```
#include <stdio.h>
/* ispiši Fahrenheit-Celsius tablicu */
main(){
      int fahr;
      for(fahr=0;fahr<=300;fahr=fahr+20)
            printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Ovo daje iste rezultate, ali svakako izgleda drugačije. Najbitnija izmjena je eliminacija najvećeg broja varijabli; ostaje samo fahr, koja postaje int. Donje i gornje granice i korak pojavljuju se samo kao konstante u for petlji, koji je konstruiran drugačije, a izraz koji izračunava temperaturu u Celsiusovim stupnjevima se sada javlja kao treći argument u printf funkciji umjesto kao posebna naredba dodjele.

Ova zadnja izmjena je primjer općeg pravila - u bilo kojem kontekstu gdje je dopušteno koristiti vrijednost varijable određenog tipa, možete upotrijebiti kompliciraniji izraz tog tipa. Kako treći argument funkcije printf mora biti realna vrijednost, tu možemo umetnuti bilo koji realan izraz.

Naredba for je petlja koja predstavlja generalizirani while oblik. Ako ih usporedimo uočit ćemo da for ima efikasnije djelovanje. U zagradama , postoje tri dijela odvojena točka-zarezom. Prvi dio, početak

```
fahr=0
```

se izvrši jednom, prije negoli se uđe u petlju. Drugi dio je uvjet koji kontrolira petlju:

```
fahr<=300
```

Taj uvjet se izračunava; u slučaju da je istinit, tijelo petlje (u ovom slučaju printf) se izvršava. Zatim se varijabla iz uvjeta inkrementira (povećava) za korak

```
fahr=fahr+20
```

a uvjet se ponovo računa. Petlja se završava ako je uvjet postao lažan. Kao i kod while, tijelo petlje može biti jedna naredba ili grupa naredbi uokvirena vitičastim zagradama. Inicijalizacija, uvjet i inkrementacija mogu biti bilo koji izrazi.

Izbor između while i for je slobodan, a temelji se na povoljnijoj soluciji. Najčešće se for koristi u petljama gdje su inicijalizacija i inkrementacija pojedinačni izrazi, ali logički povezani, što je kompaktnije nego while petlja koja ima naredbe za inicijalizaciju, uvjete i inkrementaciju međusobno razmještene.

Vježba 1-5. Izmijenite program pretvorbe temperature tako da program ispiše tablicu obrnutim redoslijedom, od 300 stupnjeva do 0.

1.4 Simboličke konstante

Možemo se posljednji put osvrnuti na pretvorbu temperature prije negoli je zaboravimo. Nije uputna upotreba "čarobnih brojeva" 300 i 20 u programu; oni donose malo podataka nekome tko bi program čitao kasnije, a teško ih je zamijeniti u cijelom programu. Jedan od načina baratanja takvim brojevima je da im se dodjele imena koja će govoriti o karakteru tih brojeva. Linija #define definira simboličko ime ili simboličku konstantu tako da bude poseban niz znakova:

```
#define     ime   tekst koji se mijenja
```

Nadalje, bilo koje pojavljivanje imena (koje nije u navodnicima i nije dio drugog imena) bit će zamijenjeno odgovarajućim tekstom koji se mijenja. Ime ima istu formu kao i ime varijable: niz slova i znamenki koji počinju slovom. Tekst zamjene može biti bilo koji niz znakova; on nije ograničen na brojeve.

```
#include <stdio.h>
#define  LOWER    0     /* donja granica tablice */
#define  UPPER    300   /* gornja granica */
#define  STEP     20    /* veličina koraka */
/* ispiši Fahrenheit-Celsius tablicu */
main(){
      int fahr;
      for(fahr=LOWER;fahr<=UPPER;fahr=fahr+STEP)
            printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Veličine LOWER, UPPER i STEP su simboličke konstante, a ne varijable pa se zato ne pojavljuju u deklaracijama. Imena simboličkih konstanti se dogovorno pišu velikim slovima pa se mogu razlikovati od imena varijabli koja se pišu malim slovima. Uočite također da nema točke-zareza na kraju #define linije.

1.5 Znakovni ulaz i izlaz

Razmotrit ćemo sada skupinu programa koje koristimo za ulaz/izlaz znakovnog tipa podataka. Vidjet ćete da su mnogi programi samo proširene verzije prototipova o kojima smo dosad govorili.

Model ulaza i izlaza kojeg podržava standardna biblioteka je vrlo jednostavan. Izlaz ili ulaz za tekst, bez obzira odakle dolazi i kamo ide, podijeljen je na nizove znakova. Tok teksta je niz znakova podijeljen na linije; svaka se sastoji od 0 ili više znakova na čijem se kraju nalazi znak nove linije. Biblioteka je zadužena za to da svaki ulazni ili izlazni tok prilagodi ovom modelu; koristeći biblioteku, programer u C-u ne treba brinuti kako su linije predstavljene izvan programa.

Standardna biblioteka određuje nekoliko funkcija za čitanje i pisanje jednog znaka, od kojih su getchar i putchar najjednostavnije. Po pozivu, getchar čita slijedeći ulazni znak iz toka teksta i vraća ga kao vrijednost. Dakle, poslije

```
c=getchar()
```

varijabla c će imati vrijednost slijedećeg znaka na ulazu. Znakovi normalno dolaze sa tastature; ulazom iz datoteke bavit ćemo se u Poglavlju 7.

Funkcija putchar ispisuje znak svaki put kad je pozvana:

```
putchar(c)
```

ispisuje znak čija je vrijednost u varijabli c, obično na ekranu. Pozivi funkcijama putchar i printf se mogu miješati; izlaz će ići po redu kojim su pozivi upućivani.

1.5.1 Kopiranje datoteka

Pomoću getchar-a i putchar-a, možete napisati uistinu mnogo korisnog koda bez većeg znanja o ulazu i izlazu. Najprostiji primjer je program koji kopira ulaz na izlaz, znak po znak:

```
pročitaj a znak
while(znak nije indikator kraja datoteke)
      ispiši upravo pročitani znak
      pročitaj a znak
```

Pretvorivši ovo u C kod, dobije se

```
#include <stdio.h>
/* kopiranje ulaza na izlaz; prva verzija */
main(){
      int c;
      c=getchar();
      while(c!=EOF){
            putchar(c);
            c=getchar();
      }
}
```

Relacijski operator != znači "nije jednako".

Ono što se pojavljuje na tipkovnici ili ekranu je, naravno, smješteno kao niz bitova. Tip char je, kako mu ime kaže, namijenjen za spremanje podataka znakovnog tipa, ali može biti uzeta i bilo koja vrijednost cjelobrojnog tipa. Ovdje koristimo int iz značajnog razloga. Problem koji bi se nametnuo jest razlikovanje znaka kraja ulaza od normalnih podataka. Funkcija getchar radi tako da uzima podatke dok na ulaz ne stigne znak za kraj ulaza (datoteke), vrijednost koja ne može biti nijedan drugi znak. Ova vrijednost se naziva EOF, zbog "End Of File". U tom pravcu, moramo prikazati varijablu c kao tip dovoljno velik da prihvati bilo koju vrijednost koju mu getchar vrati. Ne možemo koristiti char, jer c mora moći dohvatiti EOF i sve moguće znakove. Stoga koristimo int.

EOF je cjelobrojna vrijednost definirana u <stdio.h>, ali se razlikuju od sistema do sistema. Koristeći simboličku konstantu, sigurni smo da ništa u programu ne zavisi od neke specifične numeričke vrijednosti.

Program za kopiranje mogli bi ljepše napisati iskusniji programeri C-a. U C-u, bilo koja dodjela vrijednosti, kao

```
c=getchar()
```

je izraz koji ima vrijednost, a to je vrijednost lijeve strane nakon dodjele. To zapravo znači da se dodjela vrijednosti može pojaviti kao dio većeg izraza. Ako dodjelu znaka u c preselimo unutar zagrade za provjeru while petlje, program se dade napisati ovako:

```
#include <stdio.h>
/* kopiranje ulaza na izlaz; druga verzija */
main(){
      int c;
      while((c=getchar())!=EOF)
            putchar(c);
}
```

while dobije znak, prenosi ga u c, a zatim provjerava da li je znak bio signal kraja datoteke. Ako nije, tijelo petlje se izvršava, ispisujući znak. Tada se cijela priča ponavlja. Kad se napokon dosegne kraj ulaza, while se završava, a onda i main.

Ova verzija koncentrira ulaz na samo jednu referencu za getchar, pa je program kompaktniji i manji. Vi ćete se često susretati s ovim stilom pisanja (moguće je pisati i nečitljiv kod, ali to je ipak tendencija koja nije poželjna). Zagrade oko naredbe dodjele unutar uvjeta su neizostavne. Prioritet != je veći nego od =, što znači da će bez zagrada relacijski uvjet != biti odrađen prije dodjele. Tako je izraz

```
c=getchar()!=EOF
```

potpuno ekvivalentan sa

```
c=(getchar()!=EOF)
```

To ima neželjeni efekt u postavljanju c na 0 ili 1, zavisno od toga da li je getchar naišao na kraj datoteke (više o ovoj temi u Poglavlju 2.)

Vježba 1-6. Provjeriti vrijednost izraza getchar()!=EOF.

Vježba 1-7. Napišite program za ispis vrijednosti EOF-a.

1.5.2 Brojanje znakova

Slijedeći program broji znakove, slično programu kopiranja

```
#include <stdio.h>
/* brojanje znakova na ulazu; prva verzija */
main(){
      long nc;
      nc=0;
      while(getchar()!=EOF)
            nc++;
      printf("%ld\n", nc);
}

Izraz

++nc;
```

uvodi novi operator, ++, čije je značenje "povećaj za jedan". Možemo, dakle, umjesto nc=nc+1 pisati nc++ što je konciznije i vrlo često mnogo efikasnije. Postoji i odgovarajući operator za smanjivanje, --. Operatori ++ i -- mogu biti i prefiks (++nc) i sufiks operatori (nc++); ove dvije formulacije mogu dati različite vrijednosti u izrazima, kao što će biti obrađeno u Poglavlju 2, iako i jedna i druga povećavaju nc za jedan. Za ovaj trenutak ćemo se posvetiti prefiks formulaciji.

Program za brojanje znakova sumira svoje rezultate u varijabli tipa long umjesto u int. Cjelobrojne vrijednosti long imaju barem 32 bita. Iako su na nekim računalima int i long iste veličine, na drugim int ima 16 bitova, s maksimalnom vrijednošću od 32767 što je relativno mala gornja granica za brojač. Specifikacija pretvorbe %ld kaže printf-u da je odgovarajući argument long cjelobrojna vrijednost. Moguće je, dakako, operirati i s većim brojevima ako se upotrijebi double (float s dvostrukom točnošću). Koristit ćemo for petlju umjesto while radi ilustracije drugačijeg pisanja petlje.

```
#include <stdio.h>
/* brojanje znakova na ulazu; druga verzija */
main(){
      double nc;
      for(nc=0; getchar()!=EOF; ++nc)
            ;
      printf("%.0f", nc);
}
```

Funkcija printf koristi %f za ispis i float i double brojeva; %.0f ne dopušta ispis decimalne točke i decimala.

Tijelo ove for petlje je prazno, zato što se cijeli posao napravi u dijelu za provjeru uvjeta i uvećavanje. Međutim, gramatika C jezika zahtjeva da for naredba ima tijelo. Zasebna točka-zarez, nazvana nultom naredbom, tu je da bi udovoljila tom zahtjevu. Mi smo je umetnuli u posebnu liniju kako bi bila uočljivija.

Prije nego programčić za brojanje znakova prepustimo ropotarnici povijesti, zamijetimo da ako ulazni niz nema nijednog znaka, while i for provjere jave grešku pri prvom pozivanju funkcije getchar, a program daje nulu što je ispravan odgovor. Ovo je bitno. Jedna od komparativnih prednosti ovih petlji jest činjenica da one testiraju uvjete na vrhu petlje, prije ulaska u tijelo. Programi bi se trebali inteligentno ponašati kad im se zada ulaz nulte duljine. Naredbe while i for omogućuju dobar rad programima i za te granične slučajeve.

1.5.3 Brojanje linija

Slijedeći program broji ulazne linije. Kako smo ranije spomenuli, standardne biblioteka osigurava da se ulazni tok teksta pojavi kao niz linija i da se svaki završava znakom nove linije. To zapravo znači da se brojanje linija svodi na brojanje tih znakova:

```
#include <stdio.h>
/* brojanje linija na ulazu */
main(){
      int c, nl;
      nl=0;
      while((c=getchar())!=EOF)
            if(c=='\n')
                  ++nl;
      printf("%d\n", nl);
}
```

Tijelo petlje se sada sastoji od jednog if, koji kontrolira uvećavanje nl. Naredba if provjerava uvjete u zagradama, te ako je uvjet istinit, izvršava naredbu ili grupu naredbi (u vitičastim zagradama) koji slijedi. Još jednom smo htjeli pokazati što se čime kontrolira.

Dvostruki znak jednakosti == je C notacija za uvjet jednakosti (kao u Pascalu obično = ili u Fortranu .EQ.). Ovaj simbol se upotrebljava da bi se razlikovao uvjet jednakosti od običnog = koji C koristi pri dodjeli vrijednosti. Ovo je ujedno i upozorenje; novi programeri C programa ponekad pišu = umjesto ==. Kako ćemo već vidjeti u Poglavlju 2., rezultat je obično regularan izraz, pa prevoditelj neće upozoriti na vjerojatnu grešku.

Znak napisan u jednostrukim navodnicima predstavlja cjelobrojnu vrijednost jednaku ASCII vrijednosti znaka. Zovemo ga znakovnom konstantom, iako je to samo drugi način za pisanje malog cijelog broja. Tako, npr., 'A' je znakovna konstanta; u ASCII skupu znakova njena je vrijednost 65, a interno predstavlja znak A. Evidentno je bolje pisati 'A' nego 65; značenje je jasnije, a nije zavisan o skupu znakova.

ESCAPE sekvence koje koristimo u konstantama nizova su jednako tako upotrebljive u znakovnim konstantama, pa '\n' ima značenje znaka novog reda, a u ASCII tablici ima vrijednost 10. Možete zamijetiti da je '\n' jedan znak, a u izrazima on predstavlja cijeli broj; s druge strane, "\n" je konstanta niza koja ima samo jedan znak. Nizovi i znakovi bit će predmetom rasprave u Poglavlju 2.

Vježba 1-8. Napišite program koji broji razmake, tabulatore i nove linije.

Vježba 1-9. Napišite program koji kopira ulaz na izlaz, pretvarajući nizove od dva ili više razmaka samo jednim razmakom.

Vježba 1-10. Napišite program koji kopira ulaz na izlaz, mijenjajući svaki tabulator s \t, svaki backspace sa \b i svaku dvostruku kosu crtu s \\. Program treba te specijalne znakove učiniti vidljivim.

1.5.4 Brojanje riječi

Četvrti u nizu korisničkih programa broji linije, riječi i znakove, uz neformalnu definiciju da je riječ niz znakova koji nema razmaka, tabulatora ili znakova novog reda. Ovo je glavna verzija standardnog UNIX programa wc (word count).

```
#include <stdio.h>
#define     IN    1     /* unutar riječi */
#define     OUT   0     /* van riječi */
/*  zbraja linije, riječi i znakove na ulazu */
main(){
      int c, nl, nw, nc, state;
      state=OUT;
      nl=nw=nc=0;
      while((c=getchar())!=EOF){
            ++nc;
            if(c=='\n')
```

```
                ++nl;
            if(c==' '||c=='\n'||c=='\t')
                state=OUT;
            else if(state==OUT){
                state=IN;
                ++nw;
            }
        }
        printf("%d %d %d\n", nl, nw, nc);
}
```

Pri svakom susretu s prvim znakom u riječi, on odbroji jednu riječ više. Varijabla state kaže da li je program trenutno unutar riječi ili ne; njeno početno stanje je OUT, odnosno "van riječi". Simboličke konstante IN i OUT nam više odgovaraju negoli brojčane vrijednosti 0 i 1, jer, kako smo već spomenuli, čine program čitljivijim. U ovako malom programu to nam nije odviše bitno, ali u većim programima, povećanje jasnoće programa je vrijedno dodatnog napora da se program napiše od početka u tom stilu. Vi ćete isto tako zaključiti da je lakše mijenjati programe gdje se brojevi pojavljuju kao simboličke konstante.

Linija

```
nl=nw=nc=0;
```

postavlja sve tri varijable na nulu. To nije nikakva iznimka, već posljedica činjenice da je dodjeljivanje izraz koji ima vrijednost i obavlja se zdesna nalijevo. Umjesto toga mogli smo pisati

```
nl=(nw=(nc=0));
```

Operator || znači ILI (OR), pa linija

```
if(c==' '||c=='\n'||c=='\t')
```

znači "ako je c razmak ili znak za novi red ili tabulator" (prisjetimo se \t oznaka za tabulator). Postoji odgovarajući operator && za I (AND); njegov je prioritet veći nego za ||. Izrazi vezani s && ili || računaju se slijeva nadesno (!!!!), a garantira se prekid računanja čim bude poznata točnost ili netočnost. Tako, ako je c razmak, nije potrebno ispitivati da li je znak za novu liniju i tabulator, pa se ti testovi preskaču. To nije previše bitno za prikazani slučaj, ali jest za kompliciranije slučajeve što ćemo ubrzo vidjeti.

Primjer također pokazuje else, naredbu koja obavlja "nešto drugo" ako je uvjetni dio if naredbe neistinit. Opća forma je

```
if(izraz)
        naredba1
else
        naredba2
```

Izvodi se jedna i samo jedna naredba od dviju pridruženih if-else konstrukciji. Ako je izraz istinit, izvodi se naredba1; ako nije izvodi se naredba2. Svaka naredba može biti jedna ili više njih u vitičastim zagradama. U programu za brojanje riječi, nakon else imamo if koji kontrolira dvije naredbe unutar vitičastih zagrada.

Vježba 1-11. Kako biste provjerili program za brojanje riječi? Koje su vrste ulaznih podataka najpogodnije za otkrivanje netočnosti ako postoje?

Vježba 1-12. Napišite program koji ispisuje svoj ulaz tako da bude po jedna riječ u svakoj liniji.

1.6 Polja

Napišimo program koji zbraja pojavljivanja svake znamenke, specijalnih znakova (razmaci, tabulatori i sl.) i svih drugih znakova. To nije upotrebljiv primjer, ali nam omogućuje ilustraciju višeznačnu snagu C-a u jednom programu.

Postoji dvanaest kategorija ulaza, pa je pogodnije koristiti jedno polje koje sadrži broj pojavljivanja svake znamenke, nego deset posebnih varijabli. Evo jedne verzije programa:

```
#include <stdio.h>
/* zbraja znamenke, specijalne znakove i sve druge */
main(){
      int c, i, nwhite, nother;
      int ndigit[10];
      nwhite=nother=0;
      for(i=0;i<10;++i)
            ndigit[i]=0;
      while((c=getchar())!=EOF)
            if(c>='0'&&c<='9')
                  ++ndigit[c-'0'];
            else
                  if(c==' '||c=='\n'||c=='\t')
                        ++nwhite;
                  else
                        ++nother;
      printf("znamenke=");
      for(i=0;i<10;++i)
            printf(" %d", ndigit[i]);
      printf(", specijalni znakovi=%d, ostalo=%d\n", nwhite, nother);
}
```

Izlaz gore napisanog izvornog koda programa je

```
znamenke=9 3 0 0 0 0 0 0 0 1, specijalni znakovi=123, ostalo=345
```

Deklaracija

```
int ndigit[10];
```

proglašava ndigit poljem od 10 cijelih brojeva. Indeksi polja uvijek počinju od nule u C-u, pa su elementi polja ndigit[0], ndigit[1], ..., ndigit[9]. To odražava u for petljama, koje inicijaliziraju i ispisuju polje.

Indeks može biti bilo koji cjelobrojni izraz, koji ima cjelobrojnu vrijednost, kao i u našem programu, te cjelobrojne konstante.

Ovaj program se isto tako zasniva na znakovnom predstavljanju brojeva.

Npr. uvjet

```
if(c>='0'&&c<='9')
```

određuje da li je znak c znamenka. Ako jest, brojčana vrijednost te znamenke je

```
c-'0'
```

To radi samo ako '0', '1', ... , '9' imaju dosljedno rastuće vrijednosti. Srećom, to vrijedi za sve skupove znakova.

Po definiciji, char su vrlo mali cijeli brojevi, pa su char varijable i konstante istovrsne int varijablama i konstantama u aritmetičkim izrazima. To je prirodno i korisno; primjerice, c-'0' je cjelobrojni izraz s vrijednošću između 0 i 9 ovisno o znaku '0' do '9', pohranjenom u c, te je to indeks za polje ndigit.

Odluka o tipu znaka (znamenka, specijalni znak ili nešto drugo) donosi se u slijedećem dijelu programa

```
if(c>='0'&&c<='9')
      ++ndigit[c-'0'];
else
      if(c==' '||c=='\n'||c=='\t')
            ++nwhite;
      else
            ++nother;
```

Uzorak

```
if(uvjet1)
      naredba1
else
      if(uvjet2)
            naredba2
      else
            if(uvjet3)
                  naredba3
......
      ......
                  else
                        naredban
```

se često susreće u programima kao način izražavanja višeznačnih rješenja. Uvjeti se provjeravaju po redu odozgo nadolje, sve dok neki ne bude zadovoljen; tu se odgovarajuća naredba izvršava, a konstrukcija se završava (svaka se naredba može sastojati od više naredbi zatvorenih zagradama). Ako ni jedan od uvjeta nije ispunjen, izvršava se naredba iza zadnjeg else ako ovaj postoji. Ako zadnji else i naredba nisu napisani, kao u našem programu za brojanje riječi, ne obavlja se ništa. Može biti bilo koji broj

```
else
      if(uvjet)
            naredba
```

grupa između inicijalnog if i zadnjeg else.

To je stilsko pitanje, ali je preporučljivo praviti konstrukcije na prikazani način. Naredba switch, o kojoj će biti riječi u Poglavlju 3, određuje drugačiji način pisanja višeznačnog grananja koji je osobito pogodan ako neki cijeli broj ili znakovni izraz odgovara jednoj ili nizu konstanti. Tome nasuprot, pokazat ćemo switch verziju ovog programa u dijelu pod naslovom 3.4.

Vježba 1-13. Napišite program za ispis histograma dužina riječi na ulazu. Iscrtavanje histograma sa horizontalnim linijama nije težak zadatak, ali je okomita orijentacija priličan izazov.

Vježba 1-14. Napišite program za ispis histograma frekvencija pojavljivanja različitih znakova na ulazu.

1.7 Funkcije

U C-u funkcija je ekvivalentna potprogramu ili funkciji u Fortranu te proceduri ili funkciji u Pascalu. Funkcija osigurava prikladan način vršenja svakojakih izračuna, čije rezultate kasnije možemo upotrijebiti bilo gdje. Pravilno zadanim funkcijama, moguće je posao obaviti bez poznavanja unutarnje strukture funkcije; saznanje da je posao korektno obavljen je više nego dovoljno. C čini upotrebu funkcije lakom, prikladnom i efikasnom; često ćete se susretati s kratkim funkcijama definiranim i pozvanim samo jednom, tek radi pojašnjavanja dijelova koda.

Dosad smo koristili samo ponuđene funkcije tipa printf, getchar i putchar; došlo je vrijeme da sami napišemo nekoliko funkcija. Kako C nema eksponencijalni operator kao ** u Fortranu, predstavimo mehanizam definiranja funkcija pišući funkciju power(m,n) da bismo cijeli broj m podigli na pozitivni cjelobrojni eksponent n. Znači, vrijednost od power(2,5) je 32. Ova funkcija nije za praktičnu primjenu, zato što eksponent uzima samo pozitivne male cijele brojeve, ali je dovoljno dobra za ilustraciju (inače, standardna biblioteka ima funkciju pow(x,y) koja računa x^y).

Evo funkcije power i glavnog programa za njeno testiranje, kako bi mogli razlučiti čitavu njenu strukturu.

```
#include <stdio.h>
int power(int m, int n);
/* test power funkcije */
main(){
      int i;
      for(i=0;i<10;++i)
```

```
            printf("", i, power(2,i), power(-3,i));
      return 0;
}
/* power: podiže bazu na n-ti eksponent n>=0 */
int power(int base, int n){
      int i,p;
      p=1;
      for(i=1;i<=n;++i)
            p=p*base;
      return p;
}
```

Definicija funkcije ima ovaj oblik:

```
ime funkcije(deklaracija parametara, ukoliko postoje){
      deklaracije internih varijabli funkcije
      naredbe
}
```

Deklaracije funkcija se mogu pojavljivati po bilo kom redoslijedu, u jednoj ili više izvornih datoteka, iako ni jedna funkcija ne može biti podijeljena u više datoteka. Ako se izvorni program nalazi u više datoteka, može se kazati da će biti nešto više prevođenja i punjenja programa podacima nego što bi bilo da je sve u jednoj datoteci, ali to je već problem operativnog sustava i nema veze sa samim jezikom. Na trenutak ćemo pretpostaviti da su obje funkcije u istoj datoteci, pa će vrijediti sve dosada proučeno o odvijanju C programa.

Funkcija power je pozvana dva puta iz main, u liniji

```
printf("", i, power(2,i), power(-3,i));
```

Svaki poziv predaje dva argumenta funkciji power, koja nakon toga vraća cijeli broj za formatirani ispis. U jednom izrazu, power(2,i) je cijeli broj kao što su to i 2 i i (ne vraćaju sve funkcije cjelobrojnu vrijednost; o tome će biti riječi u Poglavlju 4.).

Prva linija funkcije power,

```
int power(int base, int n)
```

određuje tipove i imena parametara, te tip rezultata koji funkcija vraća. Imena koja koristi power za svoje parametre su lokalna i nisu vidljiva za bilo koju drugu funkciju: druge funkcije bez ikakvog problema mogu se služiti istim imenima. To isto tako vrijedi i za varijable i i p: i u funkciji power nema ništa s i u funkciji main.

Mi ćemo koristiti "parametar" za varijablu čije se ime nalazi na listi definicije funkcije uokvirene zagradama, a "argument" za vrijednost koja se koristi pri pozivu. Ponekad se koriste termini "formalni argument" i "aktualni argument" da se ovi pojmovi ne bi miješali.

Vrijednost koju power izračuna vraća se u main pomoću naredbe return. Bilo koji izraz može biti naveden iza return:

```
return izraz;
```

Funkcija ne mora vraćati vrijednost; naredba return bez pratećeg izraza kontrolira, ali ne vraća upotrebljivu vrijednost u program odakle je pozvana te se ta vrijednost ignorira.

Mogli ste uočiti naredbu return na kraju main funkcije. Pošto je main funkcija kao i svaka druga, njena se vrijednost može vratiti onome tko ju je pozvao, što je gotovo uvijek operativni sustav. Bilo kako bilo, vraćena vrijednost nule podrazumijeva uredan završetak; vraćena vrijednost različita od nule znak je da nešto nije bilo u redu. Da bi pojednostavnili, mi smo izostavljali naredbe return iz main funkcija, ali ćemo ih uključivati, jer programi trebaju vraćati status svome okruženju.

Deklaracija

```
int power(int m, int n);
```

kaže da je power funkcija koja treba dva cjelobrojna argumenta, a vraća jedan int. Ova deklaracija, koja se naziva prototipom funkcije, mora se slagati s definicijom power. Ako se definicija funkcije ili neki njezin poziv ne slaže s prototipom, primit ćemo dojavu o greški.

Imena parametara ne moraju se slagati. Zapravo, imena parametara u prototipu funkcije su proizvoljna, pa se dade napisati `i`

```
int power(int, int);
```

Uredno odabrana imena čine program dokumentiranim, pa ćemo ih radije koristiti.

Iz povijesti razvoja C-a znamo da je najveća razlika između ANSI C programa i ranijih verzija u načinu na koji su funkcije predstavljene i definirane. U originalnoj definiciji C programa, power funkcija je mogla biti napisana ovako:

```
/* power: diže bazu na n-ti eksponent */
/*       (starija verzija)           */
power(base, n)
int base, n;
{
      int i, p;
      p=1;
      for(i=1;i<=n;++i)
            p=p*base;
      return p;
}
```

Parametri se imenuju između zagrada, a njihovi tipovi se predstavljaju prije otvaranja lijeve vitičaste zagrade; neodređeni parametri su inicijalno (default) tipa int (tijelo funkcije ostaje isto). Deklaracija funkcije power na početku programa bi izgledala ovako:

```
int power();
```

Lista parametara nije bila određena, pa prevoditelj nije mogao na vrijeme provjeriti da li je power korektno pozvana. Zapravo, kako je funkcija power samo trebala vratiti int, kompletna deklaracija je mogla biti izostavljena.

Nova sintaksa prototipova funkcije učinila je da prevoditelji puno lakše otkrivaju greške u brojnim argumentima funkcija s obzirom na tip. Stari stil i deklaracije još su u upotrebi u ANSI C programu, barem u prijelaznom razdoblju, ali mi toplo preporučujemo korištenje nove forme ako imate prevoditelj koji je podržava.

Vježba 1-15. Nanovo napišite program za pretvorbu temperature iz dijela pod naslovom 1.2 kako biste koristili funkciju za pretvorbu.

1.8 Argumenti - pozivanje pomoću vrijednosti

Jedna osobina C funkcija može se programerima koji koriste neke druge jezike, posebice Fortran, učiniti stranom. U C-u, svi su funkcijski argumenti poredani "po vrijednosti", što znači da je pozvana funkcija dala vrijednosti argumenata privremenim varijablama prije nego originalnim. Ovdje govorimo o osobinama koje su različite od onih koje se koriste u "pozivanju preko reference" u jezicima poput Fortrana ili Pascala (var parametri) gdje pozvana rutina (potprogram, funkcija, procedura) pristupa originalnom argumentu, a ne lokalnoj kopiji.

Osnovna razlika je u činjenici da u C-u pozvana funkcija ne može direktno zamijeniti varijablu u funkciji koja ju je pozvala; ona može zamijeniti samo svoju privremenu kopiju.

Pozivanje pomoću vrijednosti djeluje aktivno, a ne pasivno. Ono obično vodi kompaktnijim programima s manje različitih varijabli, jer parametri mogu biti tretirani kao pogodno obilježene lokalne varijable u pozvanoj funkciji. Primjerice, evo jedne verzije power koja koristi ovu osobinu:

```
/* power: diže bazu na n-ti eksponent; n>=0; verzija 2 */
int power(int base, int n){
      int p;
      for(p=1;n>0;--n)
            p=p*base;
      return p;
}
```

Parametar n se koristi kao privremena varijabla, a vrijednost mu se smanjuje dok ne dođe do nule; ne postoji više potreba za varijablom i. Međutim, što god uradili sa n unutar funkcije power, nema nikakvog utjecaja na argument sa kojim je power prvobitno pozvan.

Kada bude potrebno, mi ćemo omogućiti promjenu varijable unutar pozvane funkcije. Pri pozivu jednostavno pošaljemo adresu varijable koja se uvodi kao argument (zapravo se radi o pokazivaču na varijablu), pa pozvana funkcija indirektno dohvaća varijablu i po potrebi mijenja. O pokazivačima (pointerima) više ćemo govoriti u Poglavlju 5.

Kod polja je priča nešto drugačija. Kad se ime polja koristi kao argument, vrijednost koja se predaje funkciji je lokacija ili adresa početka polja - tu nema kopiranja elemenata polja. Upisujući vrijednost kao indeks, funkcija prihvatiti bilo koji element polja. Ovo će biti tema slijedećih razmatranja.

1.9 Polja znakova

Najjednostavniji tip polja u C-u jest polje znakova. Radi ilustracije upotrebe polja znakova i funkcija koje barataju sa njima, napisat ćemo program koji čita ulaz i ispisuje najdužu liniju. Glavne crte su prilično jednostavne

```
while(ima još linija)
      if(ova linija je duža od dosada najduže){
            pohrani je
            pohrani njenu dužinu
      }
      ispiši najdužu liniju
```

Ovaj prikaz uglavnom razjašnjava kako se program prirodno rastavlja na dijelove. Jedan dio čita novu liniju, drugi je analizira, slijedeći je pamti, a ostali kontroliraju proces.

Obzirom da se zadaci tako lako dijele, bilo bi dobro napisati odgovarajući program. Prema tome, napišimo najprije funkciju getline koja uzima slijedeću liniju s ulaza. Trudit ćemo se napraviti funkciju koja bi bila upotrebljiva i u nekom drugom kontekstu. U najmanju ruku, getline treba vraćati signal o tome da se došlo (u čitanju sa ulaza) do kraja datoteke; korisnije bi bilo vraćati dužinu linije ili nulu ako je nastupio kraj datoteke. Nula je prihvatljiva vrijednost za kraj jer ta vrijednost nema smisla kao dužina linije. Svaka linija ima barem jedan znak, makar to bila oznaka nove linije (i to je linija dužine 1). Kad pronađemo liniju koja je duža od najduže prethodne, moramo je pohraniti. Ovo nam obavlja druga funkcija, copy, koja kopira novi red na sigurno mjesto. Na kraju, treba nam osnovna funkcija main za manipuliranje funkcijama getline i copy. Evo rješenja:

```
#include <stdio.h>
#define     MAXLINE     1000  /* maksimalna dužina linije */

int getline(char line[], int maxline);
void copy(char to[], char from[]);

/* ispis najduže linije s ulaza */
main(){
      int len;
      int max;
      char line[MAXLINE];
      char longest[MAXLINE];
      max=0;
      while((len=getline(line, MAXLINE))>0)
            if(len>max){
                  max=len;
                  copy(longest, line);
            }
      if(max>0)   /* ima linija */
            printf("%s", longest);
      return 0;
}
```

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

/* copy: kopira 'from' u 'to'; 'to' mora biti dovoljno veliko */
void copy(char to[], char from[]){
      int i;
      i=0;
      while((to[i]=from[i])!='\0')
      ++i;
}
```

Funkcije getline i copy deklarirane su na početku programa, a pretpostavka je da su u jednoj datoteci.

Funkcije main i getline komuniciraju preko nekoliko argumenata i povratnih vrijednosti. U getline funkciji, argumenti su predstavljeni pomoću linije

```
int getline(char s[], int lim)
```

koja određuje da je prvi argument polje s, a drugi cjelobrojni lim. Dužina polja se ne navodi, ali ona za getline nije ni bitna jer se njena veličina korigira u funkciji main. Dužina polja s nije bitna za getline pošto je njena veličina ugođena u main-u. Funkcija getline rabi return kako bi vratila vrijednost onomu tko je poziva, onako kako je to radila funkcija power. Ova linija isto tako pokazuje da getline vraća vrijednost tipa int; kako je int inicijalno (po default-u, ako se ništa drugo ne unese) tip koji funkcija vraća, tako se navođenje int u deklaraciji može izostaviti.

Neke funkcije vraćaju upotrebljive vrijednosti dok druge, kao copy, koriste se za obrade podataka, a ne vraćaju nikakve vrijednosti. Tip funkcije copy je void, što je jasna naznaka da ona ne vraća nikakvu vrijednost.

Funkcija getline postavlja znak '\0' (znak čija je vrijednost nula) na kraj polja koje stvara, kako bi označila kraj niza. Ova konvencija se koristi u C-u; kad se niz znakovni niz kakav je

```
"hello\n"
```

pojavi u C programu, on se sprema kao polje znakova koje se sastoji od znakovnog niza, a završava s '\0' kao oznakom kraja.

h	e	l	l	o	\n	\0

%s parametar formata u funkciji printf pretpostavlja da je odgovarajući argument niz u ovoj formi. Funkcija copy se također oslanja na to da ulazni argument završava s '\0', a ona kopira taj znak u izlazni argument (sve što smo rekli stoji uz pretpostavku da '\0' nije dio teksta kao svaki drugi znak).

Vrijedno je pripomenuti kako čak i tako mali program poput ovoga ima nekoliko osjetljivih kreativnih problema. Primjerice, što bi se dogodilo kad bi main naišla na liniju koja prelazi dužinu definiranu s MAXLINE? Funkcija getline radi tako da zaustavlja zbrajanje čim se polje napuni, čak i ako se ne pojavi nova linija. Testirajući dužinu i zadnji vraćeni znak, main može odrediti je li linija, a zatim je obraditi po želji. Kako bi skratili postupak, ignorirali smo rezultat.

Ne postoji način za onoga koji rabi funkciju getline da unaprijed zna koliko duga može biti ulazna linija, pa getline mora spriječiti preopterećenje. S druge strane, onaj koji koristi copy već zna (ili može znati) koliko su dugački nizovi, pa smo odlučili da od ove funkcije ne tražimo provjeru grešaka.

Vježba 1-16. Ispravite (doradite) osnovnu rutinu programa za ispis najduže linije kako bi se mogle ispisivati neograničeno duge ulazne linije i što je god moguće više teksta.

Vježba 1-17. Napišite program za ispis svih ulaznih linija dužih od 80 znakova.

Vježba 1-18. Napišite program za brisanje razmaka i tabulatora iz ulaznih linija, a koji će isto tako uklanjati prazne linije.

Vježba 1-19. Napišite funkciju reverse(s) koja obrće niz znakova s. Upotrijebite je u programu koji obrće ulaznu liniju.

1.10 Vanjske varijable i područja

Varijable u main funkciji kao što su line, longest itd. su vlastite ili lokalne za funkciju main. Kako su stvorene unutar main-a, nijedna im druga funkcija ne može direktno pristupiti. Isto vrijedi i za varijable u drugim funkcijama; npr. varijabla i u getline funkciji nije povezana s i u copy funkciji. Svaka lokalna varijabla stvara se pri pozivu funkcije, a nestaje kad se funkcija izvrši. Zato su takve varijable poznate pod imenom automatske varijable, ako se prati terminologija iz drugih programskih jezika. Mi ćemo koristiti izraz automatsko pozivanje koje će se odnositi na te lokalne varijable (u Poglavlju 4 se govori o tipu static, čije lokalne varijable zadržavaju svoje vrijednosti između poziva).

Kako se automatske varijable javljaju s pozivanjem funkcija, one ne drže svoje vrijednosti između dva poziva i moraju biti točno namještene pri svakom ulazu. Ako nisu namještene, one će prenositi i krive podatke.

Nasuprot automatskim varijablama, moguće je definirati varijable koje su vanjske za sve funkcije, dakle varijable kojima se može pristupiti pomoću imena iz bilo koje funkcije (ova tehnika je slična Fortranskoj COMMON varijabli i Pascalskim varijablama u krajnjem bloku). Kako su vanjske varijable odasvud vidljive, one se mogu rabiti umjesto liste argumenata za komunikaciju podacima između funkcija. Nadalje, kako su vanjske varijable stalno prisutne, tj., ne stvaraju se i nestaju pozivima i zatvaranjima funkcija, one drže svoje vrijednosti i nakon zatvaranja funkcija koje su ih postavile.

Vanjska varijabla mora biti točno definirana izvan svih funkcija čime se za nju određuje lokacija. Varijabla također mora biti predstavljena i u svakoj funkciji koja je želi koristiti čime se određuje tip varijable. Deklaracija može biti jedan jasan extern izraz ili izraz koji je razumljiv iz konteksta. Da bi mogli konkretno diskutirati, napišimo program za najdužu liniju s line, longest i max kao vanjskim varijablama. Ovo zahtjeva izmjenu poziva, deklaracija i tijela svih triju funkcija.

```
#include <stdio.h>

#define     MAXLINE     1000  /* maksimalna dužina linije */

int max;          /* dosad najveća dužina linije */
char line[MAXLINE];     /* tekuća linija */
char longest[MAXLINE];  /* dosad najveća linija */

int getline(void);
void copy(void);

/* ispisuje najdužu liniju; posebna verzija */
main(){
      int len;
      extern int max;
      extern char longest[];
      max=0;
      while((len=getline())>0)
            if(len>max){
                  max=len;
                  copy();
```

```
            }
      if(max>0)   /* ima linija */
            printf("%s", longest);
      return 0;
}

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

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

Vanjske varijable u funkcijama main, getline i copy su definirane prvim linijama napisanog primjera, koje određuju njihov tip i stvaraju (rezerviraju) prostor u memoriji za njihov smještaj. Sa stajališta sintakse, vanjske definicije djeluju kao definicije lokalnih varijabli, ali kako se prijavljuju izvan funkcija, varijable su vanjske. Pri preuzimanju vanjske varijable od strane funkcije, njeno ime mora biti poznato funkciji. Ime se funkciji dostavlja preko extern izraza u funkciji; deklaracija je u potpunosti identična ranijoj osim dodane ključne riječi extern.

U nekim uvjetima, extern se može izostaviti. Ako se definicija vanjske varijable pojavi u izvornoj datoteci prije njene upotrebe u posebnoj funkciji, tada ne postoji potreba za extern deklaracijom u funkciji. Deklaracije exstern u funkcijama main, getline i copy su stoga suvišne (redundantne). Zapravo, najjednostavnije je prikazati definicije svih vanjskih varijabli na početku izvorne datoteke, a onda izostaviti sve extern deklaracije.

Ako se program sastoji od više izvornih datoteka, varijabla je definirana u "datoteka1", a koristi se u "datoteka2" i "datoteka3", onda su extern deklaracije potrebne u "datoteka2" i "datoteka3" radi povezivanja varijabli. Uobičajena je praksa da se sakupe extern deklaracije varijabli i funkcija u zasebnu datoteku, zvanu zaglavlje, koju uvodimo s #include ispred svake izvorne datoteke. Ekstenzija .h je općeprihvaćena za zaglavlja. Funkcije standardne biblioteke, npr. su predstavljene kao zaglavlja u obliku <stdio.h>. O ovom će više biti riječi u Poglavlju 4, a o samoj biblioteci u Poglavlju 7 i Dodatku B.

Otkako posebne verzije getline i copy funkcija nemaju argumente, logika nam nameće mogućnost da za njihove prototipove koristimo getline() i copy(). Ipak, zbog kompatibilnosti sa starijim C programima, standard koristi praznu listu kao deklaraciju starog stila i ukida provjeru svih lista argumenata; riječ void mora se eksplicitno koristiti za prazne liste. O ovome će biti govora u Poglavlju 4.

Možete primijetiti kako riječi definicija i deklaracija koristimo oprezno kad se odnose na vanjske varijable u ovom ulomku. Riječ "definicija" se odnosi na mjesto gdje je varijabla nastala ili gdje je locirana, a "deklaracija" na mjesta gdje je jednoznačno određena priroda varijable, ali nije određen njen memorijski prostor.

Nakon svega ukazanog bilo bi možda logično čitave programe svoditi na extern varijable jer one komunikaciju čine jednostavnijom - liste argumenata i varijable su uvijek tu, čak i kad nam nisu potrebne. Programiranje naslonjeno na vanjske varijable vodi do opterećenja programa (one su u memoriji i kad nikome ne trebaju), veze među dijelovima programa nisu baš očigledne (nema lista argumenata), a programe je teško modificirati jer se treba točno znati gdje se varijable mijenjaju. Stoga je druga verzija programa slabija od prve, a osim gore navedenih razloga ona narušava općenitost dviju korisnih funkcija kakve su getline i copy, uključivanjem vanjskih varijabli.

Možemo reći da smo dosad objedinili sve ono što se naziva suštinom C-a. S ovim mnoštvom izgrađenih blokova, moguće je pisati korisničke programe razumne dužine i to bi bila dobra ideja ako imate dovoljno vremena provesti je u djelo. Vježbe koje slijede predlažu malo kompleksnije programe od prikazanih u ovom poglavlju.

Vježba 1-20. Napišite program detab za zamjenu tabulatora odgovarajućim brojem razmaka. Odredite fiksni broj razmaka za stop-tabulator, npr., n. Da li n treba biti varijabla ili simbolički parametar?

Vježba 1-21. Napišite program entab za zamjenu razmake minimalnim brojem tabulatora i razmaka. Koristite iste tabulatore kao i za detab. Kad bi i tabulator i razmak bili dovoljni da se dostigne novi stop-tabulator, kojem znaku bi dali prednost?

Vježba 1-22. Napišite program koji prelama duge ulazne linije na dvije ili više kraćih linija nakon zadnjeg znaka koji nije razmak, a koji se javlja prije n-te ulazne kolone. Uvjerite se da vaš program pravilno tretira duge linije, te da nema razmaka i tabulatora prije određene kolone.

Vježba 1-23. Napišite program koji uklanja komentare iz C programa. Ne zaboravite pravilno iskoristiti znakovne nizove za komentare i znakovne konstante. C komentare ne treba umetati.

Vježba 1-24. Napišite program za provjeru osnovnih sintaksnih grešaka u C programu, kakve su zagrade, navodnici (jednostruki i dvostruki) i komentari. Ovo je zahtjevan program ako se radi uopćeno.

POGLAVLJE 2: TIPOVI, OPERATORI I IZRAZI

Varijable i konstante su osnovni oblici podataka s kojima se radi u programu. Deklaracije imaju popis varijabli koje će se rabiti, određuju im tip, te eventualno njihovu početnu vrijednost. Operatori govore što će s njima biti učinjeno. Izrazi kombiniraju varijable i konstante kako bi proizveli nove vrijednosti. Tip nekog objekta određuje skup vrijednosti koje on može imati i operacije koje se nad njim mogu izvršavati. U ovom poglavlju će se, pored toga, govoriti i o formiranju blokova.

ANSI standard je načinio mnogo neznatnih izmjena i dodataka osnovnim tipovima i izrazima. Uvedene su signed i unsigned forme za sve tipove cijelih brojeva, te obilježavanje konstanti tipa unsigned i heksadecimalnih znakovnih konstanti. Operacije s realnim brojevima mogu se vršiti s jednostrukom i dvostrukom točnošću (postoji tip long double za dvostruku točnost). Konstante nizova daju se povezivati tokom prevođenja. Nizovi dobiveni pobrojavanjem postali su dio jezika. Objekti mogu biti deklarirani kao const, što ih čuva od promjene. Pravila za automatsku pretvorbu između aritmetičkih tipova proširena su kako bi se operiralo s većim izborom tipova.

2.1 Imena varijabli

Mada se u Poglavlju 1 nismo toga dotakli, napomenimo kako postoje neka ograničenja vezana za imena varijabli i simboličkih konstanti. Ime se sastoji od slova i znamenki pri čemu prvi znak mora biti slovo. Podcrta ("_" - underscore) se uvažava kao slovo što je ponekad korisno jer povećava čitljivost kod varijabli s dugim imenima. Ipak, ne počinjite imena varijabli podcrtom jer rutine iz standardne biblioteke često koriste takva imena. Velika i mala slova se razlikuju, pa x i X nisu isto ime. Praksa je u C-u da se mala slova rabe za imena varijabli, a velika slova za simboličke konstante.

Barem prvih 31 znakova nekog imena jesu bitni znakovi. Za imena funkcija i vanjskih varijabli broj mora biti manji od 31 jer se imena vanjskih varijabli mogu koristiti prevoditelja na strojni jezik (assembler) i učitavača (loader) nad kojima program nema kontrolu. Kod vanjskih imena standard garantira jednoznačnost samo za prvih 6 znakova. Ključne riječi kao što su if, else, int, float, itd. su rezervirane i ne mogu se upotrebljavati kao imena varijabli. One moraju biti pisane malim slovima.

Preporuka je koristiti takva imena iz kojih je vidljiva namjena varijable, a da nisu međusobno slična u tipografskom smislu. Naša je politika koristiti kratka imena za lokalne varijable, posebice za petlje, a duža za vanjske varijable.

2.2 Tipovi i veličine podataka

U C-u postoji nekoliko osnovnih tipova podataka:

`char`	jedan byte, jedan znak iz skupa znakova
`int`	cjelobrojna vrijednost, odnosi se na prirodnu veličinu cijelih brojeva na računalu
`float`	realni broj s jednostrukom točnošću
`double`	realni broj s dvostrukom točnošću

Pored toga, postoje i kvalifikatori koji se pridružuju nabrojanim osnovnim tipovima. Tako se short i long pridružuju cijelim brojevima:

```
short int sh;
long int counter;
```

Riječ int može se izostaviti u ovakvim deklaracijama, ali se obično piše.

Htjeli smo sa short i long osigurati različite dužine cijelih brojeva, uglavnom tamo gdje za to postoji valjan razlog; int je prirodna veličina za svako pojedino računalo. Tip short uglavnom ima 16 bita, long 32 bita, a int ili 16 ili 32. Svaki prevoditelj ima slobodu biranja veličine koja odgovara sklopovlju na kojemu radi, uz ograničenja da short i int moraju biti najmanje 16 bita, long najmanje 32 bita, te da short ne smije biti duži od int koji opet ne smije biti duži od long.

Kvalifikatori signed i unsigned mogu se pridružiti tipu char ili bilo kojem cijelom broju. Brojevi tipa unsigned su uvijek pozitivni ili jednaki nuli i za njih vrijedi aritmetički zakon modula 2n, gdje je n broj bitova u tipu. Tako, primjerice, ako char ima 8 bitova, varijable tipa unsigned char imaju vrijednost između 0 i 255, dok varijable tipa signed char imaju vrijednost između -128 i 127. Bilo da je char kvalificiran kao signed ili unsigned, on je ovisan o računalu, a znakovi koji se ispisuju su uvijek pozitivni.

Tip long double naznačuje realne brojeve uvećane točnosti. Kao i kod cijelih brojeva, veličine objekata realnih brojeva daju se definirati na više načina; float, double i long double mogu predstavljati jednu, dvije ili tri različite veličine.

Standardna zaglavlja <limits.h> i <float.h> sadrže simboličke konstante za sve te veličine, skupa s ostalim svojstvima računala i prevoditelja. O tome više u Dodatku B.

Vježba 2-1. Napišite program za određivanje opsega varijabli tipa char, short, int i long, kad su kvalificirani kao signed i unsigned, ispisujući odgovarajuće vrijednosti sa standardnih zaglavlja direktnim izračunavanjem. Odredite opsege različitih tipova realnih brojeva.

2.3 Konstante

Cjelobrojna konstante, kakva je npr. 1234, je tipa int. Konstanta long se piše slovom l ili L na kraju, kao npr. 123456789L, dakle cijeli broj koji je suviše velik da stane u tip int. Konstante tipa unsigned pišu se sa u ili U na kraju, dok sufiks ul ili UL naznačuje konstante tipa unsigned long.

Konstante realnih brojeva imaju decimalnu točku (123.4) ili eksponent na kraju (1e-2) ili oboje; njihov je tip double osim ako nemaju sufiks na kraju. Sufiksi f ili F naznačuju float konstantu, a l ili L long double.

Vrijednost cijelog broja dade se prikazati osim u decimalnoj notaciji, još i oktalno i heksadecimalno. Ako se na početku cjelobrojne konstante nalazi nula 0, znak je to predstavljanja u oktalnoj notaciji, a 0x ili 0X oznake su heksadecimalne notacije. Kažimo primjerom, broj 31 može se prikazati oktalnom notacijom kao 037, a 0x1f ili 0X1F heksadecimalnom. Oktalne i heksadecimalne konstante mogu se isto tako završavati s L da bi bile long ili U da bi bile unsigned; OXFUL je unsigned long konstanta koja ima vrijednost 15 decimalno.

Znakovna konstanta je cjelobrojna vrijednost, napisana kao znak između jednostrukih navodnika, npr. 'x'. Vrijednost znakovne konstante je brojčana vrijednost danog znaka u skupu znakova našeg računala. Npr. u ASCII skupu znakova, znakovna konstanta '0' ima vrijednost 48, koja nema nikakve veze s brojčanom vrijednošću 0. Ako napišemo '0' umjesto brojčane vrijednosti 48 (koja nije obavezna, ako nije u pitanju ASCII skup znakova), program je čitljiviji i nezavisan od pojedinačnih vrijednosti različitih skupova znakova. Znakovne konstante sudjeluju u brojevnim operacijama kao i druge cjelobrojne vrijednosti iako se daleko više koriste za usporedbe s ostalim znakovima.

Određeni znakovi daju se predstaviti u znakovnim konstantama i znakovnim nizovima pomoću escape nizova kao npr. '\n' (novi red ili newline); ovi nizovi izgledaju kao dva znaka, ali predstavljaju samo jedan. Jednako tako, slijed bitova može se pisati kao

```
'\ooo'
```

gdje je ooo jedna do tri oktalne znamenke (0...7) ili kao

```
'\xhh'
```

gdje je hh jedna ili više heksadecimalnih znamenki (0...9, a...f, A...F). Možemo, dakle, pisati

```
#define     VTAB  '\013'      /* ASCII okomiti tabulator */
#define BELL      '\007'      /* ASCII znak za zvučni signal */
```

ili heksadecimalno

```
#define     VTAB  '\xb' /* ASCII okomiti tabulator */
#define BELL      '\x7' /* ASCII znak za zvučni signal */
```

Potpuni skup escape nizova jest

```
\a    znak za zvučni signal        \\    obrnuta kosa crta
\b    backspace                    \?    upitnik
\f    form feed                    \'    jednostruki navodnik
\n    novi red                     \"    dvostruki navodnik
\r    carriage return              \ooo  oktalni broj
\t    horizontalni tabulator       \xhh  heksadecimalni broj
\v    okomiti tabulator
```

Znakovna konstanta '\0' predstavlja znak čija je vrijednost nula (u brojevnom smislu). Često se piše upravo '\0' umjesto 0 kako bi se naglasila znakovna priroda nekog izraza, ali brojevna vrijednost mu je nula.

Konstantni izraz je izraz koji sadrži samo konstante. Takvi se izrazi daju izračunati prilikom prevođenja, a prije samog pokretanja programa, pa se sukladno tome mogu rabiti na svakom mjestu gdje se može pojaviti konstanta, kao npr.

```
#define     MAXLINE     1000
char line[MAXLINE];
```

ili

```
#define     LEAP  1     /* u prijestupnim godinama */
int dani[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

Znakovni nizovi su nizovi nula i više znakova omeđenih dvostrukim navodnicima, kao

```
"Ja sam niz"
```

ili

```
""    /* prazan niz */
```

Navodnici nisu dio niza, već se koriste kako ograničili. Isti escape nizovi koji se rabe u znakovnim konstantama pojavljuju se i u nizovima; '\"' predstavlja dvostruki navodnik. Znakovni se nizovi mogu povezati prilikom prevođenja

```
"Hello, "   "World"
```

je isto što i

```
"Hello, World"
```

Ova osobina korisna je pri radu s velikim nizovima jer se mogu podijeliti na više osnovnih linija.

Tehnički, znakovni niz je polje znakova. Interno predstavljanje niza ima na kraju nulu, odnosno, znak '\0', tako da fizičko pohranjivanje niza zahtjeva jedan znak više od znakova napisanih među navodnicima. To ne znači da nizovi imaju nekakvo ograničenje u smislu dužine, ali programi moraju pregledati cijeli niz kako bi je odredili. Funkcija strlen(s) iz standardne biblioteke programa, vraća dužinu znakovnog niza s, pri čemu ne računa završnu nulu (znak '\0'). Evo naše verzije:

```
/* strlen: vraća dužinu niza */
int strlen(char s[]){
      int i;

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

Ostale funkcije nizova i strlen deklarirane su u standardnom zaglavlju <string.h>. Treba biti oprezan i praviti razliku između znakovne konstante i niza koji ima samo jedan znak; 'x' nije isto što i "x". Prvi znak predstavlja cjelobrojnu vrijednost znaka x u strojnom skupu znakova (bilo da se radi o ASCII ili bilo kojem drugom). Drugi je polje znakova koje ima jedan znak (x) i '\0'.

Postoji još jedna vrsta nizova, tzv. nizovi dobiveni pobrojavanjem (enumerate). Pobrojavanje je formiranje liste konstantnih cjelobrojnih vrijednosti, kao u

```
enum boolean{NO, YES};
```

Prvi naziv u enum listi ima vrijednost 0, drugi 1. Da imamo još koji element on bi imao vrijednost 2 itd. dok se god eksplicitno ne zada neka druga vrijednost. Progresija ide od one vrijednosti koja je posljednja eksplicitno zadana (inače od 0) kao u npr.

```
      enum escapes{BELL='\a', BACKSPACE='\b', TAB='\t', NEWLINE='\n', VTAB='\v',
RETURN='\r'};
      enum mjeseci{JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV,
DEC}; /* FEB=2, MAR=3, itd*/
```

Imena se u različitim pobrojavanim nizovima moraju razlikovati, a vrijednosti u jednom nizu mogu biti jednake.

Nizovi dobiveni pobrojavanjem osiguravaju zgodan način pridruživanja konstantnih vrijednosti imenima, nasuprot #define, uz očitu prednost generiranja novih vrijednosti. Iako varijable tipa enum mogu biti deklarirane, prevoditelji ne moraju provjeravati da li se u varijablu pohranjuje ispravna vrijednost za pobrojavanje. No, kako pobrojane varijable pružaju mogućnost provjere, često su bolje od #define alternative. Uz to, program za analizu izvornog koda (debugger), ispisuje vrijednosti pobrojanih varijabli u njihovoj simboličkoj formi.

2.4 Deklaracije

Sve varijable prije uporabe moraju biti deklarirane, mada određene deklaracije mogu biti obavljene tako da slijede iz konteksta. Deklaracija određuje tip, a sadrži listu od jedne ili više varijabli tog tipa, kao npr.

```
int lower, upper, step;
char c, line[1000];
```

Deklaracije se mogu dijeliti na varijable bilo kako; gore napisano se dade i drugačije napisati

```
int lower;
int upper;
int step;
char c;
char line[1000];
```

Ovakav način uzima više prostora, ali je zato zgodan za dodjelu komentara svakoj deklaraciji ili kasnije promjene.

Varijabla može biti inicijalizirana u svojoj deklaraciji. Ako iza imena varijable slijedi znak jednakosti i izraz, tada taj izraz služi kao inicijalizator kao u

```
char esc='\\';
int i=0;
int limit=MAXLINE+1;
float eps=1.0e-5;
```

Ako varijabla nije automatska, inicijalizacija se obavlja samo jednom, prije pokretanja programa, a inicijalizator mora biti konstantni izraz. Eksplicitno inicijalizirana automatska varijabla se inicijalizira svaki put kad se prolazi kroz funkciju, odnosno blok u kojem se varijabla nalazi; inicijalizator može biti bilo kakav izraz. Vanjske i statičke varijable se inicijaliziraju na nulu. Automatske varijable za koje ne postoji eksplicitni inicijalizator dobijaju nedefinirane vrijednosti (garbage).

Kvalifikator const može se dodijeliti deklaraciji bilo koje varijable kako bi označio da se ona neće mijenjati. Za polje, npr., on pokazuje da se elementi polja neće mijenjati.

```
const double e=2.71828182845905;
const char msg[]="warning:";
```

Deklaracija const se rabi i sa argumentima polja, kako bi naglasila da funkcija neće mijenjati dano polje

```
int strlen(const char[]);
```

Rezultat je jasno određen ako dođe do pokušaja promjene objekta deklariranog kao const.

2.5 Aritmetički operatori

Binarni aritmetički operatori su +, -, *, /, i modul operator %. Izraz

```
x%y
```

daje ostatak pri dijeljenju x sa y, a jednak je nuli ako je x djeljivo sa y. U primjeru, godina se smatra prestupnom ako je djeljiva s 4, ali ne sa 100, osim ako nisu djeljive s 400

```
if((godina%4==0&&godina%100!=0||godina%400==0)
      printf("%d je prestupna godina\n", godina);
else
      printf("%d nije prestupna godina\n");
```

Operator % ne može biti pridružen tipovima float i double. Smjer zaokruživanja (ili odsjecanja?!) za operator / i predznak rezultata operacije % ovisni su o tipu računala za negativne operande kao i kod pojave prekoračenja i potkoračenja (overflow i underflow).

Binarni operatori + i - imaju isti prioritet, koji je niži od prioriteta operatora *, / i %, a koji opet imaju niži prioritet od unarnih operatora + i -. Aritmetički su operatori asocijativni slijeva nadesno.

Tablica 2-1, na kraju poglavlja, prikazuje prioritete i asocijativnost svih operatora.

2.6 Relacijski i logički operatori

Relacijski operatori su >, >=, <, <=. Svi imaju isti prioritet. Odmah ispod njih po prioritetu su operatori jednakosti == i !=. Relacijski operatori imaju niži prioritet od aritmetičkih operatora, pa izraz i<lim-1 se tretira kao i i<(lim-1), što je prirodno.

Mnogo zanimljiviji su logički operatori && i ||. Izrazi povezani ovim operatorima računaju se slijeva udesno, računanje se prekida čim se ustanovi točnost ili netočnost rezultata. Mnogi C programi napisani su s jakim osloncem u ovim pravilima. Npr. evo petlje ulazne funkcije getline koju smo proučili u Poglavlju 1

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

Prije čitanja novog znaka treba obavezno provjeriti ima li za njega mjesta u znakovnom nizu s, tako da se najprije obavi provjera i<lim-1. Ako je odgovor negativan, nema smisla dalje čitati znakove. Bilo bi jednako tako pogrešno ako bi se prije pozivanja funkcije getchar, obavila provjera c!=EOF, jer pozivanje i dodjela vrijednosti moraju izvršiti prije negoli se testira znak pohranjen u c.

Prioritet && je viši od ||, a oba su operatora nižeg ranga od relacijskih i operatora jednakosti, pa zato u slijedećem izrazu nije potrebno dodavati zagrade

```
i<lim-1&&(c=getchar())!='\n'&&c!=EOF
```

No, kako je prioritet != viši od operatora dodjele, zagrade su obavezne u izrazu što slijedi

```
(c=getchar())!='\n'
```

da bi se osigurala dodjela varijabli c, a potom i usporedba sa '\n'.

Po definiciji, brojčana vrijednost relacijskog ili logičkog izraza je 1 ako je relacija točna, a 0 ako je netočna.

Operator unarne negacije ! pretvara 1 u 0 i obratno. Najčešća je upotreba ovog operatora u izrazima poput

```
if(!valid)
```

umjesto

```
if(valid==0)
```

Teško je kazati koja je forma bolja. Konstrukcije kao !valid lijepo izgledaju ("ako nije valid"), dok bi zahtjevnije konstrukcije bile i teže za čitanje.

Vježba 2-2. Napišite petlju ekvivalentnu gore napisanoj for petlji bez upotrebe && ili ||.

2.7 Konverzije tipova

Kada jedan operator ima operande različitih tipova, obavlja se pretvorba u zajednički tip s pomoću nekoliko pravila. Općenito, jedine pretvorbe koje se obavljaju automatski su one koje pretvaraju operande iz "manjeg skupa" u operande iz "većeg skupa" bez gubitka informacije, kakva je primjerice pretvorba cijelih u realne brojeve. Besmisleni izrazi, kakvi su npr. oni s brojem tipa float u eksponentu, nisu dopušteni. Pretvorba realnih u cijele brojeve, pri čemu se gubi dio informacije, izaziva upozorenje prilikom prevođenja, ali takovi izrazi nisu nedopušteni.

Kako je char mala cjelobrojna vrijednost, dade se koristiti u aritmetičkim izrazima. Ta osobina rezultira velikom fleksibilnošću pri znakovnim transformacijama. Primjer toga je i jednostavna uporaba funkcije atoi, koja pretvara znakovni niz sastavljen od brojeva u njegov brojčani ekvivalent.

```
/* atoi: pretvorba s u cijeli broj */
int atoi(char s[]){
      int i, n;

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

Kao što smo vidjeli u Poglavlju 1, izraz

```
s[i]-'0'
```

daje brojčanu vrijednost znaka pohranjenog u s[i], stoga što vrijednosti '0', '1', itd. tvore kontinuirano rastući niz.

Slijedeći primjer pretvorbe tipa char u int je funkcija lower, koja transformira pojedini znak u mala slova u ASCII setu znakova. Ako znak nije veliko slovo vraća se nepromijenjen.

```
/* lower: pretvorba c u mala slova; ASCII skup znakova */
int lower(int c){
      if(c>='A'&&c<='Z')
            return c+'a'-'A';
      else
            return c;
}
```

Ovo vrijedi za ASCII skup znakova, jer odgovarajuća velika i mala slova imaju jednak razmak od početka do kraja abecede, jer između nema nikakvih drugih znakova osim slova. Ovo zadnje ne vrijedi za EBCDIC skup znakova, pa on ne bi pretvarao samo slova.

Standardno zaglavlje <ctype.h> opisano u Dodatku B, definira cijeli niz funkcija koje osiguravaju provjeru i pretvorbu za različite skupove znakova. Primjerice, funkcija tolower(c) vraća brojčanu vrijednost malog slova varijable c ako ona reprezentira veliko slovo, odnosno, ona predstavlja zamjenu za funkciju lower koju smo napisali. Slično tome, test

```
c>='0'&&c<='9'
```

dade se zamijeniti sa isdigit(c).

Postoji jedan trik kod pretvorbe znakova u cijele brojeve. Jezik ne određuje da li su varijable tipa char veličine iz skupa pozitivnih ili negativnih brojeva. Da li se prilikom pretvorbe tipa char u tip int mogu dobiti negativni cijeli brojevi? Odgovor nije jedinstven jer varira od računala do računala. Na nekim računalima char čiji je krajnji lijevi bit 1 bit će pretvoren u negativan cijeli broj, a na drugima char se prevodi u int dodavanjem nule na krajnji lijevi bit, pa je uvijek pozitivan.

Definicija C-a jamči da bilo koji znak za ispis, u standardnom skupu znakova, nikad neće biti negativan, pa će ti znakovi uvijek predstavljati pozitivne veličine u izrazima. Ipak, neki nizovi bitova

pohranjena u znakovnim varijablama može se interpretirati kao negativna vrijednost na nekim računalima, a pozitivna na drugim. Zbog prenosivosti (portabilnosti) treba obvezno uključiti signed ili unsigned kvalifikatore ako se neznakovni podatak pohranjuje u char varijablu.

Relacijski izrazi kao i>j i logički izrazi povezani s && ili || su definirani tako da imaju vrijednost 1 ako su točni, a 0 ako nisu. U tom smislu, dodjela

```
d=c>='0'&&c<='9'
```

postavlja d na vrijednost 1 ako je c broj, a na 0 ako nije. Bilo kako bilo, funkcije poput isdigit mogu vratiti bilo koju vrijednost različitu od nule, kao signal da je ulazni znak broj. U provjeri kod if, while i for naredbi točan izraz isto ima vrijednost koja nije nula.

Implicitne aritmetičke pretvorbe urade i iznad očekivanja. Ako operator kao što je + ili * koji traži dva operanda (binarni operator) dobije operande različitih tipova, niži tip se promovira u viši prije početka same operacije. Rezultat je viši tip. Dio pod naslovom 6, Dodatka A opisuje precizno sva pravila pretvorbe (ali ne i privatizacije). Da nema unsigned operanada, slijedeći skup pravila bi bio dovoljan

-ako je jedan od dva operanda tipa long double, drugi se pretvara u long double
-ako je jedan operand tipa double, pretvara se i drugi u double.
-ako je jedan operand tipa float, pretvara se i drugi u float.
-u drugim slučajevima obavlja se pretvorba char i short u int.
-naposlijetku, ako je jedan od operanda tipa long i drugi se pretvara u long.

Primijetimo da float u izrazima ne pretvara automatski u double što je ozbiljna promjena u odnosu na originalnu definiciju. Generalno gledano, matematičke funkcije kakve su u <math.h> rabe dvostruku točnost. Glavni razlog za upotrebu float jest očuvanje memorijskog prostora u radu s velikim poljima ili, što je rjeđi slučaj, ušteda vremena na računalima na kojima je rad s dvostrukom točnošću prilično spor.

Pravila pretvorbe postaju teža kad se u igru uključe operatori tipa unsigned. Problem je usporedba vrijednosti koje mogu biti pozitivne i negativne, zavisno o računalu. Npr. pretpostavimo da int uzima 16 bitova, a long 32 bita. Tada je -1L isto što i 1U, jer je 1U koji je int promoviran u signed long. Međutim, i -1L je isto što i 1UL, jer se -1L isto tako promovira u unsigned long i pojavljuje kao veliki pozitivni cio broj.

Pretvorbe se ostvaruju preko naredbi dodjele; vrijednost s desne strane se pretvara u tip s lijeve strane, što je zapravo tip rezultata.

Znak se pretvara u cijeli broj, pozitivan ili negativan, to smo već riješili.

Duži cijeli brojevi pretvaraju se u kraće ili u vrijednosti tipa char. Tako u izrazima

```
int i;
char c;

i=c;
c=i;
```

vrijednost c ostaje nepromijenjena. To vrijedi bez obzira na to je li uračunat krajnji lijevi bit ili ne. Promjena slijeda dodjela dovodi do mogućeg gubitka informacija.

Ako je x tipa float i i tipa int, tada i za x=i i za i=x dolazi do pretvorbe pri čemu pretvorba float u int gubi dio informacije (onaj iza decimalne točke). Kad se double pretvara u float, pretvorba ovisi o primjeni bez obzira je li vrijednost zaokružena ili dolazi do gubitka decimalnog dijela.

Kada je argument poziva neke funkcije izraz, pretvorba se događa kad argument pripadne funkciji. Ako funkcije nema tip, char i short postaju int, a float postaje double. To je razlog zašto prijavljujemo argumente funkcija kao int i double čak i kad se funkcija poziva char i float tipovima.

U krajnjem slučaju, eksplicitna pretvorba tipa se može ostvariti u bilo kom izrazu, unarnim operatorom koji se zove cast. U konstrukciji

```
(tip) izraz
```

izraz se pretvara u navedeni tip pravilima pretvorbe o kojima je već bilo riječi. Pravo značenje cast operatora vidi se kad se izraz dodjeli varijabli određenog tipa, pa se ona rabi mjesto cijele konstrukcije. Npr., funkcija iz biblioteke sqrt očekuje double kao argument, pa ukoliko argument nije tog tipa rezultat nije moguće predvidjeti (sqrt je deklarirana u <math.h>). Ukoliko je n cijeli broj, možemo koristiti

```
sqrt((double) n)
```

za pretvorbu vrijednosti n u double prije negoli se nad njim obavi sqrt. Pripomenimo da cast prevodi vrijednost n u navedeni tip pri čemu se n ne mijenja. Cast operator ima visoki prioritet kao i drugi unarni operatori, što ćemo pokazati tablicom na kraju ovog poglavlja.

Ako su argumenti deklarirani funkcijskim prototipom kako se i očekuje, deklariranje pokreće automatsku pretvorbu svih argumenata prilikom poziva funkcije. Tako, dani model funkcije za sqrt

```
double sqrt(double);
```

pri pozivu

```
root2=sqrt(2);
```

pretvara cijeli broj 2 u vrijednost 2.0 tipa double bez upotrebe operatora cast.

U standardnoj biblioteci postoji mali generator slučajnih brojeva i funkcija za pokretanje, a slijedeći primjer ilustrira operator cast

```
unsigned long int next=1;

/* rand: vraća slučajni cijeli broj iz intervala [0,32767] */
int rand(void){
      next=next*1103515245+12345;
      return (unsigned int) (next/65536)%32768;
}

/* srand: postavljanje parametra za rand funkciju */
void srand(unsigned int pocetak){
      next=pocetak;
}
```

Vježba 2-3. Napisati funkciju htoi(s), koja pretvara niz heksadecimalnih brojeva (uključujući opcijske 0x ili 0X) u njenu ekvivalentnu cjelobrojnu vrijednost. Dopuštene znamenke su [0-9], [a-f] ili [A-F].

2.8 Operatori uvećavanja i umanjivanja (inkrementiranja i dekrementiranja)

C ima dva neuobičajena operatora za uvećavanje i smanjivanje (za jedan) varijabli. Operator inkrementiranja ++ dodaje 1 svom operandu, dok operator dekrementiranja oduzima 1. Često ćemo koristiti ++ za uvećavanje varijabli kao u

```
if(c=='\n')
      ++nl;
```

Neobičnost ovih operatora jest da se mogu koristiti kao prefiks operatori (++n) ili kao sufiks (n++). Učinak je u oba slučaja jednak - varijabla se uveća za jedan. Međutim, izraz ++n uvećava vrijednost n prije negoli njegova vrijednost bude upotrijebljena (pretpostavimo da je ++n dio neke relacije ili jednakosti), a n++ nakon korištenja. To zapravo znači da ta dva izraza nikako ne možemo smatrati ekvivalentnima osim ako nisu izdvojene programske linije. Neka je n 5, tada

```
x=n++;
```

postavlja x na 5, dok

```
x=++n;
```

postavlja x na 6. U oba slučaja, n postaje 6. Operatori uvećavanja i umanjivanja pridjeljuju se samo varijablama; izraz kakav je (i+j)++ jest nekorektan.

U slučaju kad se određena vrijednost ne koristi već je bitno samo uvećanje kao

```
if(c=='\n')
      nl++;
```

prefiks i sufiks oblik izraza postaju ekvivalentni. Ima situacija, međutim, kad se jedan ili drugi ciljano zovu. Primjerom kazano, načinimo prispodobu funkcije squeeze(s, c) koja uklanja znak c kad god se pojavi u nizu s.

```
/* squeeze: brisanje svih c iz s */
void squeeze(chars[], int c){
      int i,j;

      for(i=j=0;s[i]!='\0';i++)
            if(s[i]!=c)
                  s[j++]=s[i];
      s[j]='\0';
}
```

Za svaki znak različit od c, operator uvećava j da bi prihvatio slijedeći znak. Ovo je potpuno ekvivalentno s

```
if(s[i]!=c){
      s[j]=s[i];
      j++;
}
```

Slijedeći primjer je slična konstrukcija iz getline funkcije o kojoj je već bilo riječi u Poglavlju 1, gdje dio programa

```
if(c=='\n'){
      s[i]=c;
      ++i;
}
```

može biti napisan i kao

```
if(c=='\n')
      s[i++]=c;
```

Kao treći primjer, prispodobimo standardnu funkciju strcat(s, t), koja spaja kraj znakovnog niza s s početkom niza t. Funkcija strcat pretpostavlja da ima dovoljno mjesta u s da smjesti oba niza. Kako smo već napisali, strcat ne vraća nikakvu vrijednost osim pokazivača (pointera) na rezultirajući znakovni niz.

```
/* strcat: povezivanje znakovnih nizova */
void strcat(char s[], char t[]){
      int i, j;

      i=j=0;
      while(s[i]!='\0') /* Gdje je kraj niza s ? */
            i++;
      while((s[i++]=t[j++])!='\0')  /* prebaci t */
            ;
}
```

Kada se znakovi prebacuju iz t u s, sufiks ++ pridružuje se i varijabli i i varijabli j kako bi smo bili sigurni da su u poziciji za naredni prolaz kroz petlju.

Vježba 2-4. Napisati alternativnu inačicu funkcije squeeze(s1, s2) koja briše svaki znak u s1 koji postoji u s2.

Vježba 2-5. Napisati funkciju any(s1, s2) koja vraća prvo mjesto u nizu s1 gdje se nalazi bilo koji znak iz niza s2 ili -1 ako niz s1 nema nijednog znaka iz niza s2 (funkcija strcpbrk iz standardne biblioteke radi isto, osim što vraća pokazivač na mjesto).

2.9 Operatori za manipulaciju bitovima

C osigurava šest operatora za rad s bitovima. Oni rade samo s integralnim operandima kao što su char, short, int i long, bez obzira bili oni signed ili unsigned tipa. To su

&	binarni I (AND)
|	binarni ILI (OR)
^	binarno EXILI (XOR)
<<	lijevi pomak (left shift)
>>	desni pomak (right shift)
~	jedinični komplement (unarni)

Operator binarno AND & najčešće se rabi za maskiranje bitova. Tako

```
n=n&0177;
```

postavlja na nulu sve osim najnižih 7 bitova n.
Operatorom binarno OR | možemo birati bitove koje postavljamo na 1. Tako

```
x=x|SET_ON;
```

postavlja u varijabli x na 1 one bitove koji su 1 u SET_ON.
Ekskluzivno ILI, ^, je operator koji postavlja 1 na mjestima na kojima operandi imaju različite bitove, a nule gdje su bitovi isti. Možemo napisati

```
x=x^x;
```

želimo li x postaviti na nulu.
Mora se praviti razlika između operatora za manipuliranje bitovima & i |, te logičkih operatora && i ||, koji računaju slijeva nadesno istinitost izraza. Npr., ako je x=1, a y=2, tada je x&y=0 dok je x&&y=1.
Operatori pomaka (shift) << i >> određuju lijevi i desni pomak njihovog lijevog operanda pomoću broja pozicija bitova kojeg daje desni operand, obavezno pozitivan. Zato x<< pomiče vrijednost x ulijevo za dva mjesta, puneći slobodne bitove nulama, što odgovara množenju s 4. Pomicanje unsigned veličine udesno uvijek puni slobodne bitove nulama. Desno pomicanje signed veličine će popuniti slobodne bitove aritmetičkim pomicanjem na jednim računalima, a logičkim pomicanjem na drugima.
Unarni operator ~ daje jedinični komplement neke cjelobrojne vrijednosti. On zapravo obrće sve bitove (0 ide u 1, a 1 u 0). Npr.

```
x=x&~077
```

postavlja šest krajnjih desnih bitova na 0. Primijetimo da je x&~077 nezavisno od dužine riječi što je bitna prednost ispred npr., x&0177700, koje pretpostavlja da je x 16-bitna veličina. Prenosivost na druga računala nije upitna jer je ~077 konstantni izraz koji se računa za vrijeme prevođenja.
Kao zorni prikaz operatora za manipulaciju bitovima, promotrimo funkciju getbits(x, p, n) koja vraća n-bitno polje iz x (koje je krajnje desno) namješteno na poziciju p. Pretpostavka je da su n i p male pozitivne vrijednosti, a da je pozicija bita 0 s desne strane. Naša će funkcija pozivom getbits(x, 4, 3) vratiti tri desna bita namještena na bit pozicijama 4, 3, 2.

```
/* getbits: postavlja n desnih bitova na poziciju p */
unsigned getbits(unsigned x, int p, int n){
      return(x>>(p+1-n))&~(~0<<n);
}
```

Izraz x>>(p+1-n) pomiče željeno polje na desnu stranu riječi. ~0 ima samo jedan bit. Pomičući ga ulijevo n pozicija sa ~0<<n postavljamo nule u n krajnjih desnih bit pozicija. Zatim komplementirajući to sa ~ pravimo masku jedinica u n krajnjih desnih bitova.

Vježba 2-6. Napišite funkciju setbits(x, p, n, y) koja vraća x sa n bitova koji počinju na poziciji p namještenoj na n krajnjih desnih bitova od y, ostavljajući pritom ostale bitove neizmijenjenima.

Vježba 2-7. Napišite funkciju invert(x, p, n) koja vraća x sa n bitova koji počinju na invertiranoj p poziciji, ne mijenjajući druge bitove.

Vježba 2-8. Napišite funkciju rightrot(x, n) koja vraća vrijednost cjelobrojnog x rotiranog udesno pomoću n bit pozicija.

2.10 Operatori i izrazi dodjeljivanja vrijednosti

Izrazi kakav je

```
i=i+2;
```

u kojima je varijabla na lijevoj strani jedan od operanda na desnoj, mogu se pisati u komprimiranom obliku

```
i+=2;
```

Operator += naziva se operatorom dodjele vrijednosti.
Većina binarnih operatora (operatori kao +, dakle, koji imaju lijevi i desni operand) imaju odgovarajući operator dodjele vrijednosti op=, pri čemu je op jedan od

```
+     -     *     /     %     <<    >>    &     ^     |
```

Ako su imamo izraze izraz1 i izraz2, tada je

```
izraz1 op=izraz2;
```

ekvivalentno sa

```
izraz1=(izraz1)op(izraz2);
```

osim što se izraz1 računa samo jedanput. Primijetimo zagrade oko izraz2:

```
x*=y+1;
```

jest

```
x=x*(y+1);
```

a ne

```
x=x*y+1;
```

Primjera radi, napisat ćemo funkciju bitcount koja prebrojava jedinice u binarnom obliku cjelobrojne vrijednosti argumenta.

```
/* bitcount: broji jedinice binarnog broja */
int bitcount(unsigned x);{
      int b;

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

Prijavom argumenta kao unsigned, osiguravamo da se desnim pomakom slobodni bitovi ispunjavaju nulama, neovisno o računalu na kojemu pokrećemo program.

Osim kratkoće i konciznosti, operatori dodjele vrijednosti imaju prednost u komunikaciji s onim koji čita ili proučava program. Uistinu je lakše shvatiti konstrukciju tipa "povećaj i za 2" negoli "uzmi i, dodaj 2, a rezultat vrati u i". Stoga je izraz i+=2 čitljiviji nego i=i+2. Pored toga, za kompleksnije izraze kakav je

```
yyval[yypv[p3+p4]+yypv[p1+p2]]+=2;
```

operator dodjele vrijednosti čini kod lakšim za razumijevanje, jer čitatelj ne mora provjeravati da li su dva duga izraza zapravo jednaka. Operator dodjele vrijednosti ima učinkovitije djelovanje od druge konstrukcije jer je prevedeni kod efikasniji. Već smo primjere rabili u programima, najčešće kao dio izraza

```
while((c=getchar())!=EOF)
      ...
```

Ostali operatori dodjele vrijednosti (+=, -=, itd.) mogu se umetati u izraze, ali rjeđe.
U svim ovakvim izrazima, tip izraza za dodjelu vrijednosti je tip njegovog lijevog operanda, a vrijednost je ona nakon dodjele.

Vježba 2-9. U dvokomplementnom sustavu brojeva izraz x&=(x-1) briše krajnji desni bit u x. Pojasnite zašto. Iskoristite to zapažanje kako biste napisali bržu inačicu bitcount funkcije.

2.11 Uvjetni izrazi

Izrazi

```
if(a>b)
      z=a;
else
      z=b;
```

pohranjuju u z maksimum od a i b. Uvjetni izrazi, pisani pomoću ternarnog operatora ?:, određuju alternativan način pisanja ovog izraza i sličnih konstrukcija. U izrazu

```
izraz1?izraz2:izraz3;
```

najprije se računa izraz1. Ako njegova vrijednost nije nula (ako je istinit), tada se računa izraz2, a to je najveća vrijednost uvjetnog izraza. Naime, prolazi se kroz samo jedan od izraza izraz2 ili izraz3. Tako, da bi pohranili u z maksimum iz a i b pišemo

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

Morate uočiti da je uvjetni izraz zapravo samo jedan izraz koji se dade koristiti kao svaki drugi. Ako su izraz2 i izraz3 različitog tipa, tip rezultata je određen zakonima pretvorbe o kojima smo pričali u ovom poglavlju. Npr. ako je f float, a n je int, tada je izraz

```
(n>0)?f:n;
```

tipa float bez obzira je li n pozitivan.
Oko prvog uvjetnog izraza zagrade nisu obavezne, dok je prioritet od ?: nizak, samo nešto veći od uvjeta. One se ipak preporučuju radi čitljivosti programa. Uvjetni izraz je često vrlo efikasan. Npr. ova petlja ispisuje n elemenata jednog polja, deset po retku, sa stupcima koji su razdvojeni razmakom, a svaki red (uključujući i zadnji) završava znakom novog reda.

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

Znak novog reda se ispisuje nakon svakog desetog elementa, te poslije zadnjeg. Svi ostali elementi su popraćeni ispisom razmaka. To može djelovati kao doskočica, ali jest kreativnije od odgovarajuće if-else konstrukcije. Slijedeći dobar primjer je

```
printf("Imate %d elemenata%s.\n", n, n==1?"":"s");
```

Vježba 2-10. Nanovo napišite funkciju lower, koja pretvara velika u mala slova pomoću uvjetnog izraza umjesto if-else.

2.12 Prioritet i redoslijed računanja

Tablica 2-1 prikazuje pravila prioriteta i veza svih operatora, uključujući i one o kojima još nije bilo riječi. Operatori u istom retku imaju jednak prioritet, pri čemu su redovi nizani po rastućem prioritetu, pa zato *, / i % imaju isti prioritet koji je veći od prioriteta + i -. "Operator" odnosi se na pozivanje funkcija. Operatori -> i . rabe se za pristup članovima struktura o čemu ćemo diskutirati u Poglavlju 6, kao i o sizeof-u (veličini objekta). U Poglavlju 5 govori se o znaku * (indirekcija kroz pokazivač) i & (adresa objekta), a u Poglavlju 3 o operatoru , (zarez).

Možete zamijetiti kako prioritet operatora za manipulaciju bitovima &, ^ i | ima svoje mjesto između operatora == i !=. Iz toga zaključujemo da izrazi za provjeru bitova oblika

```
if((x&MASK)==0)
```

moraju biti omeđeni zagradama radi točnosti rezultata.

C, kao uostalom i mnogi jezici, nema jesno određenje u smislu reda računanja operanda u izrazu (izuzeci su logički operatori && i ||, te uvjetni izraz ?: i ,). Tako u izrazu

```
x=f()+g();
```

f() može biti izračunata prije g(), ali i obrnuto. Stoga, ako f i g mijenjaju varijablu bitnu za druge funkcije, x može imati različite vrijednosti. Međurezultati se mogu pohranjivati u privremenim varijablama kako bi se takav problem riješio.

Slično tome, ni redoslijed računanja argumenata funkcije nije determiniran pa naredba

```
printf("%d %d\n", ++n, power(2, n));      /* pogrešno */
```

može dati različite rezultate na različitim prevoditeljima, zavisno od toga da li je n povećano prije negoli je funkcija power pozvana. Problem se, jasno, dade riješiti sa

```
++n;
printf("%d %d\n", n, power(2, n));
```

Pozivi funkcija, umetanje naredbi dodjele vrijednosti, te operatora uvećavanja i umanjivanja mogu proizvesti neke nuspojave (side effect) - varijabla se može mijenjati tokom računanja izraza. U bilo kakvom izrazu u kojemu se nuspojave manifestiraju, javljaju se zamršene zavisnosti. To se redovito događa u naredbama u kojima se u izrazu pojavljuju varijable već obrađene u istom izrazu. Evo jedne nezgodne situacije

```
a[i]=i++;
```

Pitanje je da li je indeks stara ili nova vrijednost od i. Prevoditelji ovo mogu interpretirati na razne načine i generirati različite odgovore. Standard namjerno ostavlja mnoga od ovih pitanja nedefiniranim. Kad se jave nuspojave u nekom izrazu čija vrijednost ovisi o prevoditelju, rješenje nije univerzalno nego varira od računala do računala (standard kaže da se nuspojave na argumentima događaju prije negoli je funkcija pozvana, ali to ne bi pomoglo pri ranijem pozivanju printf funkcije).

Dobro je znati da pisanje koda, zavisnog o naredbi za računanje, predstavlja lošu programsku praksu. Treba, unatoč tomu, znati koje stvari moramo izbjegavati, ali bez znanja o tome kako su one realizirane na različitim platformama, nećete moći koristiti komparativne prednosti pojedinih konfiguracija.

TABLICA 2-1. PRIORITET I ASOCIJATIVNOST OPERATORA

Operatori	Asocijativnost
`()  {}  ->  .`	slijeva nadesno

`!  ~  ++  --  +  -  *  &  (tip)  sizeof`	sdesna nalijevo
`*  /  %`	slijeva nadesno
`+  -`	slijeva nadesno
`<<  >>`	slijeva nadesno
`<  <=  >  >=`	slijeva nadesno
`==  !=`	slijeva nadesno
`&`	slijeva nadesno
`^`	slijeva nadesno
`|`	slijeva nadesno
`&&`	slijeva nadesno
`||`	slijeva nadesno
`? :`	sdesna nalijevo
`=  +=  -=  *=  /=  %=  &=  ^=  |=  <<=  >>=`	sdesna nalijevo
`,`	slijeva nadesno

POGLAVLJE 3: KONTROLA TOKA

Izrazi kontrole toka u jeziku određuju red kojim će se naredbe izvršavati. Već smo vidjeli najjednostavnije konstrukcije kontrole toka ranije. Sad ćemo upotpuniti znanja i preciznije pojasniti naredbe koje smo spominjali.

3.1 Naredbe i blokovi

Izraz kao x=0 ili printf(...) postaje naredba kad je popraćen točka-zarezom, kao u

```
x=0;
i++;
printf(...);
```

U C-u se točka-zarezom završava svaka naredba. Tako je, u C-u, točka-zarez (;) terminator, a ne separator, kao, recimo u Pascalu.

Vitičaste zagrade {} rabe se za grupe deklaracija i naredbe u složenoj naredbi tj., bloku koji je sintaksno jednakovrijedan jednoj naredbi. Vitičaste zagrade koje omeđuju naredbe funkcija su klasičan primjer. Sličan primjer su takove zagrade oko mnogostrukih naredbi nakon if, else, while ili for (varijable se mogu prijaviti unutar bilo kojeg bloka o čemu ćemo pričati u Poglavlju 4). Nakon desne vitičaste zagrade kojom se okončava blok ne treba stavljati ";".

3.2 If – else

Naredba if - else koristi se za donošenje određenih odluka. Formalna sintaksa jest

```
if(izraz)
      naredba1
else
      naredba2
```

pri čemu je else neobavezan. Izraz se provjerava. Ako je istinit (ima vrijednost koja nije nula), izvršava se naredba1. Ako nije istinit (ima vrijednost nula) i ako postoji else dio, izvršava se naredba2.

Kako if testira brojevnu vrijednost nekog izraza, moguća su određena skraćenja koda. Primjerice dovoljno je napisati

```
if(izraz)
```

umjesto

```
if(izraz!=0)
```

Katkad je ovaj način pisanja ipak zgodniji i čitljiviji.

Na prvi pogled, malo je nejasno što se događa u slučaju kad if - else konstrukcija nema else dijela (jer je neobavezan). Postoji pravilo, potpuno intuitivno, koje pridružuje else najbližem prethodnom if-u koji nema else. Npr. u

```
if(n>0)
      if(a>b)
            z=a;
      else
            z=b;
```

else se pridružuje unutrašnjem if-u, što zorno pokazuje i uvučenost redova. Ako to nije željeni raspored, morate iskoristiti vitičaste zagrade kako bi pojasnili prevoditelju što zapravo želite

```
if(n>0){
      if(a>b)
            z=a;
```

```
}
else
      z=b;

Česta je greška

if(n>0)
      for(i=0;i<n;i++)
            if(s[i]>0){
                  printf("...");
                  return i;
            }
else  /* pogrešno */
      printf("greska - n je negativno\n");
```

Uvučenost redova pokazuje što programer želi, ali prevoditelj else pridružuje unutrašnjem if-u. Ovakve greške nije lako naći. Zato je topla preporuka korištenje vitičastih zagrada kad je to moguće, a ne kad se može.

Osim toga primijetite da postoje točka-zarez nakon z=a u

```
if(a>b)
      z=a;
else
      z=b;
```

Gramatički gledano, naredba ide kao posljedica if, a naredbu moramo okončati s točka-zarezom.

3.3 Else – if

Konstrukcija

```
if(izraz)
      naredba
else
      if(izraz)
            naredba
      else
            if(izraz)
                  naredba
            else
                  if(izraz)
                        naredba
                  else
                        naredba
```

je toliko česta da o tomu ne treba trošiti riječi. Ovaj niz if naredbi predstavlja najopćenitiji način pisanja višesmjernog grananja. Izrazi se računaju po redu. Ako je bilo koji izraz istinit pridružena naredba se izvršava, čime se završava čitava konstrukcija. Kao i uvijek, naredbom se podrazumijeva ili jedna naredba ili više njih omeđenih vitičastim zagradama.

Zadnja else naredba ne obavlja nijedan od gornjih slučajeva, ali izvodi slučajeve koji se izvršavaju kad nijedan od uvjeta nije zadovoljen. Ponekad takvi, podrazumijevani slučajevi, ne rade ništa, pa tada zadnji else nije ni potreban. Međutim, ipak je dobro ostaviti else, te pomoću njega tražiti nemoguće greške.

Kako bi ilustrirali grananje s tri grane, proučimo funkciju binarnog pretraživanja koja kaže da li je određena vrijednost x nalazi u sortiranom polju v. Elementi polja v moraju biti uređeni po rastućem nizu. Funkcija vraća poziciju (broj između 0 i n-1) ako se x pojavi u v, a inače -1.

Binarno traženje prvo uspoređuje ulaznu vrijednost x sa korijenskim elementom polja v. Ako je x manje od (u ovom slučaju to je srednja vrijednost) korijenske vrijednosti, pretraživanje se prebacuje na donju polovinu tablice. Inače ostaje na gornjem dijelu. U svakom slučaju, slijedeći korak je usporedba x sa korijenskim elementom izabrane polovine. Proces dijeljenja se nastavlja sve dok se ne nađe tražena vrijednost ili dok stablo ne bude pregledano.

```
/* binsearch: nađi x u v[0] <= v[1] <= ... <= v[n-1] */
int binsearch(int x, int v[], int n){
      int low, high, mid;

      low=0;
      high=n-1;
      while(low<=high){
            mid=(low+high)/2;
            if(x<v[mid])
                  high=mid-1;
            else
                  if(x>v[mid])
                        low=mid+1;
                  else  /* pronađeno */
                        return mid;
      }
      return -1;  /* nije pronađeno */
}
```

Sve se svodi na odluku da li je x manje, veće ili (daj bože) jednako elementu v[mid].

Vježba 3-1. Naše binarno pretraživanje izvodi dvije provjere unutar petlje mada bi i jedan bio dovoljan (s većim brojem vanjskih provjera). Napišite verziju samo s jednom provjerom, te provjerite razliku u vremenu izvršavanja programa.

3.4 Switch

Naredba switch je višesmjerno grananje koje provjerava da li izraz odgovara nekoj od konstantnih cjelobrojnih vrijednosti, pa se na osnovu toga grana.

```
switch(izraz){
      case konstantni_izraz: naredba
      case konstantni_izraz: naredba
      default: naredbe
}
```

Svaka mogućnost ima jednu ili više cjelobrojnih konstanti ili izraza.Ako uvjet odgovara vrijednosti izraza, počinju se obavljati naredbe za takav slučaj. Izrazi svih slučajeva moraju biti različiti, a onaj označen s default (podrazumijevani) obavlja se ako nijedan uvjet nije zadovoljen. Ovaj slučaj nije obvezan; ako ne postoji onda se za gore spomenutu situaciju neće dogoditi baš ništa. Mogućnosti možemo nizati proizvoljno, jer to neće utjecati na izvršavanje programa (jer uvjeti moraju biti isključivi).

U Poglavlju 1 pisali smo program za zbrajanje pojavljivanja znamenki, razmaka i svih drugih znakova, rabeći konstrukciju if ... else if ... else. Sad ćemo napisati isti program, ali s pomoću funkcije switch.

```
/* zbraja znamenke, razmake i druge znakove */
# include <stdio.h>
main(){
      int c, i, nwhite, nother, ndigit[10];
      nwhite=nother=0;
      for(i=0;i<10;i++)
            ndigit[i]=0;
      while((c=getchar())!=EOF){
            switch (c){
                  case '0': case '1': case '2': case '3': case '4':
                  case '5': case '6': case '7': case '8': case '9':
                        ndigit[c-'0']++;
                        break;
                  case ' ':
```

```
                        case '\n':
                        case '\t':
                              nwhite++;
                              break;
                        default:
                              nother++;
                              break;
                  }
            }
            printf("Znamenke=");
            for(i=0;i<10;i++)
                  printf(" %d", ndigit[i]);
            printf(", razmaci=%d, ostalih=%d\n", nwhite, nother);
            return 0;
      }
```

Pomoću break naredbe obavlja se trenutni izlaz iz switch petlje. Kako su mogući slučajevi samo oznake početka niza naredbi (da bi program znao točku od koje će se program nastaviti) treba eksplicitno navesti mjesto gdje program treba izaći iz switch konstrukcije, a ako se ne navede izvršavanje se nastavlja naredbama iz slijedeće mogućnosti. Naredbe break i return predstavljaju jednostavan način izlaska iz switch petlje. Naredba break može poslužiti za izlaz iz drugih tipova petlji kao while, for itd.

Prolazak kroz opcije ima dobrih i loših strana. U pozitivnom smislu, ovaj princip dopušta većem broju slučajeva da budu povezani u jedan uvjet, kako je to u ovom primjeru sa znamenkama. Međutim, ova konstrukcija pretpostavlja da se svaka opcija završava s break kako bi se onemogućio "upad" u neželjena područja. Prolazi kroz slučajeve nisu strogo definirani i programi su izuzetno osjetljivi na modifikacije. S izuzećem višestrukih oznaka za jedan slučaj, prolasci bi trebali biti racionalno korišteni, uz obavezne komentare.

Forme radi, umetnite break iza zadnje opcije (u našem primjeru default) iako je to zapravo nepotrebno. Slijedeći put kada se neka druga opcija nađe na kraju, ovaj princip će se pokazati opravdanim.

Vježba 3-2. Napišite funkciju escape(s, t), koja pretvara znakove nove linije i tabulatore u vidljive escape nizove oblika \n i \t, dok kopira niz s u t. Pri tom upotrijebite switch petlju. Napišite i funkciju za suprotni smjer, pretvarajući escape nizove u stvarne znakove.

3.5 Petlje - while i for

Već smo se susretali sa while i for petljama. U

```
while(izraz)
      naredba
```

izračunava se izraz. Ako nije nula, naredba se izvršava, a izraz se ponovo računa. Ovaj ciklus se nastavlja sve dok izraz ne postane nula, a onda se izvršava dio programa nakon naredbe. U ovom slučaju, naredba podrazumijeva i blok naredbi oivičen vitičastim zagradama.

Naredba for

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

jednakovrijedna je sa

```
izraz1;
while(izraz2){
      naredba
      izraz3;
}
```

osim za slučaj kad se pojavljuje i continue, što ćemo vidjeti u dijelu 3.7.

Gramatički promatrano, tri komponente for petlje su izrazi. Najčešće, izraz1 i izraz3 su dodjele ili pozivi funkcija, a izraz2 je relacijski izraz. Nijedan od ovih sastavnih dijelova nije obvezatan, ali točka-zarezi

moraju egzistirati. Ako izraz1 i izraz3 nedostaju ne obavlja se ništa, a ako je to slučaj s izraz2, pretpostavka je da je on točan. Tako

```
for(;;){
      ...
}
```

predstavlja beskonačnu petlju koja se dade prekinuti pomoću internih break ili return.

Odluka o uporabi while ili for petlje prepuštena je programeru. Primjerice, u

```
while((c=getchar())==' '||c=='\n'||c=='\t')     /* preskoči razmake i
escape nizove */
      ;
```

nema nikakve inicijalizacije ni početnih parametara, pa je while prirodniji.

Petlja for dobiva na značenju tamo gdje nam treba jednostavna inicijalizacija i uvećavanje, jer ima izraze za kontrolu petlje lako uočljive na vrhu petlje. Jasnoće radi napišimo

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

koji predstavlja C idiom za rad s prvih n elemenata polja, što je analogno Fortranovoj DO petlji ili Pascalovoj for petlji. Analogija nije savršena jer se indeks i granica C for petlje daju mijenjati unutar petlje, a varijabilni indeks i zadržava svoju vrijednost ako se petlja završi iz bilo kojeg mogućeg razloga. Pošto su komponente for petlje neograničeni izrazi, for petlje nisu ograničene na aritmetičke progresije. K tomu, nije preporučljivo forsirati nezavisne izračune u inicijalizaciji ili uvećavanju for petlje, puno je pametnije da se inicijalizacija i uvećavanje rabe za kontrolu petlje, za što su i namijenjena.

Kao općenitiji primjer, ovdje imamo drugu verziju atoi funkcije koja se koristi za pretvorbu niza u njegov brojčani ekvivalent. Ova je verzija malo općenitija od one u Poglavlju 2 jer uzima u obzir moguće escape nizove, te znakove + i - (u Poglavlju 4 razmatrat ćemo atof koji obavlja pretvorbu, ali za brojeve s pokretnim zarezom).

Struktura programa dosta govori i o obliku ulaznih podataka

```
preskoči escape niz, ako ga ima,
uzmi predznak u obzir, ako postoji,
uzmi cjelobrojni dio i pretvori ga
```

Svaki korak obavlja svoj dio posla, te ostavlja "pročišćeni" oblik za slijedeću operaciju. Čitav proces se okončava na prvom znaku koji nije dio nekakvog broja.

```
#include <stdio.h>
/* atoi: pretvara s u cijeli broj, druga verzija */

int atoi(char s[]){
      int i, n, sign;

      for(i=0;isspace(s[i]);i++)    /* preskoči escape niz */
            ;
      sign=(s[i]=='-')?-1:1;
      if(s[i]=='+'||s[i]=='-')      /* preskoči predznak */
            i++;
      for(n=0;isdigit(s[i];i++)
            n=10*n+(s[i]-'0';
      return sign*n;
}
```

Standardna biblioteka osigurava potpuniju funkciju strtol za pretvorbu nizova cijelih brojeva tipa long (pogledajte dio 5 Dodatka B.

Prednosti centraliziranog načina kontrole petlje su uočljivije kad imamo slučaj gniježđenja petlji. Dolje napisana funkcija služi za sortiranje polja cijelih brojeva. Osnovna ideja algoritma, koji je 1959. otkrio D.L.Shell, nalazi se u činjenici da ako sortiramo udaljene elemente odmah, u kasnijim etapama imamo

manje posla (bubble sort premeće susjedne elemente). Interval između elemenata smanjuje se postepeno do 1 kada se sortiranje svede na sortiranje susjednih elemenata.

```
/* shellsort: sortira niz */
void shellsort(int v[], int n){
      int gap, i, j, temp;

      for(gap=n/2;gap>0;gap/=2)
            for(i=gap;i<n;i++)
                  for(j=i-gap;j>0&&v[j]>v[j+gap];j-=gap){
                        temp=v[j];
                        v[j]=v[j+gap];
                        v[j+gap]=temp;
                  }
}
```

Tu postoje umetnute petlje. Krajnja vanjska petlja kontrolira razmak između elemenata koji se uspoređuju, smanjujući ga od n/2 faktorom 2, dok god ne postane nula. Središnja petlja pristupa elementima, a unutrašnja uspoređuje svaki par elemenata koji je razdvojen pomoću gap i preokreće svaki koji nije u redu. Ako je gap smanjen do 1, moguće je da su svi elementi već poredani. Dade se primijetiti kako općenitost funkcije for osigurava da vanjska petlja ima istu formu kao druge, čak i kada nije riječ o čistoj aritmetičkoj progresiji (pri tom se misli na niz koji tvori varijabla petlje).

Zadnji C operator je zarez koji se najčešće koristi upravo u for naredbi. Nekoliko izraza odvojenih zarezom izračunava se slijeva nadesno, a tip i vrijednost rezultata su tip i vrijednost desnog operanda. Stoga je u for naredbu moguće smjestiti velik broj izraza, kako bi, npr., obradili dvije naredbe koje trebaju ići paralelno. Ovo ćemo ilustrirati funkcijom reverse(s), koja preokreće znakovni niz s.

```
/* reverse: obrtanje znakovnog niza */
#include <stdio.h>

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

Zarezi koji razdvajaju argumente funkcije, varijable u deklaracijama i sl. nisu operatori, te ne garantiraju izračunavanje slijeva nadesno.

Zareze kao operatore treba koristiti grupno. Najdjelotvornije korištenje jest u konstrukcijama koje utječu jedna na drugu, kao u for petljama u funkciji reverse, te u makroima, gdje računanje u više koraka treba biti jedan izraz. Isto tako, izraz sa zarezom može su zgodno iskoristiti za manipulaciju podacima, a posebice tamo gdje je bitno da sve bude na jednom mjestu tj., u jednom izrazu. Tako se petlja iz reverse funkcije dade napisati kao

```
for(i=0, j=strlen(s)-1;i<j;i++, j--)
      c=s[i], s[i]=s[j], s[j]=c;
```

Vježba 3-3. Napišite funkciju expand(s1, s2) koja proširuje skraćenice intervala oblika "a-z" iz znakovnog niza s1 u cijele intervale abc...xyz u znakovni niz s2. Rabite i velika i mala slova i brojeve, a program treba prepoznati i ulazne konstrukcije tipa a-b-c, a-z0-9 i -a-z. Osigurajte da se - na početku ili na kraju tretira kao slovo.

3.6 Petlja do – while

Kao što smo razmotrili u Poglavlju 1, while i for petlje vrše provjeru uvjeta na vrhu petlje. Tomu nasuprot, do - while petlja takovu provjeru čini na dnu petlje, nakon prolaska kroz tijelo petlje, što znači da se ono izvrši bar jednom.

Sintaksa do - while petlje je

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

Naredba se izvršava, nakon čega se računa izraz. U slučaju da je istinit, naredba se nanovo izvršava. Kad izraz postane neistinit, petlja se završava. Za usporedbu, ova petlja je ekvivalentna repeat - until petlji u programskom jeziku Pascal.

Iskustveno znamo da se ova petlja koristi manje negoli while ili for. Unatoč tomu, zna biti korisna, kao u funkciji itoa, koja pretvara broj u znakovni niz (radi obrnut proces od atoi). Rad nije tako jednostavan kako se ispočetka čini, jer se znamenke umeću u znakovni niz pogrešnim redom (znamenka najmanje vrijednosti dolazi na desnu stranu), pa se niz na kraju balade mora obrnuti.

```
/* itoa: pretvara n u znakovni niz s */
void itoa(int n, char s[]){
      int i, sign;

      if((sign=0)<0)
            n=-n;
      i=0;
      do{
            s[i++]=n%10+'0';
      } while((n/=10)>0);

      if(sign<0)
            s[i++]='=';
      s[i]='\0';
      reverse(s);
}
```

Intervencija sa do - while petljom itekako je potrebna, jer čak i ako je n=0 barem jedan znak treba biti upisan u polje s. Isto tako, rabimo vitičaste zagrade oko jedne naredbe koja je ujedno i tijelo petlje (mada nismo to obavezni činiti) kako i onaj čitatelj koji ovu knjigu koristi kao podsjetnik ne bi pogrešno protumačio dio while kao početak while petlje.

Vježba 3-4. U dvokomplementnom predstavljanju broja, naša inačica itoa ne barata sa najvećim negativnim brojem. Pojasnite zašto. Prilagodite program tako da ispisuje tu vrijednost korektno, bez obzira na računalo na kojemu radi.

Vježba 3-5. Napišite funkciju itob(n, s, b) koja pretvara cjelobrojni dekadski broj n u broj s bazom b, te njegov znakovni prikaz u nizu s. Primjerice, itob(n, s, 16) formatira n kao heksadecimalni cijeli broj u s.

Vježba 3-6. Napišite verziju itoa koja prima treći argument minimalne širine polja. Pretvoreni broj mora biti desno poravnat ako je polje veće od broja.

3.7 Break i continue

Katkad je uputno imati mogućnost izlaska iz petlje, ali ne preko provjere uvjeta bilo na vrhu bilo na dnu petlje. Naredba break osigurava "naprasni" izlaz iz bilo koje petlje. Ona osigurava programu trenutno napuštanje petlje u kojoj se vrti.

Slijedeća funkcija, trim, uklanja razmake, tabulatore i znakove nove linije sa kraja niza, pri tom rabeći break kako bi izašla iz petlje u slučaju nailaska na krajnji desni znak koji nije razmak, tabulator ili nova linija.

```
/* trim: uklanja razmake, tabulatore i znakove nove linije */
int trim(char s[]){
      int n;
      for(n=strlen(s)-1;n>=0;n--)
            if(s[n]!=' '&&s[n]!='\t'&&s[n]!='\n')
```

```
                    break;
        s[n+1]='\0';
        return n;
}
```

Funkcija strlen vraća dužinu niza. Petlja for započinje čitanje s kraja niza, tražeći znak koji nije razmak, tabulator ili znak nove linije. Ona se prekida čim pronađe takav znak ili n dođe do nule (drugim riječima kad se pretraži cijeli niz).

Naredba continue usko je povezana s break, ali se rabi u znatnoj mjeri rjeđe. Njen je zadatak da otpočne slijedeću iteraciju petlje u kojoj se program vrti bilo da se radi o for, while ili do - while petlji. Kod while i do - while petlji to znači da se trenutno prelazi na provjeru uvjeta, a u for konstrukciji na slijedeći stupanj inkrementacije. Naredba continue koristi se samo u petljama, ali ne i kod switch naredbe (kao, recimo, break). Naredba continue unutar switch naredbe prouzročit će prelazak u slijedeću petlju.

Kao primjer promotrimo dio programa koji predaje samo pozitivne elemente u polje a, a negativne izostavlja.

```
for(i=0;i<n;i++){
        if(a[i]<0)  /* preskoči negativne elemente */
                continue;
        ...   /* obradi pozitivne elemente */
}
```

Naredba continue koristi se često kada je dio petlje koji slijedi bespotreban, a složen. Tada je postavljanje dodatnih uvjeta težak i za programera mukotrpan posao, a sam program se bespotrebno usložnjava.

3.8 Goto i labele

C ima, mada ne baš preporučljivu i upotrebljivu, goto naredbu i labele na koje se grana. Formalno, goto naredba nikad nije nezaobilazna. Možda izraz nije najbolje pogođen, ali teorija kaže da sve što goto naredbom možemo postići dade se i drugačije (i bolje). Ovaj izdanak zaostalih programskih jezika (assembly, basic i sl.) ne preporučujemo nikomu.

Ipak, ima nekoliko situacija koje čine goto ipak primjenjivim. Najčešće se koristi u nekakvim višeslojnim strukturama za prestanak odvijanja procesa koji se vrti, kakav je trenutni izlaz iz dvije ili više petlji odjednom. Naredba break djeluje samo na unutrašnju petlju pa

```
for( ... )
        for( ... ){
                ...
                if(disaster)
                        goto greska;
        }
        ...
greska:
        obrisi poruke
```

Ovakav način rada je prihvatljiv, ako program za rad s greškama (ili možda grješkama?!) nije jednostavan. Tada samo naselite goto konstrukcije na mjesta gdje se greške očekuju i ... čekate. Šalu nastranu, ali i to je na određen način zgodno.

Labele su istog oblika kao i ime varijable, a popraćene su dvotočkom. Ona može pripadati bilo kojoj naredbi koja čini funkciju goto naredbe. Područje oznake je cijela funkcija.

Primjera radi, razmotrimo problem određivanja da li dva polja a i b imaju jedan zajednički element. Jedna od mogućnosti jest

```
for(i=0;i<n;i++)
        for(j=0;j<m;j++)
                if(a[i]==b[j])
                        goto nadjeno;
/* nije pronađen nijedan zajednički element */
...
```

```
nadjeno:
/* pronađen je a[i]==b[i] */
...
```

Ovakav program dade se, jasno, načiniti bez upotrebe goto naredbe, ali trebalo bi uključiti neke dodatne provjere i varijable. Tako, pretraživanje polja postaje

```
nadjeno=0;
for(i=0;i<n&&!nadjeno;i++)
      for(j=0;j<m&&!nadjeno;j++)
            if(a[i]==b[j])
                  nadjeno=1;
if(nadjeno)
/* pronađen jedan: a[i-1]==b[j-1] */
...
else
/* nije pronađen nijedan zajednički element */
```

S nekim izuzecima koji se već navedeni, program s goto naredbama je općenito teži i nerazumljiviji nego onaj bez njih. Bez ikakvih predrasuda, mislimo da bi ovakve konstrukcije trebalo izbjegavati.

POGLAVLJE 4: FUNKCIJE I PROGRAMSKE STRUKTURE

U pokušaju definiranja funkcija možemo kazati da one pomažu u raščlanjivanju programa na manje programske cjeline, koje su primjenjive i za druge, možebitno, kasnije pisane programe. Neke funkcije posjeduju skrivene operacije (skrivene od programera) koje simplificiraju rad i obavljanje promjena.

Osnovna zamisao programskog jezika C jest u efikasnoj i jednostavnoj primjeni funkcija. C programi se obično sastoje od puno malih funkcija, a ne od nekoliko većih. Program se može bazirati na jednoj ili nekoliko izvornih datoteka. Te datoteke se daju prevoditi odvojeno, a unesene u memoriju skupa sa prethodno prevedenim funkcijama iz biblioteke. Ovdje nećemo zalaziti dublje u ovu problematiku, jer se detalji mijenjaju od sistema do sistema.

Deklaracija i definicija funkcije su pojmovi na kojima je ANSI standard napravio najveće promjene. Kako smo već zamijetili u Poglavlju 1, sad imamo mogućnost deklariranja tipova argumenata nakon deklaracije funkcije. Izmijenjena je sintaksa definicije funkcije, pa se deklaracije i definicije međusobno prepliću. To prevoditelju otvara mogućnost otkrivanja puno više grešaka negoli ranije. Štoviše, ako su argumenti pravilno deklarirani, automatski se smanjuje odgovarajući broj tipova.

Standard pojašnjava pravila koja se odnose na imena; konkretno, on zahtjeva postojanje samo jedne definicije vanjskog objekta. Inicijalizacija je općenitija jer se polja i strukture sada mogu automatski inicijalizirati.

C preprocesor također je proširen. Olakšice za korištenje novog preprocesora imaju kompletniji skup uvjeta prevođenja, mogućnost da makroargumenti kreiraju znakovne nizove i bolju kontrolu procesa makroproširenja.

4.1 Osnovni pojmovi o funkcijama

Za početak, kreirajmo program koji će ispisivati svaku ulaznu liniju s određenim uzorkom (što zapravo radi UNIX naredba grep). Primjerice, tražit ćemo linije s uzorkom 'mal'

```
Gle malu voćku poslije kiše,
puna je kapi, pa se njiše.
Al' neće više!
Jer kad kiše malo stanu,
i ona će u pilanu.
```

... pa će izlaz dati ...

```
Gle malu voćku poslije kiše,
Jer kad kiše malo stanu,
```

Program je razdijeljen na tri dijela

```
while(postoji slijedeća linija)
      if(uzorak se nalazi u liniji)
            ispiši je
```

Mada je moguće sve ovo izvesti u okviru main funkcije, bolji i (glede stjecanja programerskih principa) pogodniji način je upotreba funkcija za svaki dio programa. Općenito je lakše raditi i analizirati manje programske cjeline, što zbog obujma posla, što zbog činjenice da se nevažni detalji mogu "sakriti" po funkcijama, pa eventualnim proučavanjem programa ne dolazimo s njima u dodir. Nadalje, prednost ovakvog pristupa je u tome što se ovi programi mogu prenositi na druge programe.

Umjesto funkcije "postoji slijedeća linija" možemo rabiti getline, funkciju iz Poglavlja 1, a "ispiši je" je printf iz standardne biblioteke. To praktično znači da moramo napisati rutinu koja će donijeti odluku da li se u liniji pojavljuje uzorak.

Taj problem možemo riješiti funkcijom strindex(s, t) koja vraća poziciju ili indeks u nizu s gdje niz t počinje, a -1 ako s ne sadrži t. Kako polja u programskom jeziku C počinju indeksom 0, to će indeksi biti 0 ili pozitivni, pa se vrijednost -1 može rabiti za dojavu greške. Kad nam kasnije bude zatrebalo bolje pretraživanje nizova izmjene će biti lokalizirane na funkciji strindex dok drugi dijelovi programa mogu ostati netaknuti (spomenimo da standardna biblioteka ima funkciju strstr koja čini sve što i strindex, ali vraća pokazivač umjesto indeksa).

Možemo odmah pogledati program. Sve je iskomentirano, pa se lako uoči kako se dijelovi uklapaju. Zasad, traženi uzorak je znakovni niz, mada to nije općeniti mehanizam. Vratit ćemo se na trenutak u raspravu o inicijalizaciji znakovnog polja, a u Poglavlju 5 pokazat ćemo kako napraviti uzorak kao parametar koji se postavlja za vrijeme odvijanja programa. Dakle, postoji i malo drugačija inačica getline funkcije, pa bi možda bilo korisno usporediti je s onom u Poglavlju 1.

```
#include <stdio.h>

#define     MAXLINE     1000        /* maksimalna dužina ulazne linije */

int getline(char line[], int max);
int strindeks(char source[], char searchfor[]);
char pattern[]="mal";         /* traženi uzorak */
main(){
      char line[MAXLINE];
      int found=0;

      while(getline(line, MAXLINE)>0)
            if(strindex(line, pattern)>=0){
                  printf("%s", line);
                  found++;
            }
      return found;
}

/* getline: pohrani liniju u s, vrati dužinu */
int getline(char s[], int lim){
      int c, i;
      i=0;
      while(--lim>0&&(c=getchar())!=EOF&&c!='\n')
            s[i++]=c;
      if(c=='\n')
            s[i++]=c;
      s[i]='\0';
      return i;
}

/* strindex: vrati indeks od t u s, -1 ako ga nema */
int strindex(char s[], char t[]){
      int i, j, k;
      for(i=0;s[i]!='\0';i++){
            for(j=i;k=0;t[k]!='\0'&&s[j]==t[k];j++, k++)
                  ;
            if(k>0&&t[k]=='\0')
                  return i;
      }
      return -1;
}
```

Svaka definicija funkcije je oblika

```
tip_informacije_koja_se_vraća ime_funkcije(deklaracije argumenata){
      deklaracije i naredbe
}
```

Ovi dijelovi nisu obvezni. Funkcija može izgledati i ovako

```
primjer(){
}
```

Takva funkcija ne služi ničemu, osim što može potrošiti procesorsko vrijeme. U nedostatku tipa povratne informacije, automatski se vraća int.

Program je po definiciji samo skup definicija varijabli i funkcija. Komunikacija među funkcijama obavlja se preko argumenata i vrijednosti, koje funkcije vraćaju, te preko vanjskih (eksternih) varijabli. Funkcije se mogu umetati u izvornu datoteku po bilo kojem redu, a izvorni program se može raščlaniti na više datoteka, do te mjere da svaka funkcija bude jedna datoteka.

Naredba return jest mehanizam za vraćanje vrijednosti od pozvane funkcije do mjesta poziva. Nakon return može slijediti bilo koji izraz.

```
return izraz;
```

Izraz će biti pretvoren u tip povratne varijable ako je takova operacija potrebna. Male zagrade katkad se stavljaju oko izraza, ali su neobavezne.

Pozvana funkcija ne mora voditi brigu o vrijednosti koja se vraća. Dapače, nakon return naredbe izraz nije obvezan, a tada funkcija ne vraća nikakvu vrijednost. Tok programa nastavlja se tamo odakle je funkcija pozvana nailaskom na desnu vitičastu zagradu. Ako funkcija katkad vraća vrijednost, a katkad ne, to je redovito znak postojanja problema. Svakako, ako funkcija ne uspije vratiti informaciju, njena će vrijednost sigurno biti nekontrolirana.

Program za traženje uzoraka vraća status od funkcije main, tj., broj pronađenih podudarnosti. Tu vrijednost može rabiti okruženje koje je pozvalo program (bilo da se radi o UNIX ljusci ili nekom drugom programu).

Mehanizam prevođenja i učitavanja C programa koji počiva na višestrukim izvornim datotekama varira od sistema do sistema. Kod UNIX sistema, npr., naredba cc, spomenuta ranije, obavlja taj posao. Zamislite da su tri funkcije pohranjene u tri datoteke, s imenima main.c, getline.c i strindex.c. Tada naredba

```
cc main.c getline.c strindex.c
```

predvodi datoteke, postavljajući prevedeni objektni kod u datoteke s ekstenzijama *.o respektivno (dakle, main.o, getline.o i strindex.o), a zatim ih povezuje u izvršnu datoteku s imenom a.out. U slučaju greške, primjerice u datoteci main.c, datoteka se može sama ponovo prevesti, te rezultat ubaciti u već prethodno prevedene objektne datoteke naredbom

```
cc main.c getline.o strindex.o
```

Naredba cc rabi ekstenzije *.c i *.o radi razdvajanja izvornih od objektnih datoteka.

Vježba 4-1. Napišite funkciju strrindex(s, t) koja vraća mjesto krajnje desnog pojavljivanja t u s, a -1 ako ga nema.

4.2 Funkcije koje vraćaju necjelobrojne vrijednosti

Dosada su naši primjeri funkcija vraćali ili nikakvu vrijednost ili čisti int. Što se zbiva ako funkcija mora vraćati neki drugi tip? Mnoge matematičke funkcije, primjerice sqrt, sin i cos, vraćaju double; druge specijalizirane funkcije vraćaju druge tipove. Radi ilustracije kako treba postupati s takvim funkcijama, napišimo funkciju atof(s), koja pretvara niz s u njezin ekvivalent realnog broja dvostruke točnosti. Funkcija atof je proširena inačica funkcije atoi koja je bila obrađena u Poglavljima 2 i 3. Ona manipulira opcijskim predznakom i decimalnom točkom, te postojanjem ili nepostojanjem cjelobrojnog ili dijela iza decimalne točke. Ova inačica nije bogzna kako dobra rutina ulazne pretvorbe jer bi tada zahtijevala puno više prostora. Standardna biblioteka sadrži atof deklariranu u zaglavlju <math.h>.

Najprije, sama atof funkcija mora deklarirati tip vrijednosti koju vraća zato jer se ne radi o cijelom broju. Ime tipa prethodi imenu funkcije:

```
#include <ctype.h>
/* atof: pretvara niz s u broj tipa double */
double atof (char s[]){
      double val, power;
      int i, sign;

      for(i=0; isspace(s[i]); i++); /* preskače escape sekvence */
            ;
```

```
        sign=(s[i]=='-')?-1;1;
        if(s[i]=='+'||s[i]=='-')
              i++;
        for(val=0.0; isdigit(s[i]); i++)
              val=10.0*val+(s[i]-'0');
        if(s[i]=='.')
              i++;
        for(power=1.0; isdigit(s[i]); i++){
              val=10.0*val+(s[i]-'0');
              power*=10.0;
        }
}
```

Nadalje, a isto tako bitno, pozivna rutina mora znati da atof vraća vrijednost koja nije int. Jedan od načina da se to osigura jest jasna deklaracija atof funkcije i u pozivnoj rutini. Deklaracija je primjerom prikazana u ovom jednostavnom potprogramu. On očitava jedan broj po liniji, kome opcijski prethodi predznak, i koji ih zbraja, ispisujući trenutni zbroj nakon svakog ulaza

```
#include <stdio.h>
#define     MAXLINE     100
/* jednostavni kalkulator */
main(){
      double sum, atof(char[]);
      char line[MAXLINE];
      int getline(char line[], int max);

      sum=0;
      while(getline(line, MAXLINE)>0)
            printf("\t%g\n", sum+=atof(line));
      return 0;
}
```

Deklaracija

```
double sum, atof(char[]);
```

kaže da je sum double varijabla, a da je atof funkcija koja uzima jedan argument tipa char[], te vraća double.

Funkcija atof mora biti deklarirana i definirana na odgovarajući način. Ako su atof i njen poziv u main funkciji neusklađeni po pitanju tipa u istoj izvornoj datoteci, grešku će otkriti prevoditelj. Ali ako je (što je najvjerojatnije), atof bila prevedena posebno, ta nepodudarnost neće biti otkrivena, pa će atof vratiti double, koju funkcija main tretira kao int i imat ćemo rezultate koji nisu točni.

Činjenica koju smo naveli, kako deklaracije moraju odgovarati definicijama, možda djeluje iznenađujuće. Do neusklađenosti dolazi zato što se, bez navedenog prototipa, funkcija implicitno deklarira svojim prvim pojavljivanjem u izrazu, kakav je

```
sum+=atof(line);
```

Ako se nedeklarirano ime pojavi u izrazu, a nakon njega lijeva zagrada, podrazumijeva se da je riječ o funkciji, a inicijalno se predmnijeva da funkcija vraća int, dok se o njenim argumentima ne zna ništa. Štoviše, ako deklaracija funkcije nema argumente kao u

```
double atof();
```

to znači da se ništa ne može pretpostaviti glede argumenta atof funkcije; sve su provjere parametara isključene. Prazna lista argumenata dozvoljava prevođenje starijih C programa na novijim prevoditeljima. Unatoč tomu, nije preporučljiva njihova uporaba u novim programa. Ako funkcija uzima argumente, deklarirajte ih, inače koristite void.

Kada je funkcija atof pravilno deklarirana, možemo napisati atoi (koja pretvara niz znakova u int) na slijedeći način:

```
/* atoi: pretvara niz s u cijeli broj koristeći atof */
int atoi(char s[]){
      double atof(char[]);
      return (int) atof(s);

}
```

Obratite pažnju na strukturu deklaracija i return naredbu. Vrijednost izraza u

```
return izraz;
```

se pretvara u tip funkcije prije negoli se vrijednost vrati. Zbog toga se vrijednost funkcije atof koja je inače tipa double pretvara automatski u int kad se pojavi u naredbi return jer funkcija atoi vraća int. Ova operacija može podatak učiniti neupotrebljivim, na što mnogi prevoditelji upozoravaju. No, u našem slučaju, operacija je prijeko potrebna.

Vježba 4-2. Proširite atof tako da radi sa znanstvenom notacijom oblika:

```
123.45e-6
```

u kojoj broj sa pokretnim zarezom prate e ili E i opcijski označen eksponent.

4.3 Vanjske varijable

C program sastavljen je od vanjskih objekata, kako varijabli tako i funkcija. Pridjev "vanjski" stoji nasuprot pridjeva "nutarnji" koji opisuje argumente i varijable definirane unutar funkcija. Vanjske varijable definirane su izvan funkcija kako bi se mogle koristiti unutar više funkcija. Uz to, vanjske varijable i funkcije imaju ugodnu osobinu da sve njihove reference istog imena, čak i funkcija koje su posebno prevedene predstavljaju reference na istu stvar (što standard naziva vanjskim povezivanjem). U ovom kontekstu, vanjske varijable su analogne Fortranovim COMMON blokovima ili varijablama u najudaljenijem bloku u Pascalu. Kasnije ćemo vidjeti kako se definiraju vanjske varijable i funkcije koje su vidljive samo unutar jedne izvorne datoteke.

Zbog toga što su lokalne varijable dohvatljive, one osiguravaju alternativu za argumente funkcije i vraćene vrijednosti za razmjenu podataka među funkcijama. Svaka funkcija može pristupiti vanjskoj varijabli preko imena, ako je to ime na neki način već deklarirano.

Ako su veliki broj varijabli dijeli među funkcijama, vanjske varijable bolje rješenje od dugačkih lista argumenata. Međutim, kako je istaknuto u Poglavlju 1, u takvim razmatranjima treba biti oprezan, jer bi se to moglo loše odraziti na strukturu programa i dovesti do programa s prevelikim brojem podataka dijeljenim među funkcijama.

Vanjske varijable su isto tako korisne zbog većeg opsega, a i zbog "dužeg vijeka trajanja". Automatske varijable su unutrašnje za funkciju; one nastaju na početku funkciju, a nestaju kad se funkcija završi. Vanjske varijable su, pak, postojane, tako da zadržavaju vrijednosti od jednog poziva funkcije do drugog. Stoga, ako dvije funkcije imaju zajedničke podatke, a jedna ne poziva drugu, onda je najpogodnije da se zajednički podaci čuvaju u vanjskim varijablama, a ne da se unose i izbacuju pomoću argumenata.

Provjerimo ovo pomoću jednog općenitijeg primjera. Problem je napisati program za kalkulator koji određuje operatore +, -, * i /. Zbog lakše implementacije, kalkulator će koristiti inverznu poljsku notaciju (ova notacija koristi se u HP kalkulatorima i u jezicima kao što su Forth i Postscript).

U inverznoj poljskoj notaciji, svaki operator slijedi svoje operande; izraz tipa

```
(1-2)*(4+5)
```

unosi se kao

```
12-45+*
```

Male zagrade su nepotrebne. Notacija je jednoznačna dok znamo koliko operanda očekuje svaki operator.

Implementacija je više nego jednostavna. Svaki se operand se stavlja na stog, a nailaskom operatora operandi (za binarne operatore radi se o dva operanda) se izbacuju, obrađuju operatorom, te se rezultat vraća na stog. U gornjem primjeru 1 i 2 idu na stog, a potom se zamjenjuju njihovom razlikom, -1.

Slijedeći na stog idu 4 i 5 da bi bili zamijenjeni svojim zbrojem, 9. Produkt od -1 i 9, dakle -9, mijenja ih na stogu. Vrijednost na vrhu stoga se izbacuje i ispisuje nailaskom na kraj ulazne linije.

Struktura programa je tako petlja koja vrši odgovarajuću operaciju na svakom operatoru i operandu, redom kojim nailaze:

```
while(slijedeći operator ili operand nije znak kraja datoteke)
      if(broj)
            stavi na stog
      else
            if(operator)
                  digni operande sa stoga
                  obavi obradu
                  stavi rezultat na stog
      else
            if(novi red)
                  digni rezultat sa stoga i ispiši ga
      else
            greška
```

Operacije stavljanja i dizanja sa stoga su jednostavne, ali kad joj se doda otkrivanje i otklanjanje greške, one postaju toliko duge da ih bolje smjestiti u posebne funkcije, nego ponavljati kod kroz program. Isto tako, treba imati posebnu funkciju za dohvaćanje slijedećeg ulaznog operatora ili operanda.

Osnovno pitanje pri kreiranju, koje još nismo razmatrali, jest gdje smjestiti stog i rutine koje mu pristupaju direktno. Jedna od opcija je zadržati ga u main funkciji, a da se stog i njegova tekuća pozicija prenose do rutina stavljanja na i dizanja sa stoga. No, main funkcija ne mora znati za varijable koje manipuliraju stogom. Ona vrši samo operacije stavljanja i dizanja. Stoga ćemo stog i informacije vezane za njega u vanjskim varijablama, koje su dohvatljive push i pop funkcijama, ali ne i main funkciji.

Pretvaranje ovog principijelnog rješenja u izvorni kod relativno je lako. Zamislimo li program samo s jednom izvornom datotekom, tada će to izgledati ovako:

```
#includes
#defines
deklaracije funkcija za main
main(){ ... }
vanjske varijable za push i pop
void push (double f){ ... }
double pop(void){ ... }
int getop(char s[]){ ... }
rutine koje se pozivaju pomoću getop
```

Kasnije ćemo razmotriti kako se ovo dijeli na dvije ili više izvornih datoteka. Funkcija main predstavlja petlju koja sadrži veliku switch naredbu za tip operatora ili operanda. Ova je primjena switch naredbe znakovitija od one koja je prikazana u dijelu 3.4.

```
#include <stdio.h>
#include <math.h> /* zbog atof funkcije */
#define     MAXOP 100   /* maksimalna veličina operanda ili operatora */
#define     NUMBER      '0'   /* znak da je upisan broj */

int getop(char[]);
void push(double);
double pop(void);

/* Kalkulator s inverznom poljskom notacijom */
main(){
      int type;
      double op2;
      char s[MAXOP];
      while((type=getop(s))!=EOF){
            switch(type){
                  case NUMBER:
```

```
                        push(atof(s));
                        break;
                case '+':
                        push(pop()+pop());
                        break;
                case '*':
                        push(pop()*pop());
                        break;
                case '-':
                        op2=pop();
                        push(pop()-op2);
                        break;
                case '/':
                        op2-pop();
                        if(op2!=0.0)
                                push(pop()/op2);
                        else
                                printf("greška : dijeljenje s nulom\n");
                        break;
                case '\n':
                        printf("|t%.8g\n", pop());
                        break();
                default:
                        printf("greska : nepoznata operacija %s\n", s);
                        break;
            }
        }
        return 0;
}
```

Kako se na operatore + i * primjenjuje pravilo komutativnosti, nije bitan slijed unošenja operanda, dok se kod - i / lijevi i desni operand moraju znati. Za

```
push(pop()-pop());      /* pogrešan način */
```

nije definirano kojim će se redom ostvariti dva poziva pop funkcije. Stoga je obavezno prebaciti prvu vrijednost u privremenu varijablu, kao što smo uradili u našoj main funkciji.

```
#define     MAXVAL      100   /* maksimalna dubina stoga */

int sp=0;   /* slijedeća slobodna pozicija na stogu */
double val[MAXVAL];     /* vrijednost stoga */

/* push : potisni f na stog */
void push(double f){
        if(sp<MAXVAL)
                val[sp++]=f;
        else
                printf("greska : stog je pun, operacija prekinuta %g\n", f);
}

/* pop : dizanje vrijednosti sa stoga */
double pop(void){
        if(sp>0)
                return val[--sp];
        else{
                printf("greska : stog je prazan\n");
                return 0.0;
        }
}
```

Varijabla je vanjska ako je definirana izvan bilo koje funkcije. Stoga su stog i pokazivač stoga (koji moraju biti zajednički za push i pop funkciju) definirani izvan ovih funkcija. Ali sama main funkcija ne zna za stog i njegovu poziciju, pa predstavljanje može biti skriveno.

Razmotrimo sada implementaciju getop funkcije, koja uvodi slijedeći operator ili operand. Treba samo ignorirati razmake i tabulatore, vratiti znakove koji nisu brojevi ili decimalna točka, te tomu nasuprot, formirati niz brojeva (i s decimalnim zarezom) i vratiti vrijednost NUMBER kao znak da je učitan broj.

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getop : uzmi slijedeći operator ili operand */
int getop(char s[]){
      int i,c;
      while((s[0]=c=getch())==' '||c=='\t')
            ;
      s[1]='\';
      if(!isdigit(c)&&c!='.')
            return c;   /* nije broj */
      i=0;
      if(isdigit(c))    /* uzmi cjelobrojni dio */
            while(isdigit(s[++i]=c=getch()))
                  ;
      s[i]='\0';
      if(c!=EOF)
            ungetch(c);
      return NUMBER;
}
```

Što predstavljaju getch i ungetch funkcije? Često se događa da program ne zna da li je učitan dovoljan broj znakova za pravilan rad. Primjerice, pri učitavanju brojeva, sve do pojave znaka koji nije znamenka, broj nije potpun. No, tada je program učitao znak koji je višak.

Rješenje ovog problema bilo bi neočitavanje znaka. Tada bi se za svaki učitani znak previše, on vratio na ulaz pa bi ga daljnji tijek programa ignorirao. To je, nasreću, lako izvedivo, pomoću dvije funkcije - getch i ungetch. Funkcija getch učitava slijedeći znak, a funkcija ungetch pamti znakove koji su vraćeni na ulaz, kako bi ih kasniji pozivi getch funkcije mogli vratiti prije očitavanja novog ulaza. Isto je tako jednostavno objasniti kako one funkcioniraju zajedno. Funkcija ungetch postavlja učitani znak u zajednički pretinac - polje znakova. Funkcija getch čita iz pretinca, ako u njemu ima znakova, a poziva funkciju getchar ukoliko je pretinac prazan. Također mora postojati jedna indeksna varijabla koja pamti poziciju tekućeg znaka u pretincu.

Pošto su pretinac i indeks zajednički za funkcije getch i ungetch, a kako moraju zadržavati svoje vrijednosti između dva poziva, oni moraju biti vanjski za obje rutine. Dakle, možemo napisati funkcije getch, ungetch i njihove zajedničke varijable kao:

```
#define     BUFSIZE     100

char buf[BUFSIZE];      /* pretinac za ungetch */
int bufp=0; /* slijedeće slobodno mjesto u pretincu */

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

void ungetch(int c){    /* gurni znak nazad na ulaz */
      if(bufp>=BUFSIZE)
            printf("ungetch: previše znakova\n");
      else
            buf[bufp++]=c;
}
```

Standardna biblioteka ima funkciju ungetch, koja osigurava jedan potisnuti znak, ali to ćemo razmotriti u Poglavlju 7. Za potiskivanje smo koristili čitavo polje, a ne samo jedan znak, da bismo pokazali kako tom problemu treba pristupiti uopćeno.

Vježba 4-3. Kad je poznata osnovna struktura, nije teško proširiti funkcije kalkulatora. Dodajte funkcije modula (%) i mogućnost rada s negativnim brojevima.

Vježba 4-4. Dodajte naredbe za ispis elemenata koji su na vrhu stoga bez izbacivanja, tek radi kopiranja, a zatim zamijenite dva elementa s vrha. Dodajte naredbu za čišćenje stoga.

Vježba 4-5. Dodajte mogućnost pristupa funkcijama biblioteke kao što su sin, exp i pow (proučite zaglavlje <math.h> u Dodatku B.

Vježba 4-6. Dodajte naredbe za manipulaciju varijablama (nije problem odrediti 26 varijabli, čija se imena sastoje od samo jednog slova). Dodajte varijablu vrijednosti koja je posljednja ispisana.

Vježba 4-7. Upišite rutinu ungets(s) koja će potisnuti cijeli niz nazad na ulaz. Da li ungets treba znati za buf i bufp, ili treba koristiti ungetch?

Vježba 4-8. Zamislite da neće biti potrebe potiskivanja više od jednog znaka. Sukladno tomu, modificirajte getch i ungetch.

Vježba 4-9. Naše funkcije getch i ungetch ne manipuliraju pravilno s potisnutom EOF konstantom. Odlučite kako bi trebale reagirati funkcije ako se EOF potisne nazad, te svoju zamisao provedite u djelo.

Vježba 4-10. Kako smo funkciju getline koristili za očitavanje cijele ulazne linije, to su nam funkcije getch i ungetch nepotrebne. Prepravite kalkulatora kako bi mogao raditi na ovaj način.

4.4. Pravila opsega

Nije obavezno da se funkcije i vanjske varijable koje čine C program prevode istovremeno. Izvorni tekst može se smjestiti u nekoliko datoteka, prethodno prevedene rutine mogu se učitavati iz biblioteka. Evo nekoliko, za nas, važnijih pitanja:

- Kako napisati deklaracije varijabli da one budu pravilno deklarirane za vrijeme prevođenja?
- Kako rasporediti deklaracije da svi dijelovi budu pravilno povezani pri učitavanju programa?
- Kako organizirati deklaracije radi stvaranja samo jedne kopije?
- Kako inicijalizirati vanjske varijable?

Razmotrimo ova pitanja radi prerade programa za kalkulator i raščlanjivanja na nekoliko datoteka. U praktičnom smislu, program za kalkulator je suviše mali da bi se dijelio, no može poslužiti kao ilustracija za eventualne veće projekte.

Opseg imena je dio programa u okviru kojeg se ime može koristiti. Za automatsku varijablu koja je deklarirana na početku funkcije, opseg predstavlja funkciju u kojoj se ona deklarira. Lokalne varijable sa istim imenom u različitim funkcijama nisu u vezi. Isto se odnosi i na parametre funkcije koji su, zapravo, lokalne varijable.

Opseg vanjske varijable ili funkcije počinje od mjesta gdje se ona deklarira, a završava na kraju datoteke koja se prevodi. Primjerice, ako su funkcije i varijable main, sp, val, push i pop definirane u jednoj datoteci po prikazanom redoslijedu

```
main(){ ... }
int sp=0;
double val[MAXVAL];
void push(double f){ ... }
double pop(void){ ... }
```

tada se varijable sp i val mogu koristiti jednostavno u funkcijama push i pop odmah nakon dodjele imena. Daljnje deklaracije nisu potrebne, ali zato njihova imena nisu uočljiva u funkciji main kao ni u funkcijama push i pop.

S druge strane, ako vanjsku varijablu treba pridružiti prije definiranja ili ako je definirana u nekoj drugoj izvornoj datoteci, tada je extern deklaracija obavezna.

Važno je napraviti razliku između deklaracije vanjske varijable i njezine definicije. Deklaracijom pokazujemo osobine varijable (obično njezin tip), dok definicija obavlja dodjelu memorije. Ako se linije

```
int sp;
double val[MAXVAL];
```

pojave izvan neke funkcije, one definiraju vanjske varijable sp i val, vrše dodjelu memorije, a također služe i kao deklaracija za ostali dio izvorne datoteke. S druge strane, linije

```
extern int sp;
extern double val[];
```

deklariraju ostali dio izvorne datoteke, pa je sp cjelobrojnog tipa, a val ima tip double (čija je veličina negdje određena), no one ne kreiraju varijable, niti rezerviraju memoriju za njih.

Mora postojati samo jedna definicija vanjske varijable u svim datotekama od kojih je sastavljen izvorni program. Ostale datoteke mogu imati vanjske (extern) deklaracije radi pristupa (isto tako mogu postojati extern deklaracije u datoteci s definicijom). Veličine polja određuju se definicijom, ali su one opcijske kod extern deklaracije.

Inicijalizacija vanjske varijable obavlja se samo uz definiciju.

Iako ponešto izmijenjene strukture, funkcije push i pop mogu se definirati u jednoj datoteci, a varijable val i sp u drugoj. Tada bi za njihovu povezanost bile obavezne slijedeće definicije i deklaracije:

U datoteci 1:

```
extern int sp;
extern double val[];
void push(double f){ ... }
double pop(void){ ... }
```

U datoteci 2:

```
int sp=0;
double val[MAXVAL];
```

Kako se extern deklaracije u datoteci nalaze ispred i izvan definicija funkcije, one su primjenjive na sve funkcije. Jedan skup deklaracija je dovoljan za cijelu datoteku 1. Ista ovakva struktura bila bi potrebna ako bi se definicije za varijable sp i val pojavile nakon njihove upotrebe u jednoj datoteci.

4.5 Datoteke zaglavlja

Razmotrimo sada podjelu programa za kalkulator na nekoliko izvornih datoteka, što bi bilo praktično kad bi komponente bile veće. Funkcija main išla bi u jednu datoteku, koju bismo nazvali main.c, funkcije push i pop s njihovim varijablama idu u drugu datoteku, stack.c. Funkcija getop ide u treću koja se zove getop.c, te na kraju funkcije getch i ungetch idu u četvrtu datoteku, getch.c.

Postoji još nešto o čemu treba voditi brigu. To su definicije i deklaracije koje su zajedničke za više datoteka. Naš cilj je centralizacija, najveća moguća, radi stvaranja samo jedne kopije koja se čuva tijekom rada programa. Shodno tome, prebacit ćemo ovaj zajednički izvorni kod u datoteku zaglavlja, calc.h, koja će se uključivati po potrebi (#include linija bit će opisana nešto kasnije). Sada, konačni izgled programa jest

```
calc.h:

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

```
main.c:

#include <stdio.h>
#include <math.h>
#include "calc.h"
#define     MAXOP 100
main(){
      ...
}


getop.c:

#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop(){
      ...
}


stack.c:

#include <stdio.h>
#include "calc.h"
#define     MAXVAL      100
int sp=0;
double val[MAXVAL];
void push(double){
      ...
}
double pop(void){
      ...
}


getch.c:

#include <stdio.h>
#define     BUFSIZE     100
char buf[BUFSIZE];
int bufp=0;
int getch(void){
      ...
}
void ungetch(int){
      ...
}
```

Imamo zanimljiv sukob interesa. Želimo svakoj datoteci dati mogućnost pristupanja samo onoj informaciji koja joj je potrebna za obavljanje posla, a na drugoj strani susrećemo se s činjenicom da nije lako podržavati više datoteka zaglavlja istovremeno. Za umjerenu veličinu programa najbolje je imati samo jednu datoteku zaglavlja, u kojoj je sve zajedničko za sve dijelove programa. To bi bio nekakav zaključak iznesenog ovdje. Za znatno veći program bila bi potrebna bolja organizacija i više zaglavlja.

4.6 Statičke varijable

Varijable sp i val u potprogramu stack.c ili pak buf i bufp u potprogramu getch.c, služe za posebnu primjenu funkcija u njihovim izvornim datotekama, a ne može im pristupiti nitko drugi. Deklaracija static, koja se primjenjuje na vanjsku varijablu ili funkciju, limitira opseg tog objekta samo na preostali dio datoteke koja

se prevodi. Vanjska static varijabla tako omogućuje skrivanje imena varijabli poput buf i bufp u getch-ungetch konstrukciji koja mora biti vanjska. Isto tako ona može varijable dijeliti, ali varijable nisu vidljive za korisnike funkcija getch i ungetch.

Statička pohrana određuje se dodavanjem prefiksa static normalnoj deklaraciji. Ako se dvije funkcije i dvije varijable prevedene u jednoj datoteci, kao u

```
static char buf[BUFSIZE];
static int bufp=0;
int getch(void){
      ...
}
int ungetch(int c){
      ...
}
```

tada nijedna druga funkcija neće moći pristupiti varijablama buf i bufp, a ta imena će se bez problema moći koristiti u drugim datotekama tog programa. Isto tako, varijable koje funkcije push i pop koriste za manipulaciju stogom daju se sakriti deklariranjem varijabli sp i val kao static.

Vanjska static deklaracija najčešće se koristi za varijable, ali je primjenjiva i na funkcije. Imena funkcija su općenito vidljiva svakom dijelu programa, ali sa static deklaracijom njeno je ime nevidljivo van datoteke u kojoj je deklarirana.

Deklaracija static je, nadalje, primjenjiva i na nutarnje varijable. Unutrašnje static varijable su lokalne za određenu funkciju, baš kao i automatske varijable, no za razliku od automatskih varijabli, one ostaju trajno, što je prihvatljivije od solucije da se inicijaliziraju i nestaju pri svakom aktiviranju i deaktiviranju funkcije. To znači da unutrašnje static varijable posjeduju stalnu memoriju u okviru jedne funkcije.

Vježba 4-11. Modificirajte funkciju getop kako ona ne bi morala rabiti funkciju ungetch. Savjet: Upotrijebite unutrašnju static varijablu.

4.7 Registarske varijable

Deklaracija register sugerira prevoditelju da se spomenuta varijabla često koristi. Ideja je da se register varijable pohrane varijable u procesorske registre što može imati značajan učinak u brzini i veličini programa. Dakako, prevoditelj tu sugestiju može uredno zanemariti.

Deklaracija register izgleda kao

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

i tako dalje. Deklaracija register dade se primijeniti samo na automatske varijable i na formalne parametre funkcija. U ovom drugom slučaju, to izgleda ovako:

```
f(register unsigned m, register long n){
      register int i;
      ...
}
```

U praksi pak, postoje ograničenja u vezi s registarskim varijablama, koja su odraz ugrađenog sklopovlja. Samo nekoliko varijabli iz svake funkcije, primjerice, može se čuvati u registrima, i to samo određenih tipova i sl. Veliki broj registarskih deklaracija nipošto nije štetan, jer se ključna riječ register ignorira kad resursi postanu oskudni ili deklaracije nedozvoljene. Nije moguće imati adresu registarske varijable (o tome više u Poglavlju 5), bez obzira na to da li je varijabla stvarno pohranjena u registar. Posebna ograničenja registarskih varijabli ovisne su uvelike o sklopovlju.

4.8 Struktura bloka

C nije blokovski strukturiran jezik, kao primjerice, Pascal i sl. jer funkcije ne moraju biti definirane unutar drugih funkcija. S druge strane, varijable mogu biti definirane na blokovski strukturiran način u okviru funkcije. Deklaracije varijabli (uključujući i inicijalizacije) mogu se pojaviti nakon velike lijeve zagrade, koja uvodi bilo koji složeni izraz, a ne samo onaj kojim počinje funkcija. Varijable, deklarirane ovako, skrivaju sve

varijable s istim imenom u vanjskim blokovima i ostaju u upotrebi sve dok ne naiđe velika desna zagrada. Na primjer, u

```
if(n>0){
      int i;      /* prijava novog i */
      for(i=0;i<n;i++){
            ...
      }
}
```

opseg varijable i jest blok koji se izvrši kad je uvjet if naredbe istinit. Ova varijabla nema veze ni s jednom varijablom i koja je izvan spomenutog bloka. Automatska varijabla koja je deklarirana i inicijalizirana u bloku nanovo se inicijalizira pri svakom prolazu kroz blok. Varijabla static inicijalizira se samo pri prvom prolazu.

Automatske varijable, uključujući formalne parametre, također skrivaju vanjske varijable i funkcije istog imena. Deklaracije

```
int x;
int y;
f(double x){
      double y;
      ...
}
```

daju različite pojavnosti varijabli x i y glede tipa. Izvan funkcije, naime, i varijabla x i varijabla y imaju tip int, dok unutar funkcije imaju karakter tipa double. Više radi ljepšeg stila treba izbjegavati imena varijabli koja skrivaju imena varijabli iz vanjskog opsega, radi moguće greške.

4.9 Inicijalizacija

Dosad smo inicijalizaciju spominjali samo uzgred, te uvijek u okviru nekog drugog pitanja. Sada, nakon upoznavanja različitih načina pohrane, u ovom dijelu posložit ćemo kockice.

U nedostatku vanjske inicijalizacije, vanjske i statičke varijable su sigurno postavljene na nulu. Automatske i registarske varijable imaju nedefinirane, bolje kazano, neupotrebljive početne vrijednosti.

Brojčane varijable dadu se inicijalizirati, ako su definirane, tako što je ime popraćeno znakom jednakosti i izrazom:

```
int x=1;
char jednostruki_navodnik='\'';
long dan=1000L*60L*60L*24L;   /* broj milisekundi u jednom danu */
```

Za vanjske i statičke varijable inicijalizator mora biti konstantni izraz. Tu se inicijalizacija obavlja jednom i to prije no što počne izvršavanje programa. Kod automatskih i registarskih varijabli, to se obavlja pri svakom prolasku kroz funkciju ili blok.

Kod automatskih i registarskih varijabli, inicijalizator ne mora biti konstanta. On može biti bilo koji izraz, uključujući prethodno definirane vrijednosti, a čak i pozive funkcija. Na primjer, inicijalizicija programa za pretraživanje binarnog stabla može se napisati kao

```
int binsearch(int x, int v[], int n){
      int low=0;
      int high=n-1;
      int mid;
      ...
}

umjesto

int low, high, mid;
low=0;
high=n-1;
```

Zapravo, inicijalizacije automatskih varijabli predstavljaju skraćenice za izraze dodjeljivanja. Koji ćemo oblik koristiti, u principu jest stvar ukusa. Najčešće ćemo rabiti eksplicitna dodjeljivanja, jer se inicijalizatori u deklaracijama teže uočavaju i koriste.

Polje se može inicijalizirati nakon deklaracije listom inicijalizatora u zagradi, koja je odvojena zarezima. Primjerice, da bi inicijalizirali polje dani pomoću broja dana u svakom mjesecu:

```
int dani[]={31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Kad je veličina polja izostavljena, prevoditelj će automatski prebrojiti inicijalizatore, kojih je u našem konkretnom slučaju 12.

Ako ima manje inicijalizatora polja od precizirane veličine, preostala mjesta za inicijalizatore postaju nule za vanjske i statičke varijable,a neupotrebljive vrijednosti za automatske. Postojanje viška inicijalizatora rezultira greškom. Ne postoji način određivanja ponavljanja inicijalizatora, niti inicijalizacije elementa u sredini polja, a da se ne pozovu sve prethodne vrijednosti.

Znakovna polja predstavljaju poseban slučaj inicijalizacije. Dade se upotrijebiti niz mjesto notacije sa velikim zagradama i zarezima:

```
char uzorak[]="mal";
```

predstavlja značajno skraćenje u odnosu na

```
char uzorak[]={'m','a','l','\0'};
```

U ovom slučaju, veličina polja je četiri (tri znaka + limitator '\0').

4.10 Rekurzija

C funkcije se mogu koristiti rekurzivno. To znači da funkcija može pozivati samu sebe bilo direktno, bilo indirektno. Prispodobite da ispisujemo broj kao niz znakova. Kako smo spomenuli, znamenke se generiraju u krivom slijedu. Znamenke koje su niže po redoslijedu prihvaćaju se prije od onih koje su više, ali će se ispisati obrnuto. Postoje dva rješenja. Jedno je pohranjivanje znamenki u polje po redu generiranja, a zatim ispisivanje po obrnutom slijedu kako smo već napravili funkcijom itoa. Alternativu predstavlja rekurzivno rješenje, u kojem funkcija printd najprije poziva samu sebe kako bi obavila posao s vodećim znamenkama, a onda ispisala zadnje znamenke. Spomenimo ipak da funkcija u ovakvom obliku može griješiti kod velikih negativnih brojeva.

```
#include <stdio.h>

/* printd : ispiši n decimalno */
void printd(int n){
      if(n<0){
            putchar('-');
            n=-n;
      }
      if(n/10)
            printd(n/10);
      putchar(n%10+'0');
}
```

Kad funkcija rekurzivno pozove samu sebe, svaki poziv stvara novi set automatskih varijabli, nezavisan o prethodnom. Tako pozivom funkcije printd(123), varijabla n iz funkcije printd dobiva vrijednost 123, drugim pozivom 12, a trećim 1. Na tom nivou funkcija ispisuje broj 1, a zatim se vraća na nivo koji ju je pozvao. Ta printd funkcija ispisuje 2, te se vraća na prvi nivo gdje se ispisuje broj 3 i zatvara rekurzivna petlja.

Slijedeći dobar primjer rekurzije jest algoritam za brzo sortiranje koji je 1962. napravio C.A.Hoare. Za zadano polje izabere se jedan element, a ostali se razlože na dva podskupa elemenata. U jednom podskupu nalaze se elementi manji od korijenskog, a u drugom veći ili jednaki. Isti se princip primjeni i na podskupove elemenata, jasno, rekurzivnim putem. Kad u podskupu ima manje od dva elementa, rekurzija se stopira.

Prikazana verzija "quicksort" algoritma nije najbrža, no jedna je od jednostavnijih. Za razlaganje koristit ćemo srednji element svakog podskupa.

```
/* qsort : algoritam za brzo sortiranje */
void qsort(){
      int i, last;
      void swap(int v[], int i, int j);
      if(left>=right)   /* treba razdvojiti elemente */
            return;
      swap(v, left, (left+right)/2);
      last=left;
      for(i=left+1;i<=right;i++)
            if(v[i]<v[left])
                  swap(v, ++last, i);
      swap(v, left, last);
      qsort(v, left, last-1);
      gsort(v, last+1, right);
}
```

Sama se zamjena elemenata obavlja u funkciji swap iz prostog razloga što se poziva u funkciji qsort tri puta.

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

Standardna biblioteka ima inačicu qsort funkcije koja može sortirati objekte bilo kojeg tipa. Rekurzija možda neće biti u stanju rezervirati memoriju, zbog čega se stog vrijednosti mora negdje održati. To svakako neće biti najbrža rutina.

S druge strane, rekurzivni kod je kompaktniji, vrlo često razumljiviji od algoritama koji to nisu. Rekurzija je izuzetno pogodna za rekurzivno definirane strukture podataka kakva su, primjerice, stabla.

Vježba 4-12. Prilagodite ideje funkcije printf tako da se rekurzivno koristi funkcija itoa. Dakle, pretvorite cijeli broj u niz, pozivajući rekurzivnu rutinu.

Vježba 4-13. Upišite rekurzivnu verziju funkcije reverse(s), koja izvrće niz s.

4.11 C preprocesor

Programski jezik C omogućuje određene olakšice ako se rabi preprocesor kao prvi korak u prevođenju. Dvije konstrukcije su najčešće u upotrebi. Jedna je #include, za uključivanje datoteka za prevođenje, te #define, koja zamjenjuje simbol proizvoljnim nizom znakova. Druge konstrukcije opisane u ovom dijelu pretpostavljaju uvjetno prevođenje i makro naredbe s argumentima.

4.11.1 Uključivanje datoteke

Uključivanje datoteke olakšava manipulaciju funkcijama, deklaracijama i #define skupovima. Svaka izvorna linija oblika

```
#include "ime datoteke"

ili

#include <ime datoteke>
```

se mijenja sadržajem imena datoteke. Ako je ime datoteke pod navodnicima, datoteka se počinje tražiti u direktoriju u kojem je i izvorni program. Ako se ona tamo ne nalazi, potraga se nastavlja od mjesta koje je definirano implementacijom prevoditelja (moguće je definirati mjesta na kojima će se tražiti datoteke iz #include konstrukcije).

Često se može pojaviti nekoliko #include linija na početku izvorne datoteke, koje uključuju zajedničke #define naredbe i extern deklaracije (ako ste primijetili, #define katkad nazivam naredbom, a drugi put konstrukcijom, jer zapravo nije ni jedno ni drugo) ili pak pristupaju deklaracijama prototipa funkcije iz standardne biblioteke (kako bi bili do kraja precizni, to ne moraju biti datoteke jer način pristupanja datotekama ovisi o implementaciji).

Naredba #include je najbolja za međusobno povezivanje deklaracija velikog programa. Ona jamči da će sve izvorne datoteke vidjeti definicije i deklaracije varijabli, što će eliminirati neke poteškoće. Naravno, kad se promatrana datoteka izmjeni, sve datoteke koje je spominju u izvornom kodu, moraju se ponovo prevoditi.

4.11.2 Makrozamjena

Definicija ima oblik

```
#define ime tekst zamjene
```

Ona izvrši makrozamjenu u najjednostavnijem obliku. Učestala pojavljivanja imena simbola zamijenit će se tekstom zamjene. Ime u #define naredbi istog je oblika kao i ime varijable; tekst zamjene je proizvoljan. Jasno, tekst zamjene je ostali dio linije, no dugačka definicija može prelaziti preko više linija ako se na kraj svake linije koja se nastavlja umetne \. Opseg imena definiranog pomoću #define konstrukcije proteže se od mjesta na kojemu je definiran, pa do kraja izvorne datoteke koja se prevodi. Definicija može koristiti prethodne definicije. Zamjene se obavljaju samo na simbolima, a ne obavljaju se u nizovima unutar navodnika. Primjera radi, ako je YES definirano ime, zamjena neće biti obavljena kod

```
printf("YES");
```

ili u

```
YESMAN=123;
```

Svako ime može biti definirano pomoću bilo kojeg teksta zamjene. Tako npr.,

```
#define forever for(;;) /* za beskonačnu petlju */
```

definira novu riječ, forever, za beskonačnu petlju.

Također je moguće definirati makronaredbe s argumentima, tako da tekst zamjene može biti različit za različita pojavljivanja makronaredbe. Možemo definirati makronaredbu čije je ime max:

```
#define max(A, B) ((A) (B) ? (A) : (B))
```

Mada izgleda kao poziv funkcije, upotreba max makronaredbe se prevodi u neposredni kod. Svaka pojava formalnog parametra (A ili B) bit će zamijenjeno odgovarajućim stvarnim argumentom. Stoga će linija

```
x=max(p+q, r+s);
```

biti zamijenjena linijom

```
x=((p+q)>(r+s)?(p+q):(r+s));
```

Sve dok se dosljedno postupa s argumentima, ova makronaredba poslužit će za bilo koji tip podatka. Dakle, nema potrebe za postojanjem različitih oblika makronaredbe za različite tipove podataka što je slučaj kod funkcija.

Ako dobro razmotrimo proširivanje makronaredbe max, možemo uočiti neke nejasnoće. Izrazi se računaju dvaput. Tako, nije baš preporučljivo u argumente umetati inkrementacijske operatore

```
max(i++, j++)     /* pogrešno */
```

jer će se veći broj u gornjem izrazu uvećati dvaput. Moramo, isto tako, voditi računa i o zagradama kako bi sigurno očuvali slijed računanja. Što bi se, primjerice dogodilo kada bi makronaredbu

```
#define     square(x) x*x     /* pogrešno */
```

pozvali kao square(z+1)?

Ipak, makronaredbe su korisne. Konkretan primjer nalazimo i u zaglavlju <stdio.h>, u kojemu se funkcije getchar i putchar često definiraju kao makronaredbe da bi se skratilo vrijeme pozivanja funkcije po učitanom znaku. Isto tako, funkcije iz zaglavlja <ctype.h> obično se uvode kao makronaredbe.

Imena funkcija mogu se osigurati pomoću #undef naredbe, što se čini kad se želi naglasiti da je potprogram zbilja funkcija, a ne makronaredba.

```
#undef      getchar
int getchar(void){
      ...
}
```

Formalni parametri se ne zamjenjuju unutar nizova znakova pod navodnicima. Ako pak imenu parametra prethodi # u tekstu zamjene, kombinacija će se proširiti u niz znakova pod navodnicima pomoću parametra koji se zamjenjuje stvarnim argumentom. Tu činjenicu možemo iskoristiti s osobinom povezivanja nizova kako bi napravili makronaredbu za otklanjanje grešaka pri ispisu

```
#define     dprint(expr) printf(#expr " = %g\n", expr)
```

Kad se makronaredba pozove sa

```
dprint(x/y);
```

ona se proširuje u

```
printf("x/y" " = %g\n", x/y);
```

nizovi znakova se povezuju, pa dobivamo

```
printf("x/y = %g\n", x/y);
```

Preprocesorski operator ## osigurava način povezivanja argumenata za vrijeme obavljanja makronaredbe. Ako se parametar u tekstu nalazi u blizini ##, on se zamjenjuje argumentom, operator i eventualni okolni razmaci se premještaju, a rezultat ponovo provjerava. Primjerice, makronaredba paste povezuje svoja dva argumenta

```
#define     paste(front, back) front ## back
```

tako da

```
paste(name, 1)
```

kreira simbol name 1.

Pravila za uporabu ## su složena. Detaljnije ćemo ih proučiti kasnije (Dodatak A).

Vježba 4-14. Definirajte makronaredbu swap(t, x, y), koji obavlja zamjenu dva argumenta tipa t. Uputa: Blok struktura može biti od pomoći.

4.11.3 Uvjetno uključivanje

Moguća je kontrola samog preprocesiranja uvjetnim izrazima koji se računaju tokom preprocesiranja. To određuje selektivno uključivanje koda, ovisno o rezultatima izračunatim tokom prevođenja.

Naredba #if računa izraz konstantnog cijelog broja. Ako izraz nije nula, linije koje slijede do naredbi #endif ili #elif ili #else bit će obrađene (preprocesorska naredba #elif sliči na #else if). Izraz

```
defined(ime);
```

u #if naredbi ima vrijednost 1 ako je ime već definirano, a 0 ako nije.
Primjerom kazano, kako bismo osigurali uključivanje datoteke hdr.h samo jednom, možemo pisati

```
#if !defined(HDR)
#define HDR

/* sadržaj hdr.h se uključuje */
#endif
```

Prvo uključivanje hdr.h definira ime HDR. Eventualni kasniji pokušaji uključivanja pronaći će definirano ime, te će preskočiti preko #endif. Takav stil valja rabiti kako bi se izbjegao veliki broj uključivanja datoteka. Ako se ovaj stil dosljedno poštuje, tada svako pojedino zaglavlje može uključiti sva zaglavlja od kojih ovisi, a da korisnik ne zna ništa o njihovoj međusobnoj ovisnosti.
Ovaj niz testira ime SYSTEM kako bi donio odluku o uključivanju zaglavlja:

```
#if SYSTEM=SYSV
      #define     HDR "sysv.h"
#if SYSTEM=BSD
      #define     HDR "bsd.h"
#if SYSTEM=MSDOS
      #define     HDR "msdos.h"
#else
      #define     HDR "default.h"
#endif
#include HDR
```

Linije #ifdef i #ifndef su specijalizirane forme za ispitivanje definiranosti imena. Prvi primjer #if konstrukcije dade se napisati ovako

```
#ifndef     HDR
#define     HDR

/* sadržaj HDR se uključuje */

#endif
```

PETO POGLAVLJE: POKAZIVAČI I POLJA

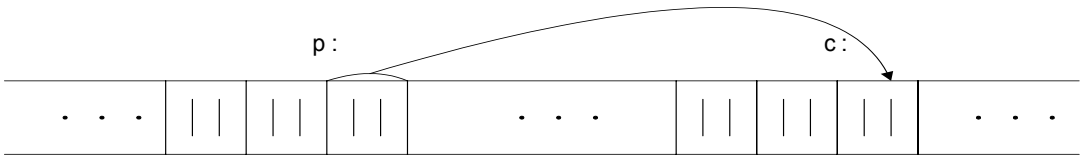
Pokazivač je varijabla koja sadržava adresu varijable. Pokazivači se intenzivno koriste u programskom jeziku C dijelom i zato jer je to katkad jedini put rješenja problema, a dijelom i zbog toga što vode kompaktnijem i efikasnijem kodu od bilo kojeg drugog načina. Pokazivači i polja tijesno su povezani. Tema je ovog poglavlja upravo veza između ta dva pojma i načini na koje se ona koristi.

Pokazivači su se rabili kod goto izraza stvarajući tako programe koje nitko ne može razumjeti. To je naročito dolazilo do izražaja kad bi se program stvarao nesavjesno, pa je sasvim vjerojatna bila pojava pokazivača usmjerenih pogrešno. Isto tako, uz samo malo pažnje, pokazivači se mogu iskoristiti da bi se postigla jasnoća i jednostavnost.

Glavna izmjena u ANSI C standardu jest u razjašnjavanju pravila o tome kako manipulirati pokazivačima, što u praksi dobri prevoditelji i programeri već primjenjuju. Isto tako, tip void * (pokazivač na void) mijenja char * kao odgovarajući tip srodnog pokazivača.

5.1 Pokazivači i adrese

Počnimo s pojednostavljenim crtežom organizacije memorije. Jedno tipično računalo ima polje koje je niz adresiranih memorijskih ćelija kojima se može manipulirati pojedinačno i grupno. Sasvim je uobičajena situacija da bilo koji oktet (byte) može biti char, par okteta može se promatrati kao short, a četiri susjedna okteta čine jedan long cjelobrojni tip. Pokazivač je skupina ćelija (često dvije ili četiri) koja može sadržavati adresu. Tako, ako je c tipa char, a p je pokazivač na nj, onda se to dade ovako predstaviti:



Unarni operator & daje adresu objekta, tako naredba

```
p=&c;
```

određuje adresu c prema varijabli p, a za p možemo reći da "pokazuje na" c. Operator & odnosi se samo na objekte u memoriji, bile to varijable ili elementi nekog polja. On se ne može primijeniti na izraze, konstante ili registarske varijable.

Unarni operator * je indirektni ili dereferentni operator. Kad se primjeni na pokazivač, on pristupa objektu na koji pokazivač pokazuje. Recimo da su x i y cijeli brojevi, a da je ip pokazivač na int. Taj neprirodan slijed pokazuje kako se deklarira pokazivač i kako se koriste operatori & i *:

```
int x=1, y=2, z[10];
int *ip;    /* ip sad pokazuje na int */

ip=&x;            /* ip sad pokazuje na x */
y=*ip;            /* y je sad 1 */
y=*ip;            /* x je sada nula */
*ip=0;            /* x je sada nula */
ip=&z[0];   /* ip sada pokazuje na z[0]*/
```

Do sada su bile dotaknute deklaracije x, y i z. Deklaracija pokazivača ip,

```
int *ip;
```

služi za pohranu podatka o podatku. Ona kaže da izraz *ip predstavlja cjelobrojnu vrijednost. Sintaksa deklaracije jedne varijable predstavlja sintaksu izraza u kojemu se varijabla može pojaviti. Takav pristup možemo primijeniti i kod deklariranja funkcija. Npr.,

```
double *dp, atof(char *);
```

kaže da u nekakvom izrazu *dp i atof(s) imaju vrijednost tipa double, a da je argument atof funkcije znakovnog tipa.

Treba uočiti implikaciju kako pokazivač obavezno pokazuje na određenu vrstu objekta (tip podatka). Međutim, i tu nalazimo neke izuzetke kakav je, primjerice, pokazivač na void koji se koristi za čuvanje bilo kojeg tipa podatka, ali ne može se sam obnavljati (o tome šire u odjeljku 5.11).

Ako ip pokazuje na cijeli broj x, onda se *ip može pojaviti u bilo kojem kontekstu u kojem bi se mogao pojaviti x, tako da

```
*ip=*ip+10;
```

povećava *ip za 10.

Unarni operatori * i & povezuju jače nego aritmetički, tako da pridijeljeni

```
y=*ip+1;
```

prihvaća sve ono na što ip pokazuje, dodaje 1 i dodjeljuje rezultat

```
*ip+=1;
```

uvećava ono na što ip pokazuje, kao što čine i

```
++*ip
```

i

```
(*ip)++
```

Mala zagrada je obavezna u zadnjem primjeru, jer bez nje izraz bi povećao ip umjesto onoga na što ip pokazuje, uzevši u obzir da unarni operatori oblika * i ++ pridružuju zdesna nalijevo.

Naposljetku, kako su pokazivači varijable, oni se mogu rabiti bez dereferenciranja. Tako, ako je iq slijedeći pokazivač na cjelobrojnu vrijednost,

```
iq=ip
```

preslika sadržaj ip u iq, pa varijabla iq pokazuje na ono na što je pokazivao ip.

5.2 Pokazivači i argumenti funkcija

Vidjeli smo da C predaje argumente funkcijama pomoću vrijednosti. Dakle, nema načina da se direktno promijeni varijabla u funkciji iz koje je druga funkcija pozvana. Tako, potprogram za sortiranje zamjenjuje dva neporedana elementa funkcijom sa imenom swap. Nije dovoljno napisati

```
swap(a, b);
```

gdje je funkcija definirana kao

```
void swap(int x, int y){      /* ovo je pogresno */
      int temp;
      temp=x;
      x=y;
      y=temp;
}
```

Zbog poziva preko vrijednosti, funkcija swap ne može utjecati na argumente a i b u funkciji koja ju je pozvala. Ona samo zamjenjuje preslike (možda je ipak razumljivije reći kopije) varijabli a i b.

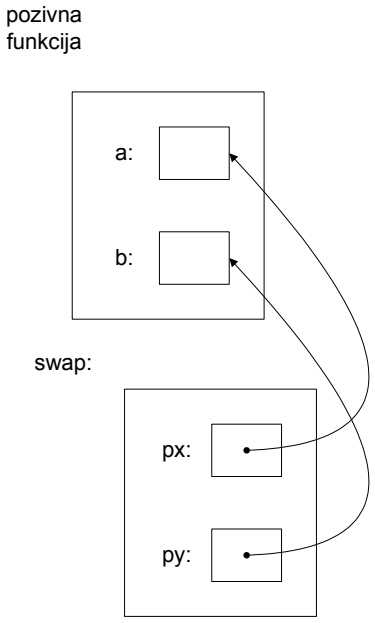
Da bi se dobio željeni efekt, potrebno je pozivnim programom predati pokazivače do vrijednosti ovih varijabli:

```
swap(&a, &b);
```

Pošto operator & daje adresu varijable, &a je pokazivač na varijablu a. U samoj funkciji swap, parametri su definirani kao pokazivači, a operandima se pristupa indirektno preko njih.

```
void swap(int *px, int *py){  /* zamijeni *px i *py */
      int temp;
      temp=*px;
      *px=*py;
      *py=temp;
}
```

Zorno:



Pokazivači argumenata omogućuju funkciji prihvaćanje i izmjenu objekata u funkciji koja ju je pozvala. Primjerice, promotrimo funkciju getint koja vrši ulaznu pretvorbu bez formatiranja, razlažući niz znakova u cjelobrojne vrijednosti, jedan cijeli broj po pozivu. Funkcija getint mora vratiti pronađenu vrijednost, te signalizirati kraj datoteke kad više nema ulaznih podataka. Ove vrijednosti moraju biti predane natrag odvojenim putovima, jer bez obzira koja vrijednost se koristi za EOF, to bi također mogla biti ulazna cjelobrojna vrijednost.

Jedno rješenje jest da funkcija getint vrati status kraja datoteke kao svoju funkcijsku vrijednost, koristeći pritom pokazivač argumenta za pohranu pretvorenog cijelog broja u funkciji koja je pozvala funkciju getint. Ovo je, inače, rješenje koje koristi funkcija scanf (kasnije, u dijelu 7.4)

Naredna petlja ispunjava polje cijelim brojevima koristeći funkciju getint:

```
int n, array[size], getint(int *);
for(n=0;n<SIZE&&getint(&array[n])!=EOF;n++)
```

Svako pozivanje namješta array[n] na slijedeći cijeli broj sa ulaza i povećava n. Primijetimo kako je od izuzetne važnosti predati adresu varijable polja array[n] funkciji getint. Inače, nema načina da funkcija getint vrati pretvoreni broj u natrag u pozivnu funkciju.

Naša inačica funkcije getint vraća EOF kao kraj datoteke, nulu ako slijedeći ulaz nije broj, a pozitivnu vrijednost ako je ulaz valjan.

```
#include <ctype.h>
int getch(void);
void ungetch(int);
/* getint : uzmi slijedeći cijeli broj sa ulaza u *pn */
int getint(int *pn){
      int c, sign;
```

```
        while(isspace(c=getch()))     /* preskoči razmak */
              ;
        if(!issdigit(c)&&c!=EOF&&c!='+'&&c!='-'){
              ungetch(c);
              return 0;
        }
        sign=(c=='-')?-1:1;
        if(c=='+'||c=='-')
              c=getch();
        for(*pn=0;isdigit(c);c=getch())
              *pn=10**pn+(c-'0');
        *pn *=sign;
        if(c!=EOF)
              ungetch(c);
        return c;
}
```

Unutar funkcije getint *pn se koristi kao najobičnija int varijabla. Isto tako, upotrijebili smo funkcije getch i ungetch (proučene u dijelu 4.3) tako da se čitanjem specijalnog znaka mogu ponovno vratiti na ulaz.

Vježba 5-1. Kako se dade vidjeti iz primjera, funkcija getint tretira + ili - znakove kao nulu ako iza njih ne slijedi broj. Pročitajte ga da bi ga vratili na ulaz.

Vježba 5-2. Napišite funkciju getfloat, koja s brojevima s pokretnim zarezom radi ono što radi funkcija getint s cijelim brojevima.

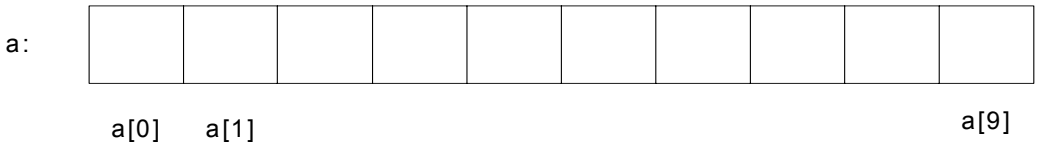
5.3 Pokazivači i polja

U programskom jeziku C postoji čvrsta veza između pokazivača i polja, pa ćemo ih paralelno proučavati. Bilo koja operacija koju možemo obaviti preko indeksa polja, dade se obaviti i preko pokazivača. Rad preko pokazivača jest principijelno brži, ali i za neiskusne teže razumljiv.

Deklaracija

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

definira polje a veličine 10, što predstavlja skup od deset objekata nazvanih a[0], a[1], a[2], ..., a[8], a[9].



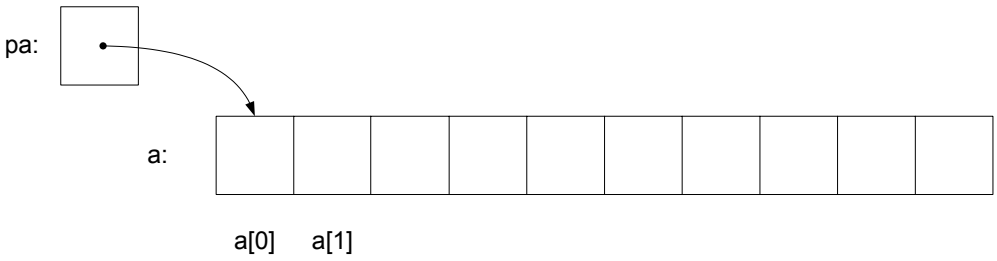
Pri tom a[i] odgovara i-tom elementu polja. Ako je pa pokazivač na cjelobrojnu vrijednost, deklariran kao

```
int *pa;
```

tada naredba

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

dodjeljuje varijabli pa vrijednost koja je ekvivalentna adresi prvog elementa polja a.



Sad naredba

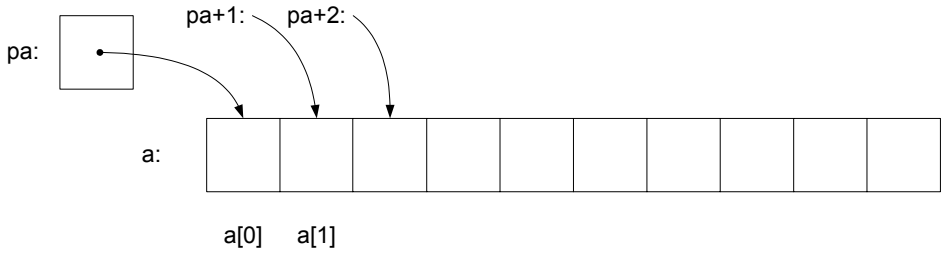
```
x=*pa;
```

prebacuje sadržaj a[0] u x.

Ako pa pokazuje na određeni element polja, onda po definiciji pa+1 pokazuje na slijedeći element, pa+i pokazuje na i-ti element poslije prvog, dok pa-i pokazuje na i-ti element prije prvog. Stoga, ako pa pokazuje na a[0],

```
*(pa+1)
```

se odnosi na sadržaj od a[1], pa+i je adresa od a[i], a *(pa+i) je sadržaj od a[i].



Ove opaske su točne bez obzira na tip i veličinu varijabli u polju a. Time tehnika rada s pokazivačima nalikuje radu s indeksima polja, a mogućnosti time nisu ni izbliza iscrpljene.

Spomenuta sličnost je uistinu velika. Po definiciji, vrijednost varijable ili izraz koji određuje tip polja je adresa prvog elementa (ili elementa s indeksom 0) polja. Zato, nakon izraza

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

pa i a imaju identične vrijednosti. Kako je ime polja sinonim za lokaciju početnog elementa, izraz pa=&a[0] dade se napisati kao

```
pa=a;
```

Vrlo je zanimljivo proučiti činjenicu da se referenca za a[i] dade napisati kao *(a+i). Pri računanju a[i], C ga odmah pretvara u *(a+i). Primjenjujući operator & na oba dijela ove jednakosti, slijedi da su &a[i] i a+i jednako tako identični izrazi. S druge strane, ako je pa pokazivač, izrazi ga mogu koristiti ako mu dodijelimo indeks jer je pa[i] identično s *(pa+i).

Postoji razlika između imena polja i pokazivača koja se mora znati. Pokazivač je varijabla, pa su pa=a i pa++ dozvoljene operacije. Ali ime polja nije varijabla, pa konstrukcije tipa a=pa i a++ nisu dopuštene.

Kad se ime polja pridjeljuje funkciji, prenosi se, jasno, lokacija početnog elementa. Unutar pozvane funkcije, ovaj argument jest lokalna varijabla, pa je parametar imena polja pokazivač, odnosno varijabla koja sadrži adresu. Možemo iskoristiti tu činjenicu kako bismo napisali još jednu verziju funkcije strlen, koja računa veličinu niza.

```
/* strlen : vraća dužinu niza */
int strlen(char *s){
      int n;
      for(n=0;*s!='\0';s++)
            n++;
      return n;
}
```

Pošto je s pokazivač, njegovo uvećavanje je dopušteno, pri čemu izraz s++ nema utjecaja na niz znakova u funkciji koja je pozvala funkciju strlen, već samo povećava pokazivač na početak niza. Naglasimo da su moguće konstrukcije tipa

```
strlen("Hello, World"); /* konstanta niza */
strlen(array);          /* char array[100]; */
strlen(ptr);            /* char *ptr; */
```

Kao formalni parametri u definiciji funkcije

```
char s[];
```

i

```
char *s;
```

jesu jednakovrijedni. Radije rabimo ovaj drugi, jer on jasno pokazuje da je parametar pokazivač. Kad se ime polja dodjeljuje funkciji, možemo kazati da funkcija dobije polje ili pokazivač, kako nam drago. Uglavnom, mogu se rabiti obje notacije ako su prikladne i jasne. Možemo tako prenijeti dio polja u funkciju prenoseći pokazivač na početak dijela polja (ako vam takav način izražavanja odgovara). Tako ako je a polje,

```
f(&a[2])
```

i

```
f(a+2)
```

prenose u funkciju f adresu dijela polja koje počinje s a[2]. U okviru funkcije f, deklaracija parametra može glasiti

```
f(int arr[]){
      ...
}

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

Tako da, dok smo god unutar funkcije f, činjenica da se parametar odnosi samo na dio polja, nebitna. Ako smo sigurni da ti elementi postoje, moguće je elemente pozvati s negativnim indeksima, npr., p[-2]. Takva je notacija također dopuštena, a odnosi se na elemente koji prethode onomu koji je prenesen u funkciju. Jasno, ako pozovemo elemente koji prelaze granice polja, možemo dobiti nebulozne rezultate.

5.4 Adresna aritmetika

Ako je p pokazivač na neki element polja, tada p++ povećava p da bi pokazivao na slijedeći element, dok p+=i ga povećava kako bi pokazivao na i-ti element nakon trenutnog. To su najjednostavniji primjeri pokazivača i adresne aritmetike.

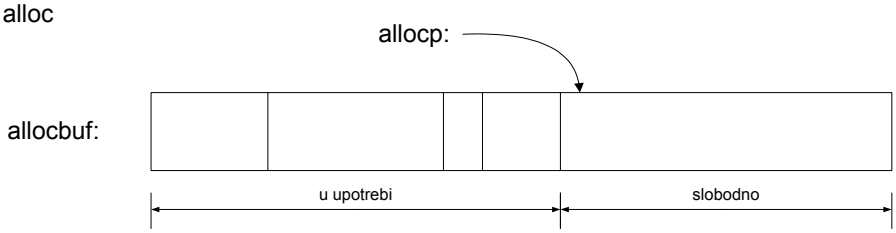
C je regularan i dosljedan u svom pristupu adresnoj aritmetici; njegova integracija polja, pokazivača i adresne aritmetike je jedna od glavnih karakteristika jezika. Ilustrirajmo to pisanjem programa za dodjelu memorijskog prostora. Postoje dva potprograma (ili funkcije). Prva funkcija, alloc, s argumentom n vraća pokazivač na n uzastopnih pozicija znakova, koje mogu biti uporabljene za pohranu znakova. Druga, funkcija afree, čiji je parametar pokazivač p, oslobađa dodijeljenu memoriju kako bi je mogli ponovo koristiti. Standardna biblioteka posjeduje analogne funkcije koje nose naziv malloc i free, a nisu ograničene redoslijedom poziva. U dijelu 8.7 pokazat ćemo kako se one daju primijeniti.

Najjednostavnija implementacija jest funkcija alloc koja manipulira dijelovima velikog polja znakova koja ćemo nazvati allocbuf. Ovo polje je bitno i za funkciju alloc i za funkciju afree. Kako rade s pokazivačima, a ne sa poljima, nijedna druga funkcija ne mora znati ime polja, koje deklariramo kao static u

izvornoj datoteci gdje su funkcije alloc i afree, te je ono nevidljivo izvan datoteke. U praksi, polje ne mora imati ime. Ono se može dobiti pozivom funkcije malloc ili upitom operativnog sistema koji pokazivač pokazuje na neimenovani memorijski blok.

Druga potrebna informacija je koliki dio polja allocbuf se koristi. Upotrijebit ćemo pokazivač pod imenom allocp koji pokazuje na slijedeći slobodan element. Kada se od funkcije alloc zatraži niz od n znakova, ona provjeri da li ima mjesta u polju allocbuf. Za slučaj da ima, funkcija alloc vraća trenutnu vrijednost allocp pokazivača (zapravo početak slobodnog bloka), te ga povećava za n da bi pokazao na slijedeće slobodno područje. Ako nema mjesta, funkcija alloc vraća nulu. Funkcija afree samo postavlja allocp na p ako je p unutar polja allocbuf.

prije poziva funkcije alloc



nakon poziva funkcije alloc


```
#define ALLOCSIZE 10000 /* veličina potrebnog prostora */
static char allocbuf[ALLOCSIZE];    /* memorija za alloc */
static char *allocp=allocbuf; /* slijedeća slobodna pozicija */
char *alloc(int n){           /* vraća pokazivač na n znakova */
      if(allocbuf+ALLOCSIZE-allocp>=n){   /* odgovara */
            allocp+=n;
            return allocp-n;  /* staro p */
      }
      else  /* nema dovoljno mjesta */
            return 0;
}

void afree(char *p){    /* slobodna memorija na koju pokazuje p */
      if(p>=allocbuf&&p<allocbuf+ALLOCSIZE)
            allocp=p;
}
```

Zapravo, pokazivač se može inicijalizirati kao i bilo koja druga varijabla, mada su jedine moguće vrijednosti nula ili izraz koji uključuje adrese prethodno definiranih podataka istog tipa. Deklaracija

```
static char *allocp=allocbuf;
```

definira allocp kao znakovni pokazivač te ga inicijalizira na početak polja allocbuf. To se dade napisati i ovako

```
static char *allocp=&allocbuf[0];
```

jer ime polja predstavlja adresu početnog elementa.
Uvjet

```
if(allocbuf+ALLOCSIZE-allocp>=n){   /* odgovara */}
```

provjerava ima li dovoljno mjesta za zadovoljenje zahtjeva za n elemenata. Ako ima, funkcija alloc vraća pokazivač na početak bloka znakova (obratite pažnju na deklaraciju same funkcije). Ako se ne zadovolji, funkcija alloc mora vratiti neki prepoznatljiv signal da više nema prostora. C garantira da nula nikad nije vrijedeći podatak za adresu, pa je možemo iskoristiti za označavanje neprihvatljivog slučaja.

Pokazivači i cijeli brojevi se ne mogu međusobno mijenjati. Jedini izuzetak (koji potvrđuje pravilo) jest nula. Konstanta nula može se pridijeliti pokazivaču, a pokazivač se njom dade i usporediti. Simbolička konstanta NULL često se rabi umjesto nule, kako bi se upamtilo da je to specijalna vrijednost pokazivača. NULL se definira u <stdio.h>,a mi ćemo ubuduće često koristiti NULL.

Uvjeti oblika

```
if(allocbuf+ALLOCSIZE-allocp>=n){   /* odgovara */    }
```

i

```
if(p>=allocbuf&&p<allocbuf+ALLOCSIZE)
```

pokazuju nekoliko bitnih karakteristika pokazivačke aritmetike. Najprije, vidljivo je da se pokazivači mogu uspoređivati pod određenim uvjetima. Ako p i q pokazuju na članove istog polja, onda su izrazi oblika ==, != i dr. ispravni. Npr.,

```
p<q;
```

jest ispravno ako p pokazuje na prethodni element polja u odnosu na element na koji pokazuje q. Znakovito je da se svaki pokazivač može usporediti kako bi se vidjelo je li ili nije jednak nuli, no ovaj postupak nije definiran za aritmetiku ili usporedbu sa pokazivačima koji ne pokazuju elemente istog polja (I ovdje imamo izuzetak. Adresa prvog elementa poslije kraja polja može se rabiti u pokazivačkoj aritmetici.)

Nadalje, već smo uočili da se pokazivač i cijeli broj mogu zbrajati i oduzimati. Konstrukcija

```
p+n
```

predstavlja adresu n-tog objekta nakon onog kojeg p trenutno pokazuje. Ovo je točno bez obzira na koji objekt p pokazuje jer n ima opseg ovisan o veličini tog objekta, do koje dolazimo pri deklaraciji pokazivača p.

Analogijom je moguće pokazivače i oduzimati. Ova se činjenica dade iskoristiti za pisanje nove inačice funkcije strlen:

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

U svojoj deklaraciji, p se inicijalizira preko niza s, što nam kaže da pokazuje na prvi znak niza. U while petlji, testiramo svaki znak da vidimo radi li se o kraju niza. Kako je p pokazivač na znakovni tip, p++ pomiče p naprijed svaki put na slijedeći znak, dok je p-s broj takvih pomaka, što zapravo daje dužinu niza kad dođemo do njegova kraja (Inače, ovakva funkcija strlen ima značajno ograničenje u činjenici da dužina znakovnog niza ne smije prelaziti veličinu opsega brojeva tipa int. Zaglavlje <stddef.h> definira tip ptrdiff_t koji ima dovoljno veliki opseg. Iz razloga opreznosti možemo uporabiti size_t za povratni tip iz funkcije strlen, kako bismo ga prilagodili standardnoj verziji. Tip size_t jest neoznačeni tip cijelog broja kojeg vraća sizeof operator.).

Pokazivačka aritmetika je dosljedna. Ako smo manipulirali podacima tipa float, koji traže više memorijskog prostora od podataka tipa char, te ako je p pokazivač na float, tada ćemo izrazom p++ pokazivač p povećati tako da pokazuje na slijedeći float podatak (drugim riječima, vrijednost pokazivača povećat će se za onoliko memorijskih lokacija koliko zauzima tipičan float podatak). Tako možemo napisati još jednu verziju funkcije alloc, a koja podržava tip float umjesto tipa char. To je izvedivo jednostavnom zamjenom tipa char u tip float u funkcijama alloc i afree, jer sve radnje s pokazivačem automatski uzimaju u obzir veličinu objekta na koji pokazuju.

Dopuštene operacije sa pokazivačima vrijede za pokazivače istog tipa jesu zbrajanje i oduzimanje pokazivača i cijelih brojeva, oduzimanje i usporedba dvaju pokazivača sa članovima istog polja, te dodjela i usporedba s nulom. Sva druga pokazivačka aritmetika nije dopuštena. Tako se ne mogu zbrajati dva pokazivača, niti dijeliti, množiti, pomjerati (shift) ili maskirati. Ne možemo im dodavati ništa tipa float ili double, pa čak, osim za void *, pridijeliti pokazivač jednog tipa pokazivaču drugog tipa bez modela.

5.5 Pokazivači i funkcije znaka

Konstanta znakovnog niza napisana kao

```
"Ja sam niz"
```

predstavlja polje znakova. To polje se završava znakom '\0' koje ima funkciju limitatora. Memorirana dužina niza jest, dakle, za jedan znak veća od broja znakova između dvostrukih navodnika.

Moguće je da se konstante znakovnog niza najčešće pojavljuju u obliku argumenta funkcije

```
printf("hello, world");
```

Kad se znakovni niz poput ovog pojavi u programu, njemu se pristupa pomoću znakovnog pokazivača. Funkcija printf dobije pokazivač na početku polja znakova.

Konstante znakovnog niza ne moraju biti argumenti funkcije. Ako je pmessage deklarirana kao

```
char *pmessage;
```

tada izraz

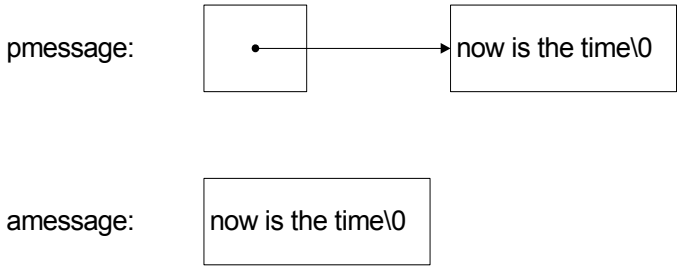
```
pmessage="now is the time";
```

pridjeljuje pokazivaču pmessage adresu prvog elementa polja znakova. To nije kopiranje niza, jer su u cijeloj operaciji zaposleni jedino pokazivači. C ne osigurava nikakve operatore za predaju niza znakova kao cjeline.

Postoji značajna razlika među ovim definicijama

```
char amessage[]="now is the time";  /* polje */
char *pmessage="now is the time";   /* pokazivač */
```

Polje amessage dovoljno je veliko da sačuva niz znakova i '\0' koji ga inicijaliziraju. Pojedine znakove u polju moguće je mijenjati, ali se ime amessage uvijek odnosi na iste memorijske lokacije. Na drugoj strani, pmessage je pokazivač, koji inicijaliziran pokazuje na konstantu znakovnog niza. Pokazivač može promijeniti svoju vrijednost, no u tom slučaju gubimo mogućnost promjene niza.



Prikazat ćemo još neke mogućnosti pokazivača i polja promatrajući dvije korisničke funkcije iz standardne biblioteke. Prva funkcija jest strcpy(s,t) koja preslikava niz t u niz s. Bilo bi zgodno napisati s=t čime bi preslikali pokazivač, ali ne i znakove. A kako bi preslikali znakove, trebamo petlju. Verzija s poljem je prva

```
/* strcpy: preslikava polje t u polje s; inačica s indeksom polja */
void strcpy(char *s, char *t){
      int i;
```

```
        i=0;
        while((s[i]=t[i])!='\0')
                i++;
}
```

Usporedbe radi, tu je i inačica funkcije strcpy s uporabom pokazivača

```
/* strcpy: preslikava polje t u polje s; prva inačica s pokazivačima */
void strcpy(char *s, char *t){
        while((*s=*t)!='\0'){
                s++;
                t++;
        }
}
```

Kako se argumenti prenose po vrijednosti, funkcija strcpy može rabiti parametre s i t kako želi. Predstavimo prikladno incijalizirane pokazivače, nanizane duž polja, znak po znak sve dok se '\0' ne preslika iz t u s.

U praksi funkcija strcpy neće biti napisana kako je gore predočeno. Iskusniji C programeri radije će napisati

```
/* strcpy: preslikava polje t u polje s; druga inačica s pokazivačima */
void strcpy(char *s, char *t){
        while((*s++=*t++)!='\0')
                ;
}
```

Mada suštinski nema razlike među inačicama, ipak smo uvećavanje pokazivača prebacili u dio testiranja uvjeta za nastavak petlje. Vrijednost *t++ jest znak na koji je t pokazivao prije provjere uvjeta. Sufiks ++ mijenja t, kako je poznato, tek nakon testa. Taj isti znak pohranjuje se u "staroj" poziciji s prije njezine promjene (uvećavanja za jedan). Ovaj znak predstavlja vrijednost koja se nakon toga uspoređuje sa znakom '\0' radi kontrole petlje, pa se dade zaključiti da se polje t preslikava u polje s zaključno sa znakom '\0'

Konačno, uočite da je usporedba s '\0' bespotrebna jer je upitno jedino to da li je izraz unutar zagrada nula ili nije. Stoga će funkcija vjerojatno biti napisana kao

```
/* strcpy: preslikava polje t u polje s; treća inačica s pokazivačima */
void strcpy(char *s, char *t){
        while(*s++=*t++)
                ;
}
```

Mada ovaj pristup može izgledati prilično neobično, na ovu programersku finesu treba se naviknuti, jer ćete ga često susretati u C programima.

Funkcija strcpy iz standardne biblioteke (zaglavlje <string.h>) vraća konačan znakovni niz kao svoju funkcijsku vrijednost.

Druga funkcija koju ćemo proučiti jest strcmp(s,t) koja uspoređuje znakovne nizove s i t, te vraća negativnu vrijednost, nulu ili pozitivnu vrijednost za niz s manji, jednak ili veći od t, respektivno. Vrijednost se dobija oduzimanjem znakova na prvom znaku u kojem se nizovi s i t ne slažu.

```
/* strcmp: vrati >0 ako je s>t, =0 za s=t i <0 za s<t */

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

Napravimo li funkciju strcmp pomoću pokazivača

```
/* strcmp: vrati >0 ako je s>t, =0 za s=t i <0 za s<t */

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

Kako su ++ i -- prefiksi ili sufiksi, mogu se pojaviti i druge kombinacije od * i +, mada ne tako često. Npr.,

```
*--p
```

najprije smanji p nakon čega p pokazuje na znak ispred onog na koji je pokazivao ranije. Zapravo, izrazi

```
*p++=val;   /* gurni vrijednost varijable val na vrh stoga */

val =*--p;  /* istisni vrh stoga u varijablu val */
```

su standardne fraze za punjenje i pražnjenje stoga (pogledajmo dio 4.3)

Zaglavlje <string.h> sadrži deklaracije funkcija spomenutih u ovom dijelu, te još mnogo drugih funkcija iz standardne biblioteke koje rade s nizovima.

Vježba 5-3. Napišite pokazivačku inačicu funkcije strcat koju smo prikazali u Poglavlju 2. Prisjetimo se da funkcija strcat(s,t) preslikava niz t na kraj niza s.

Vježba 5-4. Napišite funkciju strend(s,t) koja vraća 1 ako se niz t pojavi na kraju niza s, a nulu za bilo koji drugi slučaj.

Vježba 5-5. Napišite inačice funkcija biblioteke strncpy, strncat i strncmp, koje rade s najviše n znakova u nizovima argumenata. Primjerice, strncpy(s,t,n) preslika najviše n znakova niza t u niz s. Ovo je detaljno prikazano u Dodatku B.

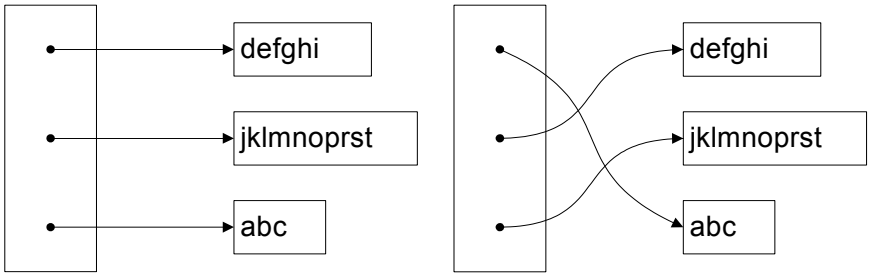
Vježba 5-6. Nanovo napišite odgovarajuće programe iz ranijih poglavlja i vježbi sa pokazivačima mjesto indeksa polja. Zanimljive mogućnosti imaju funkcija getline (Poglavlje 1 i Poglavlje 4), atoi, itoa te njihove varijante (Poglavlja 2, 3 i 4), reverse (Poglavlje 3), strindex i gretop (Poglavlje 4).

5.6 Pokazivači polja. Pokazivači na pokazivače

Kako su pokazivači varijable, oni se daju pohraniti u poljima kao i sve druge varijable. Ilustrirajmo tu činjenicu pisanjem programa koji će sortirati skup linija teksta po abecednom redu, kao što to radi Unix naredba sort.

U Poglavlju 3 smo predstavili funkciju shellsort koja sortira polje cijelih brojeva, a u Poglavlju 4 smo vidjeli funkciju quicksort koja ubrzava sortiranje. Vrijedit će, dakle, isti algoritmi, no sada radimo s linijama teksta različitih dužina, koji se ne mogu uspoređivati i prebacivati tek tako. Trebaju nam podaci koji će biti "zamjenske" vrijednosti linijama teksta varijabilne dužine.

Na ovom mjestu uvest ćemo pojam polja pokazivača. Ako su linije koje treba sortirati pohranjene u jednom velikom memorijskom bloku slijedno, tada svakoj liniji tj. prvom znaku svake linije možemo pristupiti s pomoću pokazivača. Sami pokazivači mogu se isto tako pohraniti u nekakvo polje. Dvije linije uspoređujemo prosljeđivanjem njihovih pokazivača u funkciju strcmp. Treba li dvije linije zamijeniti, mijenjamo samo pokazivače u polju pokazivača, a linije se zapravo ne diraju.



Ovakav pristup eliminira dvostruke probleme kompliciranog rukovanja memorijom i preopterećenja koje bi nastalo prebacivanjem linija.

Proces sortiranja obavlja se u tri koraka:

```
učitaj sve ulazne linije

sortiraj ih

ispiši ih po redu
```

Uobičajena je praksa podijeliti program u funkcije koje se uklapaju u tu prirodnu podjelu, s glavnom funkcijom koja ih poziva. Zaboravimo načas sortiranje i ispis, a usredotočimo se na strukturu podataka na ulazu i izlazu. Ulazna funkcija treba prikupiti i sačuvati znakove svake linije, pripadno polje pokazivača. Isto tako, ona mora izbrojiti ulazne linije, pošto je ta informacija potrebna za sortiranje i ispis. Još napomenimo da ulazna funkcija radi s konačnim brojem linija, pa će vratiti -1 kao nedopuštenu vrijednost ako je ulaz prevelik.

Izlazna funkcija mora ispisati linije po redoslijedu pojavljivanja u polju pokazivača.

```
#include <stdio.h>
#include <string.h>
#define     MAXLINES    5000  /* najveći broj linija za sortiranje */

char *lineptr[MAXLINES];      /* pokazivači na linije teksta */
int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);
void qsort(char *lineptr[], int left, int right);

main(){
      int nlines; /* broj ucitanih ulaznih linija */
      if((nlines=readlines(lineptr,MAXLINES))>=0){
            qsort(lineptr, 0, nlines-1);
            writelines(lineptr, nlines);
            return 0;
      }
      else{
            printf("error : input too big to sort\n");
            return 1;
      }
}

#define MAXLEN    1000  /* najveća dužina ulazne linije */

int getline(char *, int);
char *alloc(int);

/* readlines : učitava ulazne linije */
int readlines(char *lineptr[], int maxlines){
      int len, nlines;
      char *p, line[MAXLEN];
      nlines=0;
```

```
      while((len=getline(line, MAXLEN))>0)
            if(nlines>=maxlines||(p=alloc(len))==NULL)
                  return      -1;
            else{
                  line[len-1]='\0'; /* izbaci novu liniju */
                  strcpy(p, line);
            }
      return nlines;
}

/* writelines : ispisuje izlazne linije */
void writelines(char *lineptr[], int nlines){
      int i;
      for(i=0;i<nlines;i++)
            printf("%s\n", lineptr[i]);
}
```

Funkcija getline identična je onoj iz dijela 1.9.
Glavnu novinu predstavlja deklaracija za pokazivač lineptr:

```
char *lineptr[MAXLINES];
```

koja kaže da je lineptr polje za MAXLINES elemenata, pri čemu su elementi pokazivači na podatak tipa char. Dakle, lineptr[i] je pokazivač na znak, a *lineptr[i] znak na koji on pokazuje, odnosno prvi znak i-te pohranjene linije teksta.

Kako je lineptr ime polja, ono se može tretirati kao pokazivač, na isti način kao i u ranijim primjerima, funkcija writelines se dade napisati kao

```
void writelines(char *lineptr, int nlines){
      while(nlines-->=0)
            printf("%s\n", *lineptr++);
}
```

Na početku pokazivač lineptr pokazuje na prvu pohranjenu liniju, a svako uvećanje ga pomiče naprijed, na pokazivač slijedeće linije. Nasuprot tome, varijabla nlines odbrojava unazad.

Kad smo okončali kontrolu ulaza i izlaza možemo pristupiti sortiranju. Ubrzani program za sortiranje iz Poglavlja 4 zahtjeva male izmjene. Moramo modificirati deklaracije, operacija usporedbe mora se obaviti pozivanjem funkcije strcmp. Pseudokod ostaje isti, pa nema razloga sumnjati u korektno izvršenje programa.

```
/* qsort : sortiranje po rastućem redoslijedu */
void qsort(char *v[], int left, int right){
      int i, last;
      void swap(char *v[], int i, int j);
      if(left>=right)   /* ako polje ima više od 2 elementa */
            return;           /* ne treba ništa uraditi */
      swap(v, left, (left+right)/2);
      last=left;
      for(i=left+1;i<=right;i++);
            if(strcmp(v[i], v[left])<0)
                  swap(v, ++last, i);
      swap(v, left, last);
      qsort(v, left, last-1);
      qsort(v, last+1, right);
}
```

Slično ovomu, potprogram zamjene zahtjeva samo neznatne promjene

```
/* swap : obavlja zamjenu v[i] i v[j] */
void swap(char *v[], int i, int j){
      char *temp;
```

```
        temp=v[i];
        v[i]=v[j];
        v[j]=temp;
}
```

Kako svaki element polja v (ili lineptr) predstavlja znakovni pokazivač, tako moramo prikazati i varijablu temp jer preko nje obavljamo zamjenu.

Vježba 5-7. Napišite ponovo funkciju readlines da bi pohranili linije u polju koje je rezervirala funkcija main, što je bolje od poziva funkcije alloc. Koliko je taj način brži?

5.7 Višedimenzionalna polja

Programski jezik C određuje pravila za rukovanje višedimenzionalnim poljima, mada ih u praksi susrećemo mnogo rjeđe no polja pokazivača. Razmotrimo sada problem pretvorbe datuma. Primjerice, 1. ožujak 60. je dan u godini koja nije prestupna, a 61. dan prestupne. Definirajmo dvije funkcije koje će obaviti pretvorbu. Neka to budu funkcija day_of_the_year koja pretvara mjesec i dan u dan u godini, a funkcija month_day pretvara dan u godini u mjesec i dan. Kako ova druga funkcija računa dvije vrijednosti, argumenti za mjesec i dan bit će pokazivači

```
month_day(1996, 60, &m, &d)
```

postavlja m na 2, a d na 29 (29.veljače).

Objema funkcijama treba ista informacija, tablica broja dana u svakom mjesecu (rujan ima trideset dana). Uzevši u obzir da se broj dana u mjesecu razlikuje za prestupne godine i one koje to nisu, najlakše je problem riješiti odvajanjem tablica u dva reda dvodimenzionalnog polja. Polje i funkcije koje obavljaju pretvorbu jesu

```
static char daytab[2][13]={{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,
31}, {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}};
/* day_of_year : namjesti dan u godini ako je poznat dan u mjesecu */
int day_of_year(int year, int month, int day){
        int i, leap;
        leap=year%4==0&&year%100!=0||year%400==0;
        for(i=1;i<month;i++)
              day+=daytab[leap][i];
        return day;
}

/* month_day : namjesti dan u mjesecu ako je poznat dan u godini */
void month_day(int year, int yearday, int *pmonth, int *pday){
        int i, leap;
        leap=year%4==0&&year%100!=0||year%400==0;
        for(i=1;yearday>daytab[leap][i];i++)
              yearday-=daytab[leap][i];
        *pmonth=i
        *pday=yearday;
}
```

Podsjetimo se kako je aritmetička vrijednost logičkog izraza, poput onog za leap, ili 0 ili 1. Dakle, taj podatak možemo korisno uporabiti kao indeks polja daytab. Polje daytab mora biti vanjsko i za funkciju day_of_year i za funkciju month_day tako da se može koristiti u obje funkcije.

Prvo dvodimenzionalno polje sa kojim smo radili je polje daytab. U programskom jeziku C, dvodimenzionalno polje jest jednodimenzionalno polje, čiji je svaki element polje. Stoga su indeksi napisani kao

```
daytab[i][j]            /* [row][col] */
```

,a ne kao

```
daytab[i,j]             /* pogrešno */
```

Pored ove razlike u notaciji, dvodimenzionalno polje se može obrađivati kao i u drugim jezicima. Elementi se pohranjuju u redovima, tako da krajnji desni indeks varira najbrže ako pristupamo elementima po redu kojim su pohranjeni.

Polje se inicijalizira listom inicijalizatora unutar vitičastih zagrada, pri čemu se svaki red dvodimenzionalnog polja inicijalizira odgovarajućom podlistom. Počeli smo polje daytab stupcem u kojemu su vrijednosti 0 kako bi, zajedno s indeksima, brojevi koji označuju dane u mjesecu prirodno rasli od 1 do 12, a ne od 0 do 11.

Ako dvodimenzionalno polje treba prenijeti na funkciju, deklaracija parametara u funkciji mora imati i broj stupaca. Broj redova nije uopće bitan, jer ono što se prenosi jest pokazivač na polje redova u kojemu je svaki taj red polje od 13 cjelobrojnih vrijednosti (specijalno za ovaj naš slučaj). Stoga ako polje daytab treba prenijeti na funkciju f, deklaracija funkcije f će biti

```
f(int daytab[2][13]){
      ...
}
```

Mogla bi se ona i drugačije napisati

```
f(int daytab[][13]){
      ...
}
```

jer broj redova nije bitan. Postoji i ova mogućnost

```
f(int (*daytab)[13]){
      ...
}
```

koja jasno pokazuje da je parametar pokazivač na polje cjelobrojnih vrijednosti. Općenitije, samo prva dimenzija (ili indeks) polja je slobodna, dok druge moraju biti određene. U dijelu 5.12 nastavit ćemo pričati o složenim deklaracijama.

Vježba 5-8. U funkciji day_of_year ili month_day ne obavlja se nikakva provjera greške. Učinite sve kako bi se taj propust otklonio.

5.8 Inicijalizacija pokazivača polja

Promotrimo problem pisanja funkcije month_name(n), koja vraća pokazivač na znakovni niz s imenom mjeseca s rednim brojem n. Taj slučaj savršeno odgovara za primjenu internog polja tipa static. Funkcija month_name ima vlastito polje znakovnih nizova i vraća pokazivač na odgovarajući niz kad je pozovemo. Pogledajmo kako to polje i inicijalizirati, pri čemu je sintaksa vrlo slična prethodnim inicijalizacijama:

```
/* month_name : vraća ime n-tog mjeseca */
char *month_name(int n){
      static char *name[]={
            "Illegal month", "January", "February", "March", "April",
"May", "June", "July", "Avgust", "September", "October", "November", "December"
      };
      return (n<1||n>12) ? name[0] : name [n];
}
```

Deklaracija polja name, koja predstavlja polje znakovnih pokazivača jednaka je kao i deklaracija polja lineptr u programu sortiranja. Inicijalizator je lista znakovnih nizova, pri čemu je svaki pridijeljen odgovarajućoj poziciji u polju. Znakovi i-tog niza su negdje smješteni, a njihov pokazivač je pohranjen u polju name[i]. Kako veličina polja nije točno naznačena, prevoditelj broji inicijalizatore, te popunjava točan broj.

5.9 Pokazivači na višedimenzionalna polja

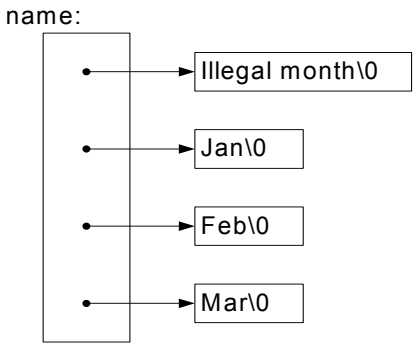
Novaci u programskom jeziku C ponekad su zbunjeni razlikom između dvodimenzionalnog polja i polja pokazivača kakvo je name u prikazanom primjeru. U definicijama

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

i a[3][4] i b[3][4] su dopuštene konstrukcije. No, a je pravo dvodimenzionalno polje kod kojeg je 10x20=200 memorijskih lokacija za cjelobrojne podatke (int) rezervirano, te se uobičajeni pravokutni izračun indeksa (20 x red + stupac) upotrebljava radi pronalaska elementa a[red, stupac]. Za polje b, međutim, definicija pridjeljuje samo deset pokazivača bez ikakve inicijalizacije. To moramo obaviti eksplicitno, bilo statički, bilo uz pomoć koda. Ako pretpostavimo da svaki element polja b pokazuje na polje dužine dvadeset elemenata, bit će onda rezervirano 200 lokacija za cijele brojeve, ali i deset lokacija za pokazivače. Bitna prednost polja pokazivača je što redovi polja mogu biti različitih dužina. Dakle, svaki element polja b ne mora pokazivati na polje od dvadeset elemenata, nego je taj broj praktično proizvoljan.

Mada smo u prethodnom razmatranju pričali o poljima cijelih brojeva, najčešća uporaba polja pokazivača jest pohrana znakovnih nizova različitih duljina, kao u funkciji month_name. Usporedite deklaraciju i sliku polja pokazivača

```
char *name[]={"Illegal month", "Jan", "Feb", "Mar"};
```



sa onom za dvodimenzionalno polje

```
char aname[][15]={"Illegal month", "Jan", "Feb", "Mar"};
```

a n a m e :

I l l e g a l m o n t h \0	J a n \0	F e b \0	M a r \0
0	1 5	3 0	4 5

Vježba 5-9. Napišite nanovo funkcije day_of_year i month_day sa poljem pokazivača umjesto indeksiranja.

5.10 Argumenti naredbene linije

U tipičnim okruženjima za programski jezik C, postoji način prenošenja argumenata naredbene linije ili parametara u program. Poziv funkcije main obavljamo s dva argumenta. Prvi (uobičajeno nazvan argc, kao argument count, za pobrojavanje argumenata) predstavlja broj argumenata naredbene linije kojom je program pozvan. Drugi (nazvan argv, kao argument vector, za pokazivač argumenata) predstavlja pokazivač na polje znakovnih nizova gdje su zapisani argumenti, po jedan u svakom znakovnom nizu. Kako smo i navikli, koristit ćemo više razina pokazivača za manipulaciju ovim argumentima.

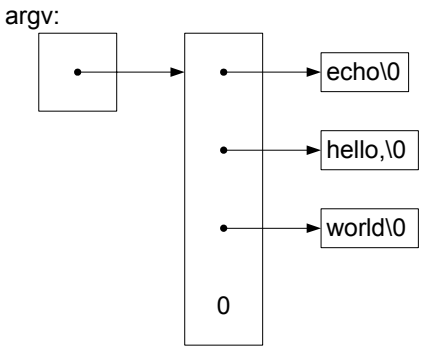
Najjednostavniji primjer dosad rečenog jest program echo, koji svoje argumente naredbene linije odvojene razmacina ispisuje u samo jednoj liniji. Pa će tako naredba

```
echo hello, world
```

ispisati

```
hello, world
```

Po dogovoru, element argv[0] je ime pomoću kojeg je program pozvan, tako da varijabla argc ima najmanju vrijednost 1. Ako je varijabla argc uistinu 1, tada zapravo nema argumenata u naredbenoj liniji. Konkretno, u gornjem primjeru varijabla argc ima vrijednost 3, dok su vrijednosti za elemente argv[0], argv[1] i argv[2] "echo", "hello" i "world" respektivno. Prvi uvjetni argument jest argv[1], a posljednji argv[argc-1]. Pored toga, standard postavlja zahtjev da element argv[argc] bude nulti pokazivač (null-pointer).



Prva inačica progama echo tretira polje argv kao polje znakovnih pokazivača

```
/* echo : argumenti naredbene linije, prva inačica */
#include <stdio.h>

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

Kako je argv pokazivač na polje pokazivača, možemo manipulirati s pokazivačem, umjesto indeksiranja polja. Slijedeća varijacija na ovu temu zasniva se na uvećanju vrijednosti pokazivača argv, koji pokazuje na podatak tipa char, dok varijablu argc smanjujemo i koristimo kao brojač.

```
/* echo : argumenti naredbene linije, druga inačica */
#include <stdio.h>

main(int argc, char *argv[]){
      while(--argc>0)
            printf("%s%s", *++argv, (argc>1) ? " " : "");
      printf("\n");
      return 0;
}
```

Jer je argv pokazivač na početak polja znakovnih nizova - argumenata, uvećanjem (++argv) on pokaže na prvi argument argv[1] umjesto dotadašnjeg elementa - naredbe, argv[0]. Svako slijedeće uvećanje miče ga na slijedeći argument , pri čemu je *argv pokazivač na taj argument. U isto vrijeme, smanjujemo varijablu argc, te kad ona postane jednaka nuli prekidamo petlju jer više nema argumenata za ispis.

Kao zamjenu, možemo napisati printf naredbu kao

```
printf((argc>1) ? "%s " : "%s", *++argv);
```

Ovaj primjer jasno pokazuje da argument printf naredbe može biti i izraz.

U drugom primjeru, unaprijedit ćemo program za traženje uzorka iz dijela 4.1. Ako se sjećate, umetnuli smo traženi uzorak unutar izvornog koda, što svakako nije baš najsretnije rješenje. Rabeći klasični

UNIX utility program grep, izmijenit ćemo program tako da traženi uzorak bude određen prvim argumentom naredbene linije.

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

#define     MAXLINE     1000

int getline(char *line, int max);

/* find : ispisi linije koje određuje uzorak iz prvog argumenta */
main(int argc, char *argv[]){
      char line[MAXLINE];
      int found=0;
      if(argc !=2)
            printf("Usage : find patern\n");
      else
            while(getline(line, MAXLINE)!=0)
                  if(strstr(line, argv[1])!=NULL){
                        printf("%s", line);
                  }
      return found;
}
```

Funkcija iz standardne biblioteke strstr(s, t) vraća pokazivač na prvo pojavljivanje niza t u nizu s ili vrijednost NULL ako u nizu s ne postoji niz t. To je deklarirano u zaglavlju <string.h>. Ovaj model dade se doraditi kako bi prikazali daljnju konstrukciju pokazivača. Prispodobimo kako želimo dopustiti pojavu dva opcijska argumenta (switch). Jedan služi za ispis svih linija koje nemaju uzorak, a drugi za ispis rednog broja linije ispred svake ispisane linije.

Zajednički dogovor za C programe na UNIX sistemima je ta da argument koji počinje sa negativnim predznakom uvodi jednu uvjetnu oznaku ili parametar. Ako izaberemo -x da označimo inverziju ("sve osim linija koje imaju uzorak") i -n za ispis rednog broja linija tada će naredba

```
find -x -n uzorak datoteka
```

ispisati svaku liniju koja nema uzorak, a ispred nje njezin redni broj u datoteci.

Treba dopustiti da opcijski argumenti budu poredani po bilo kojem slijedu, a ostali dio programa treba biti nezavisan o broju upisanih argumenata. Štoviše, korisnicima odgovara ako se opcijski argumenti mogu kombinirati kao npr.,

```
find -nx uzorak datoteka
```

Evo programa:

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

#define     MAXLINE     1000

int getline(char *line, int max);

/* find : ispisi linije koje imaju uzorak iz prvog argumenta */
main(int argc, char *argv[]){
      char line[MAXLINE];
      long lineno=0;
      int c, except=0, number=0, found=0;
      while(--argc>0&&(*++argv[0]=='-'))
            while(c==*++argv[0])
                  switch(c){
                        case 'x':
                              except=1;
```

```
                        break;
                    case 'n':
                        number=1;
                        break;
                    default:
                        printf(" find : illegal option %c\n", c);
                        argc=0;
                        found=-1;
                        break;
                }
            if(argc!=1)
                printf("Usage : find -x -n pattern\n");
            else
                while(getline(line, MAXLINE)>0){
                    lineno++;
                    if((strstr(line, *argv)!=NULL)!=except){
                        if(number)
                            printf("%ld:", lineno);
                        printf("%s", line);
                        found++;
                    }
                }
            return found;
}
```

Funkcija smanjuje varijablu argc, a uvećava pokazivač argv prije svakog opcijskog argumenta. Na završetku prolaza kroz petlju varijabla argc pokazuje koliko je argumenata ostalo neiščitano, dok je argv pokazivač na prvi od njih. Stoga, varijabla argc treba biti 1, a *argv treba pokazivati uzorak. Primijetite da je *++argv pokazivač na argument niza, pa je (*++argv)[0] njen prvi znak (Imamo i drugi način - **++argv). Zbog činjenice što uglata zagrada ([]) kao operator ima veći prioritet negoli * ili ++, mala je zagrada obvezna. Bez nje, izraz bi prevoditelj shvatio kao *++(argv[]). Nimalo slučajno, baš takav izraz upotrijebili smo u unutrašnjoj petlji, gdje smo vršili provjeru opcijskih argumenata. U unutrašnjoj petlji izraz izraz *++argv[0] uvećava pokazivač argv[0]!

Rijetko se događa da su izrazi s pokazivačima kompliciraniji od ovog primjera. Za takve slučajeve, potrebno je prilagoditi algoritme tj., raščlaniti posao na dijelove.

Vježba 5-10. Napišite funkciju expr, koja izračunava izraz pisan inverznom poljskom notacijom u naredbenoj liniji, u kojoj je svaki operator ili operand odvojeni argument. Primjerom,

```
expr 2 3 4 + *
```

```
računa 2*(3+4).
```

Vježba 5-11. Izmijenite funkcije entab i detab (napisane za vježbu u Poglavlju 1) radi prihvaćanja liste tabulatora kao argumenata. Upotrijebite tabulatorsko namještanje koje se predmnijeva ako nema argumenata.

Vježba 5-12. Proširite funkcije entab i detab radi prihvaćanja notacije

```
entab -m +n
```

što bi značilo da se tabulator zaustavlja svakih n stupaca, polazeći od stupca m. Prilagodite program korisniku (user-friendly).

Vježba 5-13. Napišite funkciju tail, koja ispisuje zadnjih n linija sa ulaza (ovo je standardna naredba svakog UNIX sustava). Pretpostavite kako je n, primjerice, 10, ali se to da izmijeniti opcijskim argumentom, pa će

```
tail -n
```

ispisati zadnjih n linija. Program bi trebao obavljati svoj posao bez obzira na (eventualno) apsurdne ulazne vrijednosti ili vrijednosti za argument n. Napišite program koji će najefikasnije koristiti raspoloživu memoriju (da budemo jasniji treba linije smještati kao u programu za sortiranje u dijelu 5.6, a ne u dvodimenzionalnom polju zadane dužine.

5.11 Pokazivači na funkcije

U programskom jeziku C sama funkcija nije promjenjiva, ali je moguće definirati pokazivače na funkcije koji mogu biti pridijeljeni, pohranjeni u polja, preneseni u funkcije i vraćeni iz njih. Ovo ćemo predočiti izmjenom postupka sortiranja koji je napisan ranije u ovom poglavlju, pa će, za uključeni opcijski argument -n, sortiranje biti obavljeno brojevno, a ne abecedno.

Sortiranje se, promatramo li pseudokod, obično sastoji od tri dijela. Tu je usporedba koja određuje poredak bilo kojeg para podataka, izmjena koja obrće njihov redoslijed, te algoritam sortiranja koji pravi usporedbe i izmjene sve dok se podaci ne urede. Algoritam sortiranja nezavisan je o operacijama usporedbe i izmjene, tako da možemo, izmjenom tih funkcija, omogućiti sortiranje po različitim kriterijima. To je pristup kojeg ćemo koristiti u još jednoj inačici programa za sortiranje.

Abecedna usporedba dvaju redova obavlja se pomoću funkcije strcmp, kao i prije. Bit će nam potrebna, međutim, i funkcija numcmp koja uspoređuje dvije linije na osnovu njihove brojčane vrijednosti, a vraća iste podatke kao i funkcija strcmp. Ove funkcije se deklariraju prije funkcije main, a pokazivač na odgovarajuću funkciju se prenosi u funkciju qsort.

Nećemo sad previše riječi trošiti na pogrešnu predaju argumenata kako bi se koncentrirali oko bitnih stvari.

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

#define     MAXLINES 5000     /* maksimalni broj linija za sortiranje */

char *lineptr[MAXLINES];      /* pokazivači na linije teksta */
int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);
void qsort(void *lineptr[], int left, int right, int (*comp)(void *, void
*));
int numcmp(char *, char *);   /* sortira ulazne linije */

main(int argc, char *argv[]){
      int nlines; /* broj učitanih ulaznih linija */
      int numeric=0;    /* 0 za abecedno, 1 za brojevno sortiranje */

      if(argc>1&&strcmp(argv[1], "-n")==0)
            numeric=1;
      if((nlines=readlines(lineptr, MAXLINES))>=0){
            qsort((void **)lineptr, 0, nlines-1, (int (*)(void *, void
*))(numeric ? numcmp : strcmp));
            writelines(lineptr, nlines);
            return 0;
      }
      else{
            printf("input too big to sort\n");
            return 1;
      }
}
```

U pozivima funkcija qsort, strcmp i numcmp jesu adrese tih funkcija. Kako znamo da se radi o funkcijama, operator & nije potreban, kao ni prije imena polja.

Napisali smo funkciju qsort koja može predati bilo koji tip podatka, a ne samo znakovne nizove. Prototipom funkcije dali smo do znanja da funkcija qsort očekuje polje pokazivača, dva cijela broja i funkciju s dva pokazivača kao argumente. Tu se rabi pokazivač tipa void * koji je zanimljiv po tome što pokazivač bilo kojeg tipa može biti model za tip void *, ne gubeći vrijednost pri povratku iz funkcije. Poboljšani model

argumenata funkcije određuje argumente funkcije za usporedbu. Zapravo, njih ne treba posebno proučavati, jer će ionako prevođenjem biti obavljena provjera.

```
/* qsort : sortira v[lijevo], ... v[desno] po rastućem redoslijedu */

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

      if(left==right)   /* ako polje ima više od 2 elementa */
            return;     /* ne treba uraditi ništa */
      swap(v, left, (left+right)/2);
      last=left;
      for(i=left+1;i<=right;i++)
            if((*comp)(v[i], v[left])>0)
                  swap(v, ++last, i);
      swap(v, left, last);
      qsort(v, left, last-1, comp);
      qsort(v, last+1, right, comp);
}
```

Deklaracije treba pažljivo proučiti. Četvrti argument funkcije qsort je

```
int (comp *)(void *, void *)
```

što kazuje da je comp pokazivač na funkciju koja ima dva void * argumenta, a koja vraća cjelobrojnu vrijednost.

Uporaba pokazivača comp u liniji

```
if((*comp)(v[i], v[left])>0)
```

slična je deklaraciji. comp je pokazivač na funkciju, *comp je funkcija, a

```
(*comp)(v[i], v[left])
```

jest njezin poziv. Male su zagrade i ovdje obvezne, kako bi elementi izraza bili pravilno povezani. Inače,

```
int *comp(void *, void *)     /* pogrešno */
```

kaže da je comp funkcija koja vraća cijeli broj, a to nije egzaktno.

Već smo pokazali funkciju strcmp, koja uspoređuje dva znakovna niza. Tu je i funkcija numcmp za usporedbu dva niza po prvoj brojčanoj vrijednosti, koja se dobija izlaznom vrijednošću funkcije atof.

```
#include <math.h>

int numcmp(char *s1, char *s2){
      double v1, v2;
      v1=atof(s1);
      v2=atof(s2);
      if(v1<v2)
            return -1;
      else
            if(v1>v2)
                  return 1;
            else
                  return 0;
}
```

Funkcija swap, koja zamjenjuje dva pokazivača, identična je ranije predstavljenoj u ovom poglavlju, osim činjenice da su deklaracije promijenjene u void *.

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

Veliki je broj opcija kojima se ovaj programčić za sortiranje može proširiti. Neke su kao stvorene za vježbu.

Vježba 5-14. Izmijenite program za sortiranje dodajući mu mogućnost rada sa opcijskim argumentom -r čime sortiranje postaje u obrnutom slijedu. Uvjerite se da -r radi sa -n.

Vježba 5-15. Dodajte opcijski argument -f kojim program za sortiranje ne pravi razliku od malih i velikih slova.

Vježba 5-16. Dodajte -d "poredak direktorija", opcijski argument koji uspoređuje samo slova, brojeve i razmake. Provjerite da li radi zajedno sa -f.

Vježba 5-17. Dodajte mogućnost baratanja poljem, kako bi se sortiranje moglo obaviti na poljima između linija, da svako polje bude sortirano u nezavisan skup opcijskih argumenata (indeks za ovu knjigu je sortiran sa -df za kategoriju indeksa i -n za brojeve strana).

5.12 Složene deklaracije

Programski jezik C doživljava kritike zbog sintakse njegovih deklaracija, posebice onih koje uvode pokazivače na funkcije. Sintaksa mora ujediniti deklariranje i način uporabe, što dobro prolazi kod jednostavnih slučajeva. Međutim, kod kompleksnijih problema, sve to djeluje konfuzno. Što zbog činjenice da se deklaracije ne mogu očitati slijeva nadesno, što zbog prečestog korištenja zagrada. Razlika između

```
int *f();         /* f :      funkcija čija je ulazna vrijednost
pokazivač, a izlazna tipa int */
```

i

```
int (*pf)();      /* pf : pokazivač na funkciju koja vraća cjelobrojnu
vrijednost */
```

zorno ilustrira problem: * je prefiksni operator, te je manjeg prioriteta od malih zagrada (), tako da su one obvezne radi pravilnog povezivanja.

Iako se kompleksne deklaracije rijetko pojavljuju u praksi, bitno je njihovo razumijevanje, a napose i primjena. Dobar način za stvaranje tih deklaracija jest pomoću ključne riječi typedef, a o čemu će biti govora u dijelu 6.7. Kao alternativu, predočit ćemo par programa koji obavljaju pretvorbu iz C programa u pseudokod i obratno. Pseudokod se čita slijeva udesno.

Prvi program, dcl, je nešto složeniji. On pretvara deklaracije u opisne riječi kao npr.

```
char **argv
      argv : pokazivač na pokazivač na char
int (*daytab)[13]
      daytab : pokazivač na polje[13] od int
int *daytab[13]
      daytab : polje[13] pokazivača na int
void *comp()
      comp : funkcija čija je ulazna vrijednost pokazivač, a izlazna void
void (*comp)()
      comp : pokazivač na funkciju koja vraća void
char(*(*x())[])()
      x : funkcija koja vraća pokazivač na polje[] pokazivača na funkciju
koja vraća char
```

```
char(*(*x[]))()[]
      x : polje[3] pokazivač na funkciju koja vraća pokazivač na polje[5]
od char
```

Program dcl temelji se na gramatici koja određuje deklarator. Ovo je njegov pojednostavljeni oblik

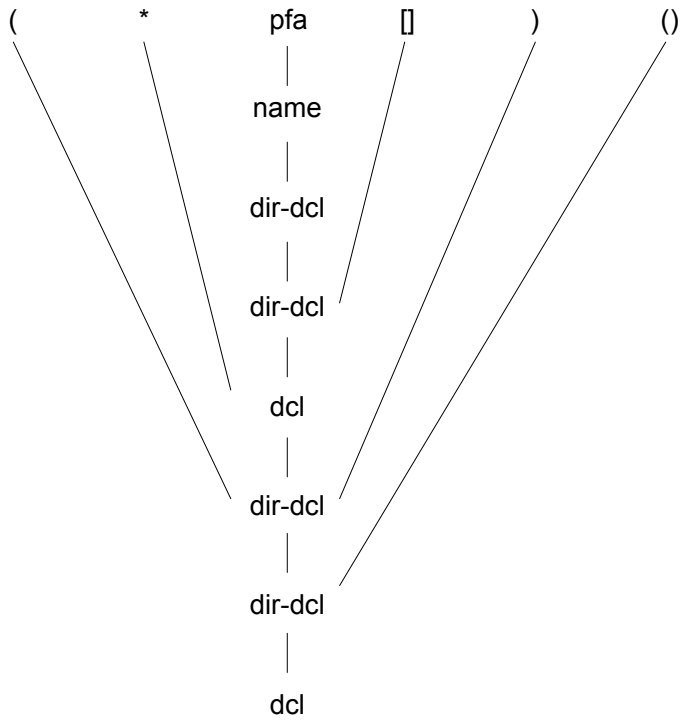
```
dcl:  opcijska oznaka pokazivača    naziv
naziv:      ime
      (dcl)
      naziv()
      naziv[opcijska veličina]
```

Riječima, deklarator dcl jest naziv kojemu može prethoditi *. naziv, s druge strane, može biti ime, deklarator u malim zagradama, naziv ispred malih zagrada ili naziv ispred uglatih zagrada s opcijskom veličinom.

Ova se gramatika može iskoristiti pri analizi deklaracija. Uzmimo tako, primjera radi, ovaj deklarator

```
(*pfa[])()
```

Deklarator pfa će biti identificiran kao ime, što znači i kao naziv. Tada je pfa[] također naziv, pa tako i *pfa[], zatim (*pfa[]), a naposljetku i (*pfa[])(). Ovo analiziranje (inače ovakav pristup uobičajen je kod pisanja prevoditelja, za sintaksnu analizu) dade se ilustrirati slijedećim crtežom



Osnovnu zamisao dcl programa čini nekoliko funkcija, konkretno dcl i dirdcl, koje analiziraju deklaraciju po prikazanoj gramatici. Gramatika je pisana rekurzivno, pa se funkcije, da se tako izrazimo, same pozivaju, kako je to kod rekurzije i uobičajeno.

```
/* dcl : analizira deklarator */
void dcl(void){
      int ns;
      for(ns=0;gettoken()=='*';)    /* zbraja pokazivače */
            ns++;
      dirdcl();
      while(ns-->0)
            strcat(out, "pointer to");
```

```
}

/* dirdcl : analizira naziv */
void dirdcl(void){
      int type;
      if(tokentype=='('){     /* da li je dcl()? */
            dcl();
            if(tokentype!=')')
                  printf("error : missing )\n");
      }
      else
            if(tokentype==NAME)     /* da li je ime varijable? */
                  strcpy(name, token);
            else
                  printf("error : expected name or (dcl)\n");
      while((type=gettoken())==PARENS||type==BRACKETS)
            if(type==PARENS)
                  strcat(out, " function returning");
            else{
                  strcat(out,"array");
                  strcat(out, token);
                  strcat(out, "of");
            }
}
```

Kako programi trebaju biti čitljivi, uveli smo značajne restrikcije u funkciji dcl. Ona može manipulirati samo jednostavnim tipovima podataka kakvi su npr. char i int, dok ne može raditi s kompleksnijim argumentima ili kvalifikatorima tipa const. Funkcija je osjetljiva na velike praznine, a teško otkriva greške, pa treba paziti i na nepravilne deklaracije. Dorada ovog programa neka vam bude dobra vježba.

Napišimo i globalne varijable, te glavne funkcije

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

#define     MAXTOKEN    100

enum NAME, PARENS, BRACKETS;
void dcl(void);
void dirdcl(void);
int gettoken(void);
int tokentype;    /* tip zadnjeg simbola */
char token[MAXTOKEN];   /* zadnji simbol u nizu */
char name[MAXTOKEN];    /* ime identifikatora */
char datatype[MAXTOKEN];      /* tip podatka = char, int, etc. */
char out[1000];   /* izlazni niz */

main(){
      while(gettoken!=EOF){   /* prvi simbol je tip podatka */
            strcpy(datatype, token);
            out[0]='\0';
            dcl();      /* analizira ostali dio linije */
            if(tokentype!='\n')
                  printf("syntax error\n");
            printf("%s : %s %s\n", name, out, datatype);
      }
      return 0;
}
```

Funkcija gettoken preskače razmake i tabulatore, da bi naišla na slijedeći ulazni simbol. Pri tome simbol može biti ime, par malih zagrada, par uglatih zagrada s (opcijski) brojevima ili pak bilo koji drugi pojedinačni znak.

```
int gettoken(void){
      int c, getch(void);
      void ungetch(int);
      char *p=token;
      while((c=getch())==' '||c=='\t')
            ;
      if(c=='('){
            if((c=getch())==')'){
                  strcpy(token, "()");
                  return tokentype=PARENS;
            }
            else{
                  ungetch(c);
                  return tokentype='(';
            }
      }
      else
            if(c=='['){
                  for(*p++=c;(*p++=getch())!=']';)
                        ;
                  *p='\0';
                  return tokentype=BRACKETS;
            }
            else
                  if(isalpha(c)){
                        for(*p++=c;isalnum(c=getch());)
                              *p++=c;
                        *p='\0';
                        ungetch(c);
                        return tokentype=NAME;
                  }
                  else
                        return tokentype=c;
}
```

O funkcijama getch i ungetch već smo govorili u Poglavlju 4.

Lakše je ići obrnutim smjerom, naročito ako nas ne zabrinjava generiranje suvišnih zagrada. Program undcl pretvara opis: "x je funkcija koja vraća pokazivač na polje pokazivača na funkcije koje vraćaju char" u ovakav izraz

```
x () * [] () char

char (*(*x())[])()
```

Skraćena ulazna sintaksa omogućuje je nam ponovnu uporabu funkcije gettoken. Funkcija undcl isto tako koristi vanjske varijable kao i dcl.

```
/* undcl : pretvara pseudokod u deklaraciju */
main(){
      int type;
      char temp[MAXTOKEN];

      while(getttoken()!=EOF){
            strcpy(out, token);
            while((type=gettoken())!='\n')
                  if(type==PARENS||type==BRACKETS)
                        strcat(out, token);
```

```
                        else
                              if(type=='*'){
                                    sprintf(temp, "(*%s)", out);
                                    strcpy(out, temp);
                              }
                              else
                                    if(type==NAME){
                                          sprintf(temp, "%s %s", token, out);
                                          strcpy(out, temp);
                                    }
                                    else
                                          printf("invalid input at %s\n",
token);
                  printf("%s\n", out);
            }
            return 0;
      }
```

Vježba 5-18. Prepravite program dcl da bude otporan na ulazne pogreške.

Vježba 5-19. Izmijenite program undcl kako ne bi pripajao suvišne zagrade deklaracijama.

Vježbe 5-20. Proširite program dcl, tj., dodajte mogućnost rada s deklaracijama s kompleksnim argumentima i kvalifikatorima tipa const, itd.

Poglavlje 6: STRUKTURE

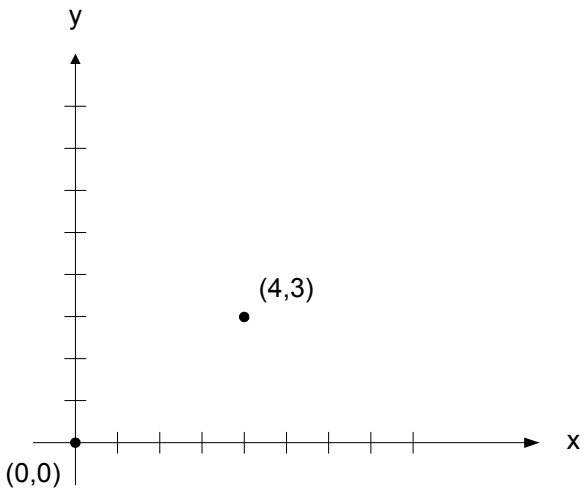
Struktura jest skup jedne ili više varijabli, koje mogu biti različitih tipova, grupiranih zajedno radi lakše manipulacije (ovakav oblik podataka susreli smo u Pascalu, gdje se nazivaju RECORD ili, prevedeno, zapisi). Strukture pomažu pri organizaciji kompleksnih podataka, posebno u velikim programima, jer one omogućuju obradu grupe međusobno povezanih varijabli kao jedne cjeline.

Tradicionalan primjer strukture je stvaranje platnog spiska: zaposlenik je opisan skupom atributa tipa imena, adrese, broja socijalnog osiguranja, plaće i sl. Neki od ovih atributa mogu se, također, predstaviti preko struktura jer se mogu sastojati od više komponenti. Drugi primjer, još znakovitiji za C, vidi se na slici. Točka je definirana uređenim parom koordinata, pravokutnik se (barem dio njih) dade definirati parom točaka i sl.

Glavna izmjena koju definira ANSI standard jest u definiranju strukture dodjeljivanja. Strukture mogu biti kopirane, a zatim dodijeljene, prenesene do funkcija i vraćene pomoću funkcija. Sve to su prevoditelji podržavali dugi niz godina, no sada su te osobine precizno definirane. Automatske strukture i polja sada također mogu biti inicijalizirane.

6.1 Osnovni pojmovi o strukturama

Kreirajmo nekoliko struktura koje su pogodne za grafički prikaz. Osnovni objekt je točka, za koju ćemo predmnijevati da ima x i y koordinatu, cijele brojeve.



Te dvije komponente dadu se smjestiti u strukturu koja je ovako deklarirana

```
struct point{
      int x;
      int y;
};
```

Ključna riječ struct upoznaje nas s deklaracijom strukture koja predstavlja listu deklaracija u vitičastoj zagradi. Opcijsko ime koje jest ime strukture prati ključnu riječ (u našem primjeru to je point). Ono imenuje ovu strukturu, te se kasnije dade uporabiti kao skraćenica za dio deklaracije u vitičastim zagradama.

Varijable deklarirane u strukturi nazivamo članovima. Član strukture ili ime i obična varijabla (koja nema veze sa strukturom) mogu nositi isto ime bez problema, jer se uvijek mogu razlikovati po kontekstu. Štoviše, ista imena članova mogu se pojaviti u različitim strukturama, ali je stilsko pitanje da treba rabiti ista imena samo kod tijesno vezanih objekata.

Deklaracija struct definira tip. Iza desne vitičaste zagrade kojom se završava lista članova može doći lista varijabli kao, uostalom, iza svakog drugog tipa. To znači,

```
struct {
      ...
} x, y, z;
```

jest sintaksno identično s

```
int x, y, z;
```

jer oba izraza deklariraju x, y, z kao varijable zadanog tipa, te stvaraju za njih mjesto.

Deklaracija strukture koja nije popraćena listom varijabli ne rezervira mjesto u memoriji nego samo opisuje vrstu ili oblik strukture. Međutim, ako je deklaracija obavljena, ona se može kasnije uporabiti u definicijama. Primjerice, zadana deklaracija strukture point u gornjem slučaju

```
struct point pt;
```

definira varijablu pt koja jest struktura tipa point. Struktura se dade inicijalizirati tako je popraćena listom inicijalizatora u kojoj je svaki od njih konstantan izraz članova strukture

```
struct point maxpt={320, 200};
```

Automatska struktura inicijalizira se i dodjelom ili pozivom funkcije koja vraća strukturu pravog tipa.

Članu određene strukture pristupa se slijedećim izrazom

```
ime_strukture.član
```

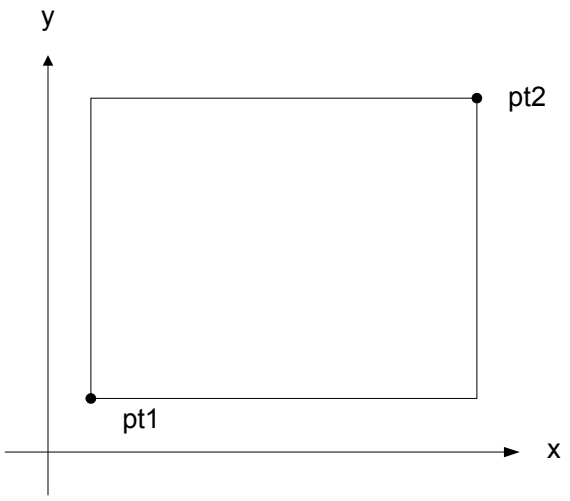
Operatorom "." povezujemo ime strukture i ime člana. Da bi se ispisale koordinate točke pt, npr.,

```
printf("%d, %d", pt.x, pt.y);
```

ili da bi se izračunala udaljenost od ishodišta do točke pt,

```
double dist, sqrt(double);
dist=sqrt((double)pt.x*pt.x+(double)pt.y*pt.y);
```

Strukture možemo ugnijezditi. Pravokutnik možemo predstaviti parom točaka koje označuju dijagonalno suprotne kutove.




```
struct rect{
      struct point pt1;
      struct point pt2;
};
```

Struktura rect ima dvije vlastite strukture point. Ako deklariramo strukturu screen kao

```
struct rect screen;
```

tada se izraz

```
struct.pt1.x
```

odnosi na x koordinatu, tj., na član pt1 strukture screen.

6.2 Strukture i funkcije

Jedine dopuštene operacije na strukturama jesu kopiranje i dodjeljivanje strukturi kao cjelini, označujući joj adresu operatorom & i pristupajući njenim članovima. Kopiranje i dodjeljivanje uključuju prenošenje argumenata na funkcije kao i vraćanje vrijednosti od strane funkcija. Strukture se na mogu uspoređivati. Struktura se dade inicijalizirati nizom članova konstantnih vrijednosti, a automatska struktura se može također inicijalizirati dodjeljivanjem.

Ispitajmo strukture opisujući funkcije za rad s točkama i trokutima.

Postoje različiti prilazi ovom problemu. Ili ćemo prenositi članove strukture posebno ili čitavu strukturu, a možemo prenositi pokazivače na strukturu. Svako od nabrojanih mogućih rješenja ima dobre i loše strane. Prva funkcija, makepoint, uporabit će dva cijela broja, te vratiti strukturu point

```
/* makepoint : stvara točku od x i y komponente */
struct point makepoint(int x, int y){
            struct point temp;
            temp.x=x;
            temp.y=y;
            return temp;
}
```

Primijetimo kako nema proturječnosti između imena argumenata i člana s istim imenom. Čak štoviše, uporaba istog imena naglašava njihovu međusobnu vezu.

Sada možemo uporabiti funkciju makepoint radi dinamičke inicijalizacije bilo koje strukture ili stvaranja strukture argumenata funkcija

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);
screen.pt1=makepoint(0, 0);
screen.pt2=makepoint(XMAX, YMAX);
middle=makepoint((scrren.pt1.x+scrren.pt2.x)/2,(screen.pt1.y+screen.pt2.y)
/2);
```

Slijedeći korak je skup funkcija koje primjenjuju linearnu algebru točaka. Primjerice,

```
/* addpoint : zbraja dva vektora */
struct point addpoint(){
      p1.x+=p2.x;
      p1.y+=p2.y;
      return p1;
}
```

Ovdje su strukture i argumenti funkcije i povratne vrijednosti. Zgodnije nam se čini uvećanje komponenti u član p1, nego uporaba eksplicitne privremene varijable koja bi istakla da su parametri strukture preneseni po vrijednosti.

Slijedeći primjer je funkcija ptinrect koja provjerava da li je točka unutar pravokutnika, uz pretpostavku da pravokutnik ima svoju lijevu i donju stranu, bez gornje i desne

```
/* ptinrect : vraća vrijednost 1 ako je točka p unutar pravokutnika, a 0
ako nije */
int ptinrect(struct point p, struct rect r){
      return p.x>=r.pt.x&&p.x<r.pt2.x&&p.y>=r.pt1.y&&r.pt2.y;
}
```

Ovo podrazumijeva da je pravokutnik prikazan u standardnom obliku koji uvjetuje da su koordinate člana pt1 manje od onih u članu pt2. Funkcija koju ćemo sada napisati vraća pravokutnik u kanonski oblik

```
#define     min(a, b)   ((a) (b) ? (a) : (b))
#define     max(a, b)   ((a) (b) ? (a) : (b))

/* canonrect : determinira koordinate pravog kuta */
struct rect canonrect(struct rect r){
      struct rect temp;
      temp.pt1.x=min(r.pt1.x, r.pt2.x);
      temp.pt1.y=min(r.pt1.y, r.pt2.y);
      temp.pt2.x=max(r.pt1.x, r.pt2.x);
      temp.pt2.y=max(r.pt1.y, r.pt2.y);
      return temp;
}
```

Ako veliku strukturu treba predati funkciji, zapravo je učinkovitije prenijeti pokazivač negoli preslikati cijelu strukturu (i time je umnožiti u memoriji). Pokazivači na strukture djeluju jednako pokazivačima na obične varijable. Deklaracija

```
struct point *pp;
```

govori da je pp pokazivač na strukturu tipa point.

Ako pp pokazuje na point strukturu, tada je *pp struktura, (*pp).x i (*pp).y su članovi. Pogledajmo slijedeći primjer

```
struct point origin, *pp;
pp=&origin;
printf("origin is (%d, %d)\n", (*pp).x, (*pp).y);
```

Mala zagrada je obvezna u konstrukciji (*pp).x, jer je prioritet strukture člana operatora "." veći od "*". Izraz *pp.x ima značenje kao *(pp.x) što nije dopušteno jer je x član, a ne pokazivač. Pokazivači na strukture se rabe prilično često, pa je uvedena i specijalna sintaksa radi jasnoće koda. Ako je p pokazivač na strukturu, tada se izraz

```
p->član_strukture
```

odnosi na taj član. Naredbu za ispis iz prethodnog primjera mogli smo, dakle, napisati kao

```
printf("origin is (%d, %d)\n", pp->x, pp->y);
```

I "." i "->" se pridružuju slijeva nadesno, pa tako ako imamo

```
struct rect r, *rp=r;
```

ova četiri izraza su ekvivalentna

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

Operatori struktura "." i "->", zajedno sa malim zagradama za pozive funkcija, te uglatim zagradama za indekse, jesu na vrhu hijerarhije prioriteta, pa su zato tijesno povezani. Npr. ako imamo deklaraciju

```
struct {
      int len;
      char *str;
} *p;
```

onda

```
++p->len;
```

uvećava vrijednost člana len, a ne strukture p, jer se predmnijeva zagrada ++(p->len). Zagradama se izraz dade izmijeniti. Tako izrazom (++p)->len uvećavamo strukturu p prije pridruživanja člana, a izrazom (p++)->len uvećavamo strukturu p nakon toga (pri čemu je zagrada nepotrebna).

Analogno tome, izraz *p->str jest vrijednost na koju član str pokazuje. Tako će izraz *p->str++ povećati vrijednost člana str po pristupanju tom članu (slično izrazu *s++), izraz (*p->str)++ uvećava sve ono na što član str pokazuje, te izraz *p++->str uvećava strukturu p nakon pristupanja podatku na koji pokazuje član str.

6.3 Polja struktura

Napišimo program koji broji pojavnosti svake ključne riječi programskog jezika C. Trebamo polje znakovnih nizova radi pohranjivanja imena i cijelih brojeva za zbrojeve. Jedna mogućnost jest uporaba dvaju paralelnih polja, imenima keyword i keycount, kao npr.,

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

, no sama činjenica paralelnih polja sugerira drugačiji pristup, tj., polje struktura. Svaki ulaz ključne riječi jest par

```
char *word;
int count;
```

, pa zato postoji polje parova. Deklaracija strukture

```
struct key{
      char *word;
      int count;
} keytab[NKEYS];
```

deklarira strukturu tipa key, definira polje struktura keytab ovog tipa i izdvaja za njih mjesto u memoriji. Svaki element polja jest struktura.

To bi se dalo napisati i ovako

```
struct key{
      char *word;
      int count;
};

struct key keytab[NKEYS];
```

Kako struktura keytab ima konstantan skup imena, najlakše je proglasiti je vanjskom varijablom i inicijalizirati je jednom zauvijek ako je definirana. Tu inicijalizaciju obavljamo kako smo već vidjeli - definiciju prati lista inicijalizatora u vitičastoj zagradi.

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

```
        "unsigned", 0,
        "void", 0,
        "volatile", 0,
        "while", 0,
};
```

Inicijalizatori su prikazani u parovima koji odgovaraju članovima strukture. Bilo bi preciznije predstaviti inicijalizatore za svaki par kao strukturu u velikoj zagradi kao

```
{"auto", 0},
{"", 0},
{"", 0},
...
```

imajući pri tom na umu da unutrašnja zagrada nije potrebna kad su inicijalizatori obične varijable ili znakovni nizovi, te kad su svi na broju. Naravno, broj objekata u polju struktura keytab bit će određen ako su inicijalizatori nabrojani i uglate zagrade ostavljene prazne.

Program za brojanje ključnih riječi počinje definicijom polja struktura keytab. Glavni potprogram očitava ulaz stalnim pozivanjem funkcije getword koja uzima slijedeću riječ ili znak sa ulaza. Svaka riječ traži se inačicom binarne funkcije pretraživanja o kojoj je bilo riječi u Poglavlju 3. Lista ključnih riječi zato mora biti poredana po rastućem redoslijedu u tablici.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define     MAXWORD     100

int getword(char *, int);
int binsearch(char *, struct key *, int);

/* program prebrojava ključne riječi programskog jezika C */
main(){
      int n;
      char word[MAXWORD];
      while(getword(word, MAXWORD)!=EOF)
            if(isalpha(word[0]))
                  if((n=binsearch(word, keytab, NKEYS))>=0)
                        keytab[n].count++;
      for(n=0; n<NKEYS; n++)
            if(keytab[n].count>0)
                  printf("%4d %s\n", keytab[n].count, keytab[n].word);
      return 0;
}
/* binsearch : pronalazi riječ u tablici */
int binsearch(char *word, struct key tab[], int n){
      int cond;
      int low, high, mid;
      low=0;
      high=n-1;
      while(low x<=high){
            mid=(low+high)/2;
            if((cond=strcmp(word, tab[mid].word))<0)
                  high=mid-1;
            else
                  if(cond>0)
                        low=mid+1;
                  else
                        return mid;
      }
      return -1;
}
```

Promotrimo na trenutak funkciju getword. Možemo napomenuti da funkcija getword pronalazi riječ koja se pohranjuje u polje istog imena kao njen prvi argument.

Veličina NKEYS jest broj ključnih riječi u strukturi keytab. Iako je ovo moguće i manuelno obaviti, puno je lakše prepustiti to računalu, naročito ako je listu potrebno mijenjati. Jedna mogućnost bila bi da na kraj liste inicijalizatora ostavimo nulti pokazivač, a zatim pretražimo strukturu keytab sve do kraja.

No, to i nije obvezno, jer je veličina strukture određena za vrijeme prevođenja. Veličina polja je veličina jednog ulaza pomnožena brojem ulaza, pa je broj ulaza

veličina mjesta rezerviranog za strukturu keytab/veličina strukture.

Programski jezik C osigurava unarni operator vremena prevođenja, nazvan sizeof, koji se može rabiti za izračun veličine tog objekta. Izrazi

```
sizeof objekt
```

i

```
sizeof(ime tipa)
```

daju cijeli broj jednak veličini navedenog objekta ili tipa izraženog u bitovima (preciznije, operator sizeof determinira pozitivnu vrijednost cijelog broja tipa size_t, definiranog u zaglavlju stddef.h). Objekt može biti varijabla, polje ili struktura. Ime tipa može biti ime osnovnog tipa kao što je int ili double, ali i izvedenog tipa kao npr., struktura ili pokazivač.

U našem slučaju, broj ključnih riječi jest veličina polja podijeljena veličinom jednog elementa. Ovo računanje rabimo u #define izrazu radi određivanja vrijednosti konstante NKEYS

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

Drugi način pisanja gornjeg izraza bio bi

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

, a njegova prednost jest u činjenici da za slučaj promjene tipa izraz ne zahtjeva izmjene.

Unarni operator sizeof se ne može rabiti u #if konstrukciji, jer preprocesor ne raščlanjuje imena tipova. Međutim, izraz unutar #define izraza se ne računa u preprocesoru, pa je operator ovdje dopušten.

Kažimo nešto o funkciji getword. Napisali smo općenitiju funkciju getword od potrebnog za ovaj program, no ni on nije kompliciran. Funkcija getword uvodi slijedeću "riječ" sa ulaza, pri čemu "riječ" može biti ili niz slova i znamenki koja počinje slovom ili jedan znak koji nije razmak (space, blank, pusti znak). Vrijednost funkcije je prvi znak riječi, ili EOF za kraj datoteke, te sami znak ako nije slovo.

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

Funkcija getword rabi funkcije getch i ungetch koje su opisane u Poglavlju 4. Kada dohvatimo neki znak, funkcija getword zaustavlja se jedan znak dalje. Pozivom funkcije ungetch taj se znak vraća na ulaz gdje iščekuje slijedeći poziv. Funkcija getword isto tako koristi funkciju getword radi preskakanja razmaka, funkciju isalpha radi identifikacije slova i znamenki. To su sve funkcije standardne biblioteke, iz zaglavlja <ctype.h>.

Vježba 6-1. Napisana verzija funkcije getword ne barata potcrtama, konstantama niza, komentarima, te preprocesorskim kontrolama na najbolji mogući način. Doradite je.

6.4 Pokazivači na strukture

Radi ilustracije pretpostavki vezanih uz pokazivače na strukture i polja struktura, napišimo program za pobrojavanje ključnih riječi još jednom, rabeći pokazivače umjesto indeksa polja.

Vanjsku deklaraciju polja struktura keytab ne treba mijenjati, ali funkcije main i binsearch zahtjevaju izmjene.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define     MAXWORD     100

int getword(char *, int);
struct key *binsearch(char *, struct key *, int);
/* pobrojava ključne riječi programskog jezika C, pomoću pokazivača */
main(){
      char word[MAXWORD];
      struct key *p;
      while(getword(word, MAXWORD)!=EOF)
            if(isalpha(word[0]))
                  if((p=binsearch(word, keytab, NKEYS))!=NULL)
                        p->count++;
      for(p=keytab;p<keytab+NKEYS;p++)
            if(p->count>0)
                  printf("%4d %s\n", p->count, p->word);
      return 0;
}

/* binsearch : pronalazi riječ u tablici */
struct key *binsearch(char *word, struct key *tab, int n){
      int cond;
      struct key *low=&tab[0];
      struct key *high=&tab[0];
      struct key *mid;
      while(low<high){
            mid=low+(high-low)/2;
            if((cond=strcmp(word, mid->word))<0);
                  high=mid;
            else
                  if(cond>0)
                        low=mid+1;
                  else
                        return mid;
      }
      return NULL;
}
```

Ovdje moramo neke stvari napomenuti. Naprije, deklaracija funkcije binsearch mora kazivati da vraća pokazivač na strukturu tipa struct key umjesto na cijeli broj. Ovo se deklarira kako u prototipu funkcije, tako i u samoj funkciji binsearch. Ako binsearch pronađe riječ, on vraća pokazivač na nju. Inače vraća vrijednost NULL.

Nadalje, sad elementima polja struktura keytab pristupaju pokazivači, što zahtjeva bitne izmjene funkcije binsearch.

Inicijalne vrijednosti za pokazivače low i high su početak i kraj promatrane tablice.

Računanje korijenskog elementa više nije trivijalno

```
mid=(low+high)/2  /* pogrešno */
```

jer zbrajanje pokazivača nije dopušteno. Kako je oduzimanje dopušteno, možemo se poslužiti trikom

```
mid=low+(high-low)/2
```

postavlja vrijednost pokazivača na element u sredini tablice.

Najbitnija izmjena jest ugađanje algoritma da ne stvara nedopušteni pokazivač ili ne pokuša pristupiti elementu izvan polja. Nekakve granice, dakle, predstavljaju adrese &tab[-1] i &tab[n]. Prva je konstrukcija nedopuštena, a nipoziv druge nije preporučljiv, mada deklaracija jezika garantira da će pokazivačka aritmetika koja uključuje prvi element izvan granica polja (odnosno element tab[n]), korektno raditi.

U funkciji main napisali smo

```
for(p=keytab;p<keytab+NKEYS;p++)
```

Ako je p pokazivač na strukturu, aritmetika primjenjena na njega uzima u obzir veličinu strukture, pa izraz p++ uvećava p za potrebnu vrijednost, a da bi se dohvatio slijedeći element polja struktura, dok uvjetni dio petlje služi samo kako bi okončao petlju u pravom trenutku.

Važno je naglasiti da veličina strukture nije zbroj veličina njenih članova. Zbog potrebe prilagođavanja različitim arhitekturama računala može doći do pojave "rupa" u strukturi (structure holes). Zato, npr., ako tip char zahtjeva jedan byte, a int četiri, struktura

```
struct{
      char c;
      int i;
};
```

može tražiti osam byteova, a ne pet, kako bi (uvjetno kazano) bilo za očekivati. U svakom slučaju, operator sizeof vratit će korektnu vrijednost.

Napokon, bacimo pogled na izgled programa. Kada funkcija vraća složeni tip kakav je pokazivač strukture

```
struct key *binsearch(char *word, struct key *tab, int n)
```

ime funkcije teško je uočljivo. Zato, katkad možemo to ovako pisati

```
struct key *
binsearch(char *word, struct key *tab, int n)
```

Sve to je, jasno, stvar osobnog ukusa. Bitno je odabrati formu koja vam se sviđa i pridržavati je se.

6.5 Samopozivajuće strukture

Pretpostavimo kako želimo riješiti općenitiji problem, tj., zbrajanje pojavljivanja svih riječi na nekom ulazu. Tablica s konačnim brojem ulaznih riječi, jasno, nije unaprijed poznata, mi je ne možemo posložiti na pravi način niti rabiti binarno pretraživanje. Isto tako, ne može se koristiti ni linearno pretraživanje jer bi s povećanjem broja riječi, program sve duže pretraživao (preciznije, vrijeme njegova odvijanja raste s kvadratom broja ulaznih riječi). Dakle, kako organizirati podatke radi uspješne obrade neograničene liste ulaznih riječi?

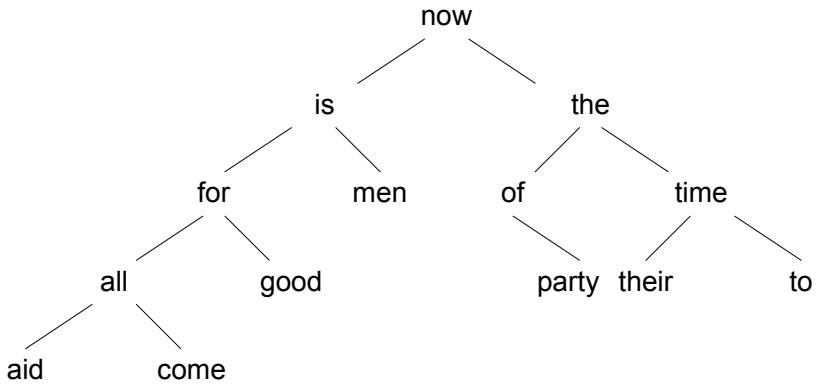
Jedno rješenje zadržati skup riječi koje su se već pojavljivale i bile sortirane, tako što ćemo ih postaviti na odgovarajuća mjesta po slijedu pojavljivanja. Ovdje se ne misli na prebacivanja unutar linearnog polja, jer je i ta operacija vremenski zahtjevna. U zamjenu, uporabit ćemo strukturu podataka koja se zove binarno stablo.

Stablo ima jedan čvor po jednoj riječi. Svaki čvor mora imati

- pokazivač na tekst riječi
- brojač broja pojavljivanja
- pokazivač na lijevu granu
- pokazivač na desnu stranu

Nijedan čvor ne može imati više od dvije grane. Čvorovi se tvore tako da u bilo kojem čvoru lijeva grana ima samo riječi leksički manje od riječi u čvoru, a desna samo veće.

Prikažimo stablo za rečenicu "now is the time for all of good men to come aid their party", koje je nastalo slaganjem riječi po slijedu nailaska.



Kako bi utvrdili je li nova riječ već u stablu, počet ćemo od korijena, te uspoređivati novu riječ sa riječima koje se već nalaze u stablu. Ako se slažu, odgovor je potvrdan. Ako se pak ne slažu, a nova riječ je manja od riječi u stablu, nastavit ćemo pretraživanje u lijevom ogranku. Ako je veća, tada u desnom ogranku. Ako nema više ogranaka u određenom smjeru, nova riječ nije u stablu, a prazan prostor ogranka je pravo mjesto za dodavanje nove riječi. Ovaj proces je rekurzivan jer se u pretraživanju bilo kojeg čvora obavlja pretraživanje jednog od njegovih ogranaka. Zbog toga će rekurzivne rutine umetanja i ispisa biti prirodnije.

Vraćajući se na opis čvora, on se lako dade predstaviti kao struktura s četiri komponente:

```
struct tnode{
      char *word; /* pokazuje na tekst */
      int count;  /* broj pojavljivanja */
      struct tnode *left;     /* lijevi ogranak */
      struct tnode *right;    /* desni ogranak */
};
```

Ova rekurzivna deklaracija čvora možda izgleda neobično, ali je korektna. Nije dopušteno da struktura sadrži sebe samu, ali

```
struct tnode *left;
```

deklarira left kao pokazivač na strukturu tnode, a ne sama struktura.

Katkad, bit će nam potrebna varijacija samopozivajućih struktura ; dvije strukture koje se međusobno pozivaju. Način ostvarivanja ovoga je

```
struct t{
      ...
      struct s *p;      /* p pokazuje na s */
};
struct s{
      ...
      struct t *q;      /* q pokazuje na t */
}
```

Kod za cijeli program je iznenađujuće malen, s mnogo dodatnih funkcija kao što je getword koja je već napisana. Glavni potprogram čita riječi pomoću funkcije getword i postavlja ih na stablo funkcijom addtree.

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

#define MAXWORD   100

struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int);

/* brojač frekvencije pojavljivanja riječi */
main(){
      struct tnode *root;
      char word[MAXWORD];

      root=NULL;
      while(getword(word, MAXWORD)!=EOF)
            if(isalpha(word[0]))
                  root=addtree(root, word);
      treeprint(root);
      return 0;
}
```

Funkcija addtree je rekurzivna. Riječ se predstavlja pomoću funkcije main na najvišem nivou (korijenski) stabla. Na svakom nivou, ta riječ se uspoređuje s riječi koja je već pohranjena u čvoru, te se filtrira u lijevi ili desni ogranak rekurzivnim pozivom funkcije addtree. Riječ se eventualno može poklopiti s nekom koja je već u stablu (u tom slučaju povećava se brojač) ili naiđe na nulti pokazivač, što govori da novi čvor treba biti kreiran i dodan u stablo. Ako se dodaje novi čvor, funkcija addtree ugađa pokazivač iz matičnog čvora (a koji je prije bio NULL) na novi čvor.

```
struct tnode *talloc(void);
char *strdup(char *);
/* addtree : dodaj čvor s w, na ili ispod p */
struct tnode *addtree(struct tnode *p, char *w)b{
      int cond;

      if(p==NULL){      /* naišli smo na novu riječ */
            p=talloc(); /* rezerviraj prostor za novi čvor */
            p->word=strdup(w);
            p->count=1;
            p->left=p->right=NULL;
      }
      else if((cond=strcmp(w, p->word))==0)
            p->count++; /* riječ koja se ponavlja */
      else if(cond<0)   /* manja nego u lijevom ogranku */
            p->left=addtree(p->left, w);
      else  /* veća nego u desnom ogranku */
            p->right=addtree(p->right, w)
      return p;
}
```

Memoriju za novi čvor osigurava funkcija talloc, koja vraća pokazivač na slobodan memorijski prostor za pohranu jednog čvora, a funkcija strdup kopira novu riječ na to mjesto. Uskoro ćemo proučiti i te funkcije. Brojač se inicijalizira, a dva ogranka postaju nula. Ovaj dio koda se obavlja samo na krajevima stabla, kad se dodan novi čvor. Preskočili smo (ne baš mudro), provjeru grešaka na vrijednostima koje vraćaju funkcije strdup i talloc.

Funkcija treeprint ispisuje stablo u sortiranoj formi; na svakom čvoru, ispisuje lijevi ogranak (sva riječi manje od ove riječi), zatim samu riječ, a potom desni ogranak (sve riječi koje su veće). Ako ne shvaćate kako rekurzija radi, simulirajte funkciju treeprint da radi na stablu prikazanom gore.

```
/* treeprint : slijedni ispis stabla p */
void treeprint(struct tnode *p){
      if(p!=NULL){
            treeprint(p->left);
            printf("%4d %s\n", p->count, p->word);
            treeprint(p->right);
      }
}
```

Praktična napomena: Ako stablo postane "neuravnoteženo" jer riječi ne dolaze slučajnom slijedom, vrijeme obavljanja programa može postati predugo. Kao najgori mogući slučaj je, ako su riječi već sortirane, da program obavlja preskupu simulaciju linearnog pretraživanja. Postoje generalizacije binarnog stabla koje rješavaju ovakove najgore moguće slučajeve, no ovdje ih nećemo opisivati.

Prije negoli ostavimo ovaj primjer, bilo bi dobro napraviti malu digresiju na problem vezan uz memorijske pridjeljivače. Očigledno je poželjno postojanje samo jednog memorijskog pridjeljivača u programu, čak i ako određuje različite vrste objekata. Ali ako jedan pridjeljivač (allocator) treba obaviti zadaću za, recimo, pokazivače na chars i pokazivače na struct tnodes, postavljaju se dva pitanja. Kao prvo, kako udovoljiti zahtjevima većine računala da objekti određenih tipova moraju zadovoljavati razna ograničenja (npr., cijeli brojevi često moraju biti smješteni na istim adresama)? S druge strane, koje deklaracije se mogu nositi s činjenicom da pridjeljivač nužno mora vratiti različite vrste pokazivača?

Različita ograničenja općenito se lako zadovoljavaju, po cijenu malo potrošenog prostora, osiguravši da pridjeljivač uvijek vraća pokazivač koji udovoljava svim ograničenjima. Funkcija alloc iz Poglavlja 5 ne garantira neki posebni raspored, pa ćemo koristiti funkciju standardne biblioteke malloc, koja to garantira. U poglavlju 8 pokazat ćemo na još jedan način korištenja funkcija malooc.

Pitanje deklaracije tipa za funkciju kakva je malloc nije ugodno za sve jezike u kojima se obavlja ozbiljna provjera tipa. U programskom jeziku C, pravilna metoda deklariranja je kad funkcija malloc vraća pokazivač na void, a zatim oblikuje pokazivač na željeni tip pomoću modela. Funkcija malloc i srodne rutine deklarirane su zaglavlju standardne biblioteke <stdlib.h>. Stoga se funkcija `talloc može pisati kao`

```
#include <stdlib.h>

/* talloc : pravi tnode */
struct tnode *talloc(void){
      return (struct tnode *) malloc(sizeof(struct tnode));
}
```

Funkcija strdup samo kopira niz znakova danu njenim argumentom na sigurno mjesto, osigurano pozivom funkcije malloc:

```
char *strdup(char *s){  /* make a duplicate of s */
      char *p;

      p=(char *) malloc(strlen(s)+1);     /* ovo +1 za  '\0' na kraju niza
*/
      if(p!=NULL)
            strcpy(p,s);
      return p;
}
```

Funkcija malloc vraća vrijednost NULL ako nema dostupnoga mjesta; funkcija strdup prenosi tu vrijednost dalje prepuštajući obradu greške pozivatelju. Memorija osigurana pozivom funkcije malloc dade se osloboditi za ponovnu upotrebu pozivom funkcije free; pogledajte Poglavlja 7 i 8.

Vježba 6-2. Napišite program koji čita C program i ispisuje po abecednom redu svaku grupu imena varijabli koja su identična u prvih šest znakova, a različita kasnije. Ne pobrojavajte riječi u znakovnim nizovima i komentarima. Učinite da je "šest" parametar koji se dade mijenjati sa ulazne linije.

Vježba 6-3. Napišite kazalo koje ispisuje listu svih riječi u dokumentu, te, za svaku riječ, listu brojeva linija u kojima se ona pojavljuje. Izbacite česte riječi poput veznika, priloga ili prijedloga.

Vježba 6-4. Napišite program koji na svom ulazu ispisuje riječi sortirane po padajućem slijedu učestalosti pojavljivanja. Svakoj riječi neka prethodi pripadajući broj.

6.6 Pretraživanje tablice

U ovom dijelu napisat ćemo sadržaj paketa za pretraživanje tablice radi ilustracije još nekih značajki struktura. Ovaj kod je tipičan primjer onoga što se nalazi u rutinama za upravljanje tablicama simbola makroprocesora ili prevoditelja. Primjerice, razmotrimo #define naredbu. Kad naiđemo na liniju tipa,

```
#define     IN    1
```

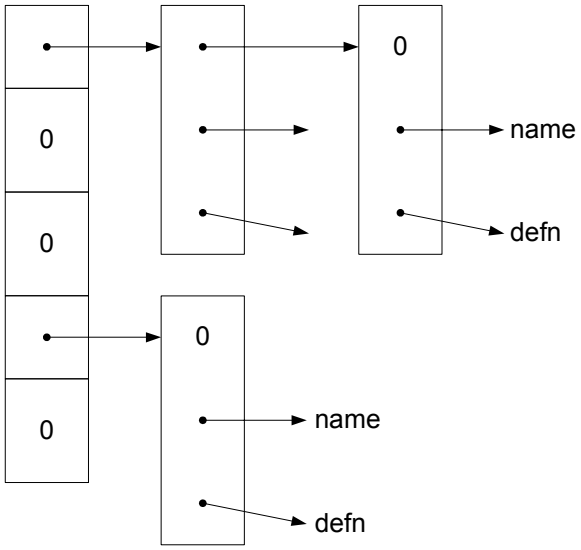
ime IN i zamjenski tekst 1 pohranjuju se u tablicu. Kasnije, kad se ime IN pojavi u izrazu kao

```
state=IN;
```

ono se mora zamijeniti s 1.

Postoje dvije rutine koje manipuliraju imenima i zamjenskim tekstovima. Funkcija install(s, t) zapisuje ime s i zamjenski tekst t u tablicu; s i t su znakovni nizovi. Funkcija lookup(s) traži ime s u tablici, te vraća pokazivač na mjesto gdje je ga je pronašla, a NULL ako ga u tablici nema.

Algoritam se zasniva na hash pretraživanju - ulazno ime se pretvara u mali nenegativni cijeli broj, koji se zatim rabi za indeksiranje u polju pokazivača. Element polja pokazuje na početak povezane liste blokova koji opisuju imena s tom vrijednošću. Ono je NULL ako nijedno ime nema tu vrijednost.



Blok u listi jest struktura koja sadrži pokazivače na ime , zamjenski tekst i slijedeći blok u listi. Slijedeći pokazivač na NULL označava kraj liste.

```
struct nlist{
      struct nlist *next;     /* slijedeći unos u lancu */
      char *name;
      char *defn;
};
```

Polje pokazivača je

```
#define     HASHSIZE    101
static struct nlist *hashtab[HASHSIZE];  /* tablica pokazivača */
```

Hash funkcija, korištena i u funkciji lookup i u funkciji install, dodaje vrijednost svakog znaka u nizu promiješanoj kombinaciji prethodnih i vraća ostatak modula veličine polja. Ovo nije idealna hash funkcija, ali je kratka i učinkovita.

```
/* hash : stvara hash vrijednost za niz s */
unsigned hash(char *s){
      unsigned hashval;

      for(hashval=0;*s!='\0';s++)
            hashval=*s+31*hashval;
      return hashval%HASHSIZE;
}
```

Aritmetika bez predznaka (unsigned) osigurava nenegativnu hash vrijednost.

Hashing proces stvara početni indeks u polju hashtab; ako je znakovni niz moguće pronaći, on će se nalaziti u listi blokova koja tu počinje. Pretraživanje obavlja funkcija lookup. Ako funkcija lookup pronađe unos koji već postoji, ona vraća pokazivač na njega; ako ne, vraća NULL vrijednost.

```
/* lookup : trazi niz s u polju hashtab */
struct nlist *lookup(char *s){
      struct nlist *np;
      for(np=hashtab[hash(s)];np!=NULL;np=np->next)
            if(strcmp(s, np->name)==0)
                  return np;  /* niz je pronađen */
      return NULL;      /* niz nije pronađen */
}
```

Petlja for u funkciji lookup jest standardna fraza za pretraživanje povezane liste:

```
for(ptr=head;ptr!=NULL;ptr=ptr->next)
      ...
```

Funkcija install koristi funkciju lookup kako bi odredila da li već postoji ime koje se uvodi; ako postoji, nova definicija mijenja staru. U protivnom, stvara se novi ulaz. Funkcija install vraća NULL ako iz bilo kojeg razloga nema mjesta za novi ulaz.

```
struct nlist *lookup(char *);
char *strdup(char *);

/* install : umetni(name, defn) u hashtab */
struct nlist *install(char *name, char *defn){
      struct nlist *np;
      unsigned hashval;

      if((np=lookup(name))==NULL){  /* nije pronađen niz */
            np=(struct nlist *) malloc(sizeof(*np));
            if(np==NULL||(np->name=strdup(name))==NULL)
                  return NULL;
            hashval=hash(name);
            np->next=hashtab[hashval];
            hashtab[hashval]=np;
      }
      else  /* već postoji */
            free((void *) np->defn);      /* oslobodi prethodni defn */
      if((np->defn=strdup(defn))==NULL)
            return NULL;
      return np;
}
```

Vježba 6-5. Napišite funkciju undef koja će ukloniti ime i definiciju iz tablice koju održavaju funkcije lookup i install.

Vježba 6-6. Primjenite jednostavnu verziju #define procesora (npr. bez argumenata) prikladnu za upotrebu u C programima, zasnovanu na rutinama iz ovog dijela. Možda će vam funkcije getch i ungetch biti od pomoći.

6.7 Typedef

Programski jezik C pruža olakšicu nazvanu typedef za kreiranje novih tipova podataka. Npr., deklaracija

```
typedef int Lenght;
```

učini da je Lenght sinonim za tip int. Tip Lenght može se rabiti u deklaracijama, modelima, itd., isto onako kao što se i tip int može:

```
Lenght len, maxlen;
Lenght *lenghts[];
```

Sličnu tomu, deklaracija

```
typedef char *String;
```

učini da je String sinonim za tip char * ili znakovni pokazivač, koji se potom može koristiti u deklaracijama i modelima:

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
p=(String) malloc(100);
```

Primjetite da se tip deklariran u typedef pojavljuje na mjestu imena varijable, a ne odmah iza ključne riječi typedef. Sintaksno, typedef je sličan memorijskim klasama extern, static, itd. Za typedef smo koristili imena napisana velikim slovima, čisto radi njihova isticanja.

Kao složeniji primjer, mogli bi napraviti typedef za čvorove stabla ranije definiranog u ovom poglavlju:

```
typedef struct tnode *Treeptr;

typedef struct tnode{
      char *word; /* pokazuje na tekst */
      int count;  /* broj pojavljivanja */
      Treeptr left;     /* lijeva grana */
      Treeptr right;    /* desna grana */
} Treenode;
```

Ovim se stvaraju dvije nove vrste ključnih riječi nazvane Treenode (struktura) i Treetr (pokazivač na strukturu). Tada funkcija talloc postaje

```
Treeptr talloc(void){
      return (Treeptr) malloc (sizeof(Treenode));
}
```

Mora se naglasiti da deklaracija typedef ne stvara novi tip; ona samo dodaje novo ime nekom postojećem tipu. Nema ni novih semantičkih varijabli; varijable ovako deklarirane imaju iste osobine kao i varijable čije su deklaracije jasno izražene. Zapravo, typedef sliči na #define, ali nakon što je obradi prevoditelj, ona može raditi s tekstualnim supstitucijama koje su veće od mogućnosti preprocesora. Npr.,

```
typedef int (*PFI) (char *, char *);
```

stvara tip PFI, kao "pokazivač na funkciju (sa dva char * argumenta) koja vraća int) (Pointer to Function returning Int), što se da upotrijebiti u kontekstima oblika

```
PFI strcmp, numcmp;
```

u programu za sortiranje iz Poglavlja 5.

Pored čisto estetskih razloga, dva su bitna razloga za upotrebu typedef. Prvi je parametriziranje programa zbog problema prenosivosti. Ako se typedef rabi za tipove podataka koji su zavisni o računalnom sustavu, tada treba samo izmijeniti typedef kad se program prenosi. Uobičajena je situacija koristiti typedef za različite cjelobrojne veličine, a zatim napraviti izbor od short, int i long prikladan za pojedino računalo. Takovi primjeri su tipovi oblika size_t i ptrdiff_t iz standardne biblioteke.

Druga je primjena typedef pružanje bolje dokumentacije program - tip naziva Treeptr razumljiviji je od onog koji je deklariran kao pokazivač na složenu strukturu.

6.8 Unije

Unija je varijabla koja može sadržavati (u različitim trenucima) objekte različitih tipova i veličina, prevoditeljem koji vodi računa o zahtjevima glede veličine i rasporeda. Unije određuju način manipuliranja različitim vrstama podataka u jednom području memorije, bez dodavanja bilo koje računalno zavisne informacije u program. One su analogne različitim zapisima u programskom jeziku Pascal.

Za primjer poput onog koji se nalazi u tablici simbola prevoditelja, prispodobimo (kako ovo dobro zvuči ...) konstanta može biti tipa int, float ili znakovni pokazivač. Vrijednost određene konstante mora biti pohranjena u varijabli odgovarajućeg tipa, ali je najzgodnije za uporabu tablica ako vrijednost zauzima istu veličinu memorije i ako je pohranjena na istom mjestu bez obzira na njen tip. Svrha unije - jedna varijabla koja može sadržavati (bolje je kazati udomiti) više različitih tipova. Sintaksa se zasniva na strukturama:

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

Varijabla u dovoljno je velika da primi najveći od tri navedena tipa; njihova veličina ovisna je o implementaciji. Bilo koji od ovih tipova može se pridijeliti varijabli u i koristiti u izrazima, tako dugo dok je upotreba konzistentna; traženi tip mora biti onaj koji je najsvježije pohranjen. Na programeru leži odgovornost praćenja toga koji tip je trenutno pohranjen u uniji; ako je nešto pohranjeno kao jedan tip i zatraženo kao drugi, rezultati će ovisiti o implementaciji.

Sintaksno, članovima unije pristupamo kao

```
ime_unije.član
```

ili

```
pokazivač_na_uniju->član
```

baš kao i kod struktura. Ako se varijabla utype koristi za praćenje tipa pohranjenog u varijabli u, tada se kod može napisati kao

```
if(utype==INT)
      printf("%d\n", u.ival);
else
      if(utype==FLOAT)
            printf("%f\n", u.fval);
else
      if(utype==STRING)
            printf("%s\n",u.sval);
else
      printf("bad type %d in utype\n", utype);
```

Unije se mogu pojaviti u strukturama i poljima, a i obratno. Notacija za pristup članu unije u strukturi (ili pak obrnuto) identična je onoj za ugniježdene strukture. Primjerice, u polju struktura definiranom kao

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

član ival se poziva s

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

i prvi znak niza sval na jedan od ovih načina

```
*symtab[i].u.sval
symtab[i].u.sval[0]
```

Zapravo, unija je struktura čiji članovi isključuju druge članove, struktura je dovoljno velika za pohranu "najglomaznijeg" člana, a raspored se prilagođava svim tipovima u uniji. Iste operacije koje su dopuštene na unijama dopuštene su i na strukturama; pridjeljivanje ili kopiranje unije kao cjeline, uzimanje adrese, te pristupanje članu.

Unija se može inicijalizirati samo vrijednošću tipa njezina prvog člana; zato se unija u koju smo opisali može inicijalizirati samo pomoću cjelobrojne vrijednosti.

Memorijski pridjeljivač u Poglavlju 8 pokazuje kako se unija može uporabiti radi postavljanja varijable na određenu memorijsku granicu.

6.9 Polja bitova

Kad memorijski prostor postane bitan, možda se ukaže potreba pakiranja nekoliko objekata u jednu računalnu riječ; česta je uporaba skupa jednobitnih zastavica u aplikacijama slično prevoditeljskoj tablici simbola. Izvana nametnuti formati podataka, kao kod sučelja prema sklopovlju, također ponekad trebaju mogućnost pristupa dijelovima riječi.

Zamislite dio prevoditelja koji manipulira tablicom simbola. Svaki identifikator u programu ima određenu informaciju vezanu za sebe, npr., radi li se o ključnoj riječi, da li je vanjska i/ili statička itd. Najkompaktniji način kodiranja takve informacije je skup jednobitnih zastavica unutar jednog znaka (char) ili cjelobrojne vrijednosti (int).

Obično se ovo radi definiranjem skupa "maski" koje odgovaraju relevantnim pozicijama bita, kao u

```
#define     KEYWORD     01
#define     EXTERNAL    02
#define     STATIC      04
```

ili

```
enum { KEYWORD=01, EXTERNAL=02, STATIC=04 };
```

Brojevi moraju biti potencije broja 2. Tada pristupanje bitovima postaje stvar "namještanja bitova" pomoću operatora pomicanja, maskiranja i komplementiranja koji su opisani u Poglavlju 2.

Neki izrazi se često javljaju:

```
flags|=EXTERNAL|STATIC;
```

pali bitove EXTERNAL i STATIC u varijabli flags, dok ih

```
flags &=~(EXTERNAL|STATIC);
```

gasi, a

```
if((flags&(EXTERNAL|STATIC))==0)
      ...
```

je istinito ako su oba bita ugašena.

Iako se ovi izrazi lako svladavaju, kao alternativu programski jezik C nudi mogućnost definiranja i pristupa poljima unutar riječi neposredno, a ne preko logičkih operatora za manipulaciju bitovima. Polje bitova, ili kraće polje, jest skup susjednih bitova unutar jedne implementacijom definirane jedinice memorije koju zovemo "riječ". Sintaksa definicije polja i pristupa polju zasnovana je na strukturama. Npr., gornja tablica simbola s #define izrazima može se zamijeniti definicijom tri polja:

```
struct{
      unsigned int is_keyword:1;
      unsigned int is_extern:1;
      unsigned int is_static:1;
} flags;
```

Ovo definira varijablu s imenom flags, koja sadrži tri jednobitna polja bitova. Broj iza znaka dvotočke predstavlja širinu polja u bitovima. Polja su deklarirana kao unsigned int tip radi osiguranja veličina bez predznaka.

Pojedina polja pozivaju se kao i drugi članovi strukture: flags.is_keyword, flags.is_static, itd. Polja se ponašaju poput malih cjelobrojnih vrijednosti, te mogu participirati u aritmetičkim izrazima baš kao i druge cijeli brojevi. Tako prethodne primjere možemo prirodnije pisati kao

```
flags.is_extern=flags.is_static=1;
```

radi postavljanja bitova na 1;

```
flags.is_extern=flags.is_static=0;
```

radi njihova postavljanja na 0; i

```
if(flags.is_extern==0&&flags.is_static==0)
      ...
```

kako bi ih testirali.

Gotovo sve što ima veze s poljima ovisno je o implementaciji. Može li polje prelaziti granicu riječi također je ovisno o njoj. Polja se ne moraju imenovati; neimenovana polja (s dvotočkom i širinom) koriste se za popunjavanje. Posebna širina 0 može se uporabiti za poravnanje na granicu slijedeće riječi.

Polja se pridjeljuju slijeva nadesno na nekim računalima, a zdesna nalijevo na drugim. To znači da iako su polja korisna za posluživanje interno definiranih struktura podataka, pitanje koji kraj predstavlja početak treba pažljivo razmotriti kod pristupanja podacima definiranim izvana; programi koji ovise o takvim stvarima nisu baš prenosivi. Polja se mogu deklarirati samo pomoću tipa int; zbog prenosivosti, jasno odredite signed ili unsigned tip. Ovo nisu klasična polja, te nemaju adrese početka, pa se tako operator & ne može primjeniti na njih.

POGLAVLJE 7: ULAZ I IZLAZ

Ulazne i izlazne mogućnosti nisu dio programskog jezika C, pa ih dosad nismo ni posebno naglašavali. Pa ipak, programi komuniciraju sa svojim okruženjem na mnogo složeniji način od već prikazanog. U ovom poglavlju opisat ćemo standardnu biblioteku, skup funkcija koje određuju ulaz i izlaz, manipulaciju nizovima, memorijom, matematičke rutine i veliki broj drugih servisa za C programe. Usredotočit ćemo se na ulaz i izlaz.

ANSI standard precizno definira ove biblioteke funkcija, pa se one javljaju u prenosivoj formi na svim platformama na kojima egzistira programski jezik C. Programi koji svode sistemske veze na one koje pruža standrdna biblioteka mogu se prenositi s jednog sistema na drugi bez izmjena.

Osobine funkcija biblioteke određene su u nekoliko zaglavlja; neke smo već spominjali, uključujući <stdio.h>, <string.h> i <ctype.h>. Ovdje nećemo prikazati cijelu biblioteku, budući nas više interesira pisanje C programa koji se njom koriste. Biblioteka je detaljno opisana u Dodatku B.

7.1 Standardni ulaz i izlaz

Kako smo već kazali u Poglavlju 1, biblioteka primjenjuje jednostavan model tekstualnog ulaza i izlaza. Tok teksta sastoji se od niza linija; svaka linija okončava se znakom nove linije (newline character). Ako sistem ne funkcionira na taj način, bibliteka čini sve potrebno kako bi izgledalo da funkcionira upravo tako. Primjerice, biblioteka može pretvarati znakove CR i LF (carriage return i linefeed) u znak NL (newline) na ulazu i vraćati ih na izlazu.

Najjednostavniji je ulazni mehanizam očitavanje pojedinog znaka sa standardnog ulaza, obično s tipkovnice, pomoću funkcije getchar:

```
int getchar(void)
```

Funkcija getchar vraća slijedeći ulazni znak svaki put kad se je pozove, a vrijednost EOF kad naiđe na kraj datoteke. Simbolička konstanta EOF definirana je u <stdio.h>. Vrijednost joj je obično -1, ali provjere treba pisati s nazivom EOF kako bi bili neovisni o njenoj vrijednosti. U brojnim okruženjima, datoteka može zamijeniti tipkovnicu koristeći < konvenciju za ulaznu redirekciju; ako program prog koristi funkciju getchar, tada naredbena linija

```
prog < infile
```

prisiljava program prog da učitava znakove iz datoteke infile umjesto s tipkovnice. Takav način unosa prekida se tako što sam program zaboravi na promjenu; konkretno, znakovni niz "< infile" nije uključen u argumente naredbene linije u polju argv. Prekidanje ulaza je isto tako nevidljivo ako ulaz dolazi iz nekog drugog programa kroz sistem cjevovoda (pipe mechanism); na nekim sistemima, naredbena linija

```
otherprog | prog
```

obavlja dva programa otherprog i prog, te usmjerava standardni izlaz iz programa otherprog na standardni ulaz programa prog.

Funkcija

```
int putchar(int)
```

rabi se za izlaz: funkcija putchar(c) postavlja znak c na standardni izlaz, koji je inicijalno ekran. Funkcija putchar vraća ili upisani znak ili EOF ako se pojavi greška. Napomenimo, izlaz se obično usmjerava u datoteku pomoću > ime_datoteke; ako prog koristi putchar

```
prog > outfile
```

usmjerit će standardni izlaz u datoteku outfile. Ako su podržani cjevovodi,

```
prog | anotherprog
```

postavit će standardni izlaz prvog programa na standardni ulaz drugog.

Izlaz načinjen pomoću funkcije printf također pronalazi svoj put do standardnog izlaza. Pozivi funkcija printf i putchar mogu se ispreplitati - izlaz se pojavljuje redom kojim su pozivi obavljeni.

Svaka izvorna datoteka koja treba ulazno/izlaznu biblioteku mora imati liniju

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

prije prvog poziva. Kad je ime omeđeno s < i > traži se zaglavlje na standardnim mjestima (primjerice, na UNIX strojevima, obično se radi o direktoriju /usr/include).

Brojni programi čitaju samo jedan ulazni tok i pišu samo jedan izlazni; za takove programe, ulaz i izlaz pomoću funkcija getchar, putchar i printf mogu biti potpuno zadovoljavajući, što je sasvim dovoljno za početak. Ovo posebice vrijedi ako se redirekcija rabi za povezivanje izlaza jednog programa sa ulazom slijedećeg. Npr., razmotrimo program lower, koji pretvara vlastiti ulaz u mala slova:

```
#include <>
#include <>
main(){
      int c;

      while((c=getchar())!=EOF)
            putchar(tolower(c));
      return      0;
}
```

Funkcija tolower definirana je u zaglavlju <ctype.h>; ona pretvara veliko slovo u malo, te vraća ostale znakove nedirnute. Kako smo prije spomenuli, "funkcije" poput getchar i putchar u zaglavlju <stdio.h> i tolower u <ctype.h> su često izvedeni kao makro, izbjegavši tako obvezni poziv funkcije za svaki znak. Prikazat ćemo kako je ovo postignuto u dijelu 8.5. Bez obzira kako su funkcije zaglavlja <ctype.h> implementirane na pojedinim strojevima, programi koji ih upotrebljavaju nemaju informaciju o skupu znakova.

Vježba 7-1. Napišite program koji pretvara velika u mala slova ili obratno, ovisno o imenu kojim je pozvan, a koje se nalazi u argv[0].

7.2 Formatirani izlaz - printf

Izlazna funkcija printf prevodi interne vrijednosti u znakove. Uprethodnim poglavljima funkciju printf rabili smo neformalno. Ovaj opis pokriva većinu tipičnih upotreba iako nije potpun; radi cijele priče, pročitajte Dodatak B.

```
int printf(char *format, arg1, arg2, ...)
```

Funkcija printf pretvara, formatira i ispisuje svoje argumente na standardnom izlazu prema pravilima iz polja format. Ona vraća broj ispisanih znakova.

Polje znakova format sadrži dvije vrste objekata: obične znakove, koji se preslikavaju na izlazni tok, te specifikacije pretvorbe, od kojih svaka pretvara i ispisuje naredni argument iz funkcije printf. Svaka pretvorbena specifikacija počinje znakom % i završava znakom pretvorbe. Između % i pretvorbenog znaka mogu postojati, po redu:

- znak minus, koji određuje lijevo podešavanje pretvorbenog argumenta.
- broj koji određuje minimalnu širinu polja. Pretvorbeni argument bit će ispisan u polju najmanje te veličine. Ako je potrebno on će biti dopunjen znakovima slijeva (ili zdesna) radi popunjavanja širine polja.
- točka, koja razdvaja širinu polja od točnosti.
- broj, točnost, koji određuje maksimalan broj znakova iz niza koji će biti ispisani, ili broj znamenki nakon decimalne točke u broju s pomičnim zarezom, ili minimalni broj znamenki za cijeli broj.
- h ako cijeli broj treba ispisati kao short, ili l (slovo l) ako broj treba ispisati kao long.

Znakovi pretvorbe prikazani su u Tablici 7-1. Ako znak nakon % nije pretvorbena specifikacija, ponašanje se nedefinirano.

TABLICA 7-1. OSNOVNE PRETVORBE FUNKCIJE PRINTF

Znak	Tip argumenta; Ispisuje se kao
d, i	int; decimalni broj
o	int; pozitivni oktalni broj (bez vodeće nule)
x, X	int; pozitivni heksadecimalni broj (bez vodećeg 0x ili 0X), koji rabi abcdef ili ABCDEF za brojeve 10, ..., 15
u	int, pozitivni decimalni broj
c	int; jedan znak
s	char *; ispisuje znakove iz niza sve do '\0' ili broja znakova određenih preciznošću
f	double; [-]m.dddddd, gdje je broj slova d zadan preciznošću (inicijalno 6)
e, E	double; [-]m.dddddde+-xx ili [-]m.ddddddE+-xx, gdje je broj slova d zadan preciznošću (inicijalno 6)
g, G	double; upotrebljava %e ili %E ako je eksponent manji od -4 ili veći ili jednak stupnju preciznosti; inače upotrebljava %f. Nepotrebne nule (na kraju) i decimalna točka se ne ispisuju.
p	void *; pokazivač (opis ovisan o implementaciji)
%	nijedan argument se ne pretvara; ispisuje %

Širina ili preciznost mogu se specificirati i pomoću *, te u tom slučaju vrijednost se računa pretvorbom slijedećeg argumenta (koji mora biti tipa int). Primjerice, kako bi ispisali najviše max znakova niza s,

```
printf("%.*s", max, s);
```

Glavnina formatnih pretvorbi već je ilustrirana u prethodnim poglavljima. Izuzetak je preciznost koja se odnosi na nizove znakova. Tablica koja slijedi pokazuje učinak različitih specifikacija u ispisivanju "hello, world" (12 znakova). Stavili smo dvotočke oko svakog polja kako bi mogli vidjeti njegovu veličinu.

```
:%s:        :hello, world:
:%10s:      :hello, world:
:%.10s:     :hello, wor:
:%-10s:     :hello, world:
:%.15s:     :hello, world:
:%-15s:     :hello, world   :
:%15.10s:   :     hello, wor:
:%s-15.10:  :hello, wor     :
```

Napomena: Funkcija printf rabi svoj prvi argument radi odluke koliko argumenata slijedi i kojeg su tipa. Ako nema dovoljno argumenata ili ako su pogrešnog tipa, nastati će zabuna, pa ćete dobiti pogrešne odgovore. Trebali bi biti svjesni razlike između ova dva poziva:

```
printf(s);  /* GRIJEŠI ako polje s sadrži znak % */
printf("%s", s);  /* SIGURNO */
```

Funkcija sprintf obavlja iste pretvorbe kao i printf, no pohranjuje izlaz u niz znakova:

```
int sprintf(char *string, char *format, arg1, arg2, ...)
```

Funkcija sprintf formatira argumente arg1, arg2, itd. prema polju format kao je već opisano, ali smješta rezultat u niz znakova string umjesto na standardni izlaz; polje string mora biti dovoljne veličine da primi rezultat.

Vježba 7-2. Napišite program koji će ispisati proizvoljan ulaz na razumljiv način. Trebao bi , kao minimum, ispisivati negrafičke znakove u oktalnoj ili heksadecimalnoj formi kako je to uobičajeno, te prelamati duge linije teksta.

7.3 Liste argumenata promjenjive dužine

Ovaj dio obrađuje primjenu najjednostavnije verzije funkcije printf, a da bi pokazao kako napisati funkciju koja procesira argumente promjenjive dužine u prenosivoj formi. Budući nas najviše interesira obrada argumenata, funkcija minprintf će obraditi niz za format i argumente, ali će ipak pozvati stvarni printf radi pretvorbe formata.

Prava deklaracija za funkciju printf jest

```
int printf(char *fmt, ...)
```

gdje deklaracija ... znači da broj argumenata i njihov tip mogu varirati. Deklaracija ... se može pojaviti samo na kraju liste argumenata. Funkcija minprintf se deklarira kao

```
void minprintf(char *fmt, ...)
```

kako nećemo vraćati brojač znakova koji vraća funkcija printf.

Interesantan dio je kako funkcija minprintf prolazi kroz listu argumenata kad lista nema čak ni ime. Standardno zaglavlje <stdarg.h> posjeduje skup makro definicija koje određuju način prolaza kroz listu argumenata. Implementacija ovog zaglavlja varira od stroja do stroja, no sučelje je uniformno.

Tip va_list rabi se pri deklaraciji varijable koja se odnosi na svaki argument u listi; u funkciji minprintf, ova varijabla naziva se ap, kao skraćenica za "argument pointer". Makro va_start inicijalizira ap pa ova pokazuje na prvi argument koji nema imena. On se mora pozvati prije korištenja varijable ap. Mora postojati barem jedan argument s imenom; zadnji argument s imenom koristi va_start kako bi se pokrenuo.

Svaki poziv funkcije va_arg vraća jedan argument i povećava varijablu ap kako bi ona sada pokazivala na slijedeći; funkcija va_arg upotrebljava ime tipa za utvrditi koji tip treba vratiti i koliko velik korak uzeti. Konačno, funkcija va_end obavlja sva potrebna brisanja. Ona mora biti pozvana prije povratka iz funkcije.

Ove osobine tvore osnovu naše pojednostavljene funkcije printf:

```
#include <stdarg.h>
/* minprintf : minimalna funkcija printf s varijabilnom listom argumenata
*/
void minprintf(char *fmt, ...){
      va_list ap; /* pokazuje na svaki argument bez imena */
      char *p, *sval;
      int ival;
      double dval;

      va_start(ap, fmt);      /* neka varijabla ap pokazuje na prvi
argument bez imena */
      for(p=fmt;*p;p++){
            if(*p!='%'){
                  putchar(*p);
                  continue;
            }
            switch(*++p){
                  case 'd':
                        ival=va_arg(ap, int);
                        printf("%d", ival);
                        break;
                  case 'f':
                        dval=va_arg(ap, double);
                        printf("%d", dval);
                        break;
                  case 's':
                        for(sval=va_arg(ap, char *);*sval; sval++)
```

```
                                putchar(*sval);
                            break;
                        default:
                            putchar(*p);
                            break;
                    }
                }
                va_end(ap); /* čišćenje na kraju */
            }
```

Vježba 7-3. Preradite funkciju minprintf kako bi imala više mogućnosti funkcije printf.

7.4 Formatirani ulaz - scanf

Funkcija scanf ulazni je ekvivalent funkcije printf, te pruža jednake pretvorbene pogodnosti u obrnutom smjeru.

```
int scanf(char *format, ...)
```

Funkcija scanf čita znakove sa standardnog ulaza, interpretira ih prema specifikacijama u polju format, te pohranjuje rezultat preko ostalih argumenata. Argument format tek ćemo opisati; drugi argumenti, od kojih svaki mora biti pokazivač, pokazuju gdje bi pretvoreni ulaz trebao biti pohranjen. Kao i s funkcijom printf, ovaj dio sažetak je najkorisnijih osobina, a ne iscrpna lista.

Funkcija scanf zaustavlja se kad iscrpi niz format ili kad unos ne odgovara kontrolnoj specifikaciji. Ona vraća kao svoju vrijednost broj uspješno unešenih i pridjeljenih ulaznih stavki. To se dade iskoristiti pri određivanju broja pronađenih stavki. Na kraju datoteke, vraća se EOF; imajte na umu da to nije 0, što znači da slijedeći ulazni znak ne odgovara prvoj specifikaciji u nizu znakova format. Slijedeći poziv funkcije scanf nastavlja pretraživanje odmah nakon pretvorbe zadnjeg znaka.

Postoji, također, funkcija sscanf koja čita iz niza znakova mjesto standardnog ulaza:

```
int sscanf(char *string, char *format, arg1, arg2, ...)
```

Ona prolazi poljem string u skladu s formatom zapisanom u polju format, te pohranjuje rezultirajuće vrijednosti u varijable arg1, arg2, itd. Ovi argumenti moraju biti pokazivači.

Niz znakova format najčešće sadrži pretvorbene specifikacije koje se koriste za pretvorbu ulaza. Niz znakova format može sadržavati:

- Razmake ili tabulatore, koji se zanemaruju.
- Obične znakove (ne %), koji se trebaju podudarati sa slijedećim znakom ulaznog niza koji nije razmak.
- Pretvorbene specifikacije, koje se sastoje od znaka %, opcijskog znaka za zabranu pridjeljivanja, opcijskog broja za određivanje maksimalne širine polja, opcijskog h, l ili L koje označava širinu cilja, te pretvorbenog znaka.

Pretvorbena specifikacija upravlja pretvorbom slijedećeg ulaznog polja. Normalno se rezultat smješta u varijablu na koju pokazuje odgovarajući argument. Ako se, međutim, označi zabrana pridjeljivanja znakom *, ulazno polje se preskače; nema pridjeljivanja. Ulazno polje definira se niz znakova koji nisu razmaci; ono se prostire do slijedećeg razmaka ili dok se širina polja, ako je specificirana, ne iscrpi. To povlači da će funkcija scanf čitati i preko granica retka ili linije kako bi pronašla svoj ulaz, jer su znakovi nove linije zapravo pusti znaci (pusti znaci su razmak, tabulator, znak nove linije, ..)

Znak pretvorbe interpretira ulazno polje. Odgovarajući argument mora biti pokazivač, kako i zahtjeva semantika pozivanja preko vrijednosti u programskom jeziku C. Znakovi pretvorbe prikazani su u Tablici 7-2.

TABLICA 7-2. OSNOVNE PRETVORBE FUNKCIJE SCANF

Znak	Ulazni podatak; Tip argumenta
d	decimalni cijeli broj; int *
i	cijeli broj; int *. Cijeli broj može biti unešen oktalno(vodeća 0) ili heksadecimalno (vodeći 0x ili 0X)

o	oktalni cijeli broj (sa i bez vodeće 0); int *
u	pozitivan cijeli broj; unsigned int *
x	heksadecimalni cijeli broj (sa i bez vodećim 0x ili 0X); int *
c	znakovi; char *. Slijedeći ulazni znakovi (inicijalno 1) smješteni su na označeno mjesto. Uobičajeno preskakanje preko razmaka isključeno je; za čitanje slijedećeg znaka koji nije pusti, koristite %1s.
s	niz znakova (nije ograničena navodnicima); char *. Pokazuje na polje znakova dovoljno veliko za niz i završni '\0' znak koji će biti dodan.
e, f, g	realni broj s opcijskim predznakom, decimalnom točkom i eksponentom; float *
%	literal %; nema pridjeljivanja.

Pretvorbenim znakovima d, i, o i x može prethoditi znak h kako bi označio da se u listi argumenata pojavljuje pokazivač na short, a ne na int, ili znak l (slovo el) kako bi označio pokazivač na long. Slično tome, znakovima pretvorbe e, f i g može prethoditi l koje označava da se u listi argumenata pojavljuje pokazivač na double, a ne na float.

Kao prvi primjer, jednostavni kalkulator iz Poglavlja 4 dade se napisati tako da funkcija scanf obavi ulaznu pretvorbu:

```
#include <stdio.h>
main(){     /* jednostavni kalkulator */
      double sum, v;

      sum=0;
      while(scanf("%lf", &v)==1)
            printf("\t%.2f\n", sum+=v);
      return 0;
}
```

Recimo da želimo čitati ulazne linije koje sadrže datume u ovakvoj formi

```
25 Dec 1988
```

Izraz za funkciju scanf izgleda ovako

```
int day, year;
char monthname[20];

scanf("%d %s %d", &day, monthname, &year);
```

Operator & ne rabi se kod imena monthname, jer je ime polja pokazivač sam po sebi.

U nizu znakova format mogu se pojaviti i literali (jednostavnije kazano - slova); oni moraju odgovarati istim znakovima na ulazu. Tako možemo čitati datume u formi mm/dd/yy pomoću ovog scanf izraza.

```
int day, month, year;

scanf("%d%d%d", &month, &day, &year);
```

Funkcija scanf ignorira razmake i tabulatore u svom znakovnom nizu format. Čak štoviše, ona preskače puste znakove (razmake, tabulatore, znakove nove linije, ...) pri traženju ulaznih vrijednosti. Za čitanje ulaza čiji format nije čvrst, često je najbolje čitati liniju po liniju, te ih razdvajati pomoću funkcije sscanf. Primjerice, pretpostavimo da želimo očitati linije koje sadrže datum u jednoj od gore spomenutih formi. Tada možemo pisati

```
while(getline(line, sizeof(line))>0){
      if(sscanf(line, "%d %s %d", &day, monthname, &year)==3)
```

```
                printf("valid: %s\n", line);  /* forma 25 Dec 1988 */
        else
                if(sscanf(line, "%d%d%d", &moth, &day, &year)==3)
                        printf("valid: %s\n", line);  /*forma mm/dd/yy */
                else
                        printf("invalid: %s\n", line);      /* pogrešna forma */
}
```

Pozivi funkcije scanf mogu se kombinirati s pozivima drugih ulaznih funkcija. Naredni poziv bilo koje ulazne funkcije počet će čitanjem prvog znaka koji koji nije pročitan funkcijom scanf.

Zadnje upozorenje: argumenti funkcija scanf i sscanf moraju biti pokazivači. Daleko najčešća greška je pisanje

```
scanf("%d", n);
```

umjesto

```
scanf("%d", &n);
```

Ova greška se općenito ne dade otkriti pri prevođenju.

Vježba 7-4. Napišite svoju inačicu funkcije scanf analogijom s funkcijom minprintf iz prethodnog dijela.

Vježba 7-5. Ponovo napišite kalkulator s reverznom Poljskom notacijom iz Poglavlja 4 tako da rabi funkcije scanf i sscanf za pretvorbu ulaza i brojeva.

7.5 Pristup datoteci

Svi dosadašnji primjeri čitali su sa standardnog ulaza i pisali na standardni izlaz, što je automatski definirano za programe od strane operativnog sustava.

Slijedeći korak jest pisanje programa koji pristupa datoteci koja nije izravno vezana s programom. Program koji ilustrira potrebu za takvim operacijama jest cat, koji spaja skup imenovanih datoteka na standardni izlaz. Program cat rabi se pri ispisu datoteka na ekran, te općenito kao ulazno sučelje za programe koji nemaju mogućnost pristupa datotekama pomoću imena. Npr., naredba

```
cat x.c y.c
```

ispisuje sadržaj datoteka x.c i y.c (i ništa više) na standardni izlaz.

Pitanje je kako omogućiti datoteci s imenom da bude pročitana - odnosno, kako povezati vanjska imena poznata korisniku s izrazima koji čitaju podatke.

Pravila su jednostavna. Prije negoli se po datoteci može čitati i pisati, ona se mora otvoriti funkcijom fopen iz biblioteke. Funkcija fopen uzima vanjsko ime kakvo je x.c ili y.c, obavi neka pospremanja i poveže s operativnim sustavom (čiji nas detalji ne trebaju zanimati), te vrati pokazivač koji će se koristiti u čitanjima i pisanjima koji slijede.

Ovaj pokazivač, nazvan file pointer, pokazuje na strukturu koja sadrži informaciju o datoteci, kao što je položaj spremnika, trenutni položaj znaka u spremniku, očitava li se datoteka ili se pak u nju upisuju podaci, te ima li grešaka ili smo došli do kraja datoteke. Korisnici ne trebaju znati detalje, jer definicije koje osigurava zaglavlje <stdio.h> uključuju deklaraciju strukture s imenom FILE. Jedina deklaracija potrebna pokazivaču datoteke predočena je s

```
FILE *fp;
FILE *fopen(char *name, char *mode);
```

Ovo kaže da je fp pokazivač na FILE, a funkcija fopen vraća pokazivač na FILE. Zamjetite kako je FILE ime tipa, poput, recimo, int, a ne oznaka strukture; ono je definirano pomoću typedef. (Detalji vezani za implementaciju na UNIX sustavima dani su u dijelu 8.5.)

Poziv funkcije fopen u programu izgleda ovako

```
fp=fopen(name, mode);
```

Prvi argument funkcije fopen znakovni je niz s imenom datoteke. Drugi argument jest mode, isto znakovni niz, koji određuje način korištenja datoteke. Dopušteni modusi su čitanje ("r"), pisanje ("w"), te dodavanje ("a"). Neki sustavi prave razliku između tekstualnih i binarnih datoteka; za ove posljednje moramo dodati "b" u znakovni niz mode.

Ako se za pisanje ili dodavanje otvori datoteka koja ne postoji, ona se kreira ako je to moguće. Otvaranje postojeće datoteke za pisanje uzrokuje brisanje ranijeg sadržaja, dok ga otvaranje radi dodavanja čuva. Pokušaj otvaranja datoteke koja ne postoji radi čitanja jest greška, a postoje i drugi uzroci grešaka, kao što je čitanje datoteke kad za to ne postoji dopuštenje (permission). Ako postoji greška, funkcija fopen vraća NULL. (Greška se dade identificirati preciznije; pogledajte raspravu o funkcijama za manipulaciju greškama na kraju prvog dijela Dodatka B.)

Slijedeće što trebamo je način čitanja i pisanja datoteke kad je ona otvorena. Postoji nekoliko mogućnosti od kojih su funkcije getc i putc najjednostavnije. Funkcija getc vraća slijedeći znak iz datoteke; ona treba pokazivač datoteke (file pointer) kako bi odredila o kojoj se datoteci radi.

```
int getc(FILE *fp)
```

Funkcija getc vraća slijedeći znak iz toka podataka na koji pokazuje pokazivač fp; ona vraća EOF ako je u pitanju greška ili kraj datoteke.

Funkcija putc ima izlazni karakter

```
int putc(int c, FILE *fp)
```

Funkcija putc piše znak c u datoteku fp i vraća upisani znak, ili EOF ako dođe do greške. Poput funkcija getchar i putchar, funkcije getc i putc mogu biti makroi mjesto funkcija.

Kad se C program pokrene, okruženje operativnog sustava odgovorno je za otvaranje triju datoteka i inicijalizaciju njihovih pokazivača. Te datoteke su standardni ulaz, standardni izlaz, te standardna greška; njihovi pokazivači datoteka (file pointers) respektivno jesu stdin, stdout i stderr, a deklarirani su u zaglavlju <stdio.h>. Prirodno je pokazivač stdin vezan uz tipkovnicu i pokazivači stdout i stderr uz zaslon, ali stdin i stdout mogu se preusmjeriti na datoteke ili cjevovode kako je opisano u dijelu 7.1.

Funkcije getchar i putchar dadu se definirati pomoću funkcija getc, putc, stdin i stdout kako slijedi:

```
#define getchar() getc(stdin)
#define     putchar(c)  putc((c), stdout)
```

Za formatirani ulaz ili izlaz datoteka, mogu se rabiti funkcije fscanf i fprintf. One su identične funkcijama scanf i printf, osim što je prvi argument pokazivač datoteke koji kaže da li će se datoteka čitati ili pisati; drugi argument jest znakovni niz format.

```
int fscanf(FILE *fp, char *format, ...)
int fprintf(FILE *fp, char *format, ...)
```

Nakon ovih neformalnih uvoda, sad smo u stanju napisati program cat koji spaja datoteke. Ovaj dizajn pokazao pogodnim za mnoge programe. Ako postoje argumenti naredbene linije, oni se interpretiraju kao imena datoteka, te obrađuju po redu. Ako argumenata nema, obrađuje se standardni ulaz.

```
#include <stdio.h>
/* cat : spaja datoteke, inačica 1 */
main(int argc, char *argv[]){
      FILE *fp;
      void filecopy(FILE *fp, FILE *fp);

      if(argc==1) /* nema argumenata; kopira standardni ulaz */
            filecopy(stdin, stdout);
      else
            while(--argc>0)
                  if((fp=fopen(*++argv, "r"))==NULL){
                        printf("cat: can't open %s\n", *argv);
                        return 1;
                  }
                  else{
                        filecopy(fp, stdout);
```

```
                        fclose(fp);
                }
        return 0;
}

/* filecopy : kopira datoteku ifp u ofp */
void filecopy(FILE *ifp, FILE *ofp){
        int c;

        while((c=getc(ifp))!=EOF)
                putc(c, ofp);
}
```

Pokazivači datoteka stdin i stdout jesu objekti tipa FILE *. Oni su konstante, naglasimo, a ne varijable, pa im se ništa ne može pridijeliti.

Funkcija

```
int fclose(FILE *fp)
```

inverzna je funkciji fopen; ona okončava vezu između pokazivača datoteke i vanjskog imena koju je uspostavila funkcija fopen, oslobađajući tako pokazivač datoteka za slijedeću datoteku. Kako većina operativnih sustava ima nekakvo ograničenje glede broja datoteka koje program može istovremeno otvoriti, dobra je strategija oslobađati pokazivače datoteka kad ih više ne trebamo, kako smo to već radili u programu cat. Ima još jedan razlog za upotrebu funkcije fclose na izlaznu datoteku - ona pobuđuje spremnik u koji funkcija putc upućuje izlaz. Funkcija fclose poziva se automatski za svaku otvorenu datoteku kad program normalno završi. (Možete isključiti datoteke stdin i stdout ako nisu potrebne. Ponovo ih možete otvoriti funkcijom iz biblioteke freopen.)

7.6 Manipulacija greškama - stderr i exit

Obrada grešaka u programu cat nije savršena. Nevolja je što ako jednoj od datoteka nije moguće zbog nečega pristupiti, dijagnostika se ispisuje na kraju spajanog izlaza. Ovo može biti prihvatljivo ako izlaz ide na zaslon, ali ne ako ide u datoteku ili u neki drugi program putem cjevovoda.

Kako bi se bolje rješavala ovakva situacija, drugi izlazni tok, nazvan stderr, pridjeljuje se programu poput tokova stdin i stdout. Izlaz upućen na tok stderr normalno izlazi na zaslon čak i ako preusmjerimo standardni izlaz.

Prepravimo program cat kako bi svoje poruke o grešci ispisivao na tok standardne greške.

```
#include <stdio.h>
/* cat : spaja datoteke, inačica 2 */
main(int argc, char argv[]){
        FILE *fp;
        void filecopy(FILE *, FILE *);
        char *prog=argv[0];     /* programsko ime za greške */

        if(argc==1) /* nema argumenata; kopira standardni ulaz */
                filecopy(stdin, stdout);
        else
                while(--argc>0)
                        if((fp=fopen(*++argv, "r"))==NULL){
                                fprintf(stderr, "%s: can't open %s\n", prog,
*argv);
                                exit(1);
                        }
                        else{
                                filecopy(fp, stdout);
                                fclose(fp);
                        }
        if(ferror(stdout)){
                fprintf(stderr, "%s: error writing stdout\n", prog);
                exit(2);
```

```
        }
        exit(0);
}
```

Program dojavljuje pojavu grešaka na dva načina. Prvo, dijagnostički izlaz kojeg napravi funkcija fprintf ide na tok stderr, pa tako izlazi na zaslonu mjesto da se izgubi u cjevovodu ili u nekoj izlaznoj datoteci. Ime programa, iz polja argv[0], uključili smo u poruku, pa ako se koristi više programa, moguće je identificirati izvor greške.

Drugo, program rabi funkciju standardne biblioteke exit, koja okončava izvršenje programa koji je pozvan. Argument funkcije exit dostupan je bilo kojem procesu koji je funkciju pozvao, pa uspjeh ili neuspjeh programa može se provjeriti drugim programom koji ovaj program kao podproces. Dogovorno, povratna vrijednost 0 označava kako je sve u redu; vrijednosti koje nisu 0 označavaju situaciju koja nije normalna. Funkcija exit poziva funkciju fclose za svaku otvorenu izlaznu datoteku, kako bi prikazala cijeli spremljeni izlaz.

Unutar funkcije main, konstrukcija return expr ekvivalentna je s exit(expr). Funkcija exit ima prednost što može biti pozvana iz druge funkcije, a ti pozivi se mogu pronaći pomoću programa za pretraživanje uzoraka kakvi su oni u Poglavlju 5.

Funkcija ferror vraća vrijednost koja nije nula ako se dogodi greška na toku fp

```
int ferror(FILE *fp)
```

Mada su izlazne greške rijetke, one se događaju (primjerice, radi zapunjenog diska), pa bi pravi programi trebali imati ove provjere.

Funkcija feof(FILE *) analogna je funkciji ferror; ona vraća vrijednost koja nije nula ako se dogodi kraj specificirane datoteke

```
int feof(FILE *)
```

Mi zapravo ne brinemo o izlaznom statusu naših malih pokaznih programčića, ali bilo koji ozbiljniji program treba voditi računa o vraćanju smislenih i korisnih statusnih vrijednosti.

7.7 Linijski ulaz i izlaz

Standardna biblioteka posjeduje ulaznu funkciju fgets koja je slična funkciji getline korištenu u ranijim poglavljima:

```
char *fgets(char *line, int maxline, FILE *fp)
```

Funkcija fgets učita narednu ulaznu liniju (uključujući znak nove linije) iz datoteke fp u znakovno polje line; bit će učitano najviše maxline-1 znakova. Rezultirajuća linija završava s '\0'. Uobičajeno funkcija fgets vraća polje line; za kraj datoteke ili grešku vraća NULL. (Naša funkcija getline vraća dužinu linije, što je korisnija vrijednost; nula označava kraj datoteke.)

Kao izlazna, funkcija fputs piše niz znakova (koji ne mora sadržavati znak nove linije) u datoteku:

```
int fputs(char *line, FILE *fp)
```

Ona vraća EOF ako dođe do greške, a inače nulu.

Funkcije biblioteke gets i puts slične su funkcijama fgets i fputs, no rade preko tokova stdin i stdout. Zabunu stvara činjenica što funkcija gets briše završno '\n', dok ga funkcija puts dodaje.

Kako bi pokazali da nema ničeg posebnog u funkcijama fgets i fputs, evo ih ovdje, preslikane iz standardne biblioteke našeg sustava:

```
/* fgets : uzima najviše n znakova iz datoteke iop */
char *fgets(char *s, int n, FILE *iop){
        register int c;
        register char *cs;

        cs=s;
        while(--n>0&&(c=getc(iop))!=EOF)
        if((*cs++=c)=='\n')
```

```
            break;
      *cs='\0';
      return (c==EOF&&cs=s) ? NULL : s;
}

/* fputs : piše znakovni niz s u datoteku iop */
int fputs(char *s, FILE *iop){
      int c;

      while(c=*s++)
            putc(c, iop);
      return ferror(iop) ? EOF : 0;
}
```

Standard određuje da funkcija ferror vraća vrijednost koja nije nula u slučaju greške; funkcija fputs vraća EOF u slučaju greške i nenegativnu vrijednost inače.

Lako je implementirati našu funkciju getline pomoću funkcije fgets:

```
/* getline : čita liniju, vraća dužinu */
int getline(char *line, int max){
      if(fgets(line, max, stdin)==NULL)
            return 0;
      else
            return strlen(line);
}
```

Vježba 7-6. Napišite program za usporedbu dviju datoteka, ispisujući prvu liniju u kojoj se razlikuju.

Vježba 7-7. Izmjeniti program za traženje uzorka iz Poglavlja 5 tako da on uzima podatke iz skupa imenovanih datoteka, a ako tog skupa nema, onda sa standardnog ulaza. Treba li ispisati ime datoteke ako pronađemo identičnu liniju.

Vježba 7-8. Napišite program koji ispisuje skup datoteka, počinjući svaku na novoj strani, s naslovom i prebrojavajući stranice svake datoteke.

7.8 Raznolike funkcije

Standardna biblioteka pruža široku lepezu funkcija. Ovaj dio je kratak pregled najkorisniji. Detaljniji pristup i brojne druge funkcije mogu se pronaći u Dodatku B.

7.8.1 Operacije s znakovnim nizovima

Već smo spominjali funkcije strlen, strcpy, strcat i strcmp, koje se nalaze u zaglavlju <string.h>. U funkcijama koje sada navodimo, argumenti s i t jesu tipa char *, dok su c i n cjelobrojni (int).

strcat(s, t)	spaja t s krajem s
strncat(s, t, n)	spaja n znakova iz t s krajem s
strcmp(s, t)	vraća negativnu, nultu ili pozitivnu vrijednost za, respektivno, s<t, s=t i s>t
strncmp(s, t, n)	isto kao i strcmp samo na prvih n znakova
strcpy(s, t)	kopira t u s
strncpy(s, t, n)	kopira najviše n znakova iz t u s
strlen(s)	vraća dužinu od s
strchr(s, c)	vraća pokazivač na prvi c u s, a NULL ako ne postoji
strrchr(s, c)	vraća pokazivač na posljednji c u s, a NULL ako ne postoji

7.8.2 Provjera i pretvorba klasa znakova

Nekoliko funkcija iz zaglavlja <ctype.h> obavlja provjeru i pretvorbu znakova. U slijedećim funkcijama, c je cjelobrojna vrijednost koji može predstavljati i tip unsigned char, ili EOF. Funkcije vraćaju vrijednost tipa int.

isalpha(c)	nije nula ako je c slovo, inače nula

isupper(c)	nije nula ako je c veliko slovo, inače nula
islower(c)	nije nula ako je c malo slovo, inače nula
isdigit(c)	nije nula ako je c znamenka, inače nula
isalnum(c)	nije nula ako je c slovo ili znamenka, inače nula
isspace(c)	nije nula ako je c pusti znak (razmak, tabulator,...), inače nula
toupper(c)	vraća znak c pretvoren u veliko slovo
tolower(c)	vraća znak c pretvoren u malo slovo

7.8.3 Funkcija ungetc

Standardna biblioteka osigurava prilično ograničenu verziju funkcije ungetch koju smo napisali u Poglavlju 4; njeno je ime ungetc.

```
int ungetc(int c, FILE *fp)
```

potiskuje znak c u datoteku fp, te vraća ili znak c ili EOF za slučaj greške. Jamči se samo jedan povratni znak po datoteci. Funkcija ungetc može se koristiti s bilo kojom od ulaznih funkcija poput scanf, getc ili getchar.

7.8.4 Izvršenje naredbe

Funkcija system(char *s) izvršava naredbu sadržanu u znakovnom nizu s, a potom nastavlja izvršavanje tekućeg programa. Sadržaj znakovnog niza potpuno ovisi o lokalnom operativnom sustavu. Kao trivijalan primjer, na UNIX sistemima, izraz

```
system("date");
```

uzrokuje pokretanje programa date; on ispisuje datum i dnevno vrijeme na standardni izlaz. Funkcija system vraća sistemski ovisnu cjelobrojnu vrijednost od izvršene naredbe. Na UNIX sistemu, povratna vrijednost je vrijednost vraćena funkcijom exit.

7.8.5 Upravljanje memorijom

Funkcije malloc i calloc dinamički pribavljaju memorijske blokove.

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

vraća pokazivač na n okteta (byte) neinicijalizirane memorije, a NULL ako ne može udovoljiti zahtjevu.

```
void *calloc(size_t n, size_t size)
```

vraća pokazivač na dovoljan prostor za polje od n objekata specificirane veličine, a NULL ako ne može udovoljiti zahtjevu. Memorija se inicijalizira na nulu.

Pokazivač kojeg vraćaju funkcije malloc i calloc ima propisno poravnanje glede objekata na koje pokazuje, no on se mora uobličiti u odgovarajući tip (pomoću operatora cast), kao u

```
int *ip;

ip=(int *) calloc(n, sizeof(int));
```

Funkcija free(p) oslobađa prostor na koji pokazuje pokazivač p, ako je p dobijen pozivom funkcija malloc ili calloc. Nema nikakvih ograničenja glede redoslijeda oslobađanja prostora, ali je stravična pogreška osloboditi nešto što nije dobijeno pozivom funkcija malloc ili calloc.

Također je pogreška upotrijebiti ono što je već oslobođeno. Tipičan, no nekorektan komadić koda je ova petlja koja oslobađa stavke iz liste:

```
for(p=head;p!=NULL;p=p->next) /* pogrešno */
      free(p);
```

Ispravan način je pohrana svega potrebnog prije oslobađanja:

```
for(p=head;p!=NULL;p=q){
      q=p->next;
      free(p);
}
```

Dio 8.7 pokazuje primjenu memorijskog pridjeljivača kakav je funkcija malloc, u kojemu se pridjeljeni blokovi daju osloboditi bilo kojim slijedom.

7.8.6 Matematičke funkcije

Više je od dvadeset matematičkih funkcija deklarirano u zaglavlju <math.h>; ovdje su neke češće korištene. Svaka treba jedan ili dva argumenta tipa double i vraća tip double.

sin(x)	sinus od x, x je izražen u radijanima
cos(x)	kosinus od x, x je izražen u radijanima
atan2(y,x)	arkus tangens od y/x, u radijanima
exp(x)	eksponencijalna funkcija ex
log(x)	prirodni logaritam (baza e) od x (x>0)
log10(x)	dekadski logaritam (baza 10) od x (x>0)
pow(x,y)	xy
sqrt(x)	drugi korijen od x (x>0)
fabs(x)	apsolutna vrijednost od x

7.8.7 Generiranje slučajnih brojeva

Funkcija rand() izračunava niz pseudo-slučajnih cijelih brojeva u intervalu od nule do vrijednosti RAND_MAX, koja je definirana u zaglavlju <stdlib.h>. Jedan od načina stvaranja slučajnih realnih brojeva većih ili jednakih nuli i manjih od jedan jest

```
#define frand() ((double) rand() / (RAND_MAX+1.0))
```

(Ako vaša biblioteka ima funkciju za slučajno generirane realne brojeve, vjerojatno će imati statistički bolje osobine od ove.)

Funkcija srand(unsigned) inicijalizira funkciju rand. Prenosiva implementacija funkcija rand i srand podržana standardom javlja se u dijelu 2.7.

Vježba 7-9. Funkcije kao isupper daju se implementirati radi uštede na prostoru i vremenu. Istražite obje mogućnosti.

POGLAVLJE 8: SUČELJE UNIX SISTEMA

Operativni sistem UNIX pruža niz pogodnosti kroz skup sistemskih poziva, a to su zapravo funkcije ugrađene u operativni sistem koje mogu biti pozvane iz korisničkih programa. Ovo poglavlje opisuje uporabu nekih bitnijih sistemskih poziva iz C programa. Ako koristite UNIX, to bi vam moglo od neposredne koristi, jer je katkad potrebno primijeniti sistemske pozive radi maksimiziranja efikasnosti, ili pak dobiti pristup do nekih pogodnosti koje nisu u biblioteci. Čak i ako rabite C na drugačijem sistemu, također, trebali bi proniknuti u C programiranje proučavajući ove primjere; mada se detalji razlikuju, sličan kod trebao bi biti na svakom sistemu. Kako je ANSI C biblioteka u velikoj mjeri modelirana prema UNIX pogodnostima, ovaj kod isto tako može pomoći vašem razumijevanju biblioteke.

Poglavlje je podijeljeno na tri osnovna dijela: ulaz/izlaz, datotečni sustav, te pridjeljivanje memorije. Prva dva dijela pretpostavljaju određenu sličnost s vanjskim karakteristikama UNIX sustava.

Poglavlje 7 obrađivalo je ulazno/izlazno sučelje jedinstveno za sve operativne sustave. Na bilo kojem sistemu, rutine standardne biblioteke trebaju biti napisane u skladu s mogućnostima tog sistema. U narednim dijelovima opisat ćemo pozive UNIX sustava za ulaz i izlaz, te ćemo pokazati kako se dijelovi standardne biblioteke daju pomoću njih implementirati.

8.1 Deskriptori datoteka

U UNIX operativnom sustavu, cijeli se ulaz i izlaz obavlja pisanjem i čitanjem datoteka, jer svi vanjski uređaji, čak i zaslon i tipkovnica, jesu datoteke u datotečnom sustavu. Ovo znači kako jedno cjelovito sučelje potpuno manipulira komunikacijom između programa i vanjskih uređaja.

U najopćenitijem slučaju, prije no što pročitate ili upišete datoteku, vi morate dati do znanja sustavu da to želite učiniti, a taj se postupak naziva otvaranje datoteke. Namjeravate li pisati u datoteku možda će biti potrebno njeno stvaranje ili brisanje njena prethodna sadržaja. Sustav provjerava vaša prava glede toga (Postoji li datoteka? Imate li pravo pristupa?), te ako je sve u redu, vraća programu mali nenegativan cijeli broj koji se zove deskriptor datoteke. Kad god treba pristupiti datoteci (bilo radi čitanja, bilo radi pisanja), rabi se deskriptor datoteke umjesto imena radi identifikacije. (Deskriptor datoteke analogan je pokazivaču datoteke kojeg upotrebljava standardna biblioteka ili manipulaciji datotekama u sistemu MSDOS). Svi podaci o otvorenoj datoteci predaju se sistemu, korisnički program obraća se datoteci samo preko deskriptora.

Kako se pojam ulaza i izlaza obično poima s tipkovnicom i zaslonom, postoje posebna rješenja koja čine takav pristup prikladnim. Kada izvođač naredbi ("shell") pokreće program, otvaraju se tri datoteke, s deskriptorima datoteka 0, 1 i 2, te imenima standardnog ulaza, izlaza i standardne greške. Ako program čita 0 i piše 1 i 2, on može pisati i čitati ne vodeći brigu o otvaranjima datoteka.

Korisnik programa može preusmjeriti U/I prema ili od datoteka pomoću < i >:

```
prog <infile >outfile
```

U ovom slučaju, radna okolina (shell) mijenja inicijalne postavke deskriptora datoteka 0 i 1 na navedene datoteke. Normalno je deskriptor datoteke 2 vezan sa zaslonom, pa se poruke o greškama mogu tamo ispisivati. Slična razmatranja vrijede za ulaz i izlaz koji su vezani na cjevovod. Program ne zna odakle dolazi ulaz ni gdje odlazi izlaz sve dok rabi datoteku 0 za ulaz, a 1 i 2 za izlaz.

8.2 Primitivni U/I - read i write

Ulaz i izlaz koriste sistemske pozive čitanja i pisanja, kojima se pristupa iz C programa preko dviju funkcija s imenima read i write. Za obje funkcije prvi argument je deskriptor datoteke. Drugi argument je znakovno polje u vašem programu u koje podaci ulaze ili iz njega dolaze. Treći argument je broj okteta (byte) koje treba prebaciti.

```
int n_read=read(int fd, char *buf, int n);

int n_written=write(int fd, char *buf, int n);
```

Svaki poziv vraća broj prebačenih okteta. Pri čitanju, broj prebačenih okteta može biti manji od traženog. Vraćena vrijednost od 0 okteta znači kraj datoteke, a -1 označava pojavu neke greške. Pri pisanju, vraćena vrijednost je broj upisanih okteta; greška se dogodila ako ovaj broj nije jednak traženom broju.

U jednom pozivu može se čitati i pisati bilo koji broj okteta. Uobičajene su vrijednosti 1, što znači jedan po jedan znak ("bez spremnika"), te brojevi poput 1024 i 4096 koji se odnose na veličinu bloka na vanjskim uređajima. Veće vrijednosti bit će efikasnije jer će napraviti manje sistemskih poziva.

Uzevši u obzir ove činjenice, možemo napisati jednostavan program koji kopira svoj ulaz na izlaz, ekvivalentan programu za kopiranje u Poglavlju 1. Ovaj program će kopirati bilo što, jer ulaz i izlaz mogu biti preusmjereni na bilo koju datoteku ili uređaj.

```
#include "syscalls.h"

main(){     /* kopira ulaz na izlaz */
      char buf[BUFSIZ];
      int n;
      while((n=read(0, buf, BUFSIZ))=0)
            write(1, buf, n);
      return 0;
}
```

Sakupili smo prototipove funkcija za sistemske pozive u datoteku pod imenom syscalls.h kako bi ih mogli uključiti u programe iz ovog poglavlja. Ovo ime, ipak, nije standard.

Parametar BUFSIZ također je definiran u zaglavlju syscalls.h; njegova vrijednost je prikladna za lokalni sistem. Ako veličina datoteke nije višekratnik veličine parametra BUFSIZ, poziv read će vratiti manji broj okteta od onog koji poziv write treba upisati; slijedeći poziv read vratit će nulu.

Poučno je vidjeti kako read i write mogu biti uporabljeni za izgradnju rutina višeg nivoa kakve su getchar, putchar, itd. Primjera radi, evo inačice funkcije getchar koja obrađuje nepohranjeni ulaz, čitajući znak po znak sa standardnog ulaza.

```
#include "syscalls.h"
/* getchar : jednoznakovni ulaz bez spremnika */
int getchar(void){
      char c;

      return(read(0, &c, 1)==1) ? (unsigned) c: EOF;
}
```

Znak c mora biti tipa char, jer poziv read treba pokazivač na znak. Prebacivanje znaka c u tip unsigned char u povratnom izrazu eliminira sve probleme oko predznaka.

Druga inačica funkcije getchar obavlja ulaz u velikim komadima, te predaje znakove jedan po jedan.

```
#include "syscalls.h"
/* getchar : jednostavna inačica sa spremnikom */
int getchar(void){
      static char buf[BUFSIZ];
      static *bufp=buf;
      static int n=0;

      if(n==0){   /* ako je spremnik prazan */
            n=read(0, buf, sizeof buf);
            bufp=buf;
      }
      return (--n>=0) ? (unsigned char) *bufp++ : EOF;
}
```

Ako bi ove inačice trebalo prevesti zajedno sa zaglavljem <stdio.h>, obavezno treba obaviti #undef imena getchar ako je implementiran kao makro.

8.3 open, creat, close, unlink

Osim za slučajeve pretpostavljenog standardnog ulaza, izlaza i greške, morat ćete eksplicitno otvoriti datoteke za njihovo čitanje ili pisanje. Za takvo što postoje dva sistemska poziva, open i creat[sic].

Sistemska funkcija open nalikuje funkciji fopen obrađenoj u Poglavlju 7, osim što namjesto povratne vrijednosti pokazivača na datoteku, ova vraća deskriptor datoteke, koja je cjelobrojna vrijednost, dakle, tipa int. Funkcija open vraća -1 ako se dogodi nekakva pogreška.

```
#include <fcntl.h>

int fd;
int open(char *name, int flags, int perms);

fd=open(name, flags, perms);
```

Kao i u funkciji fopen, argument name jest znakovni niz s imenom datoteke. Drugi argument, flags, cjelobrojna je vrijednost koja kaže kako će se datoteka otvoriti; osnovne vrijednosti su

```
O_RDONLY    otvoriti samo za čitanje
O_WRONLY    otvoriti samo za pisanje
O_RDWR      otvoriti i za čitanje i za pisanje
```

Ove konstante definirane su u zaglavlju <fcntl.h> na System V UNIX sistemima, te u zaglavlju <sys/file.h> na Berkeley (BSD) inačicama.

Kako bi otvorili postojeću datoteku za čitanje, pišemo

```
fd=open(name, O_RDONLY, 0);
```

Argument perms uvijek je nula za one uporabe funkcije open, koje ćemo mi razmatrati.

Greška je pokušaj otvaranja nepostojeće datoteke. Sistemska funkcija creat služi za stvaranje novih datoteka, ili pak ponovnog pisanja po starim datotekama.

```
int creat(char *name, int perms);

fd=creat(name, perms);
```

vraća deskriptor datoteke ako je uopće moguće kreirati datoteku, a -1 ako nije. Ako datoteka već postoji, funkcija creat će je svesti na veličinu nula, brišući njen prethodni sadržaj; dakle, nije greška kreiranje datoteke koja već postoji.

Ako datoteka već ne postoji, funkcija creat je stvara dopuštenjima definiranim u argumentu perms. Na UNIX sistemima, imamo devet bitova informacije o dopuštenjima vezanih uz datoteku koji reguliraju čitanje, pisanje i izvršavanje datoteke za vlasnika, grupu vlasnika i sve druge. Zato je troznamenkasti oktalni broj uobičajen za specificiranje dopuštenja. Npr., 0755 određuje dopuštenja za čitanje, pisanje i izvršavanje za vlasnika datoteke (7 ili binarno 111), čitanje i izvršavanje za grupu i sve ostale (55 ili binarno 101101).

Ilustracije radi, tu je pojednostavljena verzija UNIX programa cp, koji kopira jednu datoteku u drugu. Naša verzija kopira samo jednu datoteku, ne dopuštajući da drugi argument bude direktorij, a sama kreira dopuštenja, mjesto da ih kopira.

```
#include <stdio.h>
#include <fcntl.h>
#include "syscalls.h"
#define PERMS     0666  /* RW za vlasnika, grupu i druge */

void error(char *, ...);

/* cp : kopira f1 u f2 */

main(int argc, char *argv[]){
      int f1, f2, n;
      char buf[BUFSIZ];

      if(argc!=3)
            error("Usage: cp from to");
      if((f1=open(argv[1], O_RDONLY, 0))==-1)
            error("cp: can't open %s", argv[1]);
```

```
        if((f2=creat(argv[2], PERMS))==-1)
              error("cp: can't create %s, mode %030", argv[2], PERMS);
        while((n=read(f1, buf, BUFSIZ))>0)
              if(write(f2, buf, n)!=n)
                    error("cp: write error on file %s", argv[2]);
        return 0;
}
```

Ovaj program kreira izlaznu datoteku s fiksnim dopuštenjima 0666. Pomoću sistemske funkcije stat, opisane u dijelu 8.6, možemo odrediti modus postojeće datoteke i tako dati isti modus i kopiji.

Zamijetimo kako je funkcija error pozvana pomoću promjenjive liste argumenta koja sliči uvelike na funkciju printf. Primjena funkcije error pokazuje korištenje drugih funkcija iz printf familije. Funkcija standardne biblioteke, vprintf, ponaša se kao funkcija printf, osim što je lista promjenjivih argumenata zamijenjena jednim argumentom koji se inicijalizira pozivom va_start makroom. Slično, funkcije vfprintf i vsprintf odgovaraju funkcijama fprintf i sprintf.

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

/* error: ispisuje poruku o grešci i gasi se */

void error(char *fmt, ...){
      va_list args;

      va_start(args, fmt);
      fprintf(stderr, "error: ");
      vfprintf(stderr, fmt, args);
      fprintf(stderr, "\n");
      va_end(args);
      exit(1);
}
```

Postoji ograničenje (najčešće oko 20) vezano uz broj datoteka koje program može simultano otvoriti. Prema tome, svaki program koji namjerava obrađivati više datoteka mora biti pripremljen na ponovnu uporabu deskriptora datoteka. Funkcija close(int fd) prekida vezu između deskriptora datoteke i otvorene datoteke, te oslobađa deskriptor datoteke za uporabu s nekom datotekom; ona sliči na funkciju fclose u standardnoj biblioteci, osim što tu nema spremnika kojeg bi se punilo. Zatvaranje programa kroz funkciju exit ili vraćanjem iz funkcije main sve se otvorene datoteke zatvaraju.

Funkcija unlink(char *name) briše ime datoteke iz datotečnog sistema. Sliči na funkciju standardne biblioteke remove.

Vježba 8-1. Napišite ponovno program cat iz Poglavlja 7 koristeći read, write, open i close umjesto njihovih ekvivalenata iz standardne biblioteka. Obavite mjerenja radi određivanja relativnih brzina dviju inačica.

8.4 Slučajan pristup - lseek

Ulaz i izlaz su prirodno sekvencijalni; svako proces čitanja i pisanja zauzima mjesto u datoteci odmah iz prethodnog. Po potrebi, ipak, datoteka može biti čitana i pisana proizvoljnim slijedom. Sistemska funkcija lseek pruža mogućnost kretanja unutar datoteke bez čitanja ili pisanja bilo kakvih podataka:

```
long lseek(int fd, long offset, int origin);
```

postavlja tekuću poziciju unutar datoteke čiji je deskriptor fd na vrijednost varijable offset, što se uzima relativno s obzirom na lokaciju određenu varijablom origin. Naredno čitanje ili pisanje počet će s te pozicije. Varijabla origin može imati vrijednosti 0, 1 ili 2 što određuje da li će se vrijednost varijable offset mjeriti od početka, s trenutne pozicije ili s kraja datoteke, respektivno. Primjerice, radi dodavanja datoteci (što se da postići i redirekcijom >> pod UNIX ljuskom, ili opcijom "a" u funkciji fopen), ide do kraja datoteke prije pisanja:

```
lseek(fd, 0L, 2);
```

Radi povratka na početak ("rewind"),

```
lseek(fd, 0L, 0);
```

Primjetite argument 0L; on se dade pisati i kao (long) 0 ili samo 0, ako je funkcija lseek pravilno deklarirana.

Pomoću funkcije lseek, moguće je datoteke tretirati manje-više kao velika polja, uz cijenu sporijeg pristupa. Na primjer, slijedeća funkcija čita bilo koji broj okteta (byteova) s bilo kojeg mjesta u datoteci. Ona vraća broj pročitanih okteta, ili -1 u slučaju greške.

```
#include "syscalls.h"

/* get : čita n okteta s pozicije pos */
int get(int fd, long pos, char *buf, int n){
      if(lseek(fd, pos, 0)=0) /* ide na poziciju pos */
            return read(fd, buf, n);
      else
            return -1;
}
```

Povratna vrijednost funkcije lseek jest tipa long koja pokazuje na novu poziciju u datoteci, ili -1 za slučaj greške. Funkcija standardne biblioteke fseek slična je funkciji lseek osim što joj je prvi argument tipa FILE *, a povratna vrijednost nije nula ako se dogodi greška.

8.5 Primjer - Implementacija funkcija fopen i getc

Ilustrirajmo kako se ovi dijelovi slažu zajedno pokazujući implementaciju funkcija standardne biblioteke fopen i getc. Prisjetite se kako su datoteke u standardnoj biblioteci opisane pokazivačima datoteka, a ne deskriptorima. Pokazivač datoteke jest pokazivač na strukturu koja sadrži nekoliko dijelova informacije o datoteci: pokazivač na spremnik, kako bi se datoteka čitala u velikim blokovima; broj preostalih znakova ostalih u spremniku; pokazivač na slijedeću poziciju znaka u spremniku; deskriptor datoteka; zastavice koje određuju mod čitanja i pisanja, status greške itd.

Struktura podataka koja opisuje datoteku zapisana je u zaglavlju <stdio.h>, koje se mora uključiti (pomoću #include) u svakoj izvornoj datoteci koja rabi rutine iz standardne ulazno/izlazne biblioteke. Ona je, također, obuhvaćena funkcijama iz te biblioteke. U slijedećem izvatku iz tipičnog zaglavlja <stdio.h>, imena koja će koristiti samo funkcije iz biblioteke počinju potcrtom (underscore) kako bi se teže poistovjećivala s imenima iz korisničkog programa. Ovu konvenciju koriste sve rutine standardne biblioteke.

```
      #define     NULL  0
      #define     EOF   (-1)
      #define     BUFSIZ      1024
      #define     OPEN_MAX    20    /* maksimalni broj datoteka otvorenih
istovremeno */
      typedef struct _iobuf{
            int cnt;    /* preostalih znakova */
            char *ptr;  /* slijedeća pozicija znaka */
            char *base; /* mjesto spremnika */
            int flag;   /* način pristupa datoteci */
            int fd;     /* deskriptor datoteke */
      }     FILE;
      extern FILE _iob[OPEN_MAX];
      #define     stdin (&_iob[0])
      #define     stdout      (&_iob[1])
      #define     stderr      (&_iob[2])

      enum _flags{
            _READ=01,   /* datoteka otvorena za čitanje */
            _WRITE=02,  /* datoteka otvorena za pisanje */
```

```
        _UNBUF=04,  /* datoteka nije spremljena */
        _EOF=010,   /* naišlo se na EOF u datoteci */
        _ERR=020    /* naišlo se na grešku u ovoj datoteci */
};

int _fillbuf(FILE *);
int _flushbuf(int, FILE *);

#define     feof(p)     (((p)->flag&_EOF)!=0)
#define     ferorr(p)   (((p)->flag&_ERR)!=0)
#define     fileno(p)   ((p)->fd)
#define     getc(p)     (--(p)->cnt>=0 \
            ? (unsigned char) *(p)->ptr++ : _fillbuf(p))
#define     putc(x,p)   (--(p)->cnt>=0 \
            ? *(p)->ptr++ =(x) : _flushbuf((x),p))
#define getchar() getc(stdin)
#define     putchar(x)  putc((x),stdout)
```

Makro getc u normalnim okolnostima smanjuje brojač, povećava pokazivač i vraća znak (Prisjetite se kako se dugi #define izrazi nastavljaju znakom \ (backslash)). Ako pak brojač postane negativan, makro getc poziva funkciju _fillbuf da ponovo napuni spremnik, reinicijalizira sadržaj strukture i vrati znak. Vraćeni znakovi imaju unsigned tip, što osigurava njihovu pozitivnost.

Mada nećemo razmatrati detalje, dotakli smo definiciju makroa putc za pokazati kako funkcionira jako slično makrou getc, pozivajući funkciju _flushbuf kad je njegov spremnik pun. Spomenuli smo, također, makroe za pristup grešci, za oznaku kraja datoteke i deskriptor datoteke.

Sada možemo napisati funkciju fopen. Većina koda funkcije fopen bavi se otvaranjem datoteke i njenim ispravnim pozicioniranjem, te postavljanjem zastavica radi indiciranja pravog stanja. Funkcija fopen ne pridjeljuje prostor spremniku; to obavlja funkcija _fillbuf kad se datoteka prvi put učita.

```
#include <fnctl.h>
#include "syscalls.h"
#define     PERMS 0666  /* rw za vlasnika, grupu i ostale */
/* fopen : otvara datoteku, vraća datoteku ptr */
FILE *fopen(char *name, char *mode){
      int fd;
      FILE *fp;

      if (*mode!='r'&&*mode!='w'&&*mode!='a')
            return NULL;
      for(fp=_iob;fp<_iob+OPEN_MAX;fp++)
            if((fp->flag&(_READ|_WRITE))==0)
                  break;      /* pronađeno slobodno mjesto */
      if(fp>=_iob+OPEN_MAX)   /* nema slobodnog mjesta */
                  return NULL;
      if(*mode='w')
            fd=creat(name, PERMS);
      else if (*mode=='a'){
            if((fd=open(name, O_WRONLY, 0))==-1)
                  fd=creatname(name, PERMS);
            lseek(fd, 0L, 2);
      }
      else
            fd=open(name, O_RDONLY, 0);
      if(fd==-1)  /* ne može pristupiti imenu */
            return NULL;
      fp->fd=fd;
      fp->cnt=0;
      fp->base=NULL;
      fp->flag=(*mode=='r') ? _READ : _WRITE;
      return fp;
}
```

Ovaj oblik funkcije fopen ne manipulira svim mogućnostima pristupa koje pruža standard, iako njihovo dodavanje ne bi uzelo previše koda. Konkretno, naša funkcija fopen ne prepoznaje "b" kao oznaku binarnog pristupa, kako to nema većeg smisla na UNIX sistemima, niti "+" koja dopušta i čitanje i pisanje.

Prvi poziv funkcije getc zbog određene datoteke nalazi nulu na brojaču, pa se poziva funkcija _fullbuf. Ako funkcija _fillbuf uoči da datoteka nije otvorena za čitanje, odmah vraća EOF. Inače, pokušava alocirati spremnik (ako čitanje treba spremati).

Jednom kad je spremnik stvoren, funkcija _fillbuf poziva funkciju read kako bi ga popunila, postavlja brojač i pokazivače, te vraća znak na početak spremnika. Naknadni pozivi funkcije _fillbuf pronalazit će već alocirani spremnik.

```
#include "syscalls.h"
/* _fillbuf : alocira i puni spremnik */
int _fillbuf(FILE *fp){
      int bufsize;

      if((fp->flag&(_READ|_EOF|_ERR))!=_READ)
            return EOF;
      bufsize=(fp->flag&_UNBUF) ? 1 : BUFSIZ;
      if(fp->base==NULL)      /* još nema spremnika */
            if((fp->base=(char *) malloc(bufsize))==NULL)
                  return EOF; /* ne može dohvatiti spremnik */
      fp->ptr=fp->base;
      fp->cnt=read(fp->fd, fp->ptr, bufsize);
      if(--fp->cnt<0){
            if(fp->cnt==-1)
                  fp->flag|=_EOF;
            else
                  fp->flag|=_ERR;
            fp->cnt=0;
            return EOF;
      }
      return (unsigned char) *fp->ptr++;
}
```

Jedino što nismo razriješili jest kako sve to pokrenuti. Polje _iob mora biti definirano i inicijalizirano za tokove stdin, stdout i stderr:

```
FILE _iob[OPEN_MAX]={   /* stdin, stdout, stderr */
      {0, (char *) 0, (char *) 0, _READ, 0},
      {0, (char *) 0, (char *) 0, _WRITE, 0},
      {0, (char *) 0, (char *) 0, _WRITE|_UNBUF, 2},
};
```

Inicijalizacija zastavica u strukturi pokazuje kako se stdin treba čitati, stdout pisati, a stderr se treba pisati bez spremnika.

Vježba 8-2. Napišite nanovo funkcije fopen i _fillbuf s poljima umjesto eksplicitnim operacijama nad bitovima. Usporedite veličinu koda i brzinu izvršenja.

Vježba 8-3. Dizajnirajte i napišite funkcije _flushbuf, fflush i fclose.

Vježba 8-4. Funkcija iz standardne biblioteke

```
int fseek(FILE *fp, long offset, int origin)
```

jest identična s funkcijom lseek osim što je fp pokazivač datoteke, a ne deskriptor datoteke, a vraćena vrijednost je cjelobrojni status, a ne pozicija. Napišite funkciju fseek. Ovjerite se kako funkcija fseek pravilno upravlja spremanjem učinjenim za druge funkcije u biblioteci.

8.6 Primjer - Listanje direktorija

Različite vrste veza u datotečnom sistemu katkad zahtijevaju - određivanje informacije o datoteci, a ne o tome što ona sadrži. Program za listanje direktorija kakvog je primjer UNIX naredba ls - koja ispisuje imena datoteka u direktoriju, te, opcijski, druge podatke, poput veličina, dopuštenja i slično. MS-DOS ima analognu naredbu dir.

Kako se UNIX direktorij tretira kao datoteka, naredba ls je samo treba pročitati kako bi došla do željenih imena datoteka. No moramo koristiti sistemski poziv kako bi došli do drugih podataka o datoteci, kao npr. veličine. Na drugim sistemima, sistemski poziv bit će potreban čak i za pristup imenima datoteka; ovo je slučaj na MS-DOS-u npr. Ono što želimo jest omogućavanje pristupa informaciji na način relativno nezavisan od sistema, čak i ako implementacija može biti jako ovisna o sistemu.

Pokazat ćemo nešto od toga pišući program fsize. Program fsize jest poseban oblik programa ls koji ispisuje veličine svih datoteka iz popisa argumenata naredbene linije. Ako je jedna od datoteka direktorij, program se rekurzivno primjenjuje na taj direktorij. Ako argumenata uopće nema, obrađuje se radni direktorij.

Krenimo s kratkim pregledom strukture UNIX datotečnog sistema. Direktorij jest datoteka koja sadrži popis datoteka i nekakvu oznaku gdje su one smještene. "Lokacija" je oznaka drugoj tablici, pod nazivom "popis čvorova". Čvor datoteke je mjesto gdje su sve informacije vezane uz datoteku osim njenog imena. Unos jednog direktorija općenito se sastoji od samo dvije stvari, imena i broja čvora.

Na žalost, format i precizan sadržaj direktorija nisu jednaki na svim inačicama sistema. Tako ćemo posao podijeliti na dva dijela pokušavajući izolirati neprenosive dijelove. Vanjski nivo definira strukturu pod imenom Dirent i tri funkcije opendir, readdir i closedir koje omogućuju sistemski nezavisan pristup imenu i čvoru kod unosa direktorija. Napisat ćemo funkciju fsize uz pomoć ovog sučelja. Tada ćemo pokazati kako primijeniti to na sistemima koji rabe istu strukturu direktorija kao Version 7 i System V UNIX; varijacije mogu ostati za vježbu.

Struktura Dirent sadrži broj čvora i ime. Maksimalna dužina komponente imena datoteke jest NAME_MAX, koja predstavlja vrijednost zavisnu o sistemu. Funkcija opendir vraća pokazivač na strukturu s imenom DIR, slično kao FILE, koja se koristi u funkcijama readdir i closedir. Ova informacija sprema se u datoteku s imenom dirent.h

```
#define NAME_MAX 14     /* najduže ime datoteke */
                        /* zavisno o sistemu */
typedef struct{   /* univerzalan unos direktorija */
      long ino;   /* broj čvora */
      char name[NAME_MAX+1];  /* ime + '\0' */
} Dirent;

typedef struct{   /* minimalni DIR: bez spremanja itd. */
      int fd;     /* deskriptor datoteke za direktorij */
      Dirent d;   /* unos direktorija */
} DIR;

DIR *opendir(char *dirname);
Dirent *readdir(DIR *dfd);
void closedir(DIR *dfd);
```

Sistemski poziv stat uzima ime datoteke i vraća sve informacije u čvoru za tu datoteku, ili -1 za slučaj greške. Pa tako,

```
char *name;
struct stat stbuf;
int stat(char *, struct stat);

stat(name, &stbuf);
```

popunjava strukturu stbuf s čvornom informacijom o imenu datoteke. Struktura koja opisuje vrijednost vraćenu pozivom na stat je u zaglavlju <sys/stat.h> i tipično izgleda ovako:

```
struct stat{      /* čvorna informacija vraćena pomoću poziva stat */
dev_t st_dev      /* čvorni uređaj */
ino_t st_ino      /* broj čvora */
short st_mode     /* bitovi modaliteta */
short st_nlink    /* broj veza na datoteku */
short st_uid      /* korisnički ID vlasnika */
short st_gid      /* grupni ID vlasnika */
dev_t st_rdev     /* za specijalne datoteke */
off_t st_size     /* veličina datoteke u znakovima */
time_t      st_atime    /* posljednje vrijeme pristupa */
time_t      st_mtime    /* posljednje vrijeme promjene */
time_t      st_ctime    /* vrijeme zadnje izmjene čvora */
};
```

Većina ovih vrijednosti objašnjena je komentarima. Tipovi poput dev_t i ino_t definirani su u zaglavlju <sys/types.h>, koje mora biti također uključeno.

Unos st_mode sadrži skup zastavica koje opisuju datoteku. Definicije zastavica također su uključene u zaglavlje <sys/stat.h>; nama je interesantan samo onaj dio koji se bavi tipom datoteke:

```
#define     S_IFMT 016000     /* tip datoteke */
#define     S_IFDIR 0040000   /* direktorij */
#define     S_IFCHR 0020000   /* posebni znak */
#define     S_IFBLK 0060000   /* posebni blok */
#define     S_IFREG 0100000   /* regularno */

/* ... */
```

Sad smo spremni napisati program fsize. Ako modalitet dobiven iz strukture stat kaže kako datoteka nije direktorij, tada je veličina pri ruci i može odmah biti ispisana. Ako datoteka, pak, jest direktorij, morat ćemo obrađivati taj direktorij jednu po jednu datoteku; kako pri tom on može sadržavati poddirektorije, pa obrada mora ići rekurzivno.

Glavna rutina bavi se argumentima naredbene linije; ona prosljeđuje svaki argument funkciji fsize.

```
#include <stdio.h>
#include <string.h>
#include "syscalls.h"
#include <fcntl.h>      /* zastavice za čitanje i pisanje */
#include <sys/types.h>  /* definicije tipova */
#include <sys/stat.h>   /* struktura vraćena funkcijom stat */
#include "dirent.h"

void fsize(char *);

/* ispisuje veličine datoteka */
main(int argc, char **argv){

      if(argc==1) /* inicijalno: radni direktorij */
            fsize(".");
      else
            while(--argc>0)
                  fsize(*++argv);
      return 0;

}
```

Funkcija fsize ispisuje veličinu datoteke. Ako je datoteka direktorij, funkcija fsize najprije poziva funkciju dirwalk radi dohvaćanja svih datoteka u njemu. Primjetite kako se zastavice S_IFMT i S_IFDIR iz

<sys/stat.h> koriste pri određivanju datoteke kao direktorija. Zagrade igraju bitnu ulogu, jer je prioritet operatora & manji nego operatora ==.

```
int stat(char *, struct stat *);
void dirwalk(char *, void(*fcn)(char *));

/* fsize : ispisuje veličinu datoteke "name" */
void fsize(char *name){
      struct stat stbuf;

      if(stat(name, &stbuf)==-1){
            fprintf(stderr, "fsize: can't access %s\n", name);
            return;
      }
      if((stbuf.st_mode&S_IFMT)==S_IFDIR)
            dirwalk(name, fsize);
      printf("%8ld %s\n", stbuf.st_size, name);
}
```

Funkcija dirwalk jest općenita rutina koja primjenjuje funkciju na svaku datoteku u direktoriju. Ona otvara direktorij, prolazi kroz datoteke u njemu, pozivajući funkciju za svaku od njih, a zatim zatvara direktorij i vraća se. Kako funkcija fsize poziva funkciju dirwalk baš za svaki direktorij, dvije funkcije se rekurzivno pozivaju.

```
#define MAX_PATH  1024

/* dirwalk : primjeni fcn na sve datoteke u direktoriju */
void dirwalk(char *dir, void (*fcn)(char *)){
      char name[MAXPATH];
      Dirent *dp;
      DIR *dfd;

      if((dfd=opendir(dir))==NULL){
            fprintf(stderr, "dirwalk: cant't open %s\n", dir);
            return;
      }
      while((dp=readdir(dfd))!=NULL){
            if(strcmp(dp->name, ".")==0||strcmp(dp->name, "..")==0)
                  continue;   /* preskoči samog sebe i direktorij u kojem
se nalaziš */
            if(strlen(dir)+strlen(dp->name)+2>sizeof(name))
                  fprintf(stderr, "dirwalk: name %s%s too long\n", dir,
dp->name);
            else{
                  sprintf(name, "%s%s", dir, dp->name);
                  (*fcn)(name);
            }
      }
      closedir(dfd);
}
```

Svaki poziv funkcije readdir vraća pokazivač na podatak o slijedećoj datoteci, ili NULL kad datoteka više nema. Svaki direktorij sadrži podatke vezane uz njega (unos direktorija), pod imenom "." i direktorij u kojem se nalazi, pod imenom ".."; to mora biti preskočeno, kako se petlja ne bi vrtjela beskonačno.

Sve do ove razine, kod je nezavisan o načinu formatiranja direktorija. Slijedeći korak je predstavljanje minimalnih verzija funkcija opendir, readdir i closedir za određeni sistem. Ove rutine odnose se na Version 7 i System V UNIX sisteme; oni rabe podatak o direktoriju iz zaglavlja <sys/dir.h>, koji izgleda ovako:

```
#ifndef DIRSIZ
```

```
#define DIRSIZ    14
#endif
struct direct{    /* unos direktorija */
      ino_t d_ino;      /* broj čvora */
      char d_name[DIRSIZ];    /* dugačko ime nema '\0' */
};
```

Neke inačice sistema dopuštaju puno duža imena i imaju složeniju strukturu direktorija.

Tip ino_t jest typedef koji opisuje indeks u popisu čvorova. On može biti tipa unsigned short na sistemu koji normalno koristimo, no to nije podatak na koji se treba osloniti pri programiranju; razlikuje se od sistema do sistema, pa je uputno koristiti typedef. Potpun skup "sistemskih" tipova nalazi se u zaglavlju <sys/types.h>.

Funkcija opendir otvara direktorij, obavlja provjeru da li je datoteka direktorij (ovog puta pomoću sistemskog poziva fstat, koji je poput stat, ali se odnosi na deskriptor datoteke), pridjeljuje strukturu direktorija, te zapisuje informaciju:

```
int fstat(int fd, struct stat *);

/* opendir : otvara direktorij za readdir poziv */
DIR *opendir(char *dirname){
      int fd;
      struct stat stbuf;
      DIR *dp;

      if((fd=open(dirname, ORDONLY, 0))==-1||fstat(fd, &stbuf)==-
1||(stbuf.st_mode&S_IFMT)!=S_IFDIR||(dp=(DIR *) malloc(sizeof(DIR)))==NULL)
            return NULL;
      dp->fd=fd;
      return dp;
}
```

Funkcija closedir zatvara direktorij i oslobađa prostor:

```
/* closedir : zatvara direktorij otvoren pomoću funkcije opendir */
void closedir(DIR *dp){
      if(dp){
            close(dp->fd);
            free(dp);
      }
}
```

Konačno, funkcija readdir koristi read radi čitanja svakog unosa direktorija. Ako dio direktorija nije u upotrebi (jer je datoteka izbrisana), čvorni broj je nula, a pozicija se preskače. Inače, čvorni broj i ime nalaze se u strukturi static i pokazivač na nju vraća se korisniku. Svaki poziv prebriše informacije prethodnog.

```
#include <sys/dir.h>                    /* struktura lokalnog
direktorija */

/* readdir : čita ulaze direktorija u nizu */
Dirent *readdir(DIR *dp){
      struct direct dirbuf;             /* struktura lokalnog
direktorija */
      static Dirent d;            /* return : prenosiva struktura */
      while (read(dp->fd, (char *) &dirbuf,
sizeof(dirbuf))==sizeof(dirbuf)){
            if(dirbuf.d_ino==0)         /* slot koji se ne koristi */
                  continue;
            d.ino=dirbuf.d_ino;
            strncpy(d.name, dirbuf.d_name, DIRSIZ);
            d.name[DIRSIZ]='\0';        /* osigurava završetak */
            return &d;
```

```
        }
        return NULL;
}
```

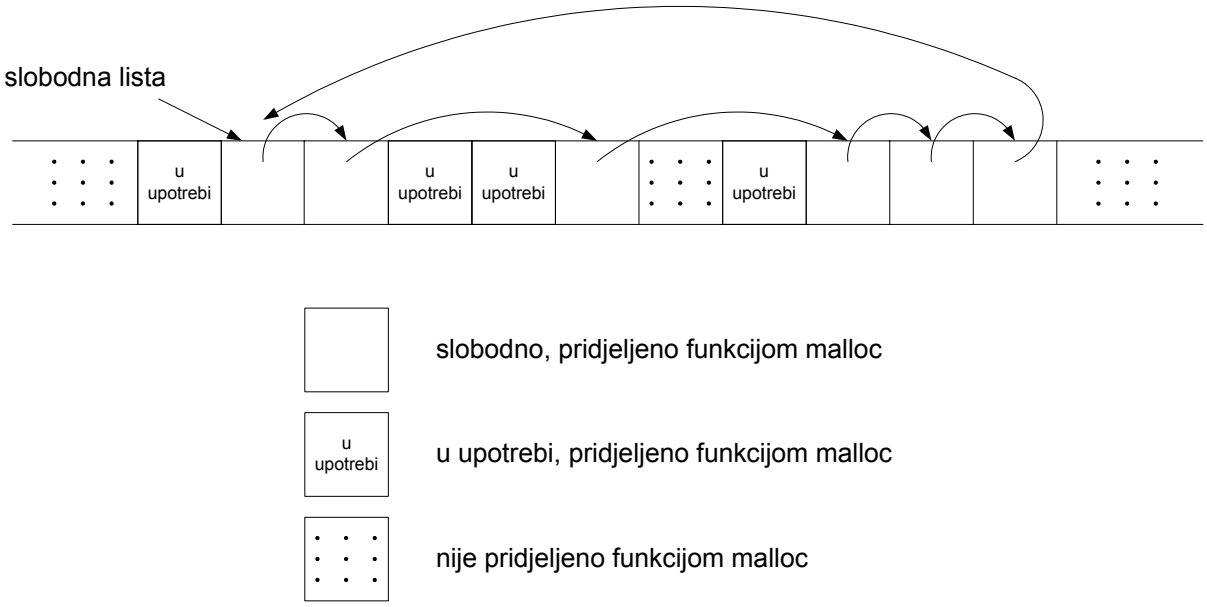
Iako je fsize namjenski program, predočava nekoliko važnih ideja. Najprije, mnogi programi nisu "sistemski programi"; oni samo koriste informacije dobivene od strane operativnog sistema. Za ovakove programe, najbitnije je da su informacije prezentirane samo u standardnim zaglavljima, te da programi uključuju te datoteke umjesto ubacivanja deklaracija. Druga značajka je da se s malo truda dade napraviti sučelje za objekte ovisne o sistemu, a koje je samo relativno neovisno o sistemu. Funkcije standardne biblioteke dobri su primjeri za to.

Vježba 8-5. Promijenite program fsize kako bi mogao ispisivati druge podatke iz čvornog ulaza.

8.7 Primjer - Pridjeljivač memorije

U Poglavlju 5 predstavili smo vrlo ograničen pridjeljivač memorije. Inačica koju ćemo sada napisati nema ograničenja. Pozivi funkcija malloc i free mogu se dogoditi bilo kojim slijedom; funkcija malloc poziva operativni sustav radi osiguravanja dovoljne količine memorije. Ove rutine ilustriraju neka od razmatranja vezana uz pisanje strojno-zavisnog koda na relativno strojno-nezavisan način, te pokazuje realnu primjenu struktura, unija i typedef-a.

Osim pridjeljivanja iz prevedenog polja fiksne veličine, funkcija malloc zahtjeva prostor od operativnog sustava po potrebi. Kako druge aktivnosti u programu mogu također zatražiti prostor bez pozivanja pridjeljivača, prostor koji nadgleda funkcija malloc ne mora biti cjelovit. Tako se njezin slobodni prostor čuva kao lista slobodnih blokova. Svaki blok sadrži veličinu, pokazivač na sljedeći blok i sami prostor. Blokovi su poredani po rastućim memorijskim adresama, a zadnji blok (najviše adrese) pokazuje na prvi.



Kada je zahtjev poslan, pretražuje se lista do pronalaženja dovoljno velikog bloka. Ovaj algoritam nazovimo "odgovarajućim" nasuprot, recimo, "najboljem" koji bi tražio najmanji blok koji udovoljava zahtjevu. Ako se pronađe blok točno tražene veličine on se briše iz liste slobodnih i predaje korisniku. Ako je blok prevelik, on se dijeli, pa se prava veličina predaje korisniku, a ostatak ostaje u listi slobodnih blokova. Ako nema dovoljno velikog bloka, uzima se sljedeći veći komad od operativnog sustava i povezuje u slobodnu listu.

Oslobađanje također uzrokuje pretragu slobodne liste, kako bi se našlo pravo mjesto umetanja oslobođenog bloka. Ako je blok koji se oslobađa susjedni već slobodnom bloku s bilo koje strane, oni se stapaju u jedan veći blok, kako ne bi bilo previše fragmentacije memorijskog prostora. Susjednost se lako određuje jer je slobodna lista uređena po rastućim adresama.

Jedan problem, na koji smo ukazali u Poglavlju 5, jest osigurati da memorijski prostor rezerviran funkcijom malloc pravilno podešena za objekte koji će se u nju smjestiti. Mada se računala razlikuju, svaki

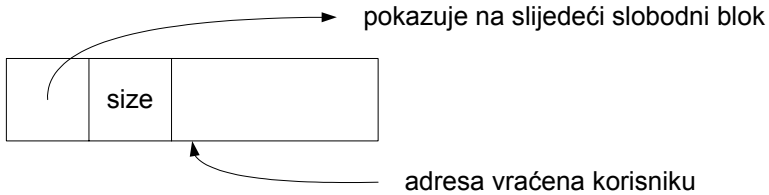
stroj ima tip podatka koji ga najviše ograničava; ako takav tip može biti pohranjen na određenoj adresi, tada to mogu i svi drugi tipovi. Na nekim računalima, najrestriktivniji tip je double, a na drugima je to int ili pak long.

Slobodan blok sadrži pokazivač na sljedeći blok u lancu, zapis o veličini bloka, te stvarni slobodan prostor; kontrolna informacija na početku naziva se "zaglavlje". Kako bi pojednostavnili podešavanje, svi blokovi su višekratnici veličine zaglavlja, a zaglavlje je pravilno podešeno. Ovo je postignuto unijom koja sadrži željenu strukturu zaglavlja i instancu najrestriktivnijeg tipa, za koji smo proizvoljno odabrali tip long.

```
      typedef long Align;                /* za određivanje veličine tipa long
*/
      union header{                      /* zaglavlje bloka */
            struct{
                  union header *ptr;     /* sljedeći blok ako je na slobodnoj
listi */
                  unsigned size;         /* veličina tog bloka */
            } s;
            Align x;                /* uredi blokove */
      };
      typedef union header Header;
```

Polje Align se ne koristi; ono samo služi da namjesti zaglavlja na najgori mogući slučaj.

U funkciji malloc, tražena veličina u znakovima se zaokružuje na pripadni broj veličina zaglavlja; blok koji će se rezervirati sadrži još jednu jedinicu, za samo zaglavlje, i to je veličina pohranjena u polju size zaglavlja. Pokazivač vraćen funkcijom malloc pokazuje na slobodan prostor, a ne na samo zaglavlje. Korisnik može činiti bilo što sa zahtjevanim prostorom, ali bilo kakvo pisanje van pridjeljenog prostora najvjerojatnije će rezultirati uništenom listom.



Blok vraćen funkcijom malloc

Polje veličine neizostavno je jer blokovi kontrolirani od strane funkcije malloc ne moraju biti kontinuirani - nije moguće izračunati veličine pokazivačkom aritmetikom.

Varijabla base rabi se za pokretanje. Ako je freep NULL, što je slučaj pri prvom pozivu funkcije malloc, kreira se degenerirana slobodna lista; ona sadrži blok veličine nula, a pokazuje na samu sebe. U svakom slučaju, slobodna se lista tada pretražuje. Traženje slobodnog bloka adekvatne veličine počinje u točki (freep) gdje je pronađen posljednji blok; ovaj pristup pomaže da lista ostane homogena. Ako je pronađen prevelik blok njegov ostatak se vraća korisniku; na ovaj način zaglavlje originala treba samo podesiti svoju veličinu. U svakom slučaju, pokazivač vraćen korisniku pokazuje na slobodan prostor unutar bloka, koji počinje jednu jedinicu poslije zaglavlja.

```
      static Header base;                /* prazna lista za pokretanje */
      static Header *freep=NULL;         /* početak prazne liste */

      /* malloc : pridjeljivač memorije opće namjene */
      void *malloc(unsigned nbytes){
            Header *p, *prevp;
            Header *morecore(unsigned);
            unsigned nunits;

            nunits=(nbytes+sizeof(Header)-1)/sizeof(Header)+1;
            if((prevp=freep)==NULL){      /* još nema slobodne liste */
                  base.s.ptr=freep=prevp=&base;
                  base.s.size=0;
```

```
        }
        for(p=prevp->s.ptr;;prevp=p, p=p->s.ptr){
              if(p->s.size>=nunits){        /* dovoljno veliko */
                    if(p->s.size==nunits)   /* točno */
                          prevp->s.ptr=p->s.ptr;
                    else{               /* pridjeli ostatak */
                          p->s.size-=nunits;
                          p+=p->s.size;
                          p->s.size=nunits;
                    }
                    freep=prevp;
                    return(void *)(p+1);
              }
              if(p==freep)              /* obavijeno oko slobodne liste */
                    if((p=morecore(nunits))==NULL)
                          return(NULL);     /* nijedan nije ostao */
        }
  }
```

Funkcija morecore dobija memorijski prostor od operativnog sistema. Detalji o tome kako se to obavlja variraju od sistema do sistema. Kako je traženje memorijskog prostora od sistema prilično zahtjevna operacija, mi ne želimo to činiti svakim pozivom funkcije malloc, pa funkcija morecore zahtjeva barem NALLOC jedinica; ovaj veći blok će se rezati po potrebi. Nakon postavljanja polja veličine, funkcija morecore ubacuje na scenu dodatnu memoriju pozivom funkcije free.

UNIX sistemski poziv sbrk(n) vraća pokazivač na još n okteta (byte) memorije. sbrk vraća -1 ako mjesta nema, mada bi NULL bilo bolje rješenje. Ovako se -1 mora prebaciti u tip char * kako bi se mogao usporediti s povratnom vrijednošću. Opet, cast operatori čine funkciju relativno otpornom na detalje izvedbe pokazivača na različitim računalima. Postoji još jedna pretpostavka, međutim, da pokazivači na različite blokove vraćeni pozivom sbrk mogu biti uspoređivani po značenju. To standard ne garantira, jer dopušta usporedbu pokazivača samo unutar istog polja. Zato je ova inačica funkcije malloc prenosiva samo među računalima koje karakterizira općenita usporedba pokazivača.

```
#define NALLOC    1024  /* zahtjev za minimalnim brojem jedinica */

/* morecore : traži od sistema još memorije */
static Header *morecore(unsigned nu){
      char *cp, sbrk(int);
      Header *up;
      if(nu<NALLOC)
            nu=NALOOC;
      cp=sbrk(nu*sizeof(Header));
      if(cp==(char *) -1)     /* nema više prostora */
            return NULL;
      up=(Header *) cp;
      up->s.size=nu;
      free((void *)(up+1));
      return freep;
}
```

Sama funkcija free jest najmanja briga. Ona pretražuje slobodnu listu, počevši od freep, tražeći mjesto za umetanje novog bloka. To mjesto je ili između dva postojeća bloka ili na kraju liste. U svakom slučaju, ako je blok koji se oslobađa nastavlja na susjedni, tada se blokovi povezuju. Jedini su problemi održavanje pokazivača kako bi pokazivali na prava mjesta i veličine.

```
/* free : stavlja blok ap na slobodnu listu */
void free(void ap){
      Header *bp, *p;
      bp=(Header *)ap-1;      /* pokazuje na zaglavlje bloka */
      for(p=freep;!(bp>p&&bp<p->s.ptr);p=p->s.ptr)
            if(p>=p->s.ptr&&(bp>p||bp<p->s.ptr))
```

```
                        break;      /* oslobođeni blok na početku ili kraju
područja */
                if(bp+bp->s.size==p->s.ptr){
                        bp->s.size+=p->s.ptr->s.size;
                        bp->ptr=p->s.ptr->s.ptr;
                }
                else
                        bp->s.ptr=p->s.ptr;
                if(p+p->s.size==bp){
                        p->s.size+=bp->s.size;
                        p->s.ptr=bp->s.ptr;
                }
                else
                        p->s.ptr=bp;
                freep=p;
        }
```

Mada pridjeljivanje memorije suštinski zavisi o računalu, gornji program pokazuje kako se zavisnosti mogu svesti na jako malen dio programa. Uporabom typedef-a i union-a obavlja se podešavanje (pod uvjetom da funkcija sbrk vraća odgovarajući pokazivač). Cast operatori brinu se da pretvorbe pokazivača budu transparentne, usprkos tome što katkad imaju posla s uistinu loše dizajniranim sistemskim sučeljem. Iako se ove pojedinosti odnose na pridjeljivanje memorije, općeniti pristup primjenjiv je i u drugim prilikama.

Vježba 8-6. Funkcija standardne biblioteke calloc(n, size) vraća pokazivač na n objekata veličine size, s memorijom inicijaliziranom na nulu. Napišite funkciju calloc, pozivajući funkciju malloc ili tako da je modificirate.

Vježba 8-7. Funkcija malloc prima zahtjev za veličinom bez provjere njegove prihvatljivosti; funkcija free vjeruje da je tražena veličina slobodnog bloka validan podatak. Poboljšajte ove funkcije kako bi se malo više pozabavile provjerom grešaka.

Vježba 8-8. Napišite funkciju bfree(p,n) koja će oslobađati proizvoljni blok p od n znakova u slobodnoj listi koju održavaju funkcije malloc i free. Rabeći bfree, korisnik može dodati statičko ili vanjsko polje slobodnoj listi kad god hoće.