➔ Sadržaj: ←

LEKCIJA 18: → Getting More from Functions ←

Kao što znate do sada, funkcije su centralne za **C** programiranje. Danas ćete naučiti još neke načine kako da koristite funkcije u vašim programima, uključujući:

- Upotreba pointera kao argumenata na funkcije
- Prosljeđivanje pointera tipa void funkcijama
- Korištenje funkcija sa varijabilnim brojem argumenata
- Vraćanje pointera od funkcije

→→→→ Prosljeđivanje pointera funkcijama

→ **Default**-ni metod za prosljeđivanje argumenta funkciji je **po vrijednosti**.
Prosljeđivanje po vrijednosti (passing by value) znači da je funkciji prosljeđena kopija vrijednosti argumenta.
Ovaj metod ima tri koraka:

1. Procjenjuje se izraz argumenta.
2. Rezultat je kopiran na **stack**, privremeni smještajni prostor u memoriji.
3. Funkcija vraća argumentovu vrijednost sa **stack**-a.

Kada je varijabla prosljeđena funkciji po vrijednosti, funkcija ima pristup vrijednosti varijabli, ali ne i oreginalnoj kopiji varijable. Kao rezultat, kood u funkciji ne može promijeniti (modifikovati) oreginalnu varijablu. Ovo je glavni razlog zašto je prosljeđivanje po vrijednosti default-ni metod prosljeđivanja argumenata: Podaci izvan funkcije su zaštićeni od nepažljive (inadvertent) modifikacije.

Prosljeđivanje argumenata po vrijednosti je moguće sa osnovnim tipovima podataka (**char**, **int**, **long**, **float**, i **double**) i **struct**urama.

→→ Postoji još jedan način prosljeđivanja argumenta funkciji, ipak: prosljeđivajući pointer varijabli argumenta, radije nego vrijednost same varijable. Ovaj metod prosljeđivanja argumenta se naziva **prosljeđivanje po referenci** (passing by reference).

Kao što ste naučili Dana 9, (“Understanding Pointers"), prosljeđivanje po referenci je jedini način da prosljedite niz ka funkciji; prosljeđivanje niza po vrijednosti nije moguće.
Sa ostalim tipovima podataka, ipak vi možete koristiti bilo koji metod.
Ako vaš program koristi velike strukture, njihovo prosljeđivanje po vrijednosti može prouzrokovati da nestane prostora na stack-u.
Pored ove opaske, prosljeđivanje argumenta po referenci, umjesto po vrijednosti, obezbjeđuje kako prednosti tako i nedostatke:

- Prednost prosljeđivanja po referenci je da funkcija može modifikovati (mijenjati) vrijednost varijablu argumenta.
- Nedostatak prosljeđivanja po referenci je da funkcija može modifikovati (mijenjati) vrijednost varijablu argumenta.

“Ša ba?”, možda se pitate. “**Prednost koja je ujedno i nedostatak??**” DA. Sve zavisi od situacije.
Ako je vašem programu potrebno da funkcija modifikuje varijablu argumenta, prosljeđivanje po referenci je prednost.
Ako ne postoji takva potreba, ona je nedostatak zbog mogućnosti nepažljive modifikacije.

Možda se pitate zašto ne koristite povratnu vrijednost funkcije za modifikovanje varijable argumenta. Vi možete uraditi to, naravno, kako je pokazano u sljedećem primjeru:

```
x = half(x);
int half(int y)
{
return y/2;
}
```

→ Ipak zapamtite, da funkcija može vratiti samo jednu vrijednost. Prosljeđivajući jedan ili više argumenata po referenci, vi dozvoljavate funkciji da “vrati” više od jedne vrijednosti pozivajućem programu. ←

Prosljeđivanje po referenci dozvoljava funkciji da modifikuje oreginalnu varijablu argumenta.

Kada vi prosljeđujete po referenci, vi morate osigurati da defeinicija funkcije i prototip reflektuju činjenicu da je argument koji je prosljeđen funkciji, pointer. Unutar tijela funkcije, vi također morate koristiti indirektni operator da pristupite varijabli(ama) prosljeđenim po referenci.

Listing **18.1** demonstrira prosljeđivanje po referenci i default-no prosljeđivanje po vrijednosti. Njen izlaz jasno pokazuje da, varijabla prosljeđena po vrijednosti, ne može biti promjenjena od strane funkcije, dok varijabla prosljeđena po referenci može biti promijenjena.
Naravno, funkcija ne mora da modifikuje varijablu prosljeđenu po referenci. U takvom slučaju, nema potrebe da prosljeđujete po referenci.

Listing 18.1. Prosljeđivanje po vrijednosti i prosljeđivanje po referenci.

```
1: /* Prosljedjivanje argumenata po vrijednosti i po referenci. */
2:
3: #include <stdio.h>
4:
5: void by_value(int a, int b, int c);
6: void by_ref(int *a, int *b, int *c);
7:
8: main()
9: {
10:     int x = 2, y = 4, z = 6;
11:
12:     printf("\nBefore calling by_value(), x = %d, y = %d, z = %d.",
13:         x, y, z);
14:
15:     by_value(x, y, z);
16:
17:     printf("\nAfter calling by_value(), x = %d, y = %d, z = %d.",
18:            x, y, z);
19:
20:     by_ref(&x, &y, &z);
21:     printf("\nAfter calling by_ref(), x = %d, y = %d, z = %d.\n",
22:            x, y, z);
23:     return(0);
24: }
25:
26: void by_value(int a, int b, int c)
27: {
28:     a = 0;
29:     b = 0;
30:     c = 0;
31: }
32:
33: void by_ref(int *a, int *b, int *c)
34: {
35:     *a = 0;
36:     *b = 0;
37:     *c = 0;
38: }
        Before calling by_value(), x = 2, y = 4, z = 6.
        After calling by_value(), x = 2, y = 4, z = 6.
        After calling by_ref(), x = 0, y = 0, z = 0.
```

ANALIZA: Ovaj program demonstrira razliku između prosljeđivanja varijabli po vrijednosti i njihovo prosljeđivanje po referenci.
Linije **5** i **6** sadrže prototipe za dvije funkcije koje su pozivane u ovom programu. Primjetite da linija **5** opisuje tri argumenta tipa **int** za **by_value()** funkciju, ali se **by_ref()** razlikuje u liniji **6** zato što zahtjeva tri pointera na varijable tipa **int** kao argumente.
Zaglavlja funkcija, za ove dvije funkcije u linijama **26** i **33** slijede isto format kao i prototipi. Tijela dviju funkcija su slična, ali ne i ista. Obje funkcije pridružuju **0** na tri varijable koje su prosljeđene njima.
U **by_value()** funkciji, **0** se pridružuje direktno na varijable. U **by_ref()** funkciji, korišteni su pointeri, tako da varijable moraju biti dereferencirane (razvezane (odvojene)).

Svaka funkcija se poziva jednom od **main()**-a. Prvo, trima varijablama, koje se prosljeđuju, su pridružene vrijednosti različite od **0** u liniji **10**.
Linija **12** printa tri vrijednosti na-ekran.
Linija **15** poziva prvu od dvije funkcije, **by_value()**.
Linija **17** printa ponovo tri varijable.
Primjetite da se one nisu promjenile. **by_value()** je primila varijable po vrijednosti, tako da nije mogla promjeniti njihov oreginalan sadržaj.
Linija **20** poziva **by_ref()**, i linija **22** printa vrijednosti ponovo. Ovaj put, sve vrijednosti su promijenjene na **0**.
Prosljeđivanje varijabli po referenci je dalo **by_ref()** pristup na stvarni sadržaj u varijablama.

Vi možete napisati funkciju koja prima (prihvata) neke argumente po referenci i ostale po vrijednosti. Samo zapamtite da ih držite pravo unutar funkcije, koristeći indirektni operator(*) (operator indirekcije) da razdvojite argumente prosljeđene po referenci.

NE prosljeđujte velike količine podataka po vrijednosti ako nije neophodno. Može vam nestati prostora na stack-u.
PROSLJEĐUJTE varijable po vrijednosti ako ne želite da se oreginalna vrijednost promjeni.
NE zaboravite da varijabla prosljeđena po referenci treba biti pointer. Također, koristite indirektni operator da razdvojite (dereferencirate) varijable u funkciji.

→→→→ Pointeri Tipa void

Vidjeli ste da je ključna riječ **void** korištena da navede ili da funkcija ne uzima argumente ili da ne vraća vrijednost.

Ključna riječ **void** također može biti korištena za kreiranje svojstvenog (generic) pointera →→→ pointer koji može pokazivati na objekat bilo kojeg tipa podataka. ←←←

Na primjer, iskaz:

```
void *x;
```

deklariše **x** kao svojstveni (generic) pointer.
x pokazuje na nešto; samo još niste naveli na šta.

Najčešća upotreba pointeera tipa **void** je u deklaraciji parametara funkcije. Vi možda želite da kreirate funkciju koja može raditi sa različitim tipovima argumenata. Vi ga možete proslijediti jednom kao tip **int**, sljedeći put tip **float**, itd. Deklarišući da funkcija uzima **void** pointer kao argument, vi se ne ograničavate na prihvatanje samo jednog tipa podataka. Ako deklarišete funkciju da uzima **void** pointer kao argument, vi možete proslijediti funkciji pointer na bilo šta.

Evo jednostavnog primjera: Vi želite funkciju koja prihvata numeričku varijablu kao argument i dijeli je sa **2**, vraćajući odgovor u varijabli argumenta. Tako da, ako varijabla **x** drži vrijednost **4**, nakon poziva do **half(x)**, varijabla **x** je jednaka sa **2**. Zato što želite da modifikujete argument, vi ga prosljeđujete po referenci. Zato što želite da koristite funkciju sa bilo kojim od **C**-ovih numeričkih tipova podataka, vi deklarišete funkciju da uzima **void** pointer:

```
void half(void *x);
```

Sad možete pozvati funkciju, prosljeđujući joj bilo koji pointer kao argument.
Ipak, postoji još jedna stvar koju trebate. Iako možete proslijediti **void** pointer bez da znate na koji tip podataka on pokazuje, vi ne možete razdvajati (dereferencirati) pointer. Prije nego što kood u funkciji može uraditi nešto sa pointerom, on mora da zna njegov tip podataka.
Ovo radite sa →→→ **typecast** ←←←, što nije ništa više nego način kako da kažete programu da tretira ovaj void pointer kao pointer na **type**. Ako je **x void** pointer, vi **typecast** kako slijedi:

```
(type *)x
```

Ovdje, **type** je odgovarajući tip podataka. Da kažete programu da je **x** pointer na tip **int**, napišite:

```
(int *)x
```

Da radvojite pointer → tj. da pristupite **int**-u na koji pokazuje **x** → napišite:

```
*(int *)x
```

typecasts se detaljnije rade Dana 20.
Vraćajući se na oreginalnu temu (prosljeđivanje **void** pointera funkciji), možete vidjeti da, da koristite pointer, funkcija mora da zna tip podataka na koji on pokazuje. U slučaju funkcije koju vi pišete da podijelite njen argument sa dva, postoje četiri mogućnosti za type: **int**, **long**, **float** i **double**. S dodatkom na **void** pointer na varijablu koja se dijeli sa dva, vi morate reći funkciji tip varijable na koju pokazuje **void** pointer.
Možete modifikovati definiciju funkcije kako slijedi:

```
void half(void *x, char type);
```

Na osnovu tipa argumenta, funkcija prebacuje **void** pointer **x** na odgovarajući tip. Onda pointer može biti razdvojen (dereferenciran), i vrijednost varijable na koju pokazuje može biti korištena. Konačna (finalna) verzija funkcije je pokazana u Listingu **18.2**.

Listing 18.2. Korištenje void pointera da se proslijede različiti tipovi podataka funkciji.

```
1: /* Upotreba pointera tipa void. */
2:
3: #include <stdio.h>
4:
5: void half(void *x, char type);
6:
7: main()
8: {
9:     /* Inicijaliziraj jednu varijablu od svakog tipa. */
10:
11:     int i = 20;
12:     long l = 100000;
13:     float f = 12.456;
14:     double d = 123.044444;
15:
16:     /* Prikazi njihove inicijalne vrijednosti. */
17:
18:     printf("\n%d", i);
19:     printf("\n%ld", l);
20:     printf("\n%f", f);
```

```
21:     printf("\n%lf\n\n", d);
22:
23:     /* Pozovi half() za svaku varijablu. */
24:
25:     half(&i, `i');
26:     half(&l, `l');
27:     half(&d, `d');
28:     half(&f, `f');
29:
30:     /* Prikazi njihove nove vrijednosti. */
31:     printf("\n%d", i);
32:     printf("\n%ld", l);
33:     printf("\n%f", f);
34:     printf("\n%lf\n", d);
35:     return(0);
36: }
37:
38: void half(void *x, char type)
39: {
40:     /* Zavisno od vrijednosti tipa, prebaci */
41:     /* pointer x prigodno (appropriately) i podijeli sa 2. */
42:
43:     switch (type)
44:     {
45:         case `i':
46:             {
47:             *((int *)x) /= 2;
48:             break;
49:             }
50:         case `l':
51:             {
52:             *((long *)x) /= 2;
53:             break;
54:             }
55:         case `f':
56:             {
57:             *((float *)x) /= 2;
58:             break;
59:             }
60:         case `d':
61:             {
62:             *((double *)x) /= 2;
63:             break;
64:             }
65:     }
66: }
20

            100000
            12.456000
            123.044444
            10
            50000
            6.228000
            61.522222
```

ANALIZA: Po implementaciji, funkcija **half()** u linijama **38** do **66** ne uključuje provjere grešaka (na primjer, ako je prosljeđen nepravilan tip argumenta). Ovo je zato što vi u realnim programima ne biste koristili funkciju da obavlja jednostavan zadatak kao što je djeljenje vrijednosti sa **2**. Ovo je samo ilustrativan primjer.

Možda mislite da bi potreba za prosljeđivanjem varijable na koju pokazuje (pointed-to variable) mogla funkciju učiniti manje flexibilnom. Funkcija bi bila uopštenija kada ne bi morala znati tip podataka objekta na koji se pokazuje (pointed-to data object), ali tako ne radi **C**.
Vi uvijek morate prebaciti (cast) **void** pointer na navedeni tip prije nego što razdvojite (dereferencirate). Uzimajući ovaj pristup, vi pišete samo jednu funkciju. Ako ne koristite **void** pointer, morali biste napisati četiri posebne funkcije → po jednu za svaki tip podataka.

Kada trebate funkciju koja može da radi sa različitim tipovima podataka, vi često možete napisati **makro** umjesto funkcije. Primjer koji je upravo predstavljen → u kojem je posao koji je odradila funkcija relativno jednostavan ← bi bio dobar kandidat za **makro**. (Dan 21 pokriva makroe).

PREBACITE (cast) **void** pointer kada koristite vrijednost na koju on pokazuje.
NE pokušavajte da inkrementirate ili dekrementirate **void** pointer.

→→→ Funkcije koje imaju varijabilan broj argumenata

Vi ste koristili nekoliko bibliotečnih funkcija, kao što su **printf()** i **scanf()**, koje uzimaju varijabilan broj argumenata. Vi možete pisati vlastite funkcije koje uzimaju varijabilnu listu argumenata.
Programi koji imaju funkcije sa varijabilnim argumentnim listama moraju uključivati file zaglavlja →→→ **STDARG.H** ←←←.

Kada deklarišete funkciju koja uzima varijabilnu listu argumenata, vi prvo listate fixne parametre → one koji su uvijek prisutni (mora postojati bar jedan fixni parametar). Onda uključite (include) ellipsis (**…**) na kraju liste parametara da indicirate da se nula ili više dodatnih argumenata prosljeđuje funkciji. Tokom ove diskusije, sjetite se razlike između parametara i argumenta, kao što je objašnjeno Dana 5, “Functions: The Basics”.

→ **Kako funkcija zna koliko joj je argumenata prosljeđeno na navedeni poziv (specific call)???**
VI JOJ KAŽETE. Jedan od fixnih parametara obavještava funkciji ukupni broj argumenata. Na primjer, kada koristite **printf()** funkciju, broj konverzionih specifikatora u format stringu govori funkciji koliko dodatnih argumenata da očekuje. Preciznije, jedan od funkcijskih fixnih argumenata može biti broj dodatnih argumenata.
Primjer kojeg ćete vidjeti uskoro koristi ovaj pristup, ali prvo treba da pogledate alate koje **C** obezbjeđuje za rad sa varijabilnom listom argumenata.

Funkcija takođe mora znati tip od svakog argumenta u varijabilnoj listi.
U slučaju za **printf()**, konverzioni specifikatori indiciraju tip svakog argumenta. U ostalim slučajevima, kao što je sljedeći primjer, svi argumenti u varijabilnoj listi su istog tipa, tako da nema problema.
Da kreirate funkciju koja prihvata različite tipove u varijabilnoj listi argumenata, vi morate smisliti (razviti (devise)) metod prosljeđivanja informacija o tipovima argumenata. Na primjer, vi možete koristiti karakterni kood, kao što je bilo urađeno za funkciju **half()** u Listingu **18.2**.

Alati za korištenje varijabilne liste argumenata su definisani u **STDARG.H**. Ovi alati su koriste unutar funkcije da dobiju (retrieve) argumente u varijabinoj listi.
Oni su kako slijedi:

va_list	Pointer tipa podataka (A pointer data type).
va_start()	Makro korišten za inicijalizaciju argumentne liste.
va_arg()	Makro korišten da dobije (retrieve) svaki argument, u povratku, iz varijabilne liste.
va_end()	Makro korišten za “čisćenje” kada su dobijeni (retrieved) svi argumenti.

Posebno je navedeno kako se ovi makroi koriste u funkciji, i onda sam uključio i primjer. Kada je funkcija pozvana, kood u funkciji mora slijediti ove korake da pristupi svojim argumentima:

1. Deklariši pointer varijablu tipa **va_list**. Ovaj pointer se koristi za pristup pojedinačnim argumentima. Česta je praxa, iako sigurno ne neophodna, da se poziva ova varijabla **arg_ptr**.
2. Pozovi makro **va_start**, prosljeđujući mu pointer **arg_ptr** kao i ime zadnjeg fixnog argumenta. Makro **va_start** nema povratnu vrijednost; on inicijalizira pointer **arg_ptr** da pokazuje na prvi argument u varijabilnoj listi.
3. Da dobijete (retrieve) svaki argument, pozovi **va_arg()**, prosljeđujući joj pointer **arg_ptr** i tip podataka sljedećeg argumenta. Povratna vrijednost od **va_arg()** je vrijednost sljedećeg argumenta. Ako je funkcija primila **n** argumenata u varijabilnoj listi, pozovi **va_arg() n** puta da dobije argumente u redosljedu u kojem su izlistani u pozivu funkcije.
4. Kada su svi argumenti iz varijabilne liste dobijeni, pozovi **va_end()**, prosljeđujući joj pointer **arg_ptr**. U nekim implementacijama, makro ne obavlja nikakvu akciju, ali u ostalim, on obavlja neophodne akcije čišćenja. Vi biste trebali dobiti naviku pozivanja **va_end()** u slučaju da koristite **C**-ove implementacije koje ga zahtjevaju.

Sad na taj primjer. Funkcija **average()** u Listingu **18.3** izračunava aritmetički prosjek liste **integer**a. Ovaj program prosljeđuje funkciji jedan fixni argument, pokazujući broj dodatnih argumenata koji slijede nakon liste brojeva.

Listing 18.3. Upotreba variabilne-veličine argumentne liste.

```
1: /* Funkcije sa varijabilnom listom argumenata. */
2:
3: #include <stdio.h>
4: #include <stdarg.h>
5:
6: float average(int num, ...);
7:
8: main()
9: {
10:     float x;
11:
12:     x = average(10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
13:     printf("\nThe first average is %f.", x);
14:     x = average(5, 121, 206, 76, 31, 5);
15:     printf("\nThe second average is %f.\n", x);
16:     return(0);
17: }
18:
19: float average(int num, ...)
20: {
21:     /* Deklarisi varijablu tipa  va_list. */
22:
23:     va_list arg_ptr;
24:     int count, total = 0;
25:
26:     /* Inicijaliziraj argument pointer. */
27:
28:     va_start(arg_ptr, num);
29:
30:     /* Dobavi (retrieve) svaki argument u varijabilnoj listi. */
31:
32:     for (count = 0; count < num; count++)
33:         total += va_arg( arg_ptr, int );
34:
35:     /* Obavi chishtchenje. */
36:
37:     va_end(arg_ptr);
38:
```

```
39:     /* Podijeli total sa brojem vrijednosti da se dobije */
40:     /* prosjek. Prebaci (cast) total na tip float, tako da je */
41:     /* povratna vrijednost tipa float. */
42:
43:     return ((float)total/num);
44: }
            The first average is 5.500000.
            The second average is 87.800000.
```

ANALIZA: Funkcija **average()** se prvo poziva u liniji **19**. Prvi proslјeđen argument, jedini fixni argument, navodi broj vrijednosti u varijabilnoj listi argumenata.
U funkciji, kako se dobavlja svaki argument u varijabilnoj listi u linijama **32** do **33**, on se dodaje na varijablin total. Nakon što su dobavljeni svi argumenti, linija **43** prebacuje total kao tip **float** i onda dijeli **total** sa **num** da dobije prosjek (average).

Još dvije stvari bi trebale biti istaknute u ovom listingu:
Linija **28** poziva **va_start()** da inicijalizira argumentnu listu. Ovo mora biti urađeno prije nego što se vrijednosti dobiju.
Linija **37** poziva **va_end()** da "očisti", jer je funkcija završila sa vrijednostima.

Vi ne biste trebali koristiti obje ove funkcije i vašem programu kada pišete funkciju sa varijabilnim brojem argumenata.

Precizno govoreći, funkcija koja prihvata broj argumenata ne mora imati fixni parametar koji je informiše o broju argumenata koji se proslјeđuju. Na primjer, vi možete markirati kraj liste argumenata sa specijalnom (posebnom) vrijednošću koja nije korišten nigdje drugo. Ova metoda postavlja granice na argumente koji se mogu proslijediti, ipak, tako da je najbolje izbjegavati.

→→→→ Funkcije koje vraćaju pointer

U prethodnom poglavlju ste vidjeli nekoliko funkcija iz **C**-ove standardne biblioteke čija je povratna vrijednost **pointer**. Vi možete napisati vaše funkcije koje vraćaju pointer.
Kao što možete očekivati, indirektni operator (*****) je korišten u deklaraciji funkcije i u definiciji funkcije.

Opšti oblik deklaracije je:

```
type *func(parameter_list);
```

Ovaj iskaz deklariše funkciju **func()** koja vraća pointer na **type**.
Evo dva konkretna primjera:

```
double *func1(parameter_list);
struct address *func2(parameter_list);
```

Prva linija deklariše funkciju koja vraća pointer na tip **double**.
Druga linija deklariše funkciju koja vraća pointer na tip **adrese** (koji pretpostavljate da je korisničko-definisana struktura).

→ **Ne miješajte funkciju koja vraća pointer sa pointerom na funkciju.** Ako uključite dodatni par zagrada u deklaraciji, vi deklarišete pointer na funkciju, kako je pokazano u sljedeća dva primjera:

```
double (*func)(...);    /* Pointer na funkciju koja vraca double. */
double *func(...);      /* Funkcija koja vraca pointer na double. */
```

→ Sad kada ste naučili format deklaracije, kako koristiti funkcije koje vraćaju pointer???

Ne postoji ništa specijalno o takvim funkcijama → vi ih koristite kao i bilo koju drugu funkciju, pridružujući njenu povratnu vrijednost varijabli prigodnog tipa (u ovom slučaju, pointer). Zato što je poziv funkcije **C**-ov izraz, vi ga možete koristiti bilo gdje, gdje biste koristili pointer tog tipa.

Listing **18.4** prezentuje jednostavan primjer, funkcija, kojoj se prosljeđuju dva argumenta i odlučuje koji je veći. Listing pokazuje dva načina da se ovo uradi: jedna funkcija vraća kao **int**, i druga vraća pointer na **int**.

Listing 18.4. Vraćanje pointera od funkcije.

```
1: /* Funkcija koja vraca pointer. */
2:
3: #include <stdio.h>
4:
5: int larger1(int x, int y);
6: int *larger2(int *x, int *y);
7:
8: main()
9: {
10:     int a, b, bigger1, *bigger2;
11:
12:     printf("Enter two integer values: ");
13:     scanf("%d %d", &a, &b);
14:
15:     bigger1 = larger1(a, b);
16:     printf("\nThe larger value is %d.", bigger1);
17:     bigger2 = larger2(&a, &b);
18:     printf("\nThe larger value is %d.\n", *bigger2);
19:     return(0);
20: }
21:
22: int larger1(int x, int y)
23: {
24:     if (y > x)
25:         return y;
26:     return x;
27: }
28:
29: int *larger2(int *x, int *y)
30: {
31:     if (*y > *x)
32:         return y;
33:
34:     return x;
35: }
            Enter two integer values: 1111 3000
            The larger value is 3000.
            The larger value is 3000.
```

ANALIZA: Ovo je relativno lak program za praćenje.
Linije **5** i **6** sadrže prototipe za ove dvije funkcije.
Prva, **larger1()**, prima dvije **int** varijable i vraća **int**.
Druga, **larger2()**, prima (receives) dva pointera na varijable **int** i vraća pointer na **int**.
main() funkcija u linijama od **8** do **20** je jasna (straightforward).
Linija **10** deklariše četiri varijable. **a** i **b** drže dvije varijable koje se uoređuju.
bigger1 i **bigger2** drže povratne vrijednosti od **larger1()** i **larger2()** funkcija, respektivno. Primjetite da je **bigger2** pointer na **int**, i **bigger1** je samo **int**.

Linija **15** poziva **larger1()** sa dva **int**-a, **a** i **b**. Vrijednost koja je vraćena od funkcije je pridružena na **bigger1**, koja se printa u liniji **16**.
Linija **17** poziva **larger2()** sa adrese od dva **int**-a. Vrijednost vraćena od **larger2()**, pointer, je pridružena na **bigger2**, takođe pointer. Ova vrijednost je odvojena (dereferenced) i printana u sljedećoj liniji.

Dvije upoređujuće funkcije su vrlo slične. One obje upoređuju dvije vrijednosti. Vraćena je veća vrijednost. Razlika između funkcija je u **larger2()**. U ovoj funkciji, vrijednosti na koje se pokazuje su upoređene u liniji **31**. Onda je vraćen pointer na veću vrijednost varijable. Primjetite da je dereferencirajući operator korišten u upoređivanjima (komparacijama), ali ne i u povratnim iskazima u linijama **32** i **34**.

U većini slučajeva, kao i u Listingu **18.4**, jednako je izvodljivo (feasible) da napišete funkciju da vrati vrijednost ili pointer. Izbor zavisi od specifikacija vašeg programa → većinom kako planirate da koristite povratnu vrijednost.

KORISTITE sve elemente koji su opisani u ovom poglavlju kada pišete funkcije koje imaju varijabilne argumente. Ovo je true čak i ako vaš kompajler ne zahtjeva sve elemente. Elementi su **va_list**, **va_start()**, **va_arg()** i **va_end()**.

NE miješajte pointere na funkcije, sa funkcijama koje vraćaju pointere.

Sažetak

U ovom poglavlju, vi ste naučili neke dodatne stvari koje vaš **C** program može raditi sa funkcijama. Naučili ste razliku između **prosljeđivanja argumenata po vrijednosti i po referenci**, i kako kasnije tehnike dozvoljavaju funkciji da "**vrati**" više od jedne vrijednosti pozivajućem programu.
Takođe ste vidjeli kako se može koristiti tip **void** da se kreira svojstven (generičan) pointer koji može pokazivati na bilo koji **C**-ov tip podatkovnog objekta. Pointeri tipa **void** su najčešće korišteni sa funkcijama kojima se mogu proslijediti argumenti koji nisu ograničeni na jedan tip podataka. Zapamtite da pointer tipa **void** mora biti prebačen (cast) u navedeni tip prije nego što ga možete odvojiti (dereferencirati).

Ovo vam je poglavlje takođe pokazalo kako se koriste **makro**i definisani u **STDARG.H** za **pisanje funkcija koje prihvataju varijabilan broj argumenata**. Takve funkcije obezbjeđuju razumnu flexibilnost programiranja. Konačno, vidjeli ste kako se pišu **funkcije koje vraćaju pointer**.

P&O

P Da li je prosljeđivanje pointera kao argumente funkciji uobičajena praxa u C programiranju???
O Definitivno!! U puno instanci, funkcija treba da promijeni vrijednost od više varijabli, i postoje dva načina kako ovo može biti postignuto. **Prvi** je da deklarišete i koristite globalne varijable. **Drugi** je da proslijedite pointere tako da funkcija može modifikovati podatke ispravno. Prva opcija se savjetuje samo kada će skoro svaka funkcija koristiti varijablu; u drugom slučaju, trebali biste je izbjegavati.

P Da li je bolje modifikovati varijablu pridružujući joj povratnu vrijednost funkcije ili prosljeđivanjem funkciji pointera na varijablu???
O Kada treba da modifikujete samo jednu varijablu u funkciji, obično je najbolje vratiti vrijednost od funkcije, nego proslijediti pointer na funkciju. Logika iza ovoga je jednostavna. NE prosljeđujući pointer, vi ne pokrećete rizik od mijenjanja nekih podataka koje ne namjeravate promjeniti, i držite funkciju nezavisnom od ostatka kooda.

Vježbe

1. Napišite prototip za funkciju koja vraća **integer**. Trebala bi uzeti pointer na karakterni niz kao svoj argument.
2. Napišite prototip za funkciju nazvanu **numbers** koja uzima tri **integer** argumenta. **Integer**i bi trebali biti prosljeđeni po referenci.
3. Pokažite kako biste pozvali brojeve (numbers) funkcije u vježbi **2** sa tri **integer**a **int1**, **int2**, i **int3**.

4. BUG BUSTER: Da li nešto nije u redu sa sljedećim??

```
void squared(void *nbr)
{
*nbr *= *nbr;
}
```

5. BUG BUSTER: Da li nešto nije u redu sa sljedećim??

```
float total( int num, ...)
{
int count, total = 0;
for ( count = 0; count < num; count++ )
total += va_arg( arg_ptr, int );
return ( total );
}
```

6. Napišite funkciju kojoj (**a**) je prosljeđen varijabilan broj stringova kao argumenti, (**b**) concatenates stringove, po redosljedu, u jedan duži string, i (**c**) vraća pointer na novi string na pozivajući program.

7. Napišite funkciju koja (**a**) prosljeđen joj je niz od bilo kojih numeričkih tipova podataka kao argument, (**b**) pronalazi najveću i najmanju vrijednost u nizu, i (**c**) vraća pointere na ove vrijednosti do pozivajućeg programa. (Hint: Treba vam neki način da kažete funkciji koliko elemenata je u niizu).

8. Napišite funkciju koja prihvata string i karakter. Funkcija bi trebala gledati na prvo pojavljivanje karaktera u stringu i vraćati pointer na tu lokaciju.

LEKCIJA 19: ➔ Exploring the C Function Library ➔

Kao što ste vidjeli kroz ovu knjigu, puno **C**-ove snage dolazi od funkcija u standardnim bibliotekama. U ovom poglavlju ćete istražiti neke od funkcija koje ne spadaju u teme ostalih poglavlja. Danas ćete naučiti o:

- Matematičkim funkcijama
- Funkcijama koje rade sa vremenom
- Funkcijama koje rješavaju greške (Error-handling functions)
- Funkcijama za traženje i sortiranje podataka

→→→ Matematičke funkcije

Standardna **C**-ova biblioteka sadrži razne funkcije koje obavljaju matematičke operacije.

Prototipi za matematičke funkcije su u file-u zaglavlja →→→ **MATH.H** ←←←.

Sve matematičke funkcije vraćaju tip **double**.
Za trigonometrijske funkcije, uglovi su izraženi u radijanima.
Zapamtite da, jedan **rad**ijan je jednak sa **57.296** stepeni, i puni krug (**360** stepeni) sadrži **2p** radijana.

→→→ Trigonometrijske funkcije

Trigonometrijske funkcije odabljaju računanja koja se koriste u nekim grafičkim i inženjerskim aplikacijama.

Funkcija	Prototip	Opis
acos()	**double acos(double x)**	Vraća **arccos** svog argumenta. Argument mora biti u opsegu **-1 <= x <= 1**, i povratna vrijednost je u opsegu **0 <= acos <= p**.
asin()	**double asin(double x)**	Vraća **arcsin** svog argumenta. Argument mora biti u opsegu **-1 <= x <= 1**, i povratna vrijednost je u opsegu **-p/2 <= asin <= p/2**.
atan()	**double atan(double x)**	Vraća **arctg** svog argumenta. Povratna vrijednost je u opsegu **-p/2 <= atan <= p/2**.
atan2()	**double atan2(double x, double y)**	Vraća **arctg** od **x/y**. Vraćena vrijednost je u opsegu **-p <= atan2 <= p**.
cos()	**double cos(double x)**	Vraća **cos** svog argumenta.
sin()	**double sin(double x)**	Vraća **sin** svog argumenta.
tan()	**double tan(double x)**	Vraća **tg** svog argumenta.

→→→ Exponencijalne i logaritamske funkcije

Exponencijalne i logaritamske funkcije su potrebne za neke tipove matematičkih proračuna.

Funkcija	Prototip	Opis
exp()	**double exp(double x)**	Vraća prirodni exponent svog argumenta, tj., $\mathbf{e^x}$ gdje je **e** jednako **2.7182818284590452354**.
log()	**double log(double x)**	Vraća prirodni logaritam svog argumenta. Argument mora biti veći od **0**.
log10()	**double log10(double x)**	Vraća logaritam s bazom **10**, svog argumenta. Argument mora biti veći od **0**.
frexp()	**double frexp(double x, int *y)**	Funkcija računa normaliziran razlomak koji predstavlja vrijednost **x**. Povratna vrijednost funkcije **r** je razlomak (frakcija) u opsegu **0.5 <= r <= 1.0**. Funkcija pridružuje **y**-u **integer** exponent takav da je **x = r * 2y**. Ako je prosljeđena vrijednost funkciji **0**, onda su i **r** i **y 0**.
ldexp()	**double ldexp(double x, int y)**	Vraća **x * 2y**.

→→→ Hiperbolične funkcije

Hiperbolične funkcije obavljaju hiperbolične trigonometrijske kalkulacija (izračunavanja).

Funkcija	Prototip	Opis
cosh()	**double cosh(double x)**	Vraća hiperbolični cos svog argumenta.
sinh()	**double sinh(double x)**	Vraća hiperbolični sin svog argumenta.
tanh()	**double tanh(double x)**	Vraća hiperbolični tg svog argumenta.

→→→ Ostale matematičke funkcije

Standardna **C**-ova biblioteka sadrži sljedeće raznovrsne (miscellaneous)matematičke funkcije:

Funkcija	Prototip	Opis
sqrt()	**double sqrt(double x)**	Vraća **kvadratni korjen** svog argumenta. Argument mora biti nula ili veći.
ceil()	**double ceil(double x)**	Vraća **najmanji integer** ne manji od svog argumenta. Na primjer, **ceil(4.5)** vraća **5.0**, i **ceil(-4.5)** vraća **-4.0**. Iako ceil() vraća integer vrijednost, ona je vraćena kao tip double.
abs()	**int abs(int x)**	Vraća **apsolutnu** vrijednost
labs()	**long labs(long x)**	svojih argumenata.
floor()	**double floor(double x)**	Vraća **najveći integer** ne veći od svog argumenta. Na primjer, **floor(4.5)** vraća **4.0**, i **floor(-4.5)** vraća **–5.0.**
modf()	**double modf(double x, double *y)**	Razdvaja **x** na integralni i razlomačke dijelove (splits **x** into integral and fractional parts), svaki sa istim znakom kao **x**. Razlomački dio je vraćen od funkcije, i integralni dio je pridružen na ***y**.
pow()	**double pow(double x, double y)**	Vraća **xy**. Greška se dešava ako je **x == 0** i **y <= 0**, ili ako **x < 0** i **y** nije integer.
fmod()	**double fmod(double x, double y)**	Vraća ostatak floating-point-a od **x/y**, sa istim znakom kao i **x**. Funkcija vraća **0** ili **x == 0**.

→→ Demonstracija matematičkih funkcija

Cijela knjiga bi se mogla popuniti sa programima koji demonstriraju sve matematičke funkcije. Listing **19.1** sadrži jedan program koji demonstrira nekoliko tih funkcija.

Listing 19.1. Upotreba C-ovih bibliotečnih matematičkih funkcija.

```
1: /* Demonstracija nekih C-ovih matematichkih funkcija */
2:
3: #include <stdio.h>
4: #include <math.h>
5:
6: main()
7: {
8:
9:     double x;
10:
11:     printf("Enter a number: ");
12:     scanf( "%lf", &x);
13:
14:     printf("\n\nOriginal value: %lf", x);
15:
16:     printf("\nCeil: %lf", ceil(x));
17:     printf("\nFloor: %lf", floor(x));
18:     if( x >= 0 )
19:         printf("\nSquare root: %lf", sqrt(x) );
20:     else
21:        printf("\nNegative number" );
22:
23:     printf("\nCosine: %lf\n", cos(x));
24:     return(0);
25: }
```

```
Enter a number: 100.95
Original value: 100.950000
Ceil: 101.000000
Floor: 100.000000
Square root: 10.047388
Cosine: 0.913482
```

ANALIZA: Ovaj listing koristi nekoliko matematičkih funkcija. Vrijednost koja je prihvaćena (accepted) u liniji **12** je printana. Onda se prosljeđuje do četiri **C**-ovih bibliotečnih matematičkih funkcija →→→ **ceil()** ←←←, →→→ **floor()** ←←←, →→→ **sqrt()** ←←← i →→→ **cos()** ←←←. Primjetite da je **sqrt()** pozvana samo ako broj nije negativan.
Po definiciji, negativni brojevi nemaju kvadratni korjen. Vi možete dodati bilo koju drugu matematičku funkciju u program kao što je ova da testira njihovu funkcionalnost.

→→→ Rad sa vremenom (Dealing with Time)

C-ova biblioteka sadrži nekoliko funkcija koje vam dopuštaju da radite sa vremenom. U **C**-u, pojam **times** (vremena) se odnosi kako na datume tako i na vrijeme.
Prototipi funkcija i definicije strukutra korištenih od puno vremenskih funkcija su u file-u zaglavlja →→→ **TIME.H** ←←←.

→→ Predstavljanje vremena

C-ove vremenske funkcije predstavljaju vrijeme na dva načina. Osnovniji metod je **broj sekundi** koje su protekle od ponoći **Januara 1, 1970**. Negativne vrijednosti su korištene da predstavljaju vrijeme prije tog datuma.

Ove vremenske vrijednosti su smještene kao **integer**i tipa **long**.
U →→→ **TIME.H** ←←←, simboli **time_t** i **clock_t** su obje definisane sa **typedef** iskazima kao **long**.
Ovi simboli se koriste u prototipima vremenskih funkcija radije nego **long**.

→ Drugi metod predstavlja vrijeme razbijeno na komponente: godina, mjesec, dan, itd. Za ovu vrstu predstavljanja, vremenske funkcije koriste strukturu **tm**, definisanu u **TIME.H** kako slijedi:

```
struct tm {
int tm_sec;     /* sekundi nakon minute - [0,59]  */
int tm_min;     /* minuta nakon sata - [0,59]    */
int tm_hour;    /* sati nakon ponoci - [0,23]  */
int tm_mday;    /* dana u mjesecu - [1,31]      */
int tm_mon;     /* mjeseci nakon Januara - [0,11]      */
int tm_year;    /* godina nakon 1900                */
int tm_wday;    /* dana nakon Nedjelje - [0,6]      */
int tm_yday;    /* dana nakon Januara 1 - [0,365] */
int tm_isdst;   /* daylight savings time flag     */
};
```

→→ Vremenske funkcije (The Time Functions)

Ova sekcija opisuje razne **C**-ove bibliotečne funkcije koje rade sa vremenom. Zapamtite da se pojam time (vrijeme) odnosi kako na datum, tako i na sate, minute, i sekunde. Demonstracioni program slijedni nakon opiisaa.

→→ Dobijanje tekućeg vremena (Obtaining the Current Time)

Da se dobije tekuće (trenutno) vrijeme kao što je postavljeno u sistemskom internom satu, koristite →→→ **time()** ←←← funkciju.
Njen prototip je:

```
time_t time(time_t *timeptr);
```

Zapamtite, **time_t** je definisan u **TIME.H** kao sinonim za **long**. Funkcija **time()** vraća broj sekundi koje su protekle nakon ponoći, Januara 1, 1970.
Ako je prosljeđeno ništa-**NULL** pointer, **time()** takođe smješta ovu vrijednost u varijablu tipa **time_t** na koju pokazuje **timeptr**. Tako da, za smještanje tekućeg vremena u varijablu tipa **time_t** sada, možete pisati:

```
time_t now;
now = time(0);
```

Takođe možete napisati:

```
time_t now;
time_t *ptr_now = &now;
time(ptr_now);
```

→→ Konverzije između predstavljanja vremena

Znati broj sekundi nakon Januara 1, 1970, često nije od velike pomoći. Tako da, **C** obezbjeđuje mogućnost konvertovanja vremena koje je proteklo, kao vrijednost **time_t** u **tm** strukturu, koristeći →→→ **localtime()** ←←← funkciju.
tm struktura sadrži dan, mjesec, godinu, i ostale informacije o vremenu u formatu koji je prigodniji za prikazivanje i printanje.
Prototip ove funkcije je:

```
struct tm *localtime(time_t *ptr);
```

Ova funkcija vraća pointer na strukturu **tm** tipa **static**, tako da ne morate deklarisati strukturu tipa **tm** da koristi → samo pointer na tip **tm**. Ova **static** struktura je ponovo koristi i prepisuje (overwritten) svaki put kada se pozove **localtime()**; ako želite da snimite (spasite (save)) vraćenu vrijednost, vaš program mora deklarisati odvojenu **tm** strukturu i kopirati vrijednosti sa **static** (statične) strukture.

Obrnuta konverzija → iz tipa **tm** strukture u vrijednost tipa **time_t** → se obavlja sa funkcijom →→→ **mktime()** ←←←.
Njen prototip je:

```
time_t mktime(struct tm *ntime);
```

Ova funkcija vraća broj sekundi između ponoći, Januara 1, 1970, i vremena koje je predstavljeno sa strukturom tipa **tm** na koju pokazuje **ntime**.

→→ Prikazivanje vremena (Displaying Times)

Da konvertujete vrijeme u formatirane stringove prigodne za prikazivanje, koristite funkcije →→→ **ctime()** ←←← i →→→ **asctime()** ←←←.
Obadvije ove funkcije vraćaju vrijeme kao string sa navedenim formatom. One se razlikuju u tome što se **ctime()**-u prosljeđuje vrijeme, kao vrijednost tipa **time_t**, dok se **asctime()**-u prosljeđuje vrijeme, kao struktura tipa **tm**.

Njihovi prototipi su:

```
char *asctime(struct tm *ptr);
char *ctime(time_t *ptr);
```

Obadvije funkcije vraćaju pointer na **static**, **null**-terminirajući, **26**-karakterni string koji daje vrijeme argumenata funkcija u sljedećem formatu:

```
Thu Jun 13 10:22:23 1991
```

Vrijeme je formatirano u 24-satno “vojno” vrijeme. Obadvije funkcije koriste **static** string, prepisujući njegov prethodni sadržaj, svaki put kada su pozvane.

Za više kontrole nad formatom vremena, koristite funkciju →→→ **strftime()** ←←←. Ovoj funkciji se prosljeđuje vrijeme kao struktura tipa **tm**. Ona formatira vrijeme prema formatu stringa.
Prototip funkcije je:

```
size_t strftime(char *s, size_t max, char *fmt, struct tm *ptr);
```

Ova funkcija uzima vrijeme u strukturi tipa **tm** na koju pokazuje **ptr**, formatira je prema format stringu **fmt**, i piše (writes) rezultat kao **null**-terminirajući string na memorijsku lokaciju na koju pokazuje **s**.
Argument **max** bi trebao navesti količinu prostora alociranog za **s**.
Ako rezultirajući string (uključujući i terminirajući **null** karakter) ima više od **max** karaktera, funkcija vraća **0**, i string je nepravilan (invalid). U drugom slučaju, funkcija vraća broj napisanih karaktera → **strlen(s)** ←.

Format string sadrži jedan ili više konverzionih specifikatora iz Tabele **19.1**.
Tabela 19.1. Konverzioni specifikatori koji mogu biti korišteni sa strftime().

Specifikator	S čim se zamijenjuje (What It's Replaced By)
%a	Skraćeno ime dana u sedmici (Abbreviated weekday name).
%A	Puno ime dana u sedmici (Full weekday name).
%b	Skraćeno ime mjeseca.
%B	Puno ime mjeseca.
%c	Predstavljanje datuma i vremena (npr., 10:41:50 30-Jun-91).
%d	Dan u mjesecu kao decimalni broj 01 do 31.
%H	Sat (24-satni saat) kao decimalni broj od 00 do 23.
%I	Sat (12-satni saat) kao decimalni broj od 00 do 11.
%j	Dan godine kao decimalni broj od 001 do 366.
%m	Mjesec kao decimalni broj od 01 do 12.
%M	Minuta kao decimalni broj od 00 do 59.
%p	AM ili PM.
%S	Sekunde kao decimalni broj od 00 do 59.
%U	Sedmica u godini kao decimalni broj od 00 do 53. Nedjelja se smatra prvim danom u sedmici.
%w	Dan u sedmici kao decimalni broj od 0 do 6 (Nedjelja = 0).
%W	Sedmica u godini kao decimalni broj od 00 do 53. Ponedjeljak se smatra prvim danom u sedmici.
%x	Predstavljanje datuma (npr., 30-Jun-91).
%X	Predstavljanje vremena (npr., 10:41:50).
%y	Godina, bez vijeka, kao decimalni broj od 00 do 99.
%Y	Godina, sa vijekom, kao decimalan broj.
%Z	Ime vremenske zone ako je informacija dostupna, ili blank ako ne.
%%	Jedan znak postotka %.

→→→ Izračunavanja vremenskih razlika (Calculating Time Differences)

Vi možete računati razlike, u sekundama, između dva vremena sa →→→ **difftime()** ←←← makroom, koji oduzima dvije **time_t** vrijednosti i vraća razliku.
Prototip je:

```
double difftime(time_t later, time_t earlier);
```

Ova funkcija oduzima ranije od kasnijeg i vraća razliku, broj sekundi između dva vremena.
difftime() se obično koristi da se izračuna proteklo vrijeme, kako je demonstrirano (zajedno sa ostalim vremenskim operacijama) u Listingu **19.2**.

Vi možete odrediti trajanje različite vrste koristeći →→→ **clock()** ←←← funkciju, koja vraća količinu vremena koje je prošlo od kad je izvršenje programa počelo, u **1/100**-sekundama jedinice.
Prototip (za **clock()**) je:

```
clock_t clock(void);
```

Da odredite trajanje nekog dijela programa, pozovite **clock()** dva puta → prije i poslije dešavanja procesa ← i oduzmite dvije vraćene vrijednosti.

→→→ Korištenje vremenkih funkcija

Listing **19.2** demonstrira kako se koriste **C**-ove bibliotečne vremenske funkcije.

Listing 19.2. Upotreba C-ovih bibliotečnih vremenskih funkcija.

```
1: /* Demonstrira vremenske funkcije. */
2:
3: #include <stdio.h>
4: #include <time.h>
5:
6: main()
7: {
8:      time_t start, finish, now;
9:      struct tm *ptr;
10:      char *c, buf1[80];
11:      double duration;
12:
13:      /* Snimi vrijeme pocetka izvrsavanja programa. */
14:
15:      start = time(0);
16:
17:      /* Snimi tekuce vrijeme, koristeci alternativni metod */
18:      /* pozivanja  time(). */
19:
20:      time(&now);
21:
22:      /* Konvertuj time_t vrijednost u strukturu tipa tm. */
23:
24:      ptr = localtime(&now);
25:
26:      /* Kreiraj i prikazi formatiran string koji sadrzi */
27:      /* tekuce vrijeme. */
28:
29:      c = asctime(ptr);
30:      puts(c);
31:      getc(stdin);
32:
33:      /* Sad koristi strftime() funkciju za kreiranje razlicitih */
34:      /* formatiranih verzija vremena. */
35:
36:      strftime(buf1, 80, "This is week %U of the year %Y", ptr);
37:      puts(buf1);
38:      getc(stdin);
39:
40:      strftime(buf1, 80, "Today is %A, %x", ptr);
41:      puts(buf1);
42:      getc(stdin);
43:
44:      strftime(buf1, 80, "It is %M minutes past hour %I.", ptr);
45:      puts(buf1);
46:      getc(stdin);
47:
48:      /* Sad uzmi tekuce vrijeme i racunaj trajanje programa. */
49:
50:      finish = time(0);
51:      duration = difftime(finish, start);
52:      printf("\nProgram execution time using time() = %f seconds.",
¬duration);
53:
54:      /* Takodje prikazi trajanje programa u stotinkama */
55:      /* koristeci clock(). */
56:
```

```
57:      printf("\nProgram execution time using clock() = %ld hundredths of
¬sec.",
58:          clock());
59:      return(0);
60: }
      Mon Jun 09 13:53:33 1997
      This is week 23 of the year 1997
      Today is Monday, 06/09/97
      It is 53 minutes past hour 01.
      Program execution time using time() = 4.000000 seconds.
      Program execution time using clock() = 4170 hundredths of sec.
```

ANALIZA: Ovaj program ima nekoliko linija komentara, tako da bi ga trebalo biti lako slijediti.
File zaglavlja **TIME.H** je uključen u liniji **4**, zato što se koriste vremenske funkcije.
Linija **8** deklariše tri varijable tipa **time_t** → **start**, **finish**, i **now**. Ove varijable mogu držati vrijeme kao offset od Januara 1, 1970, u sekundama.
Linija **9** deklariše pointer na strukturu **tm**. Struktura **tm** je opisana ranije. Ostatak varijabli ima tipove koji bi vam trebao biti poznat.

Programski snima (records) svoje početno vrijeme u liniji **15**. Ovo je urađeno sa pozivom na **time()**.
Program onda radi istu stvar na drugi način. Umjesto da koristi vrijednost vraćenu od **time()** funkcije, linija **20** prosljeđuje **time()**-u pointer na varijablu **now**.
Linija **24** radi tačno ono što kaže komentar u liniji **22**: Ona konvertuje **time_t** vrijednost od **now** u strukturu tipa **tm**.
Nekoliko sljedećih dijelova programa printaju vrijednost tekućeg vremena na ekran u različitim formatima.
Linija **29** koristi **asctime()** funkciju da pridruži informaciju na karakterni pointer, **c**.
Linija **30** printa formatiranu informaciju.
Program zatim čeka na korisnika da pritisne **Enter**.

Linije **36** do **46** koriste **strftime()** funkciju za printanje datuma u tri različita formata. Koristeći Tabelu **19.2**, trebali biste u mogućnosti da odredite šta ove tri linije printaju.

Program zatim određuje vrijeme (time) ponovo u liniji **50**. Ovo je vrijeme završetka programa.
Linija **51** koristi ovo vrijeme završavanja zajedno sa vremenom početka da izračuna trajanje programa pomoću **difftime()** funkcije. Ova vrijednost je printana u liniji **52**.
Program završava printajući vrijeme izvršavanja programa sa **clock()** funkcije.

→→→ Funkcije za rad sa greškama (Error-Handling Functions)

C-ova standardna biblioteka sadrži razne funkcije i makroe koje vam pomažu sa programskim greškama.

→→→ assert() ←←← Funkcija

Makro **assert()** može dijagnosticirati programske bug-ove.
Ona je definisana u →→→ **ASSERT.H** ←←←, i njen prototip je:

```
void assert(int expression);
```

Argument **expression** (izraz) može biti bilo šta što želite da testirate → varijabla ili bilo koji **C** izraz ←.
Ako **expression** procijeni **TRUE**, **assert()** ne radi ništa.
Ako **expression** procijeni **FALSE**, **assert()** prikazuje poruku greške na **stderr** i obustavlja izvršenje programa.

→ Kako koristite assert()?
Najčešće je korištena za otkrivanje programskih bug-ova (koji se razlikuju od kompilacijskih bug-ova). **Bug** ne sprječava program od kompajliranja, ali uzrokuje da daje (program) netačne informacije ili da ne radi ispravno (zaglavljuje se, na primjer).
Na primjer, financijsko-analitički program koji pišete može povremeno davati netačne informacije. Vi sumnjate da je problem prouzrokovan od strane varijable **interest_rate** koja uzima negativnu vrijednost, što se na bi smjelo dešavati. Da provjerite ovo, stavite iskaz:

```
assert(interest_rate >= 0);
```

na lokacijama u programu gdje se **interest_rate** koristi. Ako varijabla ikad postane negativna, **assert()** makro vas upozorava o tome. Onda možete ispitati relevantan kood da locirate uzrok problema.

Da vidite kako **assert()** radi, pokrenite Listing **19.3**. Ako unesete nenula vrijednost, program prikazuje vrijednost i terminira se normalno. Tačna poruka greške koju vidite će zavisiti od vašeg kompajlera, ali evo tipičnog primjera:

```
Assertion failed: x, file list19_3.c, line 13
```

Primjetite da, u cilju da **assert()** radi, vaš program mora biti kompajliran u debug-mode-u. Pogledajte dokumentaciju kompajlera za informaciju o omogućavanja debug mode-a (kako je objašnjeno za malo). Kada kasnije kompajlirate finalnu verziju u release mode-u, **assert()** makroi su onemogućeni.

Listing 19.3. Upotreba assert() makroa.
```
1: /*  assert() makro. */
2:
3: #include <stdio.h>
4: #include <assert.h>
5:
6: main()
7: {
8:     int x;
9:
10:     printf("\nEnter an integer value: ");
11:     scanf("%d", &x);
12:
13:     assert(x >= 0);
14:
15:     printf("You entered %d.\n", x);
16:     return(0);
17: }
```

```
Enter an integer value: 10
You entered 10.
Enter an integer value: -1
Assertion failed: x, file list19_3.c, line 13
Abnormal program termination
```

Vaša poruka greške se može razlikovati, zavisno od vašeg sistema i kompajlera, ali uopšte, ideja je ista.

ANALIZA: Pokrenite ovaj program da vidite da poruke greške, koje su prikazane sa **assert()** na liniji **13**, uključuju izraz čiji test pada (failed), ime file-a, i broj linije gdje je **assert()** lociran.

Akcija od **assert()** zavisi od drugog makroa nazvanog **NDEBUG** (što znači “no debugging”). Ako makro **NDEBUG** nije definisan (po default-u), **assert()** je aktivan. Ako je **NDEBUG** definisan, **assert()** je ugašen i nema efekta.
Ako smjestite **assert()** na razne programske lokacije da vam pomogne u debug-iranju i onda rješavanju problema, vi možete definisati **NDEBUG** da ugasi **assert()**. Ovo je mnogo lakše nego ići kroz program uklanjajući **assert()** iskaze (samo da se kasnije otkrije da želite da ih koristite ponovo).

Da definišete makro **NDEBUG**, koristite **#define** direktivu.
Ovo možete demonstrirati dodajući liniju:

```
#define NDEBUG
```

u Listing **19.3**, u liniji **2**. Sad program printa vrijednost unešenu i onda terminira normalno, čak i ako unesete **–1**.

Primjetire da **NDEBUG** ne treba da bude definisan kao ništa naročito, sve dok je uključen u **#define** direktivu. Naučićete više o **#define** direktivi Dana 21.

→→→ ERRNO.H ←←← file zaglavlja

File zaglavlja **ERRNO.H** definiše nekoliko makro-a koji se koriste da definišu i dokumentiraju runtime greške (greške u izvođenju). Ovi makroi su korišteni u konjukciji sa →→→ **perror()** ←←← funkcijom, opisanom u sljedećem dijelu.

ERRNO.H definicije uključuju externi integer nazvan **errno**. Većina **C**-ovih bibliotečnih funkcija pridružuje vrijednost na ovu varijablu ako se desi greška prilikom izvršavanja funkcije.
File **ERRNO.H** takođe definiše grupu simboličnih konstanti za ove greške, navedene u Tabeli **19.2**.

Table 19.2. Simbolične konstante grešaka definisane u ERRNO.H.

Ime	Vrijednost	Poruka i značenje (Message and Meaning)
E2BIG	**1000**	Lista argumenata je predugačka (dužina liste prelazi 128 byte-ova).
EACCES	**5**	Dozvola odbijena (npr., pokušavate da pišete u file, otvoren samo za čitanje).
EBADF	**6**	Loš opis file-a (Bad file descriptor).
EDOM	**1002**	Matematički argument van domena (argument prosljeđen do matematičke funkcije je van dozvoljenog opsega).
EEXIST	**80**	File izlazi (exists).
EMFILE	**4**	Previše otvorenih file-ova (Too many open files).
ENOENT	**2**	Ne postoji takav file ili direktorij.
ENOEXEC	**1001**	Exec format greška (Exec format error).
ENOMEM	**8**	Nema dovoljno jezgre (npr., nema dovoljno memorije za izvršenje exec() funkcije).
ENOPATH	**3**	Put nije pronađen (Path not found).
ERANGE	**1003**	Rezultat je van opsega (npr., rezultat vraćen od matematičke funkcije je prevelik ili premalen za tip povratnog podatka).

Vi možete koristiti **errno** na dva načina.
Neke funkcije signaliziraju, pomoću njihove povratne vrijednosti, da se desila greška. Ako se ovo desi, vi možete testirati vrijednost od **errno** da odredite prirodu greške i preduzmete potrebnu akciju.
U drugom slučaju, kada nemate navedenu indikaciju da se desila greška, vi možete testirati **errno**. Ako je nenula, desila se greška, i navedena (specifična) vrijednost od **errno** indicira na prirodu (vrstu) greške. Budite sigurni da resetujete **errno** na nula kada radite sa greškama.

Sljedeća sekcija objašnjava **perror()**, i onda Listing **19.4** ilustruje korištenje **errno**.

→→→ perror() ←←← Funkcija

perror() funkcija je još jedan **C**-ov alat za rad sa greškama.
Kada je pozvana, **perror()** prikazuje poruku na **stderr** opisujući najskoriju grešku koja se desilo tokom poziva bibliotečne funkcije ili sistemskog poziva.

Prototip, u **STDIO.H**, je:

```
void perror(char *msg);
```

Argument **msg** pokazuje na opcionalnu korisničko-definisanu poruku. Ova poruka se printa prvo, nakon čega slijedi colon i implementacijsko-definisana poruka koja opisuje najskoriju grešku.
Ako pozovete **perror()** kada se nije desila nijedna greška, poruka koja se prikazuje je **no error**.

Poziv na **perror()** ne radi ništa u vezi sa stanjem greške. Do programa je da poduzme akciju, koja bi se mogla sastojati od promptanja korisnika da uradi nešto kao npr. da terminira program. Akcije koje program preduzima, mogu biti odlučene testirajući vrijednost od **errno** i prirode greške.
Primjetite da program ne treba da uključi file zaglavlja **ERRNO.H** da koristi **extern**u (globalnu) varijablu **errno**. Taj file zaglavlja je potreban samo ako vaš program koristi sibolične konstante grešaka navedene u Tabeli **19.2**.

Listing **19.4** ilustruje upotrebu **perror()** i errno za rad sa greškama u toku izvršenja (runtime errors).

Listing 19.4. Korištenje perror() i errno da rad sa greškama u toku izvršenja (runtime errors).

```
1: /* Demonstracija rada sa greskama sa perror() i errno. */
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5: #include <errno.h>
6:
7: main()
8: {
9:     FILE *fp;
10:     char filename[80];
11:
12:     printf("Enter filename: ");
13:     gets(filename);
14:
15:     if (( fp = fopen(filename, "r")) == NULL)
16:     {
17:         perror("You goofed!");
18:         printf("errno = %d.\n", errno);
19:         exit(1);
20:     }
21:     else
22:     {
23:         puts("File opened for reading.");
24:         fclose(fp);
25:     }
26:     return(0);
27: }
            Enter file name: list19_4.c
            File opened for reading.
            Enter file name: notafile.xxx
            You goofed!: No such file or directory
            errno = 2.
```

ANALIZA: Ovaj program printa jednu ili dvije poruke zavisno od toga da li file može biti otvoren za čitanje ili ne.
Linija **15** pokušava da otvori file. Ako se file otvori, **else** dio od **if** petlje se izvršava, printajući sljedeću poruku:

```
File opened for reading.
```

Ako nema greške kada je file otvoren, kao što je nepostojanje file-a, linije **17** do **19** od **if** petlje se izvršavaju.
Linija **17** poziva **perror()** funkciju sa stringom **"You goofed!"**. Nakon ovoga slijedi printanje broja greške. Rezultat unošenja file-a koji ne postoji je:

```
You goofed!: No such file or directory.
errno = 2
```

UKLJUČITE ERRNO.H file zaglavlja ako mislite koristiti sibolične greške navedene u Tabeli **19.2**.

NE uključujte **ERRNO.H** file zaglavlja ako ne mislite koristiti simbolične konstante grešaka navedene u Tabeli **19.2**.

PROVJERITE moguće greške u vašim programima. Nikad ne pretpostavljajte da je sve u redu.

→→→ Traženje i Sortiranje (Searching and Sorting)

Među najčešćim programskim zadacima koje program obavlja su pretraživanje i sortiranje podataka. **C**-ova standardna biblioteka sadrži opšte-svrhne funkcije koje možete koristiti za svaki zadatak (task).

→→→ Pretraživanje sa →→→ bsearch() ←←←

Bibliotečna funkcija **bsearch()** obavlja binarno pretraživanje nizova podataka, tražeći element niza koji odgovara ključu.
Da koristite **bsearch()**, niz mora biti sortiran u opadajućem (ascending) redosljedu. Također, program mora obezbjediti upoređujuću funkciju koju koristi **bsearch()** da odredi da li je jedan predmet podatka veći od, manji od, ili jednak sa drugim predmetom.

Prototip za **bsearch()** je u **STDLIB.H**:

```
void *bsearch(void *key, void *base, size_t num, size_t width,
int (*cmp)(void *element1, void *element2));
```

Ovo je djelimično komplexan prototip, tako da prođite kroz njega oprezno.
Argument **key** (ključ) je pointer na predmet podatka koji se traži, i **base** je pointer na prvi element niza koji se pretražuje. Oba su deklarisana kao pointeri tipa **void**, tako da oni mogu pokazivati na bilo koje **C**-ove tipove objekata.

Argument **num** je broj elemenata u niizu, i **width** je veličina (u byte-ima) svakog elementa.
Specifikator tipa, **size_t**, se odnosi na tip podataka koji su vraćeni pomoću **sizeof()** operatora, koji je **unsigned**.
sizeof() operator se obično koristi da se dobiju vrijednosti za **num** i **width**.

Zadnji argument, **cmp**, je pointer na upoređujuću funkciju. Ovo može biti korisničko-napisana funkcija, ili, kada pretražujete stringove podataka, bibliotečna funkcija **strcmp()**.
Upoređujuća funkcija mora ispunjavati sljedeća dva kriterija:

- Prosljeđuju joj se pointeri na dva predmeta podataka
- Vraća tip **int** kako slijedi:
- **< 0** Element 1 je manji od element 2.
- **0** Element 1 je jednak sa element 2.
- **> 0** Element 1 je veći od element 2.

Povratna vrijednost od **bsearch()** je pointer tipa **void**.
Funkcija vraća pointer na prvi element niza koji pronađe da odgovara **key**-u (ključu), ili **NULL** ako ne pronađe odgovarajući element.
Vi morate prebaciti (cast) vraćeni pointer na odgovarajući tip prije nego što ga upotrebite.

sizeof() operator može obezbjediti **num** i **width** argumente kako slijedi. Ako je **array[]** niz koji se pretražuje, iskaz

```
sizeof(array[0]);
```

vraća vrijednost od width → veličinu (u byte-ima) od jednog elementa niza. Zato što izraz **sizeof(array)** vraća veličinu, u byte-ima, od cijelog niza, sljedeći iskaz dobija veličinu od **num**, broja elemenata u niizu:

```
sizeof(array)/sizeof(array[0])
```

Algoritam binarnog pretraživanja je vrlo efikasan; on može pretražiti veliki niiz brzo. Njegova operativnost zavisi od elemenata niza koju su u opadajućem redosljedu (ascending order).
Evo kako algoritam radi:

1. **key** (ključ) se upoređuje sa elementom u sredini niza. Ako postoji match, pretraga je gotova. U drugom slučaju, **key** mora biti ili manji ili veći od elementa niza.
2. Ako je **key** manji od elementa niza, odgovarajući (matching) element, ako ga ima, mora biti lociran u prvoj polovini niza. Tako i ako je **key** veći od elementa niza, odgovarajući element mora biti lociran u drugoj polovini niza.
3. Pretraživanje se ograničava na odgovarajuću polovinu niza, i onda se algoritam vraća na korak **1**.

Možete vidjeti da svako poređenje, koje obavlja binarno pretraživanje, eliminiše polovinu niza koji se pretražuje. Na primjer, **1,000** elementni niz može biti pretražen sa samo **10** upoređivanja, i **16,000** elementni niz može biti pretražen sa samo **14** upoređivanja.
Uopšte, binarno pretraživanje zahtjeva **n** upoređivanja za pretraživanje niza od 2^n elemenata.

→→→ Sortiranje sa →→→ qsort() ←←←

Bibliotečna funkcija **qsort()** je implementacija quicksort algoritma, kojeg je izmoslio C.A.R. Hoare. Ova funkcija sortira niz u redosljedu. Obično je rezultat u opadajućem (ascending) redosljedu, ali **qsort()** može biti korišten i za rastući redosljed takođe.

Prototip funkcije, definisan u **STDLIB.H** je:

```
void qsort(void *base, size_t num, size_t size,
int (*cmp)(void *element1, void *element2));
```

Argument **base** pokazuje na prvi element niza, **num** je broj elemenata u niizu, i **size** je veličina (u byte-ima) jednog elementa niza.
Argument **cmp** je pointer na upoređujuću funkciju. Pravila za upoređujuću funkciju su ista kao i za upoređujuću funkciju korištenu od **bsearch()**-a, opisanu u prethodnom dijelu: Vi često koristite istu upoređujuću funkciju i za **bsearch()** i za **qsort()**.
Funkcija **qsort()** nema povratnu vrijednost.

→→ Pretraživanje i sortiranje : Dvije demonstracije

Listing **19.5** demonstrira korištenje **qsort()** i **bsearch()**. Program sortira i pretražuje niz od vrijednosti (array of values). Primjetite da je korištena **ne-ANSI** funkcija **getch()**. Ako je vaš kompajler ne podržava, trebali biste je zamjeniti sa **ANSI** standardnom funkcijom **getchar()**.

Listing 19.5. Upotreba qsort() i bsearch() funkcija sa vrijednostima.

```
1: /* Koristenje qsort() i bsearch() sa vrijednostima.*/
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5:
6: #define MAX 20
7:
8: int intcmp(const void *v1, const void *v2);
9:
10: main()
11: {
12:      int arr[MAX], count, key, *ptr;
13:
14:      /* Unesi nekoliko integera od korisnika. */
15:
16:      printf("Enter %d integer values; press Enter after each.\n", MAX);
17:
18:      for (count = 0; count < MAX; count++)
19:          scanf("%d", &arr[count]);
20:
21:      puts("Press Enter to sort the values.");
22:      getc(stdin);
23:
24:      /* Sortiraj niz u opadajucem (ascending) redosljedu. */
25:
26:      qsort(arr, MAX, sizeof(arr[0]), intcmp);
27:
28:      /* Prikazi sortiran niz. */
29:
30:      for (count = 0; count < MAX; count++)
31:          printf("\narr[%d] = %d.", count, arr[count]);
32:
33:      puts("\nPress Enter to continue.");
34:      getc(stdin);
35:
36:      /* Unesi kljuc (key) za pretrazivanje. */
37:
38:      printf("Enter a value to search for: ");
39:      scanf("%d", &key);
40:
41:      /* Obavi pretrazivanje (Perform the search). */
42:
43:      ptr = (int *)bsearch(&key, arr, MAX, sizeof(arr[0]),intcmp);
44:
45:      if ( ptr != NULL )
46:          printf("%d found at arr[%d].", key, (ptr - arr));
47:      else
48:          printf("%d not found.", key);
49:      return(0);
50: }
51:
52: int intcmp(const void *v1, const void *v2)
53: {
54:      return (*(int *)v1 - *(int *)v2);
```

```
55: }
            Enter 20 integer values; press Enter after each.
            45
            12
            999
            1000
            321
            123
            2300
            954
            1968
            12
            2
            1999
            1776
            1812
            1456
            1
            9999
            3
            76
            200
            Press Enter to sort the values.
            arr[0] = 1.
            arr[1] = 2.
            arr[2] = 3.
            arr[3] = 12.
            arr[4] = 12.
            arr[5] = 45.
            arr[6] = 76.
            arr[7] = 123.
            arr[8] = 200.
            arr[9] = 321.
            arr[10] = 954.
            arr[11] = 999.
            arr[12] = 1000.
            arr[13] = 1456.
            arr[14] = 1776.
            arr[15] = 1812.
            arr[16] = 1968.
            arr[17] = 1999.
            arr[18] = 2300.
            arr[19] = 9999.
            Press Enter to continue.
            Enter a value to search for:
            1776
            1776 found at arr[14]
```

ANALIZA: Listing **19.5** obuhvata sve što je opisano prije o sortiranju i pretraživanju.
Ovaj program vam dopušta da unesete **MAX** vrijednosti (**20** u ovom slučaju). On sortira vrijednosti i printa ih u redosljedu. Onda vam dopušta da unesete vrijednost za pretraživanje u niizu. Printana poruka govori rezultat pretraživanja.

Poznati kood je korišten da se dobiju vrijednosti za niiz u linijama **18** i **19**.
Linija **26** sadrži poziv na **qsort()** da se sortira niz. Prvi argument je pointer na prvi element niza. Nakon ovoga slijedi **MAX**, broj elemenata u nizu. Zatim je obezbjeđena veličina od prvog elementa, tako da **qsort()** zna **width** (širinu) svakog predmeta. Poziv je završen sa argumentom za sort funkciju, **intcmp**.

Funkcija **intcmp()** je definisana u linijama **52** do **55**. Ona vraća razliku od dvije vrijednosti koje su joj prosljeđene. Ovo možda izgleda prejednostavno u početku, ali zapamtite koje vrijednosti upoređujuća funkcija treba da vrati. Ako su elementi **jednaki**, **0** treba biti vraćena. Ako je element **one veći** od element **two**, **pozitivan** broj treba biti vraćen. Ako je element **one manji** od element **two**, **negativan** broj treba biti vraćen. Ovo je tačno šta **intcmp()** radi.

Pretraživanje je gotovo sa **bsearch()**. Primjetite da su njeni argumenti prktično isti kao oni od **qsort()**. Razlika je da je prvi argument od **bsearch()** ključ za kojim se traga.
bsearch() vraća pointer na lokaciju pronađenog ključa ili **NULL** ako ključ nije pronađen. U liniji **43**, **ptr**-u je pridružena povratna (vraćena) vrijednost od **bsearch()**. **ptr** je korišten u **if** petlji u linijama **45** do **48** da printa status pretraživanja.

Listing **19.6** ima istu funkcionalnost kao i Listing **19.5**; ipak, Listing **19.6** sortira i pretražuje **stringove**.

Listing 19.6. Korištenje qsort() i bsearch() sa stringovima.

```
1: /* Upotreba  qsort() i bsearch()  sa stringovima. */
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5: #include <string.h>
6:
7: #define MAX 20
8:
9: int comp(const void *s1, const void *s2);
10:
11: main()
12: {
13:     char *data[MAX], buf[80], *ptr, *key, **key1;
14:     int count;
15:
16:     /* Unesi listu rijechi. */
17:
18:     printf("Enter %d words, pressing Enter after each.\n",MAX);
19:
20:     for (count = 0; count < MAX; count++)
21:     {
22:         printf("Word %d: ", count+1);
23:         gets(buf);
24:         data[count] = malloc(strlen(buf)+1);
25:         strcpy(data[count], buf);
26:     }
27:
28:     /* Sortiraj rijechi (u stvari, sortiraj pointere). */
29:
30:     qsort(data, MAX, sizeof(data[0]), comp);
31:
32:     /* Prikazi sortirane riječi. */
33:
34:     for (count = 0; count < MAX; count++)
35:         printf("\n%d: %s", count+1, data[count]);
36:
37:     /* Uzmi kljuch za pretrazivanje (Get a search key). */
38:
39:     printf("\n\nEnter a search key: ");
40:     gets(buf);
41:
42:     /* Obavi pretrazivanje. Prvo, napravi key1 kao pointer */
43:     /* na pointer na ključ (key) za pretrazivanje.*/
44:
45:     key = buf;
46:     key1 = &key;
47:     ptr = bsearch(key1, data, MAX, sizeof(data[0]), comp);
48:
49:     if (ptr != NULL)
50:         printf("%s found.\n", buf);
51:     else
52:         printf("%s not found.\n", buf);
53:     return(0);
54: }
55:
56: int comp(const void *s1, const void *s2)
57: {
58:     return (strcmp(*(char **)s1, *(char **)s2));
59: }
```

```
Enter 20 words, pressing Enter after each.
Word 1: apple
Word 2: orange
Word 3: grapefruit
Word 4: peach
Word 5: plum
Word 6: pear
Word 7: cherries
Word 8: banana
Word 9: lime
Word 10: lemon
Word 11: tangerine
Word 12: star
Word 13: watermelon
Word 14: cantaloupe
Word 15: musk melon
Word 16: strawberry
Word 17: blackberry
Word 18: blueberry
Word 19: grape
Word 20: cranberry
1: apple
2: banana
3: blackberry
4: blueberry
5: cantaloupe
6: cherries
7: cranberry
8: grape
9: grapefruit
10: lemon
11: lime
12: musk melon
13: orange
14: peach
15: pear
16: plum
17: star
18: strawberry
19: tangerine
20: watermelon
Enter a search key: orange
orange found.
```

ANALIZA: Nekoliko stvari u Listingu **19.6** zaslužuje pominjanje. Ovaj program koristi niiz pointera na stringove, tehniku koja je predstavljena Dana 15. Kao što ste vidjeli Dana 15, vi možete "sortirati" stringove tako što im sortirate niiz pointera. Ipak, ovaj metod zahtjeva modifikaciju (izmjenu) u upoređujućoj funkciji. Ovoj funkciji su prosljeđeni pointeri na dva predmeta u niizu u kojem se upoređuju. Ipak, vi želite da je niiz pointera sortiran ne na osnovu vrijednosti samih pointera, nego na vrijednostima stringova na koje oni pokazuju.

Zbog ovoga, vi morate koristiti upoređujuću funkciju kojoj se prosljeđuju pointeri na pointere. Svaki argument na **comp()** je pointer na element niza, i zato što je svaki element niza sam po sebi pointer (na string), argument je tako pointer na pointer. Unutar same funkcije, vi dereferencirate (odvojite) pointere tako da povratna vrijednost od **comp()** zavisi od vrijednosti stringova na koje pokazuju.

Činjenica da su argumenti koji su prosljeđeni ka **comp()**, pointeri na pointere, stvara novi problem. Vi smještate ključ (**key**) za pretraživanje u **buff[]**, i vi takođe znate da je ime niza (**buf** u ovom slučaju) pointer na niiz. Ipak, vi treba da prosljedite ne saami **buf**, već pointer na **buf**. Problem je što je **buf** pointer konstanta, a ne pointer varijabla.
Saam **buf** nema adresu u memoriji; to je simbol koji procjenjuje na adresu niza. Zbog ovoga, vi ne možete kreirati pointer koji pokazuje na **buf** koristeći address-of operator ispred **buf**, kao u **&buf**.

Šta raditi??? Prvo, kreirajte pointer varijablu i pridružite joj vrijednost od **buf**. U programu, ova pointer varijabla ima ime **key**. Zato što je **key** pointer varijabla, ona ima adresu, i vi možete kreirati pointer koji sadrži tu adresu → u ovom slučaju, **key1**. Kada konačno pozovete **bsearch()**, prvi argument je **key1**, pointer na pointer na **key** string. Funkcija **bsearch()** prosljeđuje taj argument na do **comp()**, i sve radi kako treba.

NE zaboravite da poredate svoj niz koji se pretražuje u opadajući (ascending) redosljed prije korištenja **bsearch()**.

Sažetak

Ovo je poglavlje istražilo neke od najkorisnijih funkcija koje su obezbjeđene u **C**-ovoj biblioteci funkcija. Tu su funkcije koje obavljaju matematičke kalkulacije, rade sa vremenom, i pomažu vašm programu da radi sa greškama. Funkcije za sortiranje i pretraživanje podataka su naročito korisne; one vam mogu uštediti dosta vremena kada pišete vaše programe.

P&O

P Zašto skoro sve matematičke funkcije vraćaju double-ove???
O Odgovor na ovo pitanje je preciznost, ne konzistentnost. **double** je precizniji od ostalih tipova varijabli; tako da su vaši odgovori precizniji. Dana 20, ćete naučiti specifikacije za prebacujuće (casting) varijable i promociju (napredovanje) varijeble. Ove teme se takođe odnose na preciznost.

P Da li su bsearch() i qsort() jedini način u C-u za sortiranje i pretraživanje???
O Ove dvije funkcije su obezbjeđene u standardnoj biblioteci; ipak, vi ih ne morate koristiti. Većina programerskih knjiga vas uči kako da napišete vlastite pretraživačke i sortirajuće programe. **C** sadrži sve komande koje su vam potrebne da ih sami napišete. Možete kuupiti specijalno napisane pretraživačke i sortirajuće rutine. Najveća prednost **bsearch()** i **qsort()**-a je da su one već napisane i da su obezbjeđene sa bilo kojim **ANSI**-kompatibilnim kompajlerom.

P Da li matematičke funkcije procjenjuju (validate) loše (bad) podatke???
O Nikad ne pretpostavljajte da su unešeni podaci ispravni. Uvijek procjenjujte korisničko-unešene podatke.
Na primjer, ako vi prosljedite nekativnu vrijednost do **sqrt()**-a, funkcija generiše grešku. Ako formatirate izlaz, vi vjerovatno ne želite da vam se ova greška prikaže kakva jeste (as it is). Uklonite **if** iskaz u Listingu **19.1** i unesite negativan broj da vidite na šta aludiram.

Vježbe (za lekciju 19):

1. Napišite poziv do **bsearch()**. Niz koji se pretražuje je nazvan **names**, i vrijednosti su karakteri. Upoređujuća funkcija je nazvana **comp_names()**. Pretpostavite da su sva imena iste veličine.

2. BUG BUSTER: Šta nije u redu sa sljedećim programom??

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int values[10], count, key, *ptr;
printf("Enter values");
for( ctr = 0; ctr < 10; ctr++ )
scanf( "%d", &values[ctr] );
qsort(values, 10, compare_function());
}
```

3. BUG BUSTER: Da li nešto nije u redu sa sljedećom upoređujućom funkcijom???

```
int intcmp( int element1, int element2)
{
if ( element 1 > element 2 )
return -1;
else if ( element 1 < element2 )
return 1;
else
return 0;
}
```

4. Modifikujte Listing **19.1** tako da **sqrt()** funkcija radi sa negativnim brojevima. Ovo uradite tako što uzmete apsolutnu vrijednost od **x**.

5. Napišite program koji se sastoji od meni-a koji obavlja razne matematičke funkcije. Koristite matematičkih funkcija koliko možete.

6. Koristeći vremenske funkcije, o kojima smo govorili u ovom poglavlju, napišite funkciju koja prouzrokuje da se program pauzira za otprilike pet sekundi.

7. Dodajte **assert()** funkciju na program u vježbi **4**. Program bi trebao printati poruku ako je unešena negativna vrijednost.

8. Napišite program koji prihvata **30** imena i sortira ih koristeći **qsort()**. Program bi trebao printati sortirana imena.

9. Modifikujte program u vježbi **8** tako da ako korisnik unese **QUIT**, program zaustavlja prihvat ulaza (unosa) i sortira unešene vrijednosti.

10. Pogledajte Dan 15 za "brute-force" metodu sortiranja nizova pointera koja mjeri vrijeme potrebno za sortiranje velikih nizova pointera sa tom metodom i onda upoređuje to vrijeme sa vremenom koje je potrebno da se obavi isto sortiranje sa bibliotečnom funkcijom **qsort()**.

LEKCIJA 20: → Working with Memory ←

Ovo poglavlje pokriva neke od najnaprednijih pogleda na upravljanje memorijom unutar vaših **C** programa. Danas ćete naučiti:

- O konverzijama tipova
- Kako da alocirate i oslobodite memorijski prostor
- Kako da manipulišete memorijskim blokovima
- Kako da manipulišete pojedinačnim bitovima

→→→ Konverzije tipova (Type Conversions)

Svi **C**-ovi podatkovni objekti imaju specifičan tip. Numerička varijabla može biti **int** ili **float**, pointer može biti pointer na **double** ili **char**, itd. Programi često trebaju da se ti različiti tipovi kombinuju u izrazima i iskazima. **Šta se dešava u takvim slučajevima???** Ponekad **C** automatski radi sa različitim tipovima, tako da ne trebate biti zabrinuti. U drugim slučajevima, vi morate explicitno konvertovati jedan tip podataka u drugi da izbjegnete rezultate sa greškama. Vidjeli ste ranije u poglavljima kada ste morali konvertovati ili prebaciti (cast) pointer tipa **void** u navedeni (specifični) tip prije nego što ga koristite. U ovoj, i drugim situacijama, vi trebate jasno razumjevanje kad su explicitne konverzije tipova potrebne i koji tipovi grešaka mogu nastati kada se pravilne konverzije ne primjene. Sljedeća sekcija pokriva **C**-ove automatske i explicitne konverzije.

→→ Konverzije Automatic tipa

Kao što ime implicira, konverzije tipa **auto**matic se obavljaju automatski od strane **C** kompajlera bez potrebe da vi išta radite. Ipak, trebate biti oprezni o tome šta se dešava, tako da možete razumjeti kako **C** procjenjuje izraze.

→→→→→ Promocija tipa u izrazima (Type Promotion in Expressions)

Kada se **C** izraz procjenjuje (evaluated), rezultirajuća vrijednost ima naročit tip podataka. Ako sve komponente u izrazu imaju isti tip, rezultirajući tip je isto tog tipa također.
Na primjer, ako su **x** i **y** tipa **int**, sljedeći izraz je tipa **int**:

```
x + y
```

→→ Šta ako komponente u izrazu imaju različite tipove???
U tom slučaju, izraz ima isti tip kao i njegova najobuhvatnija (najsvestranija (comprehensive)) komponenta.
Od najmanje obuhvatnijih do najviše obuhvatnijih, numerički tipovi podataka su:

char

int

long

float

double

Tako da, izraz koji sadrži **int** i **char**, se procjenjuje na tip **int**, izraz koji sadrži **long** i **float**, se procjenjuje na tip **float**, itd.

Unutar izraza, pojedinačni operandi se promovišu kako je potrebno, da odgovaraju pridruženim operandima u izrazu. Operandi su promovisani, u parovima, za svaki binarni operator u izrazu. Naravno, promocija nije potrebna ako su oba operanda istog tipa. Ako oni to nisu, promocija slijedi sljedeća pravila:

- Ako je ijedan od operanada **double**, drugi operand se promoviše na tip **double**.
- Ako je ijedan od operanada **float**, drugi operand se promoviše na tip **float**.
- Ako je ijedan od operanada **long**, drugi operand se konvertuje na tip **long**.

Na primjer, ako je **x** tipa **int** i **y** je **float**, procjena (evaluating) izraza **x/y** uzrokuje da se **x** promoviše u tip **float** prije nego što se izraz procjeni. Ovo ne znači da je tip varijable **x** promijenjen. Ovo znači da je kreirana kopija od **x** tipa **float** i korištena u procjeni izraza. Vrijednost izraza je, kao što ste upravo naučili, tipa **float.** Tako i, ako je **x** tipa **double** i **y** je tipa **float**, **y** će biti promovisan u **double**.

→→→ Konverzija po pridruživanju (Conversion by Assignment)

Promocije se takođe dešavaju sa operatorom pridruživanja. Izraz na desnoj strani od iskaza pridruživanja se uvijek promoviše na tip objekta podataka na lijevoj strani operatora pridruživanja.

Primjetite da ovo može prouzrokovati “demotion” (derangiranje) prije nego promociju.
Ako je **f** tipa **float** i **i** je tipa **int**, **i** je promovisan na tip **float** u ovom iskazu pridruživanja:

```
f = i;
```

U supotnom, iskaz pridruživanja:

```
i = f;
```

uzrokuje da se **f** derangira na tip **int**. Njen ostatak (fractional part) je izgubljen pri pridruživanju na **i**. Zapamtite da **f** samo po sebi nije uopšte promjenjeno; promocija utiče samo na kopiju vrijednosti. Tako da, nakon što se izvrše sljedeći iskazi:

```
float f = 1.23;
int i;
i = f;
```

varijabla **i** ima vrijednost **1**, i **f** još ima vrijednost **1.23**. Kao što ovaj primjer ilustruje, ostatak je izgubljen kada se broj u **float**ing-pointu konvertuje u tip **int**eger.

Trebate biti upoznati da kad se tip **int**eger konvertuje u tip **float**ing-point, rezultirajuća vrijednost **float**ing-point-a može da ne odgovara u potpunosti **int**eger vrijednosti.
Ovo je zato što format **float**ing-point-a, korišten interno od strane kompjutera, ne može tačno predstaviti svaki mogući **int**eger broj. Na primjer, sljedeći kood bi mogao rezultirati u prikazivanju **2.999995** umjesto **3**:

```
float f;
int i = 3;
f = i;
printf("%f", f);
```

U većini slučajeva, svaka izgubljena preciznost, prouzrokovana s ovim, bi bila zanemarljiva. Da budete sigurni, ipak, držite **int**eger vrijednosti u varijablama tipa **int** ili tipa **long**.

→→→ Explicitne konverzije korištenjem →→ Typecasts ←←

Typecast koristi operator prebacivanja (cast operator) da explicitno kontroliše konverzije u vašem programu.
Typecast se sastoji od imena tipa, u zagradama, prije izraza. Prebacivanje (cast) se može obaviti sa aritmetičkim izrazima i pointerima. Rezultat je da je izraz konvertovan u navedeni tip od strane cast-a (prebacivača).
Na ovaj način, vi možete kontrolisati tipove izraza u vašim programima radije nego se oslanjati na **C**-ove automatske konverzije.

→→ Aritmetički izrazi prebacivanja (Casting Arithmetic Expressions)

Prebacivanje aritmetičkih izraza govori kompajleru da predstavi vrijednost izraza na neki način. S efektom, cast (prebacivanje) je sličan promociji, o čemu je govoreno ranije. Ipak, prebacivanje je pod vašom kontrolom, ne kompajlerovom.
Na primjer, ako je **i** tipa **int**, izraz:

```
(float)i
```

prebacuje **i** u tip **float**. Drugim riječima, program čini internu kopiju vrijednosti od **i** u formatu **float**ing-point-a.

→ Kad biste koristili typecast sa aritmetičkim izrazima???
Najčešće korištenje je da se izbjegne gubljenje ostatka u djeljenju **int**egerom. Listing **20.1** ilustruje ovo. Trebali biste kompajlirati i pokrenuti ovaj program.

Listing 20.1. Kada se jedan integer podijeli sa drugim, svaki ostatak u odgovoru (rješenju) je izgubljen.

```
1: #include <stdio.h>
2:
3: main()
4: {
5:     int i1 = 100, i2 = 40;
6:     float f1;
7:
8:     f1 = i1/i2;
9:     printf("%lf\n", f1);
10:    return(0);
11: }
         2.000000
```

ANALIZA: Odgovor (answer) koji je prikazan od strane programa je **2.000000**, ali **100/40** se procjenjuje na **2.5**. Šta se desilo???
Izraz **i1/i2** u liniji **8** sadrži dvije varijable tipa **int**. Sljeedeći pravila koja su objašnjena ranije u ovom poglavlju, vrijednost saamog izraza je tipa **int**. Kao takav, on može predstavljati samo cijele brojeve, tako da je ostatak odgovora izgubljen.

Možda mislite da pridruživanje rezultata od **i1/i2** na varijablu tipa **float**, ga promoviše na tip **float**. To je tačno, ali je sad prekasno; ostatak odgovora je već izgubljen.

Da izbjegnete ovu vrstu nepravilnosti, vi možete prebaciti jednu varijablu tipa **int** u tip **float**. Ako je jedna varijabla prebačena u tip **float**, prethodna pravila vam govore da će i druga varijabla biti automatski promovisana u tip **float**, i vrijednost izraza je takođe tipa **float**.
Ovako je sačuvan ostatak odgovora. Da demonstriramo ovo, promijenite liniju **8** u izvornom koodu tako da iskaz pridruživanja čita kako slijedi:

```
f1 = (float)i1/i2;
```

Program će tada prikazati tačan odgovor.

→→→ Prebacivanje pointera (Casting Pointers)

Vi ste već bili upoznati sa prebacivanjem pointera. Kao što ste vidjeli Dana 18, pointer tipa **void** je generički pointer; on može pokazivati na bilo šta.
Prije nego što možete koristiti **void** pointer, vi ga morate prebaciti (cast) u prigodan tip. Primjetite da vi ne treba da prebacujete pointer s ciljem da mu pridružite vrijednost ili da ga uspoređujete sa **NULL**. Ipak, vi ga morate prebaciti prije nego što ga dereferencirte (odvojite) ili radite pointer aritmetiku s njim. Za više detalja i prebacivanju **void** pointera, ponovite Dan 18.

KORISTITE prebacivanje (cast) da provoišete (promote) ili derangirate (demote) vrijednosti varijabli kada je to potrebno.
NE koristite prebacivanje samo da spriječite upozorenja kompajlera. Možda ćete naći da prebacivanje sprječava upozorenja, ali prije nego uklonite upozorenje na ovaj način, budite sigurni da razumijete zašto dobijate upozorenja.

→→→ Alociranje memorijskog smještajnog prostora

C-ova biblioteka sadrži funkcije za alocrianje memorijskog smještajnog prostora za vrijeme izvršenja (runtime), proces nazvan **dinamička alokacija memorije**. Ova tehnika može imati značajne prednosti nad explicitnom alokacijom memorije u programskom izvornom koodu sa deklarisanjem varijabli, struktura, i nizova.
Ovaj kasniji metod, nazvan **statička alokacija memorije**, od vas zahtjeva da znate, kad pišete program, tačno koliko vam je memorije potrebno.
Dinamička alokacija memorije dopušta programu da reaguje, dok se izvršava, da traži memoriju, kao npr., za korisnički unos.
Sve funkcije za rad sa dinamičkom alokacijom memorije zahtjevaju file zaglavlja **STDLIB.H**; sa nekim kompajlerima, **MALLOC.H** je potreban takođe.
Primjetite da sve alokacione funkcije vraćaju pointer tipa **void**. Kao što ste naučili Dana 18, pointer tipa **void** mora biti prebačen u odgovarajući (prigodan) tip prije nego što ga koristite.

Prije nego što se baacimo da detalje, nekoliko riječi o **memorijskoj alokaciji**.
Šta ona tačno znači???
Svaki kompjuter ima određenu količinu memorije (random access memory, ili RAM) instaliranu. Ova količina varira od sistema do sistema. Kada pokrenete program, bilo word processor, grafički program, ili **C** program koji ste sami napisali, program je učitan sa diska u kompjutersku memoriju. Memorijski prostor koji zauzima program uključuje kako programski kood, tako i prostor za sve programske statične podatke → tj., podatkovne predmete koji su deklarisani u izvornom koodu. Memorija koja preostane je to što je na raspolaganju za alociranje koristeći funkcije u ovom dijelu poglavlja.

Koliko je memorije dostupno za alokaciju??? Sve zavisi. Ako vi pokrenete veliki program na sistemu sa skromnom količinom instalirane memorije, količina slobodne memorije će biti mala, i obratno, kada radi mali program na multigigabyte-nom sistemu, puno memorije je dostupno. To znači da vaši programu ne mogu praviti nikakve pretpostavke o dostupnosti memorije.
Kada se pozove funkcija za alokaciju memorije, vi morate provjeriti njenu povratnu vrijednost da se uvjerite da je memorija alocirana uspješno. S dodatkom, vaši programi moraju biti u mogućnosti da (gracefully) rade sa situacijom kada zahtjev za alociranje memorije ne uspije. Kasnije u ovom poglavlju, naučit ćete tehnike za određivanje tačno koliko vam je memorije dostupno.

Takođe primjetite da vaš operativni sistem možda nema efekta na dostupnost memorije. Neki operativni sistemi imaju samo mali dio fizičkog RAM-a dostupno. DOS 6.x i raniji spadaju u ovu kategoriju. Čak i ako vaš sistem ima više-megabyte-a RAM-a, DOS program će imati direktan pristup na samo prvih 640KB. (specijalne tehnike mogu biti korištene da se pristupi ostaloj memoriji, ali je to van dometa ove knjige). U kontrastu, UNIX će obično učiniti svu fizičku RAM memoriju dostupnom programu. Da dalje još zakomplikujemo stvari, neki operativni sistemi, kao što je Windows i OS/2, obezbjeđuju virtualnu memoriju, koja dozvoljava smještajni prostor na hard disku bude alociran, kao da je RAM. U ovim situacijama, količina dostupne memorije za program uključuje ne samo instalirani RAM, već i virtualni-memorijski prostor na hard disku.

Za većinu stvari, ove razlike operativnih sistema u memorijskom alokaciji bi vam trebale biti jasne. Ako vi koristite jednu od **C**-ovih funkcija da alocirate memoriju, poziv ili uspije ili padne (ne uspije), i vi ne treba da brinete o detaljima o tome šta se desilo.

→→→ malloc() ←←← Funkcija

U ranijim poglavljiva, vi ste naučili kako da koristite **malloc()** bibliotečnu funkciju da alocirate smještajni prostor za stringove.
malloc() funkcija, naravno, nije ograničena samo na alociranje stringova; ona može alocirati prostor za bilo koje potrebno smještanje. Ova funkcija alocira memoriju po byte-u.
Sjetite se da je prototip za **malloc()**:

```
void *malloc(size_t num);
```

Argument **size_t** je definisan u **STDLIB.H** kao **unsigned**.
Funkcija **malloc()** alocira **num** byte-ova smještajnog prostora i vraća pointer na prvi byte. Ova funkcija vraća **NULL** ako zahtjevani smještajni prostor nije bio alociran ili ako je **num == 0**. (Dan 10).

Listing **20.2** pokazuje kako se koristi **malloc()** da se odluči količina slobodne memorije koja je dostupna u vašem sistemu. Ovaj program radi fino u DOS-u, ali u DOS box-u pod Windows-om. Budite upozoreni, ipak, da možete dobiti čudne rezultate na sistemima kao što su OS/2 i UNIX, koji koriste hard disk prostor da obezbjede "virtualnu" memoriju. Program može uzeti puno vremena da iscrpi dostupnu memoriju.

Listing 20.2. Korištenje malloc() za određivanje koliko ima slobodne memorije.

```
1: /* Upotreba  malloc()  za odredjivanje slobodne memorije.*/
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5:
6: /* Definicija strukture koja je
7:    velichine 1024 byte-a (1 kilobyte).  */
8:
9: struct kilo {
10:    struct kilo *next;
11:    char dummy[1022];
12: };
13:
14: int FreeMem(void);
15:
16: main()
17: {
18:
19:    printf("You have %d kilobytes free.\n", FreeMem());
20:    return(0);
21: }
22:
23: int FreeMem(void)
24: {
25:    /* Vraca broj  kilobyte-a (1024 byte-a)
26:    slobodne memorije. */
27:
28:    int counter;
29:    struct kilo *head, *current, *nextone;
30:
31:    current = head = (struct kilo*) malloc(sizeof(struct kilo));
32:
33:    if (head == NULL)
```

```
34:       return 0;      /*No memory available.*/
35:
36:    counter = 0;
37:    do
38:    {
39:       counter++;
40:       current->next = (struct kilo*) malloc(sizeof(struct kilo));
41:       current = current->next;
42:    } while (current != NULL);
43:
44:    /* Sad brojach drzi broj od struktura tipa kilo
45:       koje smo bili u mogucnosti alocirati. Mi
46:       ih moramo osloboditi prije povratka. */
47:
48:    current = head;
49:
50:    do
51:    {
52:       nextone = current->next;
53:       free(current);
54:       current = nextone;
55:    } while (nextone != NULL);
56:
57:    return counter;
58: }
              You have 60 kilobytes free.
```

ANALIZA: Listing **20.2** radi na brute-force (čista-sila) način. On jednostavno peetlja, alocirajući blokove memorije, sve dok se **malloc()** funkciji ne vrati **NULL**, indicirajući da nema više dostupne memorije. Količina slobodne memorije je onda jednaka sa brojem alociranih blokova pomnožen sa veličinom bloka. Funkcija zatim oslobađa sve alocirane blokove i vraća broj alociranih blokova do pozivajućeg programa.
Čineći svaki blok jedan kilobyte-om, vraćena vrijednost indicira (pokazuje) direktno broj kilobyte-a slobodne memorije. Kao što znate, kilobyte nije tačno 1000 byte-a, već 1024 byte-a (2 na 10-u). Mi dobijemo (obtain) 1024 byte-ni predmet tako što definišemo strukturu, koju smo mudro nazvali **kilo**, koja sadrži 1022-byte-ni niz plus 2-byte-ni pointer.

Funkcija **FreeMem()** koristi tehniku uvezanih liista (iz Dana 15). Ukratko, uvezana lista se sastoji iz struktura koje sadrže pointer na njihov tip (to their own type) (s dodatkom na ostale podatkovne članove).
Tu je također i **head** pointer koji pokazuje na prvi predmet u listi (varijabla **head**, pointer na tip **kilo**). Prvi predmet u listi pokazuje na drugi, drugi pokazuje na treći, itd. Posljednji predmet u listi je definisan sa **NULL** pointer članom. (Dan 15 za više informacija).

→→→ calloc() ←←← Funkcija

calloc() funkcija također alocira memoriju.
Za razliku od **malloc()**, koja alocira grupu byte-ova, **calloc()** alocira grupu objekata.
Prototip funkcije je:

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

Zapamtite da je **size_t** sinonim za **unsigned** na većini kompajlera.
Argument **num** je broj objekata koji se alociraju, i **size** je veličina (u byte-ima) svakog objekta.
Ako je alokacija uspješna, sva alocirana memorija je očišćena (postavljena na **0**), i funkcija vraća pointer na prvi byte. Ako alokacija ne uspije ili, ako je ili **num** ili **size 0**, funkcija vraća **NULL**.

Listing **20.3** ilustruje upotrebu **calloc()**-a.

Listing 20.3. Korištenje calloc() funkcije da se alocira memorijski smještajni prostor dinamično.

```
1: /* Demonstracija calloc(). */
2:
3: #include <stdlib.h>
4: #include <stdio.h>
5:
6: main()
7: {
8:     unsigned num;
9:     int *ptr;
10:
11:     printf("Enter the number of type int to allocate: ");
12:     scanf("%d", &num);
13:
14:     ptr = (int*)calloc(num, sizeof(int));
15:
16:     if (ptr != NULL)
17:         puts("Memory allocation was successful.");
18:     else
19:         puts("Memory allocation failed.");
20:     return(0);
21: }
        Enter the number of type int to allocate: 100
        Memory allocation was successful.
        Enter the number of type int to allocate: 99999999
        Memory allocation failed.
```

Ovaj program prompta za vrijednost u linijama **11** i **12**. Ovaj broj odlučuje koliko će prostora program pokušati da alocira. Program pokušava da alocira dovoljno prostora (linija **14**) da drži navedeni broj **int** varijabli. Ako alokacija ne uspije, vraćena vrijednost od **calloc()** je **NULL**; u drugom slučaju, ona je pointer na alociranu memoriju.
U slučaju ovog programa, vraćena vrijednost od **calloc()** je smještena u **int** pointer, **ptr**.
Iskaz **if** u linijama **16** do **19** provjerava status alokacije na osnovu vrijednosti **ptr**-a i printa prigodnu poruku.
Unesite različite vrijednosti da vidite koliko memorije može biti uspješno alocirano. Maximalna količina zavisi, donekle, od konfiguracije vašeg sistema. Neki sistemi mogu alocirati prostor za 25,000 pojavljivanja tipa **int**, dok 30,000 ne uspijeva. Zapamtite da veličina od **int**-a zavisi od vašeg sistema.

→→→ realloc() ←←← Funkcija

realloc() funkcija mijenja veličinu bloka memorije koja je prethodno alocirana sa **malloc()** ili **calloc()**.
Prototip funkcije je:

```
void *realloc(void *ptr, size_t size);
```

Argument **ptr** je pointer na oreginalni blok memorije. Nova veličina, u byte-ovima, je navedena sa **size**.
Postoji nekoliko mogućih izlaza sa **realloc()**:

- Ako dovoljno prostora postoji da se proširi blok memorije na koji pokazuje **ptr**, dodatna memorija se alocira i funkcija vraća **ptr**.
- Ako ne postoji dovoljno prostora, da se proširi tekući blok, na svojoj tekućoj lokaciji, novi blok od **size** za veličinu se alocira, postojeći podaci je kopiraju sa starog bloka na početak novog bloka. Stari blok je oslobođen, i funkcija vraća pointer na novi blok.
- Ako je argument **ptr NULL**, funkcija radi kao **malloc()**, alocirajući blok veličine **size**, byte-ova, i vraća pointer na njega.
- Ako je argument **size 0**, memorija ne koju pokazuje **ptr** je oslobođena, i funkcija vraća **NULL**.
- Ako je nedovoljno memorije za realokaciju (ili proširenje starog bloka ili alociranje novog), funkcija vraća **NULL**, i oreginalni blok je nepromijenjen.

Listing **20.4** demonstrira korištenje **realloc()**-a.

Listing 20.4. Korištenje realloc() da se poveća veličina bloka od dinamično alocirane memorije.

```
1: /* Korishtenje realloc() da se promijeni alokacija memorije. */
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5: #include <string.h>
6:
7: main()
8: {
9:     char buf[80], *message;
10:
11:     /* Unesi string. */
12:
13:     puts("Enter a line of text.");
14:     gets(buf);
15:
16:     /* Alociraj inicijalni blok i kopiraj string u njega. */
17:
18:     message = realloc(NULL, strlen(buf)+1);
19:     strcpy(message, buf);
20:
21:     /* Prikazi poruku. */
22:
23:     puts(message);
24:
25:     /* Uzmi drugi string od korisnika. */
26:
27:     puts("Enter another line of text.");
28:     gets(buf);
29:
30:     /* Povecaj alokaciju, onda dodaj (concatenate) string ne njega. */
31:
32:     message = realloc(message,(strlen(message) + strlen(buf)+1));
33:     strcat(message, buf);
34:
35:     /* Prikazi novu poruku. */
36:     puts(message);
37:     return(0);
38: }
        Enter a line of text.
        This is the first line of text.
        This is the first line of text.
        Enter another line of text.
        This is the second line of text.
        This is the first line of text.This is the second line of text.
```

ANALIZA: Ovaj program uzima ulazni string u liniji **14**, čitajući ga u niz karaktera nazvan **buf**. String se onda kopira u memorijsku lokaciju na koju pokazuje **message** (linija 19).
message je alocirana koristeći **realloc()** u liniji **18**.
realloc() je pozvana čak iako nije bilo prethodne alokacije. Prosljeđujući **NULL** kao prvi parametar, **realloc()** zna da je ovo prva lokacija.

Linija **28** uzima drugi string u buffer-u **buf**. Ovaj string je dodan na string kojeg već drži **message**. Zato što je **message** tačno velik toliko da drži prvi string, on treba da se realocira da načini mjesta, da može da drži obadva, prvi i drugi, string. Ovo je tačno ono što linija **32** radi.
Program završava sa printanjem konačnog spojenog stringa.

→→→ free() ←←←Funkcija

Kada vi alocirate memoriju ili sa **malloc()** ili sa **calloc()**, ona je uzeta iz bazena dinamičke memorije koja je dostupna vašem program. Ovaj bazen (pool) se ponekad naziva **heap**, i on je konačan.
Kada vaš program završi sa korištenjem nekog bloka dinamičko alocirane memorije, vi trebate dealocirati, ili osloboditi, memoriju da je učinite dostupnom za kasniju upotrebu.
Da oslobodite memoriju koja je alocirana dinamički, koristite → **free()** ←.
Njen prototip je:

```
void free(void *ptr);
```

free() funkcija otpušta memoriju na koju pokazuje **ptr**. Ova memorija mora biti alocirana prethodno sa **malloc()**, **calloc()** ili **realloc()**.
Ako je **ptr NULL**, **free()** ne radi ništa.

Listing **20.5** demonstrira **free()** funkciju. (također je korištena u Listingu **20.2**).

Listing 20.5. Korištenje free() da se oslobodi prethodno alocirana dinamička memorija.

```
1: /* Koristenje  free()  da se oslobodi alocirana dinamicka memorija. */
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5: #include <string.h>
6:
7: #define BLOCKSIZE 30000
8:
9: main()
10: {
11:     void *ptr1, *ptr2;
12:
13:     /* Alociraj jedan blok. */
14:
15:     ptr1 = malloc(BLOCKSIZE);
16:
17:     if (ptr1 != NULL)
18:         printf("\nFirst allocation of %d bytes successful.",BLOCKSIZE);
19:     else
20:     {
21:         printf("\nAttempt to allocate %d bytes failed.\n",BLOCKSIZE);
22:         exit(1);
23:     }
24:
25:     /* Pokushaj da alocirash josh jedan blok. */
26:
27:     ptr2 = malloc(BLOCKSIZE);
28:
29:     if (ptr2 != NULL)
30:     {
31:         /* Ako je alokacija uspjesna, printaj poruku i exit. */
32:
33:         printf("\nSecond allocation of %d bytes successful.\n",
34:                BLOCKSIZE);
35:         exit(0);
36:     }
37:
38:     /* Ako nije uspjesna, oslobodi prvi blok i pokusaj ponovo.*/
39:
40:     printf("\nSecond attempt to allocate %d bytes failed.",BLOCKSIZE);
41:     free(ptr1);
42:     printf("\nFreeing first block.");
43:
```

```
44:     ptr2 = malloc(BLOCKSIZE);
45:
46:     if (ptr2 != NULL)
47:         printf("\nAfter free(), allocation of %d bytes successful.\n",
48:                BLOCKSIZE);
49:     return(0);
50: }
            First allocation of 30000 bytes successful.
            Second allocation of 30000 bytes successful.
```

ANALIZA: Ovaj program pokušava da dinamički alocira dva bloka memorije. On koristi definisanu konstantu **BLOCKSIZE** da odluči (odredi) koliko da alocira.
Linija **15** radi prvu alokaciju koristeći **malloc()**.
Linije **17** do **23** provjeravaju status alokacije tako što provjeravaju da vide da li je vraćena vrijednost jednaka sa **NULL**.
Poruka je prikazana, koja govori status alokacije. Ako alokacija nije bila uspješna, program izlazi (exits).
Linija **27** pokušava da alocira drugi blok memorije, ponovo provjeravajući da vidi da li je alokacija bila uspješna, poziv na **exit()** prekida program. Ako nije bila uspješna, poruka govori da je pokušaj da se alocira memorija propao. Prvi blok je onda oslobođen sa **free()** (linija **41**), i novi pokušaj se pravi da se alocira drugi blok.

Vi možda treba da modifikujete vrijednost simbolične konstante **BLOCKSIZE**. Na nekim sistemima, vrijednost od **30000** proizvodi sljedeći programski izlaz:

```
First allocation of 30000 bytes successful.
Second attempt to allocate 30000 bytes failed.
Freeing first block.
After free(), allocation of 30000 bytes successful.
```

Na sistemima sa virtualnom memorijom, naravno, alokacija će uvijek biti uspješna.

OSLOBODITE alociranu memoriju kada završite s njom.
NE pretpostavljajte da je poziv ka **malloc()**, **calloc()**, ili **realloc()** bio uspješan. Drugim riječima, uvijek provjerite da vidite da li stvarno memorija alocirana.

→→→→ Manipulacija memorijskim blokovima

Danas ste do sada vidjeli kako se alociraju i oslobađaju blokovi memorije. **C**-ova biblioteka takođe sadrži funkcije koje se mogu koristiti za manipulaciju blokovima memorije → postavljajući sve byte-ove na navedenu (specified) vrijednost, i kopiranje i premještanje informacija sa jedne na drugu lokaciju.

→→→ memset() ←←← Funkcija

Da postavite sve byte-ove u bloku na neku određenu vrijednost, koristite **memset()**.

Prototip funkcije je:

```
void * memset(void *dest, int c, size_t count);
```

Argument **dest** pokazuje na blok memorije.
c je vrijednost na koju se postavlja, i **count** je broj byte-ova, s početkom sa **dest**, koji se postavljaju (to be set).

Primjetite da dok je **c** tipa **int**, on se tretira kao tip **char**. Drugim riječima, koriste se samo low-order byte (s nižim redosljedom!), i vi možete navesti vrijednosti **c**-a samo u opsegu od **0** do **255**.

Koristite **memset()** da inicijalizirate blok memorije na navedenu (određenu (specified)) vrijednost.
Zbog ovoga, funkcija može koristiti samo tip **char** kao inicijalizacijsku vrijednost, nije korisno raditi sa blokovima podataka koji nisu tipa **char**, osim kada želite da inicijalizirate na **0**.
Drugim riječima, ne bi bilo efikasno koristiti **memset()** za inicijaliziranje niza tipa **int** na vrijednost **99**, ali vi možete inicijalizirati sve elemente niza na vrijednost 0.
memset() će biti demonstriran u Listingu **20.6**.

→→→ memcpy() ←←← Funkcija

memcpy() kopira byte-ove podataka između memorijskih blokova, ponekad nazvani **buffer**-i.
Ova funkcija ne brine o tipu podataka koji se kopiraju → ona jednostavno čini tačnu byte-po-byte kopiju.
Prototip ove funkcije je:

```
void *memcpy(void *dest, void *src, size_t count);
```

Argumenti **dest** i **src** pokazuju na odredište i izvor memorijskih blokova, respektivno.
count navodi broj byte-ova koji će se kopirati. Povratna (vraćena) vrijednost je **dest**.
Ako se dva memorijska bloka preklope (overlap), funkcija možda neće raditi ispravno → neki od podataka u **src**-u mogu biti prepisani (overwritten) prije nego što se kopiraju.
Koristite **memmove()** funkciju, o kojoj ćemo govoriti sljedeće, za rad sa preklapajućim memorijskim blokovima.
memcpy() će biti demonstrirana u Listingu **20.6**.

→→→ memmove() ←←← Funkcija

memmove() je veoma slična sa **memcpy()**, kopirajući navedeni broj byte-ova sa jednog memorijskog bloka u drugi. Ipak, ona je flexibilnija, zato što radi (handle) preklapanje memorijskih blokova ispravno.

Zato što **memmove()** može raditi sve što radi **memcpy()**, sa dodanom flexibilnošću sa radom blokova koji se preklapaju, vi rijetko, ako ikad, imate razlog da koristite **memcpy()**.

Prototip za **memove()** funkciju je:

```
void *memmove(void *dest, void *src, size_t count);
```

dest i **src** pokazuju na odredište (destination) i izvor (source) memorijskih blokova, i **count** navodi (specificira) broj byte-ova koji se kopiraju. Povratna vrijednost je **dest**.
Ako se blokovi preklapaju, ova funkcija osigurava da se podaci u izvoru, u preklapajućem regionu, kopiraju prije nego što se prepišu (overwritten).

Listing **20.6** demonstrira **memset()**, **memcpy()**, i **memmove()**.

Listing 20.6. Demonstracija od memset(), memcpy(), i memmove().

```
1: /* Demonstriranje  memset(), memcpy(), i memmove(). */
2:
3: #include <stdio.h>
4: #include <string.h>
4:
5: char message1[60] = "Four score and seven years ago ...";
6: char message2[60] = "abcdefghijklmnopqrstuvwxyz";
7: char temp[60];
8:
9: main()
10: {
```

```
11:    printf("\nmessage1[] before memset():\t%s", message1);
12:    memset(message1 + 5, `@', 10);
13:    printf("\nmessage1[] after memset():\t%s", message1);
14:
15:    strcpy(temp, message2);
16:    printf("\n\nOriginal message: %s", temp);
17:    memcpy(temp + 4, temp + 16, 10);
18:    printf("\nAfter memcpy() without overlap:\t%s", temp);
19:    strcpy(temp, message2);
20:    memcpy(temp + 6, temp + 4, 10);
21:    printf("\nAfter memcpy() with overlap:\t%s", temp);
22:
23:    strcpy(temp, message2);
24:    printf("\n\nOriginal message: %s", temp);
25:    memmove(temp + 4, temp + 16, 10);
26:    printf("\nAfter memmove() without overlap:\t%s", temp);
27:    strcpy(temp, message2);
28:    memmove(temp + 6, temp + 4, 10);
29:    printf("\nAfter memmove() with overlap:\t%s\n", temp);
30:
31: }
      message1[] before memset():     Four score and seven years ago ...
      message1[] after memset():      Four @@@@@@@@@@seven years ago ...
      Original message: abcdefghijklmnopqrstuvwxyz
      After memcpy() without overlap: abcdqrstuvwxyzopqrstuvwxyz
      After memcpy() with overlap:    abcdefefefefefefqrstuvwxyz
      Original message: abcdefghijklmnopqrstuvwxyz
      After memmove() without overlap:        abcdqrstuvwxyzopqrstuvwxyz
      After memmove() with overlap:   abcdefefghijklmnqrstuvwxyz
```

ANALIZA: Operacija od **memset( )** je jasna (straightforward). Primjetite kako je korištena pointer notacija **message1 + 5** da se navede da će **memset()** početi postavljati karaktere na šestom karakteru od **message1[ ]** (zapamtite, niizovu su nula-bazirani). Kao rezultat, šesti do petnaestog karaktera u **message1[ ]** su promjenjeni u **@**.

Kada se izvor i odredište ne preklapaju, **memcpy()** radi fino. **10** karaktera iz **temp[ ]**, s početkom u poziciji **17** (slova **q** do **z**), se kopiraju na pozicije **5** do **14**, gdje su oreginalno bila locirana slova od **e** do **n**. Kada funkcija pokuša da kopira **10** karaktera sa početkom u poziciji **4** do pozicije **6**, preklapanje osme (**8**) pozicije se desi. Vi možda očekujete slova, od **e** do **n**, da se kopiraju preko slova **g** do **p**. Umjesto toga, slova **e** do **f** se ponavljaju **pet** puta.

Ako nema preklapanja, **memmove()** radi isto kao i **memcpy()**. Ipak, sa preklapanjem, **memmove()** kopira oreginalne karaktere izvora na odredište.

KORISTITE memmove() umjesto **memcpy()** u slučaju da radite sa preklapajućim memorijskim regijama.
NE pokušavajte da koristite **memset()** za inicijaliziranje tipa **int**, **float**, ili **double** nizova ni na koju vrijednost, osim **0**.

→→→→ Rad sa bit-ovima (Working with Bits)

Kao što možda znate, najosnovnija jedinica kopjuterskih smještajnih podataka je **bit**. Nekad je jako korisna mogućnost manipulacije pojedinačnim bit-ovima u vašem **C** programu. **C** ima nekoliko alata koji vam ovo dopuštaju.

C-ovi **bitwise operatori** vam dozvoljavaju da manipulišete sa pojedinačnim bit-ovima **int**eger varijabli. Zapamtite, bit je najmanja moguća jedinica smještajnih podataka, i može imati samo dvije vrijednosti: **0** ili **1**.
Bitwise operatori se mogu koristiti samo sa **int**eger tipovima: binarna notacija → način na koji kompjuter interno smješta **int**egere.

Bitwise operatori se najčešće koriste kada vaš **C** program diretkno (međusobno) radi (interacts) sa hardverom vašeg sistema → tema koja je van dometa ove knjige.
Ipak oni imaju i druge upotrebe, koje ovo poglavlje uvodi.

→→ Shift ←← Operatori

Dva shift operatora shift (pomjeraju) bit-ove u **int**eger varijabli za navedeni broj mjesta.
<< operator pomjera (shifts) bit-ove na **lijevo**, i **>>** operator pomjera bitove na **desno**.
Syntax-a za ove bit-ne operatore je:

```
        x << n
```

i

```
        x >> n
```

Svaki operator pomjera bit-ove u **x**-u za **n** pozicija u navedenom smjeru.
Za **desni** pomak (shift), **nule** se smještaju na **n viših** bit-ova (higher-order bits) varijable; za **lijevi** pomak, **nule** se smještaju u **n nižih** bit-ova varijable.
Evo par primjera:

Binarno 00001100 (decimalno 12) pomjereno udesno (right-shifted) za 2 se procjenjuje (evaluates) na binarno 00000011 (decimalno 3).

Binarno 00001100 (decimalno 12) pomjereno ulijevo za 3 se procjenjuje na binarno 01100000 (decimalno 96).

Binarno 00001100 (decimalno 12) pomjereno udesno by 3 se procjenjuje na binarno 00000001 (decimalno 1).

Binarno 00110000 (decimalno 48) pomjereno ulijevo za 3 se procjenjuje na binarno 10000000 (decimalno 128).

Pod određenim okolnostima, operacije pomjeranja (shift-ovanja) se mogu koristiti za množenje i djeljenje integer varijable sa **2** (i sa power of 2).
Pomjeranje **ulijevo** (left-shifting) **int**egera za **n** mjeesta ima isti efekat kao i **množenje** njega sa **2n**, i pomjeranje **udesno** ima isti efekat kao i djeljenje njega sa **2n**.
Rezultat množenja sa pomjeranjem ulijevo je tačan samo ako ne postoji overflow (prenošenja) → tj., ako nikakvi bit-ovi nisu "izgubljeni" sa njihovim pomjeranjem van pozicije višeg-reda (higher-order positions). Djeljenje pomjeranjem udesno je **int**eger djeljenje, u kojem bilo koji ostatak rezultata je izgubljen.
Na primjer, ako pomjerite udesno vrijednost 5 (binarno 00000101) za jedno mjesto, s namjerom da dijelite sa 2, rezultat će biti 2 (binarno 00000010) umjesto tačnog 2.5, zato što je ostatak (.5) izgubljen.
Listing **20.7** demonstrira operacije pomjeranja (shift operations).

Listing 20.7. Korištenje operacija pomjeranja (shift operators).

```
1:  /* Demonstracija opracija pomjeranja (shiftovanja (shift operators)).*/
2:
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:       unsigned int y, x = 255;
8:       int count;
9:
10:      printf("Decimal\t\tshift left by\tresult\n");
11:
12:      for (count = 1; count < 8; count++)
13:      {
14:          y = x << count;
15:          printf("%d\t\t%d\t\t%d\n", x, count, y);
16:      }
17:      printf("\n\nDecimal\t\tshift right by\tresult\n");
18:
19:      for (count = 1; count < 8; count++)
20:      {
21:           y = x >> count;
22:           printf("%d\t\t%d\t\t%d\n", x, count, y);
23:      }
24:      return(0);
25: }
```

```
Decimal         shift left by   result
255             1               254
255             2               252
255             3               248
255             4               240
255             5               224
255             6               192
255             7               128
Decimal         shift right by  result
255             1               127
255             2               63
255             3               31
255             4               15
255             5               7
255             6               3
255             7               1
```

→→→→ Bitwise Logični operatori

Tri bitwise logična operatora se koriste za manipulaciju pojedinačnih bit-ova u **int**eger tipu podataka, kako je pokazano u Tebeli **20.1**.
Ovi operatori imaju imena koja su slična **TRUE/FALSE** (tačno/netačno (istina/laž)) logičkim operatorima, koje ste naučili u ranijim poglavljima, ali se njihove operacije razlikuju.

→ Table 20.1. Bitwise logični operatori.

Operator	Opis
&	AND (I)
|	Inclusive OR
^	Exclusive OR

Ovo su sve tri binarna operatora, koji postavljaju bit-ove u rezultatu na **1** ili na **0**, zavisno od bit-a u operandima. Oni rade kako slijedi:

- Bitwise **AND** postavlja rezultat na **1** samo ako su odgovarajući bit-ovi u oba operanda **1**; u drugim slučajevima, bit je postavljen na **0**. **AND** operator se koristi da se ugasi (turn off), ili očisti (clear), jedan ili više bit-ova u vrijednosti.
- Bitwise **inclusive OR** postavlja bit u rezultatu na **0** samo ako su odgovarajući bit-ovi u oba operanda jednaki **0**; u drugim slučajevima, bit se postavlja na **1**. **OR** operator se koristi da upali (turn on), ili postavi (set), jedan ili više bit-ova u vrijednosti.
- Bitwise **exclusive OR** postavlja bit u rezultatu na **1**, ako su odgovarajući bit-ovi u operandima različiti (ako je jedan **1** a drugi **0**); u drugim slučajevima, bit je postavljen na **0**.

→ Slijede primjeri kako ovi operatori rade:

Operacija	Primjer
AND	11110000
	& 01010101

	01010000
Inclusive OR	11110000
	| 01010101

	11110101
Exclusive OR	11110000
	^ 01010101

	10100101

Upravo ste proćitali da se bitwise **AND** i bitwise **Inclusive OR** mogu koristiti da očiste ili postave, respektivno, navedene bit-ove u integer vrijednosti. Evo šta to znači.
Pretpostavimo da imamo varijablu tipa **char**, i vi želite da osigurate da se bit-ovi na pozicijama **0** i **4** očiste (tj., da budu jednaki **0**) i da ostali bit-ovi ostanu na svojim oreginalnim vrijednostima. Ako vi **AND** varijablu sa drugom vrijednosti koja ima binarnu vrijednosti 11101110, vi ćete dobiti željeni rezultat. Evo kako ovo radi:

Na svakoj poziciji gdje druga vrijednost ima **1**, rezultat će imati istu vrijednost, **0** ili **1**, kao što je prisutna na toj poziciji u oreginalnoj varijabli:

```
0 & 1 == 0
1 & 1 == 1
```

Na svakoj poziciji gdje druga vrijednost ima **0**, rezultat će imati **0**, bez obzira na vrijednost koja je prisutna na toj poziciji u oreginalnoj varijabli:

```
0 & 0 == 0
1 & 0 == 0
```

Postavljanje bit-ova sa **OR** radi na sličan način. Na svakoj poziciji, gdje druga vrijednost ima **1**, rezultat će imati **1**, i na svakoj poziciji, gdje druga vrijednost ima **0**, rezultat će biti nepromjenjen.

```
0 | 1 == 1
1 | 1 == 1
0 | 0 == 0
1 | 0 == 1
```

→→→ Komplementarni Operator →→→ ~ ←←←

Posljednji bitwise operator je komplementarni operator **~**. Ovo je unarni operator.
Njegova akcija je da **inveruje** svaki bit u svom operandu, mijenjajući sve **0**-e u **1**-e, i obratno (vice versa).
Na primjer, **~254** (binarno 11111110) se procjenjuje (evaluates) na **1** (binarno 00000001).

Svi primjeri u ovom dijeelu su koristili varijable tipa **char** koje sadrže **8** bit-ova. Za veće varijable, kao što su tip **int** i tip **long**, stvari rade na isti način.

→→→→ Bit Polja u Strukturana (Bit Fields in Structures)

Posljednja tema koja se odnosi na bit-ove je upotreba **biit polja** u strukturama. Dana 11 ste naučili kako da definišete vlastite strukture podataka, prilagođavajući ih da odgovaraju potrebama vašeg programa.
Koristeći polja bit-ova, vi možete obaviti još veće prilagođavanje i uštediti memorijski prostor takođe.

Bit polje (bit field) je član strukture koji sadrži navedeni (specified) broj bit-ova. Vi možete deklarisati bit polje da sadrži jedan bit, dva bit-a, ili bilo koji broj bit-ova koji je potreban da drži podatke smještene u polju.

→ Koje prednosti ovo obezbjeđuje???

Pretpostavimo da vi programirate program baze podataka zaposlenih, koja drži podatke (records) o zaposlenim u vašoj kompaniji. Većina predmeta informacija koje smješta baza podataka su u varijanti da/ne (**yes/no**), kao npr., “Da li je zaposlenik uključen u dentalni plan?” ili “Da li je zaposlenik diplomirao na fakultetu?” Svaki komad yes/no informacije može biti smješten u jedan bit, s tim da **1** predstavlja **yes** i **0** predstavlja **no**.

Koristeći **C**-ove standardne tipove podataka, najmanji tip koji možete koristiti u strukturi je tip **char**. Vi naravno možete koristiti član strukture tipa **char** da drži yes/no podatke, ali sedam od osam bit-ova bi bilo neiskorišteno. Koristeći bit polja, vi možete smjestiti osam yes/no vrijednosti u samo jedan **char**.

Bit polja nisu ograničena na samo yes/no vrijednosti. Nastavljajući sa primjerom baze podatala, zamislite da vaša firma ima tri različita plana za zdravstveno osiguranje. Vaša baza podataka treba da smješta podatke o planu u kojem je svaki od zaposlenih uključen (ako jeste). Vi možete koristiti **0** da predstavlja neimanje zdravstvenog osiguranja i koristiti vrijednosti **1**, **2**, i **3** za predstavljanje tri plana. Bit polje koje sadrži **dva** bita je dovoljno, zato što dva binarna bit-a predstavljaju vrijednosti od **0** do **3**. Tako i, bit polje koje sadrži **tri** bit-a, može držati vrijednosti u opsegu od **0** do **7**, **četiri** bit-a mogu držati vrijednosti u opsegu od **0** do **15**, itd.

Bit polja se nazivaju, i pristupa im se kao regularnom članu strukture. Sva bit polja imaju tip **unsigned int**, i vi navedete veličinu polja (u bit-ovima) nakon čega slijedi ime člana sa colon-om (dvotačkom) i brojem bit-ova. Da definišete strukturu sa jedno-bit-nim članom nazvanim **dental**, drugim jedno-bit-nim članom nazvanim **college**, i dvo-bit-nim članom nazvanim health, vi biste napisali sljedeće:

```
struct emp_data {
unsigned dental      : 1;
unsigned college     : 1;
unsigned health      : 2;
...
};
```

Ellipsis (**...**) indiciraju za druge članove strukture.
Članovi mogu biti bit polja ili polja koja su napravljena kao regularni tipovi podataka. Primjetite da bit polja moraju biti smještena prva u definiciji strukture, prije bilo kojih ne-bit-nih članova strukture.
Da pristupite bit polju, koristite operator člana strukture (structure member operator), baš kao što ste radili i sa bilo kojim članom strukture.

Na primjer, vi možete proširiti definiciju strukture na nešto korisnije:

```
struct emp_data {
unsigned dental      : 1;
unsigned college     : 1;
unsigned health      : 2;
char fname[20];
char lname[20];
char ssnumber[10];
};
```

Onda možete deklarisati niiz struktuura:

```
struct emp_data workers[100];
```

Da pridružite vrijednosti prvom elementu niiza, napišite nešto ovako:

```
workers[0].dental = 1;
workers[0].college = 0;
workers[0].health = 2;
strcpy(workers[0].fname, "Mildred");
```

Naravno, vaš kood bi bio jasniji, ako koristite simboličke konstante **YES** i **NO** sa vrijednostima **1** i **0** kada radite sa jedno-bit-nim poljima. U svakom slučaju, vi tretirajte svako bit polje kao mali, **unsigned int**eger sa daatim brojem bit-ova.
Opseg vrijednosti koje mogu biti pridružene na bit polje sa **n** bit-ova je od **0** do $\mathbf{2^{n-1}}$. Ako pokušate da pridružite vrijednost van-opsega (out-of-range) na bit polje, kompajler vam neće prijaviti grešku, ali ćete dobiti neočekivane rezultate.

KORISTITE definisane konstante **YES** i **NO** ili **TRUE** i **FALSE** kada radite sa bit-ovima. Ovo je puno lakši način da se pročita i shvati, nego **1** i **0**.
NE definišite polje koje uzima **8** ili **16** bit-ova. To je isto kao i druge varijable kao što su tipa **char** ili **int**.

Sažetak

Ovo je poglavlje pokrilo razne teme **C** programiranja. Vi ste naučili da alocirate, realocirate i oslobodite memoriju za vrijeme izvršenja, i vidjeli ste komande koja vam daju flexibilnost u alociranju smještajnog prostora za programske podatke. Takođe ste vidjeli kako i kad se koriste **typecast**-i sa varijablama i pointerima. Zaboravljajući na **typecast**-e, ili koristeći ih nepravilno, je često težak-za-pronaći (hard-to-find) bug, tako da je ovo tema koju treba ponoviti!!!
Takođe ste naučili kako se koriste **memset()**, **memcpy()** i **memmove()** funkcije za manipulaciju blokova memorije. Konačno, vidjeli ste načine na koje možete manipulisati i koristiti pojedinačne bit-ove u vašim programima.

P&O

P Šta se desi ako ponovo koristim strin bez pozivanja realloc()???
O Vi ne trebate da pozivate **realloc()** ako je string koji koristite alocirao dovoljno mjesta. Pozovite **realloc()** kada vaš tekući string nije dovoljno velik. Zapamtite da, **C** kompajler vam dopušta da radite skoro sve, čak i stvari koje ne bi trebali! Vi možete prepisati (overwrite) jedan string sa većim stringom sve dok je dužina nonog stringa jednaka sa, ili manja od, alociranog prostora oreginalnog stringa. Ipak, ako je novi string veći, vi biste takođe prepisali (overwritte) ono što slijedi nakon stringa u memoriji. Ovo može biti ništa, ali isto tako mogu biti neki vitalni podaci. Ako vi trebate veću alociranu sekciju memorije, pozovite **realloc()**.

P Koje su prednosti od memset(), memcpy(), i memmove() funkcija? Zašto ja ne mogu jednostavno koristiti petlju sa iskazom pridruživanja da inicijaliziram ili kopiram memoriju???
O Vi možete koristiti petlju sa iskazom pridruživanja da incijalizirate memoriju u nekim slučajevima. U stvari, nekad je ovo jedini način da se to uradi → na primjer, postavljanje svih elemenata niza tipa **float** na vrijednost **1.23**. U drugim situacijama, ipak, memorija neće biti pridružena niizu ili listi, i **mem…()** funkcije su vaš jedini izbor. Postoje vremena kada će petlje i iskazi pridruživanja raditi, ali **mem…()** funkcije su jednostavnije i brže.

P Kada ću koristiti operatore pomjeranja (shift operators) i bitwise logične operatore???
O Najčešća upotreba ovih operatora je kad program međusobno radi (is interacting) direktno sa kompjuterskim hardverom → zadatak (task) koji često zahtjeva navedenu šemu bit-ova (specific bit pattern) da se generišu i interpretiraju. Ova je tema van dometa ove knjige. Čak i ako nekad trebate da manipulišete sa hardverom direktno, vi možete koristiti operatore pomjeranja, u nekim slučajevima, da dijelite ili množite **int**eger vrijednosti sa 2 (with powers of 2).

P Da li stvarno dobijam mnogo s korištenjem bit poolja???
O Da, vi dobijate prilično puno sa bit poljima. Razmislite okolnost sličnu primjeru u ovom poglavlju u kojem se file sastoji od informacije iz ankete. Ljude se pita na odgovor **YES** ili **NO** na postavljeno pitanje. Ako vi postavite **100** pitanja **10,000** ljuudi, i smještate svaki odgovor kao tip **char**, kao **T** ili **F**, vama će biti potrebno **10,000 * 100** byte-a smještajnog prostora (zato što je **1** karakter **1** byte). Ovo je **1 milion** byte-a prostora. Ako koristite bit polja, umjesto toga, i alocirate jedan bit za svaki odgovor, vama će biti potrebno **10,000 * 100** bit-ova. Zbog toga što **1** byte ima **8** bit-ova, ovaj iznosi **130,000** byte-a podataka, što je znatno manje od **1 miliona** byte-a.

Vježbe

1. Napišite **malloc()** komandu koja alocira memoriju za **1,000 long**-ova.
2. Napišite **calloc()** komandu koja alocira memoriju za **1,000 long**-ova.
3. Pretpostavite da ste deklarisali niz kako slijedi:

```
float data[1000];
```

Pokažite dva načina da inicijalizirate sve elemente niiza na **0**. Koristite petlju i iskaz pridruživanja za jednu metodu, i **memset()** funkciju za drugi način.
4. BUG BUSTER: Da li nešto nije u redu sa sljedećim koodom??

```
void func()
{
int number1 = 100, number2 = 3;
float answer;
answer = number1 / number2;
printf("%d/%d = %lf", number1, number2, answer)
}
```

5. BUG BUSTER: Da li, ako išta, nije u redu sa sljedećim koodom??

```
void *p;
p = (float*) malloc(sizeof(float));
*p = 1.23;
```

6. BUG BUSTER: Da li je sljedeća struktura dozvoljena???

```
struct quiz_answers {
char student_name[15];
unsigned answer1   : 1;
unsigned answer2   : 1;
unsigned answer3   : 1;
unsigned answer4   : 1;
unsigned answer5   : 1;
}
```

Odgovori nisu obezbjeđeni za sljedeće vježbe.
7. Napišite program koji koristi svaki od bitwise logičkih operatora. Program bi trebao primjenjivati bitwise operatore sa brojem, i onda ih ponovo primjenjivati na rezultat. Trebali biste nadgledati izlaz da budete sigurni da razumijete šta se dešava.
8. Napišite program koji prikazuje binarne vrijednosti broja. Na primjer, ako korisnik unese **3**, program bi trebao prikazati **00000011**. (Hint: Trebaćete koristiti bitwise operatore.)

LEKCIJA 21: →→ Advanced Compiler Use ←←

Ovo poglavljem posljednje od vaših 21 dan, pokriva neke dodatne karakteristike (features) **C**-ovog kompajlera. Danas ćete učiti o:

- Programiranje sa više file-ova izvornog-kooda
- Korištenje **C**-ovog preprocesora
- Korištenje argumenata komandne-linije

→→→→→→ Programiranje sa Više izvornih file-ova (Multiple Source Files) ←←←←←←

Do sada, svi vaši **C**-ovi programi su se sastojali od jednog file-a izvornog-kooda, isključivo file-ova zaglavlja. Jedan izvorni-kood file je često sve što trebate, naročito za male programe, ali vi također možete podijeliti izvorni kood, za jedan program, na dva ili više file-ova, praxa nazvana kao **modularno programiranje**.
Zašto biste vi ovo radili??? Sljedeća sekcija objašnjava.

→→→ Prednosti Modularnog Programiranja

Primarni razlog da koristite modularno programiranje je usko vezano za strukturalno programiranje i njeno pouzdavanje na funkcije. Nakon što postanete iskusniji programer, vi razvijate više opšte-namjenske funkcije koje možete koristiti, ne samo u programu za koje su oreginalno napisane, već i u drugim programima također.
Na primjer, vi možete napisati kolekciju opšte-namjenskih funkcija za prikazivanje informacija ne-ekran. Držeći ove funkcije u posebnim (odvojenim) file-ovima, vi ih možete ponovo koristiti u različitim programima koji također prikazuju informacije na-ekran.
Kada napišete program koji se sastoji iz više file-ova izvornog-kooda, svaki izvorni file se naziva **modul**.

→→→ Tehnike Modularnog Programiranja

C program može imati samo jednu **main()** funkciju.

Modul može sadržavati **main()** funkciju nazvanu **main module** (glavni modul), i ostale module koji se nazivaju **secondary modules** (sekundarni moduli).

Odvojeni (separate) file zaglavlja je često povezan sa svakim sekundarnim modulom (učit ćete ovo kasnije u ovom poglavlju). Za sada, pogledajte nekoliko jednostavnih primjera koji ilustruju osnove programiranja višestrukim modulima.
Listinzi **21.1**, **21.2**, i **21.3** pokazuju glavni modul (main module), sekundarni modul, i file zaglavlja, respektivno, za program koji unosi broj od korisnika i prikazuje njegov kvadrat (square).

Listing 21.1. LIST21_1.C: main module (glavni modul).

```
1: /* Unesi broj i prikazi njegov kvadrat. */
2:
3: #include <stdio.h>
4: #include "calc.h"
5:
6: main()
7: {
8:    int x;
9:
10:    printf("Enter an integer value: ");
11:    scanf("%d", &x);
12:    printf("\nThe square of %d is %ld.\n", x, sqr(x));
13:    return(0);
14: }
```

Listing 21.2. CALC.C: sekundarni modul.

```
1: /* Module koji sadrzi kalkulacijske funkcije. */
2:
3: #include "calc.h"
4:
5: long sqr(int x)
6: {
7:    return ((long)x * x);
8: }
```

Listing 21.3. CALC.H: file zaglavlja za CALC.C.

```
1: /* CALC.H: file zaglavlja za  CALC.C. */
2:
3: long sqr(int x);
4:
5: /* kraj od  CALC.H */
        Enter an integer value: 100
        The square of 100 is 10000.
```

ANALIZA: Pogledajmo komponente ovih triju file-ova detaljnije.
File zaglavlja, **CALC.H**, sadrži prototip za **sqr()** funkciju u **CALC.C**.
Zato što bilo koji modul koji koristi **sqr()** funkciju, mora da zna prototip od **sqr()**, modul mora uključivati **CALC.H**.

Sekundarni modul file, **CALC.C**, sadrži definiciju **sqr()** funkcije.
#include direktiva je korištena da uključi (include) file zaglavlja, **CALC.H**.
Primjetite da je ime file-a zaglavlja zatvoreno u znake navoda prije nego u (angle (<>)) zagrade. (razlog za ovo ćete saznati kasnije u ovom poglavlju.)

Glavni modul (main module), **LIST21_1.C**, sadrži **main()** funkciju. Ovaj modul također uključuje file zaglavlja, **CALC.H**.

Nakon što koristite vaš editor da kreirate ova tri file-a, kako vi kompajlirate i povezujete (linkujete) konačni exekutabilni program???
Vaš kompajler kontroliše ovo umjesto vas. U komandnoj liniji, unesite:

```
xxx list21_1.c calc.c
```

gdje je **xxx** komanda vašeg kompajlera. Ona naređuje komponentama kompajlera da obave sljedeće zadatke:

1. Kompajliraj **LIST21_1.C**, kreirajući **LIST21_1.OBJ** (ili **LIST21_1.O** na **UNIX** sistemima). Ako se naiđe na bilo koje greške, kompajler prikazuje opisne poruke greške.
2. Kompajliraj **CALC.C**, kreirajući **CALC.OBJ** (ili **CALC.O** na **UNIX** sistemima). Ponovo, pojavljuju se poruke greške, ako je potrebno.
3. Linkuj (poveži) **LIST21_1.OBJ**, **CALC.OBJ**, i sve potrebne funkcije iz standardne biblioteke da se kreira konačni izvršni (exekutabilni) program **LIST21_1.EXE**.

→→→→→→→ Komponente Modula ←

Kao što možete vidjeti, mehanika kompajliranja i povezivanja (linkovanja) višestruko-modul-nih programa je relativno jednostavna. Jedino pravo pitanje je **šta staviti u svaki file?** Ova vam sekcija daje neke opšte smjernice.

Sekundarni modul bi trebao sadržavati opšte korisne (utility functions) funkcije → tj., funkcije koje ćete možda koristiti u drugim programima.
Česta je praxa da je kreira jedan sekundarni modul za svaki tip funkcije → npr., **KEYBOARD.C** za funkcije vaše tastature, **SCREEN.C** za funkcije za prikazivanje na ekranu, itd.

Da kompajlirate i povežete (linkujete) više od dva modula, navedite (list all…) sve izvorne file-ove u komandnoj liniji:

```
tcc mainmod.c screen.c keyboard.c
```

Naravno, **glavni modul** bi trebao sadržavati **main()**, i bilo koje ostale funkcije koje su specifične-za-program (znači da nemaju opštu korist (utility)).

Obično postoji jedan file zaglavlja za svaki sekundarni modul. Svaki file ima isto ime kao i pridruženi modul, sa **.H** extenzijom.
U file zaglavlja, stavite:

- Prototipe za funkcije u sekundarnom modulu
- **#define** direktive za bilo koje simbolične konstante i makroe korištene u modulu
- Definicije bilo kojih struktura ili externih varijabli korištenih u modulu

Zbog toga što, ovaj file zaglavlja može biti uključen u više od jedan izvorni file, vi želite da spriječite njegove dijelove od kompajliranja više od jedanput. Ovo možete uraditi koristeći **preprocesorke direktive za uslovno** (kondicionalno) **kompajliranje** (raspravljano kasnije u ovom poglavlju).

→→→ Externe Varijable i Modularno Programiranje

→ U većini slučajeva, jedina komunikacija podataka između glavnog (main) modula i sekundarnog modula je kroz prosljeđivanje argumenata do i vraćanje od funkcija.
U ovom slučaju, vi ne trebate poduzimati posebne korake što se tiče vidljivosti podataka, **ali šta je sa externim (globalnim) varijablama koje trebaju biti vidljive u oba modula???**

Sjetite se Dana 12, "Understanding Variable Scope," da je **extern**a varijabla ona koja je deklarisana van bilo koje funkcije. **Extern**a varijabla je vidljiva kroz cijeli izvorni kood, uključujući i ostale module. Da je učinite vidljivom, vi morate deklarisati varijablu u svakom modulu, koristeći ključnu riječ **extern**.

Na primjer, ako imate **extern**u varijablu deklarisanu u glavnom modulu (main module) kao:

```
float interest_rate;
```

vi učinite **interest_rate** vidljivom u sekundarnom modulu uključujući (by including) sljedeće deklaracije u taj modul (van bilo koje funkcije):

```
extern float interest_rate;
```

Ključna riječ **extern** govori kompajleru da je oreginalna deklaracija od **interest_rate** (ona koja ostavlja sa strane smještajni prostor za nju) locirana negdje drugo, ali da bi varijabla trebala biti vidljiva u ovom modulu.
Sve **extern**e varijable imaju statično trajanje (static duration) i vidljive su za sve funkcije u modulu.

→→→ Korištenje .OBJ File-ova

Nakon što ste napisali i detaljno debug-ovali sekundarni modul, vi ga ne trebate rekompajlirati svaki put kada ga koristite u programu. Jedanom kada imate objektni file za modul-ni kood, sve što treba da uradite je da ga povežete (linkujete) sa svakim programom koji koristi funkcije u modulu.

Kada kompajlirate program, kompajler kreira objektni file koji ima isto ime kao i **C**-ov file izvornog kooda, i **.OBJ** extenziju (ili **.O** na **UNIX** sistemima).
Na primjer, pretpostavimo da vi razvijate modul nazvan **KEYBOARD.C** i kompajlirate ga, zajedno sa glavnim modulom **DATABASE.C**, koristeći sljedeću komandu:

```
tcc database.c keyboard.c
```

KEYBOARD.OBJ file je takođe na vašem disku. Jednom kada znate da funkcije u **KEYBOARD.C** rade ispravno, vi možete prekinuti njegovo kompajliranje svaki put kada rekompajlirate **DATABASE.C** (ili bilo koji drugi program koji ga koristi), i umjesto toga povezati (linkovati) postojeći objektni file.
Da uradite ovo, koristite sljedeću komandu:

```
tcc database.c keyboard.obj
```

Kompajler onda kompajlira **DATABASE.C** i povezuje (linkuje) rezultirajući objektni file **DATABASE.OBJ** sa **KEYBOARD.OBJ** da kreira konačan exekutabilan (izvršni) file **DATABASE.EXE**.
Ovo štedi vrijeme, zato što kompajler ne mora da rekompajlira kood u **KEYBOARD.C**. Ipak, ako vi modifikujete kood u **KEYBOARD.C**, vi ga morate rekompajlirati. S dodatkom, ako vi modifikujete (izmjenite) file zaglavlja, vi morate rekompajlirati sve module koji ga koriste.

NE pokušavajte da kompajlirate više izvornih file-ova zajedno ako više od jednog modula sadrži **main()** funkciju. Možete imati samo jedan **main()**.
KREIRAjTE generične funkcije u njihovom vlastitom izvornom file-u. Na taj način, one mogu biti povezane (linkovane) u bilo koji drugi program kojem su potrebne.
NE koristite uvijek **C** izvorne file-ove kada kompajlirate više file-ova zajedno. Ako vi kompajlirate izvorni file u objektni file, rekompajlirajte ga samo kada se file promijeni. Ovo štedi dosta vremena.

→→ Korištenje koriisti 'pravi' (Using the make Utility)

Skoro svi **C** kompajleri dolaze sa "napravi dodatkom (koriist)" →→ od sada **make utility** ←←← koji može pojednostaviti i ubrzati zadatak rada sa više file-ova izvornog kooda.
Uva korist (utility), koja se često naziva **NMAKE.EXE**, vam dopušta da pišete tzv., **make file** koji definiše ovisnosti (dependencies) između komponenti vaših raznih programa.
Šta znači ta ovisnost (dependency)???

Zamislite projekt koji ima glavni modul nazvan **PROGRAM.C** i sekundarni modul nazvan **SECOND.C**. Tu su takođe i dva file-a zaglavlja, **PROGRAM.H** i **SECOND.H**.
PROGRAM.C uključuje oba file-a zaglavlja, dok **SECOND.C** uključuje samo **SECOND.H**.
Kood u **PROGRAM.C** poziva funkcije u **SECOND.C**.

PROGRAM.C je ovisan (dependent) na ova dva file-a zaglavlja zato što ih on uključuje oba. Ako vi napravite izmjene u bilo kojem file-u zaglavlja, vi morate rekompajlirati **PROGRAM.C** tako da će on uključiti te izmjene.
U suprotnom, **SECOND.C** je ovisan od **SECOND.H**, ali ne i o **PROGRAM.H**. Ako vi promjenite **PROGRAM.H**, nema potrebe za rekompajliranjem **SECOND.C** → vi možete samo povezati (linkovati) postojeći objektni file **SECOND.OBJ** koji je kreiran kada je **SECOND.C** zadnji put kompajliran.

Make file opisuje ovisnosti kao one upravo opisane koje postoje u vašem projektu.
Svaki put kada editujete jedan ili više vaših izvornih file-ova, vi koristite **NMAKE** utility da "pokrenete" (run) make file. Ova utility (korist) ispituje vrijeme i datumske markice (stamps) na izvornom koodu i objektnom file-u, i na osnovu ovisnosti koje ste definisali, naređuje kompajleru da rekompajlira samo one file-ove koji su ovisni o modifikovanim file-ovima (ili file-u). Rezultat svega toga je da nije urađena nikakva bespotrebna kompilacija, i možete radtiti sa maximalnom efikasnošću.

Za projekte koji miješaju (involve) jedan ili dva izvorna file-a, obično nije vrijedno truda definisati make file. Ipak, za veće projekte, to je velika korist. Pogledajte dokumentaciju vašeg kompajlera na kako koristiti **NMAKE** utility.

→→→→ C-ov Preprocessor ←

Preprocessor je dio svih **C** kompajler paketa. Kada vi kompajlirate **C** program, preprocesor je prva komponenta kompajlera koja preocesuje (obrađuje) vaš program. U većini **C** kompajlera, preprocesor je dio kompajlerskog programa. Kada vi pokrenete kompajler, on automatski pokrene preprocesor.

Preprocesor mijenja vaš izvorni kood na osnovu instrukcija, ili **preprocesorskih direktiva**, u izvornom koodu.
Izlaz (output) iz preprocesora je modifikovan izvorni kood koji se onda koristi kao ulaz (input) za sljedeći korak kompilacije. Normalno, vi nikad ne vidite ovaj file, zato što ih kompajler izbriše, nakon upotrebe. Ipak, kasnije u ovom poglavlju, naučit ćete kako da gledate u ovaj posrednički file. Prvo, trebate naučiti o preprocesorskim direktivama, koje sve počinju sa simbolom →→→ **#** ←←←.

→→→ #define ←←← Preprocessorska Direktiva

#define preprocessorska direktiva ima dvije upotrebe: **kreiranje simboličkih konstanti i kreiranje makroa.**

→→→ Jednostavna zamjena Makroa korištenjem #define

Naučili ste o zamjeni makroa Dana 3,"Storing Data: Variables and Constants," iako je pojam korišten da ih opiše u tom poglavlju bio **simboličke konstante**. Vi kreirate zamjenski makro koristeći **#define** da zamjenite text sa drugim textom.
Na primjer, da zamjenite **text1** sa **text2**, napišete:

```
#define text1 text2
```

Ova direktiva prouzrokuje da preprocesor ide kroz cijeli file izvornog kooda, zamjenjujući svako pojavljivanje od **text1** sa **text2**. Jedini izuzetak je dešava ako je **text1** pronađen unutar znakova navoda, u čijem slučaju se ne pravi nikakva promjena.

Najčešća upotreba zamjenskih makroa je kreiranje simboličkih konstanti, kao što je objašnjeno Dana 3. Na primjer, ako vaš program sadrži sljedeće linije:

```
#define MAX 1000
x = y * MAX;
z = MAX - 12;
```

tokom preprocesuiranja, izvorni kood je promjenjen da se čita kako slijedi:

```
x = y * 1000;
z = 1000 - 12;
```

Efekat je isti kao i korištenje search-and-replace vašeg editora, s ciljem da promijenite svako pojavljivanje od **MAX** u **1000**.
Naravno, vaš oreginalni file izvornog kooda nije promjenjen. Umjesto toga, privremena kopija se kreira sa promjenama.
Primjetite da **#define** nije ograničena na kreiranje numeričkih simboličkih konstanti.
Na primjer, mogli ste napisati:

```
#define ZINGBOFFLE printf
ZINGBOFFLE("Hello, world.");
```

iako je malo razloga za to. Takođe trebate biti upoznati da se neki autori obraćaju simboličkim konstantama, definisanim sa **#define**, kao da su saami makroi. (Simbolične konstante se takođe nazivaju **manifestne konstante**.) Ipak, u ovoj knjizi, riječ **makro** je rezervisana za tip konstrukcije koja je opisana sljedeća.

→→→ Kreiranje funkcijskih Makroa sa #define

Vi možete koristiti **#define** direktivu, također, da kreirate funkcijske makroe.

→ **Funkcijski makro** je tip "kratkorukog" (shorthand), koji koristi nešto jednostavno da predstavlja nešto mnogo složenije.
Razlog za ime "funkcije" je da ovaj tip makroa može prihvatati argumente, baš kao što može i realna (prava) **C**-ova funkcija.
Jedna prednost funkcijskog makroa je da njeni argumenti **nisu osjetljivi na tip**. Tako da, vi možete proslijediti bilo koji tip numeričke varijable do funkcijskog makroa koji očekuje numerički argument.

Pogledajmo primjer. Preprocesorska direktiva:

```
#define HALFOF(value) ((value)/2)
```

definiše makro nazvan **HALFOF** koji uzima parametar nazvan **value**. Kad god preprocesorska direktiva susretne text **HALFOF(value)** u izvornom file-u, on je zamjenjuje sa definisanim textom i ubacuje argument, ako je potrebno. Tako da, linija izvornog kooda:

```
result = HALFOF(10);
```

je zamijenjena sa ovom linijom:

```
result = ((10)/2);
```

Tako i, programska linija:

```
printf("%f", HALFOF(x[1] + y[2]));
```

je zamjenjena sa ovom linijom:

```
printf("%f", ((x[1] + y[2])/2));
```

Makro može imati više od jednog parametra, i svaki se parametar može koristiti više od jednom u zamjenskom textu.
Na primjer, sljedeći makro, koji izračunava prosjek od pet vrijednosti, ima pet parametara:

```
#define AVG5(v, w, x, y, z) (((v)+(w)+(x)+(y)+(z))/5)
```

Sljedeći makro, u kojem kondicionalni (uslovni) operator određuje (odlučuje) veću od dvije vrijednosti, također koristi svaki od svojih parametara dva puta. (Naučili ste o kondicionalnom operatoru Dana 4, "Statements, Expressions, and Operators.")

```
#define LARGER(x, y) ((x) > (y) ? (x) : (y))
```

Makro može imati onoliko parametara koliko je potrebno, ali svi parametri u listi moraju biti korišteni u zamjenskom stringu (substitution string).
Na primjer, makro definicija:

```
#define ADD(x, y, z) ((x) + (y))
```

je neispravna, zato što parametar **z** nije korišten u zamjenskom stringu. Još i, kada pobudite (invoke) makro, vi mu morate proslijediti tačan broj argumenata.

Kada pišete makro definiciju (ili definiciju makroa), **otvorena zagrada** mora slijediti odmah nakon imena makroa; ne može biti bijelog prostora (white space). Otvorena zagrada govori preprocesoru da je funkcija makro definisana i da nije tipa jednostavne simbolične konstante.
Pogledajte sljedeću definiciju:

```
#define SUM (x, y, z) ((x)+(y)+(z))
```

Zato što je prostor između **SUM** i **(** , preprocesor tretira ovo kao jednostavni zamjenski makro. Svako pojavljivanje od **SUM** u izvornom koodu je zamjenjeno sa **(x, y, z) ((x)+(y)+(z))**, što, jasno, niste želili.

Još napomenimo da u zamjenskom stringu, svaki parametar je zatvoren u zagrade. Ovo je neophodno da se izbjegnu neželjeni popratni efekti kada prosljeđujete izraze kao argument do makroa.

Pogledajte sljedeći primjer makroa koji je definisan bez zagrada:

```
#define SQUARE(x) x*x
```

Ako pobudite (invoke) ovaj makro sa jednostavnom varijablom kao argumentom, nema problema.
→ **Ali šta ako prosljedite izraz kao argument???**

```
result = SQUARE(x + y);
```

Rezultirajući makro je kako slijedi, koji ne daje odgovarajući rezultat:

```
result = x + y * x + y;
```

Ako koristite zagrade, vi možete izbjeći ovaj problem, kako je pokazano u ovom primjeru:

```
define SQUARE(x) (x)*(x)
```

Ova definicija se proširuje na sljedeću liniju, koja daje odgovarajući rezultat:

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

→ Vi možete dobiti dodatnu flexibilnost u makro definicijama koristeći **stringujući operator (#)** (stringizing operator) (ponekad nazvan i string-literal operator).
Kada **#** prethodi ispred makro parametra u zamjenskom stringu, argument je konvertovan u string sa znacima navoda, kada je makro proširen.

Tako da, ako vi definišete makro kao:

```
define OUT(x) printf(#x)
```

I pobudite ga sa iskazom

```
OUT(Hello Mom);
```

on se proširuje na ovaj iskaz:

```
printf("Hello Mom");
```

Konverzija koja je urađena pomoću stringujućeg operatora uzima specijalne karaktere u obzir. Tako da, ako karakter u argumentu normalno zahtjeva escape character, onda **#** operator ubacuje backslash (\) prije karaktera.
Nastavljajući sa primjerom, pobuđivanje:

```
OUT("Hello Mom");
```

se proširuje na:

```
printf("\"Hello Mom\"");
```

operator je demonstriran u Listingu **21.4**. Prvo, vi trebate pogledati na jedan drugi operator korišten u makroima, **concatenation operator** (→→→ **##** ←←←) (operator spajanja).
Ovaj operator concatenates, ili spaja, dva stringa u makro expanziji (širenju). On ne uključuje znake navoda ili specijalni (posebni) tretman escape characters-a.
Njegova glavna upotreba je za kreiranje sekvenci **C** izvornog kooda.

Na primjer, ako vi definišete i pobudite makro kao:

```
#define CHOP(x) func ## x
salad = CHOP(3)(q, w);
```

pobuđeni makro u drugoj liniji je proširen na:

```
salad = func3 (q, w);
```

Možete vidjeti da koristeći **##** operator, vi odlučujete (određujete) koja se funkcija poziva. Vi ste u stvari modifikovali **C**-ov izvorni kood.

Listing **21.4** pokazuje primjer jednog načina korištenja **#** operatora.

Listing 21.4. Korištenje # operatora u expanziji (širenju) makroa.

```
1: /* Demonstrira  #  operator u makro expanziji. */
2:
3: #include <stdio.h>
4:
5: #define OUT(x) printf(#x " is equal to %d.\n", x)
6:
7: main()
8: {
9:    int value = 123;
10:   OUT(value);
11:   return(0);
12: }
        value is equal to 123.
```

ANALIZA: Koristeći **#** operator u liniji **5**, poziv na makro se proširuje sa imenom varijable **value** kao string sa znacima navoda prosljeđen do **printf()** funkcije. Nakon expanzije u liniji **9**, makro **OUT** izgleda ovako:

```
printf("value" " is equal to %d.",  value );
```

→→→ Makroi nasuprot Funkcijama

Vidjeli ste da makroi funkcija mogu biti korišteni na mjestima realnih funkcija, bar u sitacijama gdje je rezultirajući kood relativno kratak.
Makroi funkcija se mogu proširiti van jedne linije, ali obično postaju nepraktični nakon nekoliko linija.
Kada možete koristiti ili funkciju ili makro, šta biste trebali koristiti???
To je razmjena između brzine programa i veličine programa.

Makroova definicija je proširena u kood svaki put kada je susretnut makro u izvornom koodu. Ako vaš program pobudi makro **100** puta, **100** kopija proširenog makro kooda su u finalnom programu, U kontrastu, funkcijski kood postoji samo kao jedna kopija. Tako da, riječima **programske veličine**, bolji izbor je istinska funkcija.

Kada program pozove funkciju, potrebna određena količina procesujućeg overhead-a (prekoračenje) s ciljem da se proslijedi izvršenje do funkcijskom kooda i onda vrati izvršenje do pozivajućeg programa. Ne postoji procesuirajući overhead (prekoračenje) u "pozivanju" makroa, zato što je kood tu u programu. Riječima **brzine**, funkcijski makro ima prednost.

Ova veličina/brzina razmatranja često nisu nešto što previše brine programera početnika. Samo sa velikim, vremenski-kritičnim aplikacijama one postaju bitne.

→→→ Gledanje proširenja Makro-a (Viewing Macro Expansion)

Nekad, vi možda želite da vidite kako izgledaju proširenja vaših makroa, naročito kada oni ne rade ispravno.
Da vidite proširene makroe, vi naredite (instruct) kompajleru da kreira file koji uključuje makro proširenje nakon prvog prolaza kompajlera kroz kood.
Možda nećete biti u mogućnosti da uradite ono ako vaš kompajler koristi Integrated Development Environment (IDE); možda ćete morati raditi iz komandnog prompta.
Većina kompajlera ima zastavicu (flag) koja treba biti postavljena tokom kompilacije. Ova je zastavica prosljeđena kompajleru kao komandna-linija parametar.

Na primjer, da prekompajlirate program nazvan **PROGRAM.C** sa Microsoftovim kompajlerom, vi biste unijeli:

```
cl /E program.c
```

Na UNIX kompajleru, vi biste unijeli:

```
cc -E program.c
```

Preprocesor pravi prvi prolaz kroz vaš izvorni kood. Uključeni su svi file-ovi zaglavlja, **#define** makroi su prošireni, i ostale preprocesorske direktive su urađene. Zavisno od vašeg kompajlera, izlaz (output) ide ili do **stdout** (tj., ekrana) ili na disk file sa imenom programa i posebnom extenzijom.
Microsoft-ov kompajler šalje preprocesoriran izlaz (output) do **stdout**. Nažalost, uopšte nije korisno imati procesiran kood (whip by (možda raštkan)) na vašem ekranu! Možete koristiti komandu redirekcije da pošaljete izlaz na file, kao u ovom primjeru:

```
cl /E program.c > program.pre
```

Onda možete napuniti (load) file u vaš editor za printanje ili gledanje.

KORISTITE #define-ove, naročito za simboličke konstante. Simbolične konstante čine vaš program puno čitljivijim. Primjeri šta da stavite u definisane simbolične konstante su boje, true/false, yes/no, tipke tastature, i maximalne vrijednosti. Simbolične konstante su korištene kroz ovu knjigu.
NE koristite previše makro funkcije. Koristite ih kada je potrebno, ali budite sigurni da su one bolji izbor od normalnih funkcija.

→→→ #include ←←← Direktiva

Već ste naučili kako se koristi **#include** preprocesorska direktiva za uključivanje file-ova zaglavlja u vaš program.
Kada on naiđe na **#include** direktivu, preprocesor čita navedeni file i ubacuje ga na lokaciju direktive.
Vi NE možete koristiti ***** ili **?** joker znakove (wildcards) za čitanje u grupi file-ova sa jednom **#include** direktivom.
Vi MOŽETE, ipak, ugnijezditi **#include** direktive. Drugim riječima, uključeni file može sadržavati **#include** direktive, koje mogu sadržavati **#include** direktive, itd.
Većina kompajlera ograničavaju broj nivoa dubine koji možete ugnijezditi, ali obično možete ugnijezditi do **10** nivoa.

Postoje dva načina kako da navedete ime file-a za **#include** direktivu.
Ako je ime file-a zatvoreno u zagradama (angle brackets), kao npr. **#include <stdio.h>** (kao što ste vidjeli kroz ovu knjigu), preprocesor prvo gleda na file u standardnom direktoriju. Ako file nije pronađen, ili standardni direktorij nije naveden, preprocesor gleda na file u tekućem direktoriju.

"**Šta je to standardni direktorij???**" možda se pitate.
U DOS-u, to je direktorij ili direktoriji navedeni sa DOS INCLUDE okružnom varijablom (environment variable). DOS dokumentacija sadrži kompletne informacije o DOS okruženju. Da sažmemo, ipak, vi postavite okružnu varijablu sa **SET** komandom (obično, ali ne i neophodno, u vašem AUTOEXEC.BAT file-u). Većina kompajlera automatski postavi (setuje) INCLUDE varijablu u AUTOEXEC.BAT file kada je kompajler instaliran.

Druga metoda navođenja file-a da bude uključen (included) je zatvaranje imena file-a u znake navoda: **#include "myfile.h"** . U ovom slučaju, preprocesor ne pretražuje standardne direktorije; umjesto toga, on gleda u direktoriju koji sadrži file izvornog kood-a koji se kompajlira.
Opšte govoreći, file-ovi zaglavlja koje vi pišete trebaju biti držani u istom direktoriju gdje i **C**-ovi file-ovi izvornog kooda, i oni su uključeni (included) koristeći znake navoda. Standardni direktorij je rezervisan za file-ove zaglavlja koji su dostavljeni sa vašim kompajlerom.

→→→ Korištenje →→→→ #if, #elif, #else, i #endif ←←←←

Ovih pet preprocesorskih direktiva kontrolišu uslovnu (kondicionalnu) kompilaciju.
Pojam **uslovna kompilacija** znači da su kompajlirani blokovi izvornog **C**-ovog kooda samo ako se naiđe na određene uslove.
U većini slučajeva, →→→ **#if** ←←← familija preprocesorskih direktiva radi kao i **if** iskaz **C** jezika. Razlika je da **if** kontroliše da li su izvršeni određeni iskazi, dok **#if** kontroliše da li su kompajlirani.

Struktura **#if** bloka je kako slijedi:

```
#if condition_1
statement_block_1
#elif condition_2
statement_block_2
...
#elif condition_n
statement_block_n
#else
default_statement_block
#endif
```

Testni izraz koji **#if** koristi može biti skoro bilo koji izraz koji se procjenjuje (evaluates) na konstantu.
Vi ne možete koristiti **sizeof()** operator, **typecast**, ili **float** tip.
Najčešće koristite **#if** da testirate simbolične konstante kreirane sa **#define** direktivom.

Svaki **statement_block** (iskazni blok) se sastoji od jednog ili više **C**-ovih iskaza bilo kojeg tipa, uključujući preprocesorske direktive. Oni ne trebaju biti zatvoreni u zagradama, iako mogu biti.

→→ **#if** ←← i →→ **#endif** ←← direktive su potrebne, ali →→ **#elif** ←← i →→ **#else** ←← su opcionalne.

Vi možete imati **#elif** direktiva koliko želite, ali samo jednu **#else**.
Kada vaš kompajler dođe do **#if** direktive, on testira pridruženi uslov.
Ako se procijeni na **TRUE** (nenula), kompajliraje se iskazi koji slijede nakon **##**.
Ako se procjeni na **FALSE** (nula), kompajler testira, po redu, uslove koji su pridruženi sa svakom **#elif** direktivom.

Kompajliraju se iskazi koji su povezani sa prvim **TRUE #elif**-om. Ako se nijedan od uslova ne procjeni na **TRUE**, iskazi koji slijede nakon **#elif** direktive se kompajliraju.

Primjetite da, većinom, jedan blok iskaza unutar **#if…#endif** konstrukcije je kompajliran. Ako kompajler ne pronađe **#else** direktive, on možda neće kompajlirati nikakve iskaze.

Moguće upotrebe ovih uslovnih kompilacijskih direktiva su ograničene samo vašom maštom.
Evo jednog primjera. Pretpostavimo da pišete program koji koristi prilično puno country-specific (zemljo-specifičnih) informacija. Ove informacije su sadržane u file-u zaglavlja za svaku zemlju (country). Kada kompajlirate program za upotrebu u drugim zemljama, vi možete koristiti **#if…#endif** konstrukciju kako slijedi:

```
#if ENGLAND == 1
#include "england.h"
#elif FRANCE == 1
#include "france.h"
#elif ITALY == 1
#include "italy.h"
#else
#include "usa.h"
#endif
```

Onda, koristeći **#define** da definišete prigodnu simboličnu konstantu, vi možete kontrolisati koji je file zaglavlja uključen tokom kompilacije.

→→ Upotreba →→ #if...#endif ←← kao pomoć pri Debug-ovanju

Još jedna česta upotreba od **#if…#endif** –a je da se uključi dodatni debug-irajući kood u program.
Vi možete definisati **DEBUG** simboličnu konstantu koje je postavljena ili na **1** ili na **0**. Kroz program, vi možete unositi debug-irjaući kood, kako slijedi:

```
#if DEBUG == 1
debugging code here
#endif
```

Tokom razvijanja programa, ako vi definišete **DEBUG** kao **1**, debug-irajući kood je uključen da pomogne otkrivanju nekog bug-a.
Nakon što program radi ispravno, vi možete redefinisati **DEBUG** kao **0** i rekompajlirati program bez debug-irajućeg kooda.

→→→ **defined()** ←←← operator je koristan kada pišete uslovne kompilacijske direktive. Ovaj operator testira da vidi da li je neko naročito ime definisano. Tako, izraz:

```
defined( NAME )
```

se procjenjuje ili na **TRUE** ili na **FALSE**, zavisno od toga da li je **NAME** definisano.
Koristeći **defined()**, vi možete kontrolisati kompilaciju programa, na osnovu prethodnih definicija, bez obzira na navedenu vrijednost od name-a.
Vraćajući se na prethodni debug-irajući kood-ni primjer, vi možete ponovo napisati **#if…#endif** sekciju kako slijedi:

```
#if defined( DEBUG )
debugging code here
#endif
```

Takođe možete koristiti **defined()** da pridružite definiciju na name samo ako prethodno nije definisana. Koristite **NOT** operator (**!**) kako slijedi:

```
#if !defined( TRUE )     /* ako TRUE nije definisano. */
#define TRUE 1
#endif
```

Primjetite da **defined()** operator ne zahtjeva da ime (name) koje se definiše bude ništa naročito. Na primjer, nakon ove programske linije, ime (name) **RED** je definisano, ali kao ništa naročito:

```
#define RED
```

Čak i tada, izraz **defined(RED)** se procjenjuje kao **TRUE**. Naravno, pojavljivanja od **RED** u izvornom koodu su uklonjena i nisu zamjenjena nisčim, tako da morate biti oprezni.

→→→ Izbjegavanja višestrukih Include-ova od file-ova zaglavlja (Avoiding Multiple Inclusions of Header Files)

Kako program raste, ili kako koristite file-ove zaglavlja češće, povećava se rizik da slučajno uključite file zaglavlja više od jednom. Ovo može prouzrokovati to da se kompajler zaglavi u konfuziji.
Koristeći direktive koje ste naučili, lako možete izbjeći ovaj problem.
Pogledajte primjer prikazan u Listingu **21.5**.

Listing 21.5. Korištenje preprocesorskih direktiva sa file-ovima zaglavlja.

```
1: /* PROG.H - File zaglavlja sa provjerom za prevenciju visestrukih
iclude-ova! */
2:
3. #if defined( PROG_H )
4: /* file je vec bio ukljucen (included) */
5: #else
6: #define PROG_H
7:
8: /* Ovdje idu informacije file-a zaglavlja... */
9:
10:
11:
12: #endif
```

ANALIZA: Ispitajte šta ovaj file zaglavlja radi.
U liniji **3**, on provjerava da li je **PROG_H** definisan. Primjetite da je **PROG_H** sličan imenu file-a zaglavlja.
Ako je **PROG_H** definisan, komentar je uključen u liniji **4**, i program gleda na **#endif** na kraju file-a zaglavlja. To znači da ništa više nije urađeno.

Kako PROG_H biiva definisan???
On je definisan u liniji **6**. Prvi put kada je ovaj file zaglavlja uključen, preprocesor provjerava da li je **PROG_H** definisan. On neće biti, tako da se kontrola prebacuje (ili ide na) **#else** iskaz. Prva stvar koja se radi nakon **#else** –a je da se definiše **PROG_H** tako da bilo koje druge incluzije (inclusions) ovog file-a preskoče tijelo file-a.
Linije **7** do **11** mogu sadržavati bilo koji broj komandi ili deklaracija.

→→→ #undef ←←← Direktiva

#undef direktiva je suprotna (opozitna) direktivi **#define** → ona uklanja definicije iz imena (name-a).
Evo primjerćićićića:

```
#define DEBUG 1
/* U ovoj sekciji programa, pojavljivanja DEBUG-a     */
/* su zamjenjena sa 1, i izraz  defined( DEBUG ) */
/* se procjenjuje na  TRUE. *.
#undef DEBUG
/* U ovoj sekciji programa, pojavljivanja DEBUG-a  */
/* nisu zamjenjena, i izraz defined( DEBUG ) */
/* se procjenjuje na  FALSE. */
```

Vi možete koristiti **#undef** da kreirate ime (name) koje je definisano samo u dijelu vašeg izvornog kooda. Ovo možete koristiti u kombinaciji sa **#if** direktivom, kao što je objašnjeno ranije, za veću kontrolu nad uslovnim kompilacijama (conditional compilations).

→→→→ Predefinisani Makroi

Većina kompajlera ima predefinisane makroe.
Najkorisniji su od njih su **__DATE__**, **__TIME__**, **__LINE__**, i **__FILE__**.
Primjetite da je svaki od njih prethođen i slijeđen sa duplim donjim ćizama (double underscores).
Ovo je urađeno zato da vas spriječi od vašeg redefiniranja, s pretpostavkom da programeri nerado kreiraju svoje definicije sa prethođenim i slijeđenim donjim linijama.

Ovi makroi rade kao i makroi opisani ranije u ovom poglavlju.
→ Kada prekompajler susretne neke od ovih makroa, on zamjenjuje makro sa makroovim koodom. ←

__DATE__ i **__TIME__** su zamjenjeni sa tekućim (trenutnim) datumom i vremenom. Ovo je datum i vrijeme prekompajliranja izvornog file-a. Ovo može biti korisna informacija kada radite sa različitim verzijama programa. Imajući opciju da program prikaže datum i vrijeme svoje kompilacije, vi možete reći da li pokrećete najnoviju verziju programa ili raniju.

Druga dva makroa su čak još vrijedniji.
__LINE__ je zamjenjen sa tekućim brojem linije izvornog file-a.
__FILE__ je zamjenjen sa imenom file-a tekućeg izvornog kooda.
Ova dva makroa su najbolje korištena kada pokušavate debug-irati program ili radite sa greškama.
Razmotrite sljedeći **printf()** iskaz:

```
31:
32: printf( "Program %s: (%d) Error opening file ", __FILE__, __LINE__ );
33:
```

Da su ove linije dio programa nazvanog **MYPROG.C**, one bi printale:

```
Program MYPROG.C: (32) Error opening file
```

Ovo možda ne izgleda važno u ovoj tački, ali kako vaši programi rastu i šire se kroz više izvornih file-ova, nalaženje grešaka postaje sve teže.
Korištenjem **__LINE__** i **__FILE__** čini debug-iranje puno lakšim.

NE koristite **__LINE__** i **__FILE__** makroe da učinite korisnijim vaše poruke graške.
NE zaboravite **#endif** kada koristite **#if** iskaz.
STAVITE zagrade oko vrijednosti koja se prosljeđuje makrou. Ovo sprječava greške. Na primjer, koristite ovo:

```
#define CUBE(x)    (x)*(x)*(x)
```

umjesto ovoga:

```
#define CUBE(x)    x*x*x
```

→→→ Korištenje argumenata komandne-linije

Vaš **C** program može pristupiti argumentima koji su prosljeđeni programu u komandnoj liniji. Ovo se odnosi na informacije nakon imena programa kada pokrenete program. Ako vi pokrenete program nazvan **PROGNAME** sa **C:\>** prompt-a, npr., možete napisati:

```
C:\>progname smith jones
```

Dva argumenta komandne-linije **smith** i **jones** mogu biti dobavljeni (retrieved) tokom izvršenja programa.
Možda mislite o ovoj informaciji kao argumentima prosljeđenim do **main()** funkcije programa. Takvi argumenti komandne-linije dozvoljavaju da se prosljede informacije programu na početku (at startup), prije nego tokom izvršenja programa, koje nekad može biti pogodnije.
Vi možete prosljediti argumenta komandne-linije koliko želite.
Primjetite da argumenti komandne-linije mogu biti dobavljeni samo unutar **main()**-a. Da uradite to, deklarišite **main()** kako slijedi:

```
main(int argc, char *argv[])
{
/* Statements go here */
}
```

Prvi parametar, **argc**, je **int**eger koji daje broj dostupnih argumenata komandne-linije. Ova vrijednost je uvijek najmanje **1**, zato što je ime programa ubrojano kao prvi argument.
Parametar **argv[ ]** je niz pointera na stringove. Pravilni indexi za ovaj niiz su od **0** do **argc – 1**.
Pointer **argv[0]** pokazuje na ime programa (uključujući informacije o putu (path info)), **argv[1]** pokazuje na prvi argument koji slijedi nakon imena programa, itd.
Primjetite da imena **argc** i **argv** nisu potrebna → vi možete koristiti validna **C** imena koja želite da dobijete argumente komandne-linije. Ipak, ova se dva imena tradicionalno koriste za ovu svrhu, tako da biste se vjerovatno trebali držati njih.

Komandna-linija je podjeljena u diskretne argumente sa bilo kojim bijelim prostorom (white space). Ako treba da prosljedite argument koji uključuje space, zatvorite cijeli argument u znake navoda. Na primjer, ako unesete:

```
C:>progname smith "and jones"
```

smith je prvi argument (na koji pokazuje **argv[1]**), i **and jones** je drugi argument (na koji pokazuje **argv[2]**).
Listing **21.6** demonstrira kako da pristupite argumentima komandne-linije.

Listing 21.6. Prosljeđivanje argumenata komandne-linije na main().

```
1: /* Pristupanje argumentima komandne-linije. */
2:
3: #include <stdio.h>
4:
5: main(int argc, char *argv[])
6: {
7:     int count;
8:
9:     printf("Program name: %s\n", argv[0]);
10:
11:     if (argc > 1)
12:     {
13:         for (count = 1; count < argc; count++)
14:             printf("Argument %d: %s\n", count, argv[count]);
15:     }
16:     else
17:         puts("No command line arguments entered.");
18:     return(0);
19: }
```

```
list21_6
Program name: C:\LIST21_6.EXE
No command line arguments entered.
list21_6 first second "3 4"
Program name: C:\LIST21_6.EXE
Argument 1: first
Argument 2: second
Argument 3: 3 4
```

ANALIZA: Ovaj program ne radi ništa više do printanja parametara komandne-linije, koji su uneseni od strane korisnika.
Primjetite da linija **5** koristi **argc** i **argv** parametre pokazane ranije.
Linija **9** printa jedan parametar komandne-linije koji uvijek imate, ime programa. Primjetite da je ovo **argv[0]**.
Linija **11** provjerava da vidi da li postoji više od jednog parametra komandne-linije. (uvijek postoji više ili jednako jedan (jedan je ime programa (uvijek))).
Ako postoje dodatni argumenti, onda **for** petlja printa svaki na-ekran (linije **13** i **14**). U drugom slučaju, printa se prigodna poruka (linija **17**).

Argumenti komandne-linije uopšte spadaju u dvije kategorije:
→ one koji su potrebni zato što program ne može da radi bez njih,
→ i one koji su opcionalni, kao npr., zastavice (flags) koje govore (instructs) programu da radi na određeni način.

Na primjer, zamislite program koji sortira podatke u file-u. Ako pišete program koji prihvata ulaz imena file-a sa komandne linije, ime je zahtjevana informacija.
Ako korisnik zaboravi da unese ulaz imena file-a u komandnoj liniji, program mora nekako djelovati svrsishodno situaciji.
Program bi takođe mogao gledati na argumente **/r**, koji signaliziraju **sortiranje u obrnutnom-redosljedu**. Ovaj argument nije neophodan (potreban); program gleda na njega i ponaša se na način ako je pronađen ili na drugi način ako nije.

KORISTIE argc i **argv** kao imena varijabli za argumente komandne-linije za **main()**. Većini **C** programera su poznata ova imena.
NE pretpostavite da će korisnik unijeti tačan (ispravan) broj parametara komandne-linije. Provjerite da budete sigunri da jeste, i ako nije, prikažite poruku koja objašnjava argumente koje treba da unese.

Sažetak

Ovo je poglavlje pokrilo neke od naprednijih programerskih alata dostupnih sa **C** kompajlerima. Naučili ste kako da pišete program koji ima izvorni file koji je podijeljen među više file-ova ili modula. Ova praxa, nazvana modularno programiranje, čini lakšim za ponovno korištenje opšte-namjenskih funkcija u više od jednom programu. Vidjeli ste kako možete koristiti preprocesorske direktive da kreirate funkcijske makroe, za uslovnu kompilaciju, i druge zadatke. Konačno, vidjeli ste da kompajler obezbjeđuje neke funkcije makroe za vas.

P&O

P Kada kompajliram više file-ova, kako kompajler zna koje ime file-a da koristi za exekutabilni file???
O Možda mislite da kompajler koristi ime file-a sadržanog u **main()** funkciji; ipak, ovo obično nije slučaj. Kada kompajlirate iz komandne-linije, prvi file naveden (listed) je korišten da odluči

ime. Na primjer, ako vi kompajlirate sljedeće sa Borland-ovim Turbo **C**-om, exekutabilni bi bio nazvan **FILE1.EXE**:

```
tcc file1.c main.c prog.c
```

P Da li file-ovi zaglavlja moraju imati .H extenziju???
O Ne. Vi možete dati file-u zaglavlja bilo koje ime želite. Standardna je praxa da se koristi **.H** extenzija.

P Kada uključujemo (including) file-ove zaglavlja, da li mogu explicitno koristiti put (path)???
O Da. Ako želite da navedete put gdje je file koji treba da bude uključen, možete. U tom slučaju, stavite ime od include file-a u znake navoda.

P Jesu li svi predefinisani makroi i preprocesorske direktive prezentovane u ovom poglavlju???
O Ne. Predefinisani makroi i direktive prezentovane u ovom poglavlju su one koje su najčešće na većini kompajlera. Ipak, većina kompajlera ima i dodatne makroe i konstante.

P Da li je sljedeće zaglavlje također prihvatljivo kada koristim main() sa parametrima komandne-linije???

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

O Vi vjerovatno možete odgovoriti na ovo pitanje sami. Ova deklaracija koristi pointer na karakterni pointer umjesto pointer na karakterni niiz. Zato što je niz pointer, ova definicija je praktično ista kao i ona prezentovana u ovom poglavlju. Ova deklaracija je također često korištena. (Vidi Dan 8, "Using Numeric Arrays," i Dan 10, "Characters and Strings," za više detalja.)

Vježbe

Zbog postojanja puno rješenja za sljedeće vježbe, odgovori nisu obezbjeđeni.

1. Koristite vaš kompajler da kompajlirate višestruke izvorne file-ove u jedan exekutabilni file. (Možete koristiti Listinge **21.1**, **21.2**, i **21.3** ili vaše lične listinge.)
2. Napišite rutinu error, koja prihvata (error) broj greške, broj linije, i ime modula. Rutina bi trebala printati formatiranu poruku greške i onda izaći iz programa. Koristite predefinisane makroe za broj linije i ime modula (prosljedite broj linije i ime modula sa lokacije gdje se greška desila). Evo mogući primjer formatirane greške:

```
module.c (Line ##): Error number ##
```

3. Modifikujte vježbu **2** da učinite poruku greše više opisnom. Kreirajte textualni file sa vašim editorom koji sadrži (error) broj greške i poruku. Nazovite ovaj file **ERRORS.TXT**. On može sadržavati informacije kao što slijedi:

```
1    Error number 1
2    Error number 2
90   Error opening file
100  Error reading file
```

Neka vaša error rutina pretražuje ovaj file i prikazuje prigodnu poruku greške na osnovu broja koji joj je proslјeđen.
4. Neki file-ovi zaglavlja mogu biti uključeni više od jednom kada pišete modularan program. Koristite preprocesorske direktive da napišete kostur za file zaglavlja koji se kompajlira samo prvi put kada se susretne tokom kompilacije.
5. Napišite program koji uzima dva imena file-a kao parametre komandne-linije. Program bi trebao kopirati prvi file u drugi file. (pogledajte Dan 16, "Using Disk Files," ako trebate pomoć za rad sa file-ovima.)
6. Ovo je posljednja vježba u ovoj knjizi, i njen sadržaj zavisi od vas. Izaberite programerski zadatak od vašeg interesa koji također zadovoljava realnu potrebu koju imate. Na primjer, mogli biste napisati program za katalogizaciju vaših CD-ova, ili računanje financijalni iznosa vezano za planiranje kupovine kuće. Ne postoji zamjena za borbu sa real-world problemima programiranja u cilju da naoštrite vaše programerske sposobnosti i pomognu da zapamtite sve stvari koje ste naučili u ovoj knjizi.

LEKCIJA 14 → Working with the Screen, Printer, and Keyboard ←

→→→ Streams (stream-ovi) i C

Svi **C**-ovi **input/output (ulaz/izlaz**) se rade sa stream-ovima (tookovima), bez obzira odakle dolazi ulaz ili gdje ide izlaz.

Prvo, treba da znate tačno šta znače termini **input** (ulaz) i **output** (izlaz).

- Podaci mogu dolaziti sa neke lokacije koja je izvan (external) programa (to the program). Podaci premješteni iz externe lokacije u RAM, gdje im program ima pristup, se zovu **input** (ulaz). Tastatura i disk file-ovi su najčešći izvori programskog ulaza (input-a).
- Podaci mogu biti poslani na lokaciju koja je vanjska (external) programu; ovo se zove **output** (izlaz). Najčešće destinacije za izlaz su ekran, printer i disk file-ovi.

Izvorima ulaza i odredištima (destincaije) izlaza se kolektivno obraća sa, uređaj (devices (odsad uređaj ☺)).
Tastatura je uređaj, ekran je uređaj, itd. Neki uređaji (tastatura) služe samo za ulaz, drugi (ekran) služe samo za izlaz, dok ostali (disk file-ovi) su za obadvoje, i ulaz i izlaz.
Bilo koji da je uređaj (device), i bilo da obavlja ulaz ili da obavlja izlaz, **C** obavlja sav ulaz i izlaz operacije pomoću stream-ova (by means of streams).

→→→ Šta je Stream?

Stream je sekvenca karaktera.
Preciznije, **to je sekvenca byte-ova podataka**.
Sekvenca byte-ova koji utiču u program su ulazni stream; sekvenca byte-ova koji ističu iz programa su izlazni stream.
Fokusirajući se na stream-ove, vi ne morate brinuti puno o tome gdje oni idu ili odakle dolaze. Tako da je, osnovna prednost stream-ova, da je ulaz/izlaz (input/output) programiranja uređaja nezavisno (independent).
Programeri ne treba da pišu specijalne ulaz/izlaz funkcije za svaki uređaj (device) (tastatura, disk, itd.).
Program vidi ulaz/izlaz kao kontinualni stream byte-ova, bez obrzira odakle ulaz (input) dolazi ili kamo ide.

Ulaz i izlaz se mogu nalaziti između vašeg programa i raznih externih (vanjskih) uređaja (devices).

→ **Svaki C-ov stream je spojen (povezan) sa file-om.** U ovom kontextu, pod pojmom file se ne podrazumjeva disk file. Nego, to je srednji (intermediate) korak koji se koristi za ulaz i izlaz. Za većinu dijeela, **C**-ov programer početnik, ne treba da se brine oko ovih file-ova, zato što detalje oko interakcije između stream-ova, file-ova, i uređaja (devices) automatski obrađuje kompajler pomoću **C**-ovih bibliotečnih funkcija i operativnog sistema.

→→ Text nasuprot Binarnim Stream-ovima

C-ov stream spada u dva mooda (modes): text i binarni.
Text stream su organizovani u linijama, koje mogu biti do **255** karaktera dugačke i terminirane pomoću end-of-line, ili novelinije, karaktera. Neki karakteri u text stream-u se prepoznaju kao da imaju specijalno značenje, kao što je karakter novalinija. Ovo poglavlje obrađuje text karaktere.

Binarni stream može raditi sa bilo kojom vrstom podataka, uključujući, ali ne ograničujući se sa, textualnim podacima. Byte-ovi podataka u binarnom sistemu se ne prevode ili interpretiraju ninakakav poseban način; oni se čitaju i pišu onakvi kakvi jesu (as-is). Binarni sistem se primarno koristi sa disk file-ovima, koji se rade Dana 16.

→ Predefinisani Stream-ovi

ANSI C ima tri predefinisana stream-a, koji se takođe zovu i standardni ulaz/izlaz file-ovi.
Ako programirate na IBM-kompatibilnom PC-u pokretajući DOS, dva dodatna standardna stream-a su vam dostupna. Ovi stream-ovi se automatski otvaraju kada **C** program počne izvršenje i zatvaraju se kada se program terminira.
Programer ne treba da poduzima nikakve posebne akcije da učini ove stream-ove dostupnim.
Tabela 14.1 nabraja standardne stream-ove i uređaje koji su spojeni na njih normalno. Svih pet standardnih stream-ova su text-mode stream-ovi.

Tabela 14.1. Pet standardnih stream-ova.

Ime	Stream	Uređaj (Device)
stdin	Standardni ulaz (input)	Tastatura
stdout	Standardni izlaz (output)	Ekran
stderr	Standardna greška (error)	Ekran
Stdprn*	Standardni printer	Printer (LPT1:)
Stdaux*	Standard pomoćni (auxiliary)	Seriski port (COM1:)
*Podržano samo pod DOS-om.		

Kad god ste koristili **printf()** ili **puts()** funkcije za prikazivanje texta na-ekran, koristili ste **stdout** stream.
U drugom slučaju, kada ste koristili **gets()** ili **scanf()** za čitanje ulaza sa tastature, koristili ste **stdin** stream.
Standardni stream-ovi se otvaraju automatski, ali drugi stream-ovi, kao oni što se koriste za manipulaciju informacija smještenih na disku, se moraju otvoriti explicitno (“ručno”). Ovo ćete naučiti Dana 16. Ostatak ovog poglavlja radi sa standardnim stream-ovima.

→ C-ove Stream Funkcije

C-ova standardna biblioteka ima raznolike funkcije koje rade sa stream-ovima ulaza i izlaza. Većina ovih funkcija dolazi u dvije varijante: jedna koja uvijek koristi jedan od standardnih stream-ova, i druga kojoj je potrebno da programer navede (specificira) stream.
Ove funkcije su navedene u Tabeli **14.2.** Ova tabela ne navodi sve **C**-ove ulaz/izlaz funkcije, niti su sve funkcije pokrivene u ovom poglavlju.

Tabela 14.2. Standardne bibliotečne stream ulaz/izlaz (input/output) funkcije.

Koristi jedan od standardnih stream-ova	Zahtjeva ime stream-a	Opis
printf()	**fprintf()**	Formatiran izlaz
vprintf()	**vfprintf()**	Formatiran izlaz sa varijablinom argumentnom listom
puts()	**fputs()**	String izlaz
putchar()	**putc(), fputc()**	Karakter izlaz
scanf()	**fscanf()**	Formatiran ulaz
gets()	**fgets()**	String ulaz
getchar()	**getc(), fgetc()**	Karakter ulaz
perror()		String izlaz ka **stderr** samo

Svim ovim funkcijama je potrebno da uključite **STDLIB.H** file zaglavlja.
Funkcija **perror()**, takođe može zahtjevati **STDLIB.H**.

Funkcije **vprintf()** i **vfprintf()** takođe zahtjevaju **STDARGS.H**.
Na **UNIX** sistemima, **vprintf()** i **vfprintf()** mogu zahtjevati **VARARGS.H**.
Bibliotečna referenca vašeg kompajlera će vam reći svaki dodatni ili alternativni file zaglavlja koji je potreban.

Primjer
Kratki program u Listing-u **14.1** demonstrira ekvivalenciju stream-ova.
Listing 14.1. Ekvivalencija stream-ova.

```
1:  /* Demonstrira ekvivalenciju stream-ovog ulaza i izlaza. */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char buffer[256];
7:
8:     /* Unesi liniju, onda je odmah prikazi (output it). */
9:
10:    puts(gets(buffer));
11:
12:    return 0;
13: }
```

U liniji **10**, **gets()** funkcija se koristi za unos linije texta sa tastature (**stdin**).
Jer **gets()** vraća pointer na string, on se može koristiti kao argument na **puts()**, koji prikazuje string na-ekran (**stdout**).
Kada se pokrene, ovaj program unosi liniju texta od korisnika i onda odmah prikazuje string na-ekranu.

> **ISKORISTITE** prednosti standardnih ulaz/izlaz stream-ova koje **C** obezbjeđuje.
> **NE** preimenujte ili mijenjajte standardne stream-ove bez potrebe.
> **NE** pokušavajte da koristite ulazni stream kao **stdin** za izlaznu funkciju, kao što je **fprintf()**.

→ Prihvatanje unosa (input-a) sa tastature

Većini **C**-ovih programa je potreban neki unos sa tastature (tj., sa **stdin**).
Unosne (input) funkcije su podijeljene u hijerarhije sa tri nivoa: karakterni unos, linijski unos i formatiran unos.

→ Karakterni unos (Character Input)

Funkcije za karakterni unos čitaju unos sa stream-a jedan po jedan karakter (one character at a time).
Kada se pozove, svaka od ovih funkcija vraća sljedeći karakter u stream, ili **EOF** ako je dostignut kraj file-a ili je desila neka greška.

→ **EOF** je simbolična konstanta definisana u **STDIO.H** kao **–1**.

Funkcije karakternog unosa je odlikuju u smislu buffer-ovanja i eho-vanja (buffering and echoing).

- Neke funkcije karakternog unosa su buffer-ovane. To znači da operativni sistem drži sve karaktere u privremenom smještajnom prostoru dok vi ne pritisnete **<enter>**, i onda sistem šalje karaktere do **stdin** stream-a. Drugi (ostali) su nebuffer-ovani, što znači da se svaki karakter šalje odmah nakon što se pritisne tipka.
- Neke ulazne (unosne) funkcije automatski eho-išu svaki karakter do **stdout** odmah nakon primanja. Drugi (ostali) ne eho-išu (don't echo); karakter se šalje do **stdin**, a ne **stdout**. Zato što je **stdout** pridružen ekranu, tu je gdje je unos (input) eho-isan (echoed).

Upotreba buffer-isanih, nebuffer-isanih, eho-isanih i neeho-isanih karakternih unosa je objašnjena u sljedećoj sekciji (dijelu).

→→→ getchar() Funkcija

getchar() funkcija dobija sljedeći karakter od stream-a **stdin**. On obezbjeđuje buffer-isan karakterni unos sa ehom, i njegov prototip je:

```
int getchar(void);
```

Upotreba **getchar()** je demonstrirana u Listing **14.2**.
Primjetite da **putchar()** funkcija, detaljnije objašnjena u ovom poglavlju, jednostavno prikazuje jedan karakter na-ekran (on-screen).

Listing 14.2. getchar() funkcija.

```
1: /* Demonstrira getchar() funkciju. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     int ch;
8:
9:     while ((ch = getchar()) != `\n')
10:         putchar(ch);
11:
12:    return 0;
13: }
          This is what's typed in.
          This is what's typed in.
```

ANALIZA: U liniji **9**, **getchar()** funkcija se poziva i čeka da primi karakter od **stdin**-a.
Zato što je **getchar()** buffer-ovana unosna funkcija, nikakvi karakteri se ne primaju dok ne pritisnete **<enter>**. Ipak, svaka tipka koju pritisnete je eho-isana odmah na-ekranu.

Kad pritisnete **<enter>**, svi karakteri koje unesete, uključujući novuliniju, se šalju do **stdin**-a od strane operativnog sistema.
getchar() funkcija vraća jedan po jedan karakter, pridružujući svaki po redu ka **ch**.

Svaki karakter je upoređen sa novalinija karakterom, **\n**, i, ako nije jednak, prikazuje se na-ekranu sa **putchar()**. Kada je novalinija vraćen od **getchar()**-a, while ptelja se terminira (prekida).
getchar() funkcija se može koristit za unos cijele linije texta, kako je pokazano u Listing-u **14.3**. Ipak, druge unoosne funkcije su bolje za ovaj zadatak, kao što ćete naučiti kasnije u ovom poglavlju.

Listing 14.3. Upotreba getchar() funkcije za unoos cijele linije texta.

```
1: /* Upotreba getchar() za unos stringa. */
2:
3: #include <stdio.h>
4:
5: #define MAX 80
6:
7: main()
8: {
9:     char ch, buffer[MAX+1];
10:     int x = 0;
11:
12:     while ((ch = getchar()) != `\n' && x < MAX)
13:         buffer[x++] = ch;
14:
```

```
15:     buffer[x] = `\0';
16:
17:     printf("%s\n", buffer);
18:
19:     return 0;
20: }
          This is a string
          This is a string
```

ANALIZA: Ovaj program je sličan onome u Listing-u **14.2**. na način kako koristi **getchar()**.
Dodatni uslov je dodan u petlji. Ovaj put **while** petlja prihvata karaktere od **getchar()**, dok ne naiđe ili karakter novalinija ili se ne pročita **80** karaktera.
Svaki karakter se pridružuje niizu, koji je nazvan **buffer**. Kada je karakter unešen, linija **15** stavlja **null** na kraj niiza, tako da **printf()** funkcija, u liniji **17**, može printati unešeni string.

U liniji **9**, zašto je buffer deklarisan sa veličinom **MAX+1**, umjesto sa **MAX**???
Ako deklarišete buffer sa veličinom **MAX+1**, string može biti sa **80** karaktera, plus **null** terminirajući karakter. Ne zaboravite da uključite prostor za **null** nerminirajući karakter na kraju vašeg stringa.

→→→ getch() Funkcija

getch() funkcija dobija (obtains) sljedeći karakter od stream-a **stdin**. Ona obezbjeđuje nebufferovan karakterni unos bez eha.
getch() funkcija nije dio **ANSI** standarda. To znači da možda nije dostupna na svakom sistemu, i još, možda zahtjeva da su uključi drugačiji file zaglavlja. Uopšte, prototip za **getch()** je file zaglavlja **CONIO.H**, kako slijedi:

```
int getch(void);
```

Zato što je nebufferovan, **getch()** vraća svaki karakter odmah nakon pritiskanja tipke, bez čekanja na korisnika da pritisne **<enter>**. Zato što **getch()** ne eho-iše svoj ulaz (unos (input)), karakteri se ne prikazuju na-ekranu.

Listing **14.4** ilustruje upotrebu **getch()**.

> **UPOZORENJE:** Sljedeći listing koristi **getch()** , koji nije **ANSI**-kompatibilan. Trebali biste biti oprezni kada koristite ne-**ANSI** funkcije, jer ne postoje garancije da ih podržavaju svi kompajleri. Ako dobijete greške od sljedećeg listinga, možda je razlog za to što vaš kompajler ne podržava **getch()**.

Listing 14.4. Upotreba getch() funkcije.

```
1: /* Demonstrira getch() funkciju. */
2: /* Ne-ANSI kood */
3: #include <stdio.h>
4: #include <conio.h>
5:
6: main()
7: {
8:     int ch;
9:
10:     while ((ch = getch()) != `\r')
11:         putchar(ch);
12:
13:     return 0;
14:}
          Testing the getch() function
```

ANALIZA: Kada se ovaj program pokrene, **getch()** vraća svaki karakter odmah čim pritisnete tipku → on ne čeka na vas da pritisnete **<enter>**.
On nema eho (echo), tako da je jedini razlog zašto se svaki karakter ispisuje na-ekranu, poziv funkcije **putchar()**.
Da biste bolje razumili kako radi **getch()**, dodajte znak tačka-zarez (“**;**”) na kraj linije **10** i uklonite liniju **11** (**putchar(ch)**). Kada ponovo pokrenete program, primjetićete da ništa što utipkate se ne eho-iše na ekranu.
getch() funkcija uzima karaktere bez da ih eho-iše na (do) ekran(a). Znate da su karakteri dobijeni jer je oreginalni listing koristio **putchar()** za njihovo prikazivanje.

Zašto ovaj prgram upoređuje svaki karakter sa \r umjesto sa \n???
Kood **\r** je escape sequence za **carriage povratni karakter**. Kada pritisnete **<enter>**, uređaj tastature šalje carriage povratnu vrijednost (return) ka **stdin**. Funkcija bufferovanog karakternog unosa (ulaz (input)), automatski prevodi carriage return na novu-liniju, tako da program mora testirati **\n** da odluči da li je pritisnut **<enter>**.
Funkcije nebufferovanog karakternog unosa ne prevode, tako da carriage return je unos kao **\r**, i to je ono za što program mora testirati.

Listing **14.5** koristi **getch()** za unos cijele linije texta. Pokretanje ovog programa jasno ilustruje da **getch()** neeho-iše svoj ulaz (unos (input)). Sa izuzetkom zamjene **getch()** za **getchar()**, ovaj program je bukvlano identičan sa Listingom **14.3**.
Listing 14.5. korištenje getch() funkcije za unos cijele linije.

```
1: /* Upotreba getch() za unos stringova. */
2: /* Ne-ANSI kood */
3: #include <stdio.h>
4: #include <conio.h>
5:
6: #define MAX 80
7:
8: main()
9: {
10:     char ch, buffer[MAX+1];
11:     int x = 0;
12:
13:     while ((ch = getch()) != `\r' && x < MAX)
14:         buffer[x++] = ch;
15:
16:     buffer[x] = `\0';
17:
18:     printf("%s", buffer);
19:
20:     return 0;
21:}
```

```
Here's a string
Here's a string
```

UPOZORENJE: Zapamtite da **getch()** nije **ANSI**-standardna komanda. To znači da ga vaš kompajler (I drugi kompajleri) može ili ne može podržavati. **getch()** je podržan sa Symantec i Borland. Microsoft podržava **getch()**. Ako imate problema sa ovom komandom, trebali biste provjeriti vaš kompajler da vidite da li podržava **getch()**. Ako ste zabrinuti zbog portabilnosti, ne biste trebali koristiti ne-**ANSI** funkcije.

→→→ getche() Funkcija

Ovo je kratki dio, jer je **getche()** isti kao i **getch()**, s razlikom da on eho-iše karakter do **stdout**.
Modifikujte program u Listingu **14.4** da koristi **getche()** umjesto **getch()**. Kada se program pokrene, svaka tipka koju pritisnete se prikazuje na-ekranu dvaputa-jednom (twice-once) kao eho od **getche()**, i jednom kao eho od **putchar()**.

> **UPOZORENJE: getche()** NIJE **ANSI**-standardna komanda, ali je podržava većina **C** kompajlera.

→→→ getc() ←←← i →→→ fgetc() ←←← Funkcije

getc() i **fgetc()** karakterni unos ne radi automatski sa **stdin**.
Umjesto toga, od dopušta programu da specificira ulazni stream (input stream). One se primarno koriste za čitanje karaktera sa disk file-ova. Vidi Dan 16 za više detalja.

> **RAZUMITE** razliku između eho-isanog i neeho-isanog ulaza (unosa (input-a)).
> **RAZUMITE** razliku između buffer-ovanog i nebuffer-ovanog ulaza (unosa (input-a)).
> **NE** koristite ne-**ANSI** standardne funkcije, ako je portabilnost u pitanju.

"nedobijanje" ("Ungetting") karaktera pomoću ungetc()

Šta znači "nedobijanje" ("ungetting") karaktera???
Primjer bi vam trebao pomoći da ovo razumijete.
Pretpostavimo da vaš program čita karaktere sa ulaznog stream-a i može otkriti (detektovati) kraj ulaza (unosa) samo čitajući jedan karakter viška. Na primjer, možete unositi samo brojeve, tako da znate da je unos završio kada se naiđe na prvi nebroj karakter.
Ovaj prvi nebroj karakter može biti važan dio sljedećeg (sebsequent) podatka, ali je on odrstranjen iz ulaznog stream-a (input stream). **Da li je izgubljen???**
NE, on može biti "nedobijen" ("ungotten") ili vraćen ulaznom stream-u, gdje je tada prvi karakter pročitan sa sljedećom operacijom unosa na tom stream-u.

Da "nedobijete" karakter, koristite → → → **ungetc()** ← ← ← bibliotečnu funkciju.
Njen prototip je:

```
int ungetc(int ch, FILE *fp);
```

Argument **ch** je karakter koji treba da bude vraćen.
Argument ***fp** specificira stream, na koji karakter treba da se vrati, koji može biti bilo koji ulazni (unešeni (input stream)) stream.
Za sada, jednostavno navedite **stdin** kao drugi argument: **ungetc(ch, stdin);**.
Notacija **FILE *fp** se koristi sa stream-ovim koji su povezani sa disk file-ovima; ovo ćete naučiti Dana 16.

Možete nedobiti samo jedan karakter na stream-u, između čitanja, i ne možete nedobiti **EOF** u bilo koje vrijeme.
Funkcija **ungetc()** vraća **ch** sa uspjehom i **EOF** ako karakter ne može biti vraćem stream-u.

→→ Linijski ulaz (unos (input))

Funkcije linijskog ulaza čitaju liniju sa ulaznog stream-a → one čitaju sve karaktere do karaktera sljedeća novalinija. Standardna biblioteka koja ima funkcije dvije linije ulaza, **gets()** i **fgets()**.

→→→ gets() ←←← Funkcija

Već ste se upoznali sa **gets()** funkcijom Dana 10. Ovo je pravonaprijed funkcija, koja čita liniju sa **stdin** i smješta je na string.
Prototip funkcije je:

```
char *gets(char *str);
```

gets() uzima pointer na tip **char** kao svoj argument i vraća pointer na tip **char**.
gets() funkcija čita karaktere iz **stdin**, dok ne naiđe na novalinija (**\n**) ili **e**nd-**o**f-**f**ile; novalinija se zamijenjuje sa **null** karakterom, i string se smješta na lokaciju na koju pokazuje (indicira) **str**.

Povratna vrijednost je pointer na string (isto kao i **str**). Ako **gets()** naiđe na grešku ili pročita end-of-file prije nego što se unese ijedan karakter, vraća se **null** pointer.

Prije pozivanja **gets()**-a, vi morate alocirati dovoljno memorijskog prostora da smjestite string, koristeći metode koje su pokrivene Dana 10.
Ova funkcija nema način kako da zna da li je prostor na koji pokazuje **ptr** alociran; string je ulaz (input) i smješten s početkom na **ptr** u svakom slučaju. Ako prostor nije alociran, string može prepisati (overwrite) neke podatke i prouzrokovati programske greške.

Listinzi **10.5** i **10.6** koriste **gets()**.

→→→ fgets() ←←← Funkcija

fgets() bibliotečna funkcija je slična sa **gets()** u tome što ona čita liniju texta sa ulazno stream-a (input stream). Flexibilnija je, zato što dozvoljava programeru da specificira ulazni stream za korištenje i maximalni broj karaktera koji će da se unesu.
fgets() funckija se često koristi za unos texta sa disk file-ova, što se radi Dana 16.

Da bi je koristili za ulaz (unos (input)) od **stdin**, morate specificirati (navesti) **stdin** kao ulazni stream (input stream).

Prototip za **fgets()** je:

```
char *fgets(char *str, int n, FILE *fp);
```

Zadnji parametar, **FILE *fp**, se korsti da specificira ulazni stream. Za sada, jednostavno specificirajte standardni ulazni stream, **stdin**, kao stream argument.

Pointer **str** pokazuje gdje je ulazni string smješten.
Argument **n** specificira maximalan broj karaktera koji se unose.
fgets() funkcija čita karaktere sa ulazno stream-a dok ne naiđe na **n**ovalinija ili **e**nd-**o**f-**l**ine ili **n-1** karaktera se ne pročita. Novalinija je uključena u string i prekida (terminira) se sa **\0** prije nego što se smjesti. Povratne vrijednosti od **fgets()** su iste kao što je opisano ranije za **gets()**.

Precizno govoreći, **fgets()** ne unosi ni jednu liniju texta (ako definišete liniju kao sekvencu karaktera koja završava sa novalinija).
Ona može da čita manje od pune linije ako linija sadrži više od **n-1** karaktera.

Kada se koristi sa **stdin**, izvršenje se ne vraća od **fgets()** sve dok ne pritisnete **<enter>**, ali samo je prvih **n-1** karaktera smješteno u string. Novalinija je uključena u string samo ako spadne pod prvih **n-1** karaktera.

Listing **14.6** demonstrira **fgets()** funkciju.

Listing 14.6. Upotreba fgets() funkcije za unos s tastature.

```
1:  /* Demonstrira fgets() funkciju. */
2:
3:  #include <stdio.h>
4:
5:  #define MAXLEN 10
6:
7:  main()
8:  {
9:     char buffer[MAXLEN];
10:
11:    puts("Enter text a line at a time; enter a blank to exit.");
12:
13:    while (1)
14:    {
15:         fgets(buffer, MAXLEN, stdin);
16:
17:         if (buffer[0] == `\n')
18:             break;
19:
20:         puts(buffer);
21:    }
22:    return 0;
23: }
```

```
Enter text a line at a time; enter a blank to exit.
Roses are red
Roses are
 red
Violets are blue
Violets a
re blue
Programming in C
Programmi
ng in C
Is for people like you!
Is for pe
ople like
 you!
```

Linija **15** sadrži **fgets()** funkciju. Kada se program pokrene, unesite linije dužine manje nago što je **MAXLEN** da vidite šta se dešava. Ako je unesena linija veća nego **MAXLEN**, prvih **MAXLEN-1** karaktera se čita sa prvim pozivom na **fgets()**; preostali karakteri ostaju u tastaturinom buffer-u i oni se čitaju pomoću sljedećeg poziva **fgets()**-a ili bilo koje druge funkcije koja čita sa **stdin**. Program izlazi kada se unese prazna linija (blank line) (linije **17** i **18**).

→→ Formatirani ulaz (Formatted Input)

Ulazna funkcija pokrivena do ove tačke je jednostavno uzimala jedan ili više karaktera iz ulaznog stream-a i stavljala ih negdje u memoriju. Nije urađena nikakva interpretacija formatiranja ulaza, i još uvijek nemate način kako da smještate numeričke vrijednosti.

Na primjer, kako biste unijeli vrijednost **12.86** sa tastature i pridružili je na varijablu tipa **float**??? Unesite **scanf()** i **fscanf()** funkcije. Upoznali ste se s **scanf()** Dana 7.

Ove dvije funkcije su identične, s razlikom što **scanf()** UVIJEK koristi **stdin**, dok korisnik može navesti (specificirati) ulazni stream u **fscanf()** funkciji.
Ovaj dio pokriva **scanf()**;
fscanf() uopšte se korsti sa disk file-ovima koji su pokriveni Dana 16.

Argumenti funkcije scanf()

scanf() funkcija uzima varijablin broj argumenata; potrebna su joj minimalno dva.
Prvi argument je format string koji koristi specijalne karaktere, koji kaže **scanf()**-u kako da interpretira (shvati) ulaz.
Drugi i ostali argumenti su adrese od varijable-i (množina) kojima su pridruženi ulazni podaci.

Ego ga i primjerćić:

```
scanf("%d", &x);
```

Prvi argument, **"%d"**, je format string. U ovom slučaju, **%d** govori **scanf()**-u da gleda na jednu **signed integer** vrijednost.
Drugi argument koristi adresa-od (address-of) operator (**&**) da kaže **scanf()**-u da pridruži ulaznu vrijednost varijabli **x**.

Sad možete pogledati detalje format string-a:

scanf() format string može sadržavati sljedeće:

- Spaces i tabs, koji su ignorisani (mogu biti korišteni da se format string napravi čitljivijim).
- Karaktere (ali ne **%**), koji se uspoređuju sa nebijelim (nonwhitespace) karakterima u ulazu
- Jedan ili više konverzionih specifikatora, koji sadrže **%** karakter nakon kojeg slijedi specijalni karakter, Uopšte, format string sadrži jedan konverzioni specifikator za svaku varijablu.

Jedini potrebni dio format string-a je konverzioni specifikator. Svako konverzioni specifikator počinje sa **%** karakterom i sadrži opcionalne i potrebne komponente u određenom redosljedu.
scanf() funkcija primjenjuje konverzionie specifikatore u format string-u, po redu, na ulazna polja (input fields).
Ulazno polje (input field) je sekvenca nebijelih karaktera koje završava kada sljedeći bijeli space naiđe ili kada se dostigne širina polja, ako je specificirana.

Komponente konverzionih specifikatora uključuju sljedeće:

- Opcionalnu pridružnu ugušujuću (**suppression flag**) zastavicu (*****) odmah nakon **%**. Ako je prisutna, ovaj karakter govori **scanf()**-u da obavi konverzije koje odgovaraju (korespondiraju) trenutnim konverzionom specifikatoru, ali da ignoriše rezultat (ne pridružuje se ni jednoj varijabli).
- Sljedeća komponenta, širina polja (**field width**), je također opcionalna. Širina polja je decimalni broj koji specificira (označava) širinu, u karakterima, izlaznog polja. Drugim riječima, širina polja označava koliko se karaktera od (from) **stdin scanf()** treba ispitati za trenutnu konverziju. Ako širina polja nije navedena, ulazno polje se širi do sljedećeg bijelog znaka.
- Sljedeća komponenta je **opcionalni modifikator preciznosti**, jedan karakter koji može biti **h**, **l**, ili **L**. Ako je prisutan, modifikator preciznosti mijenja značenje specifikatora tipa koji ga slijedi. Detalji su dati kasnije u ovom poglavlju.
- Jedina potrebna kompomenta od konverzionog specifikatora (osim **%**) je specifikator tipa. Ovi karakteri su navedeni i opisani u Tabeli **14.3**. Na primjer, specifikatoru tipa **d** je potreban **int *** (pointer na tip **int**).

→ Tabela 14.3. Specifikator tipa karaktera korišten u scanf() konverzionim specifikatorima.

Tip	Argument	Značenje tipa
d	**int ***	Decimalni integer.
i	**int ***	Integer u decimalnoj, oktalnoj (sa prethodnom **0**), ili hexadecimalnoj (sa prethodnim **0X** ili **0x**) notaciji.
o	**int ***	Integer u oktalnoj notaciji sa ili bez vodeće (prethodne) **0**.
u	**unsigned int ***	unsigned decimalni integer.
x	**int ***	Hexadecimalni integer sa ili bez vodećih (prethodnih) **0X** or **0x**.
c	**char ***	Jedan ili više karaktera se čitaju i pridružuju sekvencijalno memorijskoj lokaciji na koju pokazuje argument. Ne dodaje se terminator **\0**. Ako argument širine polja nije daat, čita se jedan karakter. Ako je dat argument širine polja, taj broj karaktera, uključujući bijeli space (ako ga ima), se čita.
s	**char ***	String od nonwhitespace karaktera se čita u specificiranu (navedenu) memorijsku lokaciju, i dodaje se terminator **\0**.
e,f,g	**float ***	floating-point broj. Brojevi mogu biti unešeni u decimalnoj ili scientific (sa decimalnom tačkom) notaciji.
[...]	**char ***	String. Prihvataju se samo oni karakteri koji su navedeni između zagrada. Ulaz prestaje: čim se naiđe na neodgovarajući karakter, dostigne se navedena širina polja, ili se pritisne **<enter>**. Da prihvatite **]** karakter, navedite ga prvo: **[]…]**. **\0** se dodaje na kraju stringa.
[^...]	**char ***	Isto kao i **[...]**, samo što se prihvataju karakteri koji nisu navedeni unutar zagrada.
%	**None**	Literal **%**: Čita **%** karakter. Ne prave se nikakva pridruživanja.

Prije nego što vidite neke primjere za **scanf()**, morate shvatiti modifikatore preciznosti, koji su navedeni u Tabeli **14.4**.

→ Tabela 14.4. Modifikatori preciznosti.

Modifikator preciznosti	Značenje
h	Kada se smjesti prije specifikatora tipa **d**, **I**, **o**, **u**, ili **x**, modifikator **h** specificira da je argument pointer na tip **short** umjest na tip **int**. Na PC-u, tip **short** je isti kao i tip **int**, tako da modifikator preciznosti **h**, nije nikad potreban.
l	Kada se smjesti ispred specifikatora tipa **d**, **i**, **o**, **u**, ili **x**, modifikator **l** specificira da je argument pointer na tip **long**. Kada se smjesti ispred specifikatora tipa **e**, **f**, ili **g**, modifikator **l** specificira da je argument pointer na tip **double**.
L	Kada se smjesti prije specifikatora tipa **e**, **f**, ili **g**, modifikator **L** specificira da je argument pointer na tip **long double**.

→ Upravljanje viškom karaktera (Handling Extra Characters)

Ulaz (unos (input)) od **scanf()**-a je buffer-ovan; nikakvi karakteri u stvari se ne primaju od **stdin** dok korisnik ne pritisne **<enter>**. Cijela linija karaktera tada “pristiže” od **stdin**, i procesiraju se, po redu, od strane **scanf()**-a.
Izvršenje se vraća od **scanf()** samo onda, kada se privi dovoljno unosa da odgovara specifikacijama u format string-u. Takođe i, **scanf()** procesuira samo dovoljno karaktera od **stdin**-a da zadovolji svoj format string. Dodatni (extra), nepotrebni karakteri, ako ih ima, ostaju da čekaju u **stdin**. Ovi karakteri mogu prouzrokovati probleme.
Pogledajte bliže operacije **scanf()**-a da vidite kako.

Kada se poziv na **scanf()** izvrši, i korisnik je unio jednu liniju, možete imati tri situacije.
Za ove primjere, pretpostavite da **scanf("%d %d", &x, &y);** je izvršen; drugim riječina, **scanf()** očekuje dva decimalna **integer**-a.
Evo mogućnosti:

- Linija koju korisnik unese, odgovara format string-u. Na primjer, pretpostavimo da korisnik unese **12 13** nakn čega slijedi **<enter>**. U ovom slučaju, nema problema. **scanf()** je zadovoljen, i nikakvi karakteri se ne ostavljaju u **stdin**.
- Linija koju korisnik unese ima premalo elemenata da odgovaraju format string-u. Na primjer, pretpostavimo da korisnik unese **12** nakon čega pritisne **<enter>**. U ovom slučaju, **scanf()** nastavlja da čeka na nedostajući unos. Jednom kad je unos primljen, izvršenje se nastavlja, i nikakvi karakteri se ne ostavljaju u **stdin**.
- Linija koju korisnik unese ima više elemenata nego što je potrebno od format string-a. Na primjer, pretpostavimo da korisnik unese **12 14 16** nakon čega pritisne **<enter>**. U ovom slučaju, **scanf()** čita **12** i **14** i onda se vraća. Dodatni (extra) karakteri, **1** i **6**, su ostavljeni da čekaju u **stdin**.

U ovoj trećoj situaciji (naročito, ovi preostali karakteri) koji mogu prouzrokovati probleme. Oni ostaju na čekanju sve dok se vaš program izvršava, do sljedećeg puta kad vaš program čita ulaz (input) iz **stdin**-a. Tada se ovi preostali karakteri prvi čitaju, ispred bilo kojeg unosa koje napravi korisnik u međuvremenu. Jasno je kako bi ovo prouzrokovalo grešku.
Na primjer, sljedeći kod piita korisnika da unese **integer**, pa onda string:

```
puts("Enter your age.");
scanf("%d", &age);
puts("Enter your first name.");
scanf("%s", name);
```

Recimo, na primjer, da u odgovoru na prvi propt, korisnik odluči da bude precizniji i unese **29.00** i onda pritisne **<enter>**. Prvi poziv na **scanf()** traži **integer** vrijednost, tako da čita karaktere **29** iz **stdin**-a i pridružuje vrijednost varijabli **age**. Karakteri **.00** su ostavljeni da čekaju u **stdin**-u. Sljedeći poziv na **scanf()** je traženje stringa. Ona ide do **stdin** za unos i nalazi **.00** koji čeka tamo. Rezultat je da se string **.00** pridružuje na **name**.

Kako izbjeći ovaj problem???
Ako ljudi koji koriste vaš program nikad ne prave greške pri unosu informacija, postoji jedno rješenje → ali je prilično nepraktično.

Bolje rješenje je da budete sigurni da nema nikakvih dodatnih karaktera koji čekaju u **stdin**-u prije prompt-anja korisnika za unos.
Ovo možete uraditi pozivajući **gets()**, koji čita sve preostale karaktere iz **stdin**-a, sve do i uključujući kraj linije (end of the line).
Radije, nego pozivajući **gets()** direktno iz programa, možete ga staviti u posebnu funkciju sa opisnim imenom kao **clear_kb()**. Ova funkcija je pokazana u Listingu **14.7**.

Listing 14.7. Čišćenje stdin od dodatnih karaktera, da bi se izbjegle greške.

```
1: /* Ciscenje stdin od dodatnih karaktera. */
2:
3: #include <stdio.h>
4:
5: void clear_kb(void);
6:
7: main()
8: {
9:     int age;
10:     char name[20];
11:
12:     /* Prompt za korisnikove godine (age). */
13:
```

```
14:     puts("Enter your age.");
15:     scanf("%d", &age);
16:
17:     /* Ocisti stdin od bilo kojih dodatnih karaktera. */
18:
19:     clear_kb();
20:
21:     /* Sad prompt-a za korisnikovo ime (name). */
22:
23:     puts("Enter your first name.");
24:     scanf("%s", name);
25:     /* Prikazi podatke. */
26:
27:     printf("Your age is %d.\n", age);
28:     printf("Your name is %s.\n", name);
29:
30:     return 0;
31: }
32:
33: void clear_kb(void)
34:
35: /* Cisti stdin od bilo kojih karaktera koji chekaju. */
36: {
37:     char junk[80];
38:     gets(junk);
39: }
```

```
Enter your age.
29 and never older!
Enter your first name.
Bradley
Your age is 29.
Your name is Bradley.
```

ANALIZA: Kada pokrenete Listing **14.7**, unesite neke dodatne karaktere nakon vaših godina, i pritisnite **<enter>**. Uvjerite se da ih program ignoriše i da vas pravilno promt-a za vaše ime. Onda modifikujte program odstranjujući poziv za **clear_kb()**, i pokrenite ga ponovo. Svi dodatni karakteri na istoj liniji kao gdje su vaše godine, se pridružuju na ime (name).

→→→ Upravljanje viškom karaktera sa →→→ fflush() ←←←

Postoji drugi način na koji možete očistiti dodatne karaktere koji su utipkani.
→→ **fflush()** funkcija baca (flushes) informacije u stream. → uključujući standarni ulazni stream (input stream).
fflush() se uopšteno koristi sa disk file-ovima (Dan 16); ipak, takođe se mogu koristiti sa Listingom **14.7**, da bi se pojednostavio.
Listing **17.8** koristi **fflush()** funkciju, umjesto **clear_kb()** funkcije koja je kreirana u Listingu **14.7**.

Listing 14.8. Čišćenje stdin-a od dodatnih karaktera korišćenjem fflush().

```
1:   /* Ciscenje stdin od dodatnih karaktera. */
2:   /* Upotreba fflush() funkcije           */
3:   #include <stdio.h>
4:
5:   main()
6:   {
7:      int age;
8:      char name[20];
9:
10:     /* Prompt za korisnikove godine. */
```

```
11:     puts("Enter your age.");
12:     scanf("%d", &age);
13:
14:     /* Ocisti stdin od bilo kojih dodatnih karaktera. */
15:     fflush(stdin);
16:
17:     /* Novi prompt za korisnikovo ime. */
18:     puts("Enter your first name.");
19:     scanf("%s", name);
20:
21:     /* Prikazi podatke. */
22:     printf("Your age is %d.\n", age);
23:     printf("Your name is %s.\n", name);
24:
25:     return 0;
26: }
```

```
Enter your age.
29 and never older!
Enter your first name.
Bradley
Your age is 29.
Your name is Bradley.
```

ANALIZA: Kao što možete vidjeti u liniji **15**, **fflush()** funkcija se koristi.
Prototip za **fflush()** funkciju je:

```
int fflush( FILE *stream);
```

stream je stream koji treba da se baci. U Listingu **14.8**, standardni ulazni stream, **stdin**, se prosljeđuje ka **stream**.

→→ scanf() primjeri

Najbolji način kako da se upoznate sa operacijama **scanf()** funkcije je da je koristite.
Ona je moćna funkcija, ali nekad može biti zbunjujuća.
Listing **14.9** demonstrira neke od neuobičajenih načina da se koristi **scanf()**.
Trebali biste kompajlirati i pokrenuti ovaj program, a onda experimantisati, praveći izmjene na **scanf()** format string-u.

Listing 14.9. Nekoliko načina korištenja scanf()-a za unos s tastature.

```
1:  /* Demonstrira neke upotrebe scanf()-a. */
2:
3:  #include <stdio.h>
4:
5:
6:
7:  main()
8:  {
9:      int i1, i2;
10:     long l1;
11:
12:     double d1;
13:     char buf1[80], buf2[80];
14:
15:     /* Koristi 1 modifikator za unos long integer-a i double-ova.*/
16:
17:     puts("Enter an integer and a floating point number.");
18:     scanf("%ld %lf", &l1, &d1);
```

```
19:     printf("\nYou entered %ld and %lf.\n",l1, d1);
20:     puts("The scanf() format string used the l modifier to store");
21:     puts("your input in a type long and a type double.\n");
22:
23:     fflush(stdin);
24:
25:     /* Koristi sirinu polja da podijeli unos (ulaz (input)). */
26:
27:     puts("Enter a 5 digit integer (for example, 54321).");
28:     scanf("%2d%3d", &i1, &i2);
29:
30:     printf("\nYou entered %d and %d.\n", i1, i2);
31:     puts("Note how the field width specifier in the scanf() format");
32:     puts("string split your input into two values.\n");
33:
34:     fflush(stdin);
35:
36:     /* Koristeci iskljuceni space (excluded space) da podijeli */
37:     /* liniju unosa u dva stringa na space (at the space). */
38:
39:     puts("Enter your first and last names separated by a space.");
40:     scanf("%[^ ]%s", buf1, buf2);
41:     printf("\nYour first name is %s\n", buf1);
42:     printf("Your last name is %s\n", buf2);
43:     puts("Note how [^ ] in the scanf() format string, by excluding");
44:     puts("the space character, caused the input to be split.");
45:
46:     return 0;
47: }
```

```
Enter an integer and a floating point number.
123 45.6789
You entered 123 and 45.678900.
The scanf() format string used the l modifier to store
your input in a type long and a type double.
Enter a 5 digit integer (for example, 54321).
54321
You entered 54 and 321.
Note how the field width specifier in the scanf() format
string split your input into two values.
Enter your first and last names separated by a space.
Gayle Johnson
Your first name is Gayle
Your last name is Johnson
Note how [^ ] in the scanf() format string, by excluding
the space character, caused the input to be split.
```

ANALIZA: Ovaj listing počinje sa definicijama neoliko varijabli u linijama **9** do **13** za unos podataka. Onda vas vodi kroz korake različitih unešenih tipova podataka.
Linije **17** do **21** imaju za unos i printanje **long integer**-e i **double**.
Linija **23** poziva **fflush()** funkciju da očisti neželjene karaktere iz standardnog ulaznog stream-a (**stdin**).
Linije **27** i **28** uzimaju sljedeću vrijednost, peto-karakterni integer. Zato što postoje specifikatori širine, peto-cifreni **integer** se dijeli (cijepa) u dva **integera**;
jedan koji ima dva karaktera, i drugi koji ima tri karaktera.
Linija **34** poziva **fflush()** da očisti tastaturu ponovo.
Posljednji primjer, u linijama **36** do **44**, koristi isključujući karakter (excluce character).
Linija **40** koristi **"%[^]%s"**, koji govori **scanf()**-u da uzme string, ali da se zaustavi na bilo kojem razmaku (space). Ovo efektivno dijeli unos (ulaz (input)).

scanf() funkcija se može koristiti za većinu vaših ulaznih potreba, naročito onih koji se tiču brojeva (stringovi se mogu unijeti na lakši način sa **gets()**).
Često se isplati, ipak, da napišete vaše specijalne (specialized) ulazne funkcije.
Možete vidjeti neke primjere o korisničko definisanim (user-defined) funkcijama Dana 18.

> **SIKORISTITE** prednost poduženih (extended) karaktera u vašim programima. Kada koristite produžene karaktere, trebali biste ostati konzistentni sa drugim programima.
> **NE** zaboravite da provjerite ulazni stream za dodatne karaktere.
> **KORISTITE gets()** i **scanf()** funkcije umjesto **fgets()** i **fscanf()** funkcija kada samo koristite standardni ulazni file (input file) **stdin**.

→→→ Ekran izlaz (Screen Output)

Funkcije ekranskog izlaza su podijeljene u tri generalne kategorije zajedno sa nekim linijama kao ulaznim funkcijama: karakterni izlaz, linijski izlaz, i formatirani izlaz.

Vi ste upoznati sa nekim od ovih funkcija u ranijim poglavljima. Ovaj dio ih pokrive sve detaljno.

→→ Karakterni izlaz sa putchar(), putc(), i fputc()

C-ova bibliotečna karakterna izlazna funkcija šalje jedan karakter ka stream-u.
Funkcija **putchar()** šalje svoj izlaz ka **stdout** (obično ekran).
Funkcije **fputc()** i **putc()** šalju svoj izlaz ka stream-u koji je naveden u listi argumenata.

Korištenje putchar() Funkcije

Prototip za **putchar()**, koji se nalazi u **STDIO.H**, je kako slijedi:

```
int putchar(int c);
```

Ova funkcija piše karaktere koji su smješteni u **c** na **stdout**.
Iako prototip navodi (specificira) argument tipa **int**, vi prosljeđujete **putchar()**-u tip **char**. Takođe ga možete prosljediti (argument) kao tip **int** sve dok je vrijednost prigodna za karakter (tj., u opsegu od **0** do **255**). Funkcija vraća karakter baš onako kako je napisan, ili **EOF** ako se desi greška.

Vidjeli ste **putchar()** demonstriran u Listingu **14.2**.
Listing **14.10** prikazuje karaktere sa **ASCII** vrijednostima između **14** i **127**.

Listing 14.10. putchar() funkcija.

```
1: /* Demonstrira putchar(). */
2:
3: #include <stdio.h>
4: main()
5: {
6:     int count;
7:
8:     for (count = 14; count < 128; )
9:          putchar(count++);
10:
11:   return 0;
12: }
```

Vi također možete prikazati string sa **putchar()** funkcijom (kako je pokazano u Listingu **14.11**), iako su druge funkcije bolje rješenje za ovu svrhu.

Listing 14.11. Prikazivanje stringa sa putchar().

```
1: /* Koristenje  putchar()  za prikazivanje stringova. */
2:
3: #include <stdio.h>
4:
5: #define MAXSTRING 80
6:
7: char message[] = "Displayed with putchar().";
8: main()
9: {
10:     int count;
11:
12:     for (count = 0; count < MAXSTRING; count++)
13:     {
14:
15:         /* Gledajte na kraj stringa. Kada se nadje, */
16:         /* napisite karakter novalinija i izadjite iz petlje. */
17:
18:         if (message[count] == `\0')
19:         {
20:             putchar(`\n');
21:             break;
22:         }
23:         else
24:
25:         /* Ako se ne nadje kraj stringa, pishite sljedeci karakter. */
26:
27:             putchar(message[count]);
28:     }
29:     return 0;
30: }
              Displayed with putchar().
```

→→→ Korištenje putc() i fputc() Funkcija

Ove dvije funkcije obavljaju iste zadatke → šalju jedan karakter na navedeni (specificirani) stream. **putc()** je makro implementacija od **fputc()**. O makroima ćete učiti Dana 21, za sada samo se držite za **fputc()**.
Njen prototip je:

```
int fputc(int c, FILE *fp);
```

FILE *fp dio vas može zbuniti.
Vi prosljeđujete **fputc()** na izlazni stream u ovom argumentu (više Dana 16).
Ako navedete **stdout** kao stream, **fputc()** se ponaša isto kao i **putchar()**. Tako da, sljedeća dva iskaza su ekvivalentna:

```
putchar(`x');
fputc(`x', stdout);
```

→→→ Korištenje puts() i fputs() za izlaz stringa (String Output)

Vaš program prikazuje stringove na-ekran češće nego što prikazuje jedinstven (single) karakter. Bibliotečna funkcija **puts()** prikazuje stringove.
Funkcije **fputs()** šalje string na navedeni stream; u drugom slučaju, ona je identična sa **puts()**.
Prototip za **puts()** je:

```
int puts(char *cp);
```

***cp** je pointer na prvi karakter stringa koji želite da prikažete.
puts() funkcija prikazuje cijeli string sve do, ali ne uključujući, terminirajućeg **null** karaktera, dodajući novu liniju na kraju. Onda **puts()** vraća pozitivnu vrijednost ako je uspješno ili **EOF** u slučaju greške. (Zapamtite, **EOF** je simbolična konstanta sa vrdijednošću **–1**; ona je definisana u **STDIO.H**).

puts() funkcija se može koristiti za prokazivanje stringa bilo kojeg tipa, kako je demonstrirano u Listingu **14.12**.

Listing 14.12. Korištenje puts() funkcije za prikazivanje stringova.

```
1: /* Demonstrira puts(). */
2:
3: #include <stdio.h>
4:
5: /* Deklarisi i inicijaliziraj niiz pointera. */
6:
7: char *messages[5] = { "This", "is", "a", "short", "message." };
8:
9: main()
10: {
11:     int x;
12:
13:     for (x=0; x<5; x++)
14:         puts(messages[x]);
15:
16:     puts("And this is the end!");
17:
18:     return 0;
19: }
```

```
This
is
a
short
message.
And this is the end!
```

ANALIZA: Ovaj listing deklariše niiz pointera, tema koja još nije pokrivena. (biće sutra.)
Linije **13** i **14** pritaju svaki od stringova smještenih u niizu **message** (poruke).

→→→ Korištenje printf() i fprintf() za formatirani izlaz

Do sada, izlazne funkcije se prikazivale samo jedinstvene (single) karaktere i stringove. **Šta je sa brojevima???**
Da prikažete broj, morate koristiti **C**-ove bibliotečne formatirane izlazne funkcije, **printf()** i **fprintf()**. Ove funkcije, takođe, mogu prikazivati stringove i karaktere. Oficijalno ste upoznati sa **printf()** Dana 7, i koristili ste ga u skoro svakom poglavlju. Ovaj dio obezbjeđuje ostatak detalja.

Dvije funkcije **printf()** i **fprintf()** su identične, s razlikom što **printf()** uvijek šalje izlaz ka **stdout**, dok **fprintf()** navodi (specificira) izlazni stream.

fprintf() se obično koristi za izlaz ka disk file-ovima. (Dan 16).

printf() funkcija uzima varijablin broj argumenata, sa minimumom od jednog (1).
Prvi i jedini potrebni argument je format string, koji govori **printf()**-u kako da formatira izlaz. Opcionalni argumenti su varijable i izrazi, čije vrijednosti želite da prikažete.

Pogledajte ovih sljedećih par jednostavnih primjera, koji daju osjećaj (miris) za **printf()**, prije nego što uđete da postanete tata-mata:

- iskaz **printf("Hello, world.");** prikazuje poruku Hello, world. na-ekran. Ovo je primjer korištenja **printf()**-a sa samo jednim argumentom, format string-om. U ovom slučaju, format string sadrži samo literalni string koji će da se prikaže na-ekran.
- iskaz **printf("%d", I);** prikazuje **integer** varijablu i na-ekran. Format string sadrži samo format specifikator **%d**, koji govori **printf()**-u da prikaže jedan (single) decimalni **integer**. Drugi argument **I**-a je ime od varijable čija će vrijednost da se prikaže.
- Iskaz **printf("%d plus %d equals %d.", a, b, a+b);** prikazuje **2** plus **3** jednako **5** na-ekran (pretpostavljajući da su **a** i **b integer** varijable sa vrijednostima **2** i **3**, respektivno). Ovakva upotreba **printf()**-a ima četiri argumenta: format string koji sadrži literalnu text kao i specifikatore formata, i dvije varijable i izraz čije vrijednosti će da budu priakazane.

Sad pogledajte **printf()** format string više detaljno. On sadrže sljedeće:

- Nijedan, jedan ili više konverzionih specifikatora koji govore **printf()**-u kako da prikaže vrijednosti u svojoj argumentnoj listi. Konverziona komanda sadrži **%f** nakon čega slijedi jedan ili više karaktera.
- Karakteri, koji nisu dio komande konverzije, se prikazuju onako-kako-jesu (as-is).

Treći primjer format stringa je **%d** plus **%d** jednako **%d**. U ovom slučaju, tri **%d**-a su konverzione komande, i ostatak stringa, uključujući spaces, su literalni karakteri koji se direktno prikazuju.

Sad možete secirati konverzione komande. Komponente komande su date ovdje i objašnjene sljedeće. Komponente u zagradama su opcionalne.

```
%[flag][field_width][.[precision]][l]conversion_char
```

conversion_char je jedini potrebni dio konverzione komande (osim **%**).
Tabela **14.5** navodi (lists) konverzione karaktere i njihova značenja.

→ Table 14.5. printf() i fprintf() konverzioni karakteri.

Konverzioni karakter	Značenje
d, i	Prikazuje **signed integer** u decimalnoj notaciji.
u	Prikazuje **unsigned integer** u decimalnoj notaciji.
o	Prikazuje **integer** u **unsigned octal** notaciji.
x, X	Prikazuje **integer** u **unsigned hexadecimal** notaciji. Koristi **x** za lowercase izlaz i **X** za uppercase izlaz (output).
c	Prikazuje jedan (single) karakter (argument daje karakterov **ASCII** kood).
e, E	Prikazuje **float** ili **double** u scientific notaciji (npr.,, 123.45 se prikazuje kao 1.234500e+002). Šest cifara se prikazuju s desna od decimalne tačke, osim ako je druga preciznost naglašena sa **f** specifikatorom. Koristite **e** ili **E** za kontrolu slučaja (case) izlaza.
f	Prikazuje **float** ili **double** u decimalnoj notaciji (npr., 123.45 se prikazuje kao 123.450000). Šest cifara se prikazuje s desne strane decimalne tačke, osim ako je druga preciznost naglašena.
g, G	Koristite **e**, **E**, ili **f** format. **e** ili **E** format se koristi ako je exponent manji od **-3** ili veći nego što je preciznost (koja je po default-u to **6**). **f** format se koristi u drugom slučaju (otherwise). Povučene nule trunc-irane (Trailing zeros are truncated).
n	Ništa se prikazuje. Odgovarajući argument **n** konverzionoj komandi je pointer na tip **int**. **printf()** funkcija pridružuje ovoj varijabli broj karaktera koji su do sada izašli.
s	Prikazuje string. Argument je pointer na **char**. Karakteri se prikazuju sve dok ne naiđe **null** karakter ili se ne prikaže broj karaktera navedenih sa preciznošću (po default-u je 32767). Terminirajući **null** karakter nije izlaz (output).
%	Prikazuje **%** karakter.

Možete smjestiti i modifikator prije konverzionog karaktera. Ovaj se modifikator primjenjuje samo na konverzione karaktere **o**, **u**, **x**, **X**, **l**, **d** i **b**. Kada se primjeni, ovaj modifikator navodi da je argument tipa **long**, a ne tipa **int**.
Ako se modifikator i primjeni na konverzione karaktere **e**, **E**, **f**, **g**, i **G**, on navodi da je argument tipa **double**.
Ako se i smjesti prije nekog drugog konverzionog karaktera, on se ignoše.

Specifikator preciznosti se sastoji iz decimalne tačke (**.**) samo ili nakon njega brojem.
Specifikator preciznosti (precision specifier) se primjenjuje samo na konverzione karaktere **e**, **E**, **f**, **g**, **G**, i **s**. On navodi broj cifara koji se prikazuju s desne strane decimalne tačke, ili kada se koristi sa **s**, broj karaktera ka izlazu.
Ako se koristi decimalna tačka sama, ona navodi preciznost od **0**.

Specifikator širine polja odlučuje minimalni broj karaktera izlaza. Širina-polja specifikator može biti sljedeće:

- Decimalni **integer** koji ne počinje sa **0**. Izlaz se postavlja lijevo sa prostorima (spaces) da se popuni namijenjena širina polja.
- Decimalni **integer** koji počinje sa **0**. Izlaz se postavlja lijevo sa nulama da se popuni namijenjena širina polja.
- ***** karakter. Vrijednost sljedećeg argumenta (koji mora biti **int**) se koristi kao širina polja. Na primjer, ako je **w** tipa **int** sa vrijdenošću **10**, iskaz **printf("%d", w, a);** printa vrijednosti sa širinom polja **10**.

Ako se ne navede širina polja, ili ako je specifikator širine polja uži od izlaza, izlazno polje je široko onoliko koliko je potrebno.

Posljednji opcionalni dio od **printf()** format stringa je zastavica (**flag**), koja odmah slijedi nakon **%** karaktera.

Evo četiri dustupne **zastavice**:
- Ovo znači da je izlaz pozicioniran-lijevo u svom polju, umjesti pozicioniran-desno, što je default.

+ Ovo znači da se signed brojevi uvijek prikazuju sa **+** ili **–** predznakom.

`` Razmak (space) znači da pozitivnim brojevima prethodi space (razmak).

Ovo se primjenjuje samo na **x**, **X**, i **o** konverzione karaktere. On navodi da nenula broj se prikazuje sa prethodnim **0X** ili **0x** (za **x** i **X**) ili nakon vodeće **0** (za **o**).

Kada koristite **printf()**, format string može biti literalni string zatvoren između navodnika u **printf()** argumentoj listi. Takođe može biti smješten u memoriju **null**-terminirajući string, u kojem slučaju vi prosljeđujete pointer na string od **printf()**.
Na primjer, ovaj iskaz:

```
char *fmt = "The answer is %f.";
printf(fmt, x);
```

je ekvivalentan sa ovim iskazom:

```
printf("The answer is %f.", x);
```

Ako što je objašnjeno Dana 7, **printf()** format string može sadržavati escape sequence koji obezbjeđuju kontrolu nad izlazom.

Tabela **14.6** navodi najčešće korištene escape sequence. Na primjer, uključujući sequence novalinija (**\n**) u format string,pruzrokuje da se kasniji izlaz prikaže sa početkom u novoj liniji ekrana.

→ Tabela 14.6. Najčešće korišteni escape sequences.

Sequence	Značenje
\a	Zvono (alert)
\b	Backspace
\n	Novalinija
\t	Horizontalni tab
\\	Backslash
\?	Question mark
\'	Single quote
\"	Double quote

printf() je nešto komplikovaniji. Najbolji način da ovo naučite je da pogledate primjere i onda experimentišete sami.
Listing **14.13** demonstrira neke od načina kako možete koristiti **printf()**.
Listing 14.13. Neki načini upotrebe printf() funkcije.

```
1:  /* Demonstracija  printf()-a. */
2:
3:  #include <stdio.h>
4:
5:  char *m1 = "Binary";
6:  char *m2 = "Decimal";
7:  char *m3 = "Octal";
8:  char *m4 = "Hexadecimal";
```

```
9:
10: main()
11: {
12:     float d1 = 10000.123;
13:     int n, f;
14:
15:
16:     puts("Outputting a number with different field widths.\n");
17:
18:     printf("%5f\n", d1);
19:     printf("%10f\n", d1);
20:     printf("%15f\n", d1);
21:     printf("%20f\n", d1);
22:     printf("%25f\n", d1);
23:
24:     puts("\n Press Enter to continue...");
25:     fflush(stdin);
26:     getchar();
27:
28:     puts("\nUse the * field width specifier to obtain field width");
29:     puts("from a variable in the argument list.\n");
30:
31:     for (n=5;n<=25; n+=5)
32:         printf("%*f\n", n, d1);
33:
34:     puts("\n Press Enter to continue...");
35:     fflush(stdin);
36:     getchar();
37:
38:     puts("\nInclude leading zeros.\n");
39:
40:     printf("%05f\n", d1);
41:     printf("%010f\n", d1);
42:     printf("%015f\n", d1);
43:     printf("%020f\n", d1);
44:     printf("%025f\n", d1);
45:
46:     puts("\n Press Enter to continue...");
47:     fflush(stdin);
48:     getchar();
49:
50:     puts("\nDisplay in octal, decimal, and hexadecimal.");
51:     puts("Use # to precede octal and hex output with 0 and 0X.");
52:     puts("Use - to left-justify each value in its field.");
53:     puts("First display column labels.\n");
54:
55:     printf("%-15s%-15s%-15s", m2, m3, m4);
56:
57:     for (n = 1;n< 20; n++)
58:         printf("\n%-15d%-#15o%-#15X", n, n, n);
59:
60:     puts("\n Press Enter to continue...");
61:     fflush(stdin);
62:     getchar();
63:
64:     puts("\n\nUse the %n conversion command to count characters.\n");
65:
66:     printf("%s%s%s%s%n", m1, m2, m3, m4, &n);
67:
68:     printf("\n\nThe last printf() output %d characters.\n", n);
69:
```

```
70:     return 0;
71: }
        Outputting a number with different field widths.
        10000.123047
        10000.123047
           10000.123047
                10000.123047
                     10000.123047
         Press Enter to continue...
        Use the * field width specifier to obtain field width
        from a variable in the argument list.
        10000.123047
        10000.123047
           10000.123047
                10000.123047
                     10000.123047
         Press Enter to continue...
        Include leading zeros.
        10000.123047
        10000.123047
        00010000.123047
        0000000010000.123047
        000000000000010000.123047
         Press Enter to continue...
        Display in octal, decimal, and hexadecimal.
        Use # to precede octal and hex output with 0 and 0X.
        Use - to left-justify each value in its field.
        First display column labels.
        Decimal        Octal          Hexadecimal
        1              01             0X1
        2              02             0X2
        3              03             0X3
        4              04             0X4
        5              05             0X5
        6              06             0X6
        7              07             0X7
        8              010            0X8
        9              011            0X9
        10             012            0XA
        11             013            0XB
        12             014            0XC
        13             015            0XD
        14             016            0XE
        15             017            0XF
        16             020            0X10
        17             021            0X11
        18             022            0X12
        19             023            0X13
         Press Enter to continue...
        Use the %n conversion command to count characters.
        BinaryDecimalOctalHexadecimal
        The last printf() output 29 characters.
```

→→ Preusmjeravanje ulaza i izlaza

Program koji koristi **stdin** i **stdout** može iskoristiti osobinu operativnog sistema nazvanu preusmjeravanje (redirection). Preusmjeravanje vam dozvoljava da uradite sljedeće:

- Izlaz poslan do **stdout**-a može biti poslan na disk file ili na printer umjesto na ekran.
- Programski ulaz iz **stdin**-a može doći sa disk file-a umjesti sa tastature.

Vi ne koodirate preusmjeravanja u vaš program; vi ga navedete na komandnoj liniji kada pokrenete program.
U DOS-u, kao u UNIX-u, simboli za preusmjeravanje su **<** i **>** .

Prvo ćemo govoriti o preusmjeravanju izlaza.

Sjećate li se vašeg prvog programa u **C**-u, **HELLO.C**? On je koristio **printf()** bibliotečnu funkciju da prikaže poruku **Hello, world** na-ekran-u. Kao što sada znate, **printf()** šalje izlaz do **stdout**, tako da se on može preusmjeriti. Kada napišete ime programa na komandnoj-liniji prompt-a, slijedite je sa **>** simbolom i imenom novog odredišta (destinacije):

```
hello > destination
```

Tako da, ako utipkate **hello >prn**, programski izlaz ide na printer umjesto na ekran (**prn** je DOS-ovo ime za printer koji je priključen na **LPT1:**).
Ako utipkate **hello >hello.txt**, izlaz se smješta na disk file sa imenom **HELLO.TXT**.

Kada preusmjeravate izlaz na disk file, budite oprezni. Ako je file već postojao, stara kopija se briše i zamjenjuje sa novim file-om. Ako file nije postojao, on se kreira. Kada se preusmjerava na file, također možete koristiti **>>** simbol. Ako navedeno odredišni file već postoji, programski izlaz se lijepi na kraj tog file-a.

Listing **14.14** demonstrira preusmjeravanje.
Listing 14.14. Preusmjeravanje ulaza i izlaza.

```
1: /*Moze se koristiti kao demonstracija preusmjeravanja ulaza i izlaza. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:    char buf[80];
8:
9:    gets(buf);
10:   printf("The input was: %s\n", buf);
11:   return 0;
12: }
```

ANALIZA: Ovaj program prihvata jednu liniju ulaza sa **stdin**-a i onda šalje liniju ka **stdout**-u, čemu prethodi **The input was:**.
Nakom kompajliranja i povezivanja (linkovanja) programa, pokrenite ga bez preusmjeravanja (pretpostavljajući da je program nazvan **LIST1414**) unoseći **LIST1414** u komandnoj-liniji prompt-a.
Ako unesete **I am teaching myself C**, program prikaže sljedeće na-ekran:

```
The input was: I am teaching myself C
```

Ako pokrenete program unoseći **LIST1414 >test.txt** i čineći isti unos, ništa se ne prikazuje na-ekran-u. Umjesto, file nazvan **TEST.TXT** se kreira na disku.
Ako koristite DOS TYPE (ili ekvivalentni) komandu da prikažete sadržaj file-a:

```
type test.txt
```

vidjet ćete da file sadrži samo liniju The input was: **I am teaching myself C**.

→→ Preusmjeravanje ulaza

Sad pogledajmo preusmjeravanje ulaza. Prvo trebate izvorni file. Koristite editor da kreirate file nazvan **INPUT.TXT** koji sadrži jednu liniju **William Shakespeare**.
Sad pokrenite Listing **14.14** unoseći sljedeće u DOS prompt:

```
list1414 < INPUT.TXT
```

Program ne čeka na vas da napravite unos sa tastature.
Umjesto toga, on odmah prikazuje sljedeću poruku na-ekran:

```
The input was: William Shakespeare
```

Stream **stdin** je preusmjeren na disk file **INPUT.TXT**, tako da programski poziv na **gets()** čita one linije texta iz file-a, umjesto sa tastature.

Možete preusmjeriti ulaz i izlaz istovremeno.
Pokušajte pokrenuti program sa sljedećom komandom da preusmjerite **stdin** na file **INPUT.TXT** i preusmjerite **stdout** na **JUNK.TXT**:

```
list1414 < INPUT.TXT > JUNK.TXT
```

Preusmjeravanje **stdin** i **stdout** može biti korisno u nekim situacijama. Na primjer, sortirajući program može sortirati ili tastaturin ulaz ili sadržaj nekog disk file-a.
Tako i, mailling lista program može prikazati adresu na-ekran, slati je na printer za mailling oznake, ili smjestiti ih u file za neku drugu upotrebu.

> **NAPOMENA:** Zapamtite da je preusmjeravanje **stdin** i **stdout**-a osobina operativnog sistema, a ne samog **C** jezika. Ipak, on obezbjeđuje još jedan primjer flexibilnost stream-ova.

→→ Kada koristiti fprintf()

Kao što je rečeno ranije, bibliotečna funkcija **fprintf()** je identična sa **printf()**, s razlikom što možete navesti stream na koji će se slati izlaz. Glavna upotreba **fprintf()**-a uključuje disk file-ove (Dan 16). Postoje još dvije upotrebe, objašnjene ovdje.

→→→ Upotreba stderr

Jedan od **C**-ovih predefinisanih stream-ova je **stderr** (standard error). Poruka programske greške se tradicionalno šalje ka stream-u **stderr**, a ne **stdout**-u. **Zašto je to tako???**

Kao što ste upravo naučili, izlaz na **stdout** se može preusmjeriti na odredište različito od prikaza ekrana. Ako se **stdout** preusmjeri, korisnik možda neće biti svjestan o bilo kojoj grešci koju program šalje do **stdout**-a. Za razliku od **stdout**-a, **stderr** ne može biti preusmjeren i uvijek je povezan sa ekranom (bar u DOS-u - UNIX sistemi mogu dozvoliti preusmjeravanje **stderr**-a).
Usmjeravajući poruke greške na **stderr**, možete biti sigurni da ih korisnik uvijek vidi. Ovo radite sa **fprintf()**:

```
fprintf(stderr, "An error has occurred.");
```

Možete napisati funkciju da radi sa porukama greške i onda zove funkciju kada se greška desi umjesto da zove **fprintf()**:

```
error_message("An error has occurred.");
void error_message(char *msg)
{
    fprintf(stderr, msg);
}
```

Koristeći vašu funkciju umjesto direktnog pozivanja **fprintf()**-a, vi ste obezbjedili dodatnu flexibilnost (jednu od prednosti u strukturnom programiranju).
Na primjer, u specijalnim okolnostima možda želite da poruke greške vašeg programa idu na printer ili na disk file. Sve što treba da uradite je da modifikujete **error_message()** funkciju tako da se izlaz šalje na željeno odredište.

→ Printer izlaz pod DOS-om

Na DOS ili Windows sistemima, vi šaljete izlaz do vašeg printera, pristupajući predefinisanom stream-u **stdprn**. Na IBM-ovim PC-ovima i kompatibilnim, stream **stdprn** je spojen sa uređajem **LPT1:** (**prvi paralelni printer port**).
Listing **14.15** predstavlja jednostavan primjer.

> **NAPOMENA:** Da koristite **stdprn**, morate isključiti **ASCII** kompatibilnost u vašem kompajleru.

Listing 14.15. Slanje izlaza na printer.

```
1: /* Demonstrira  printer-ski izlaz. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     float f = 2.0134;
8:
9:      fprintf(stdprn, "\nThis message is printed.\r\n");
10:     fprintf(stdprn, "And now some numbers:\r\n");
11:     fprintf(stdprn, "The square of %f is %f.", f, f*f);
12:
13:     /* Send a form feed. */
14:     fprintf(stdprn, "\f");
15:
16:     return 0;
17: }
          This message is printed.
          And now some numbers:
          The square of 2.013400 is 4.053780.
```

> **NAPOMENA:** Izlaz se printa na printeru. Neće se pojaviti na-ekranu.

ANALIZA: Ako vaš DOS sistem ima printer koji je spojen na port **LPT1:**, možete kompajlirati i pokrenuti ovaj program. On printa tri linije na stranici. Linija **14** šalje **"\f"** printeru. **\f** je escape sequence za **form feed**, komanda koja prouzrokuje da printer nastavi na sljedeću stranicu (ili, u slučaju laserskih printera, da izbaci trenutnu stranicu).

NIKAD ne pokušavajte da preusmjerite **stderr**.
KORSITIE fprintf() da kreirate programe koji mogu slati izlaz na **stdout**, **stderr**, **stdprn**, ili neki drugi stream.
KORISTITE fprintf() sa **stderr** da printate poruke greške na ekran.
NE koristite **stderr** za svrhe koje nisu printanje poruka greške ili upozorenja.
KREIRAJTE funkcije kao što su **error_message** da učinite vaš kood struktuiranijim i održavanijim.

Sažetak

Ovo je bio dug dan pun važnih informacija o programskom ulazu i izlazu. Naučili ste kako **C** koristi stream-ove, tretirajuće sav ulaz kao sekvencu byte-ova. Također ste naučili da **C** ima pet predefinisanih stream-ova:

Stdin	Tastature
Stdout	Ekran
Stderr	Ekran
Stdprn	Printer
stdaux	Komunikacioni port

Ulaz sa tastature stiže sa **stdin** stream-a. Koristeći **C**-ove standardne bibliotečne funkcije, vi možete prihvatiti ulaz s tastature karakter po karakter, liniju po liniju (at a time), ili kao formatirane brojeve i stringove. Karakterni ulaz može biti buffer-ovan ili nebuffer-ovan, eho-isan ili neeho-isan.

Izlaz za prikaz na ekranu se normalno radi sa **stdout** stream-om. Kao i ulaz, programski izlaz može biti pomoću karaktera, linije, ili kao formatirani brojevi i stringovi. Za izlaz na printer, koristili ste **fprintf()** da šaljete podatke na stream **stdprn**.

Kada koristite **stdin** i **stdout**, vi možete preusmjeriti ulaz i izlaz. Ulaz može doći sa disk file-a umjesto sa tastature, i izlaz može ići na disk file ili na printer, umjesto na prikazni ekran.

Konačno, naučili ste zašto poruke greške trebaju biti poslane do **stderr** stream-a umjesto **stdout**. Zato što je **stderr** obično spojen sa prikaznim ekranom, vi ste sigurni da ćete vidjeti grešku poruke, čak i ako se programski izlaz preusmjeri.

P&O

Q Šta se desi ako pokušam da dobijem ulaz sa izlaznog strem-a?
A Možete napisati **C** program da ovo radi, ali neće raditi. Na primjer, ako pokušate da koristite **stdprn** sa **scanf()**-om, program se kompajlira u exekutivni file, ali je printer nesposoban da šalje ulaz, tako da program ne radi kako je namjenjen.

Q Šta se desi ako preusmjerim jedan od standardnih stream-ova?
A Radeći ovo može izazvati probleme kasnije u programu. Ako preusmjerite stream, morate ga vratiti nazad ako treba da se koristi kasnije u programu. Većina funkcija opisanih u ovom poglavlju koristi standarde stream-ove. Svi oni koriste iste stream-ove, tako da ako promjenite stream na jednom mjestu, vi ih promjenite za sve funkcije. Na primjer, pridružite **stdout** jednako sa **stdprn** u jednom od listinga u ovom poglavlju, i pogledajte šta će se desiti.

Q Zašto ne bi uvijek koristio fprintf() umjesto printf()? Ili fscanf() umjesto scanf()?
A Ako koristite standardni izlazne ili ulazne stream-ove, trebali biste koristiti **printf()** i **scanf()**. Koristeći ove jednostavnije funkcije, ne morate se peglati sa ostalim stream-ovima.

Exercises

1. Napišite iskaz koji printa **"Hello World"** na ekran.
2. Koristite dvije različite **C** funkcije da urade istu stvar kao i funkcija u primjeru **1**.
3. Napišite iskaz da printa **"Hello Auxiliary Port"** na standardni pomoćni port.
4. Napišite iskaz koji uzima string od **30** karaktera ili kraći. Ako se naiđe na asterisk (*), truncate string.
5. Napišite jedan iskaz koji printa sljedeće:

```
Jack asked, "What is a backslash?"
Jill said, "It is `\'"
```

6. ON YOUR OWN: Napišite program koji preusmjerava file na printer jedan karakter at a time.
7. ON YOUR OWN: Napišite program koji koristi preusmjeravanje za prihvat ulaza sa disk file-a, broji broj nailazaka na slovo u file-u, i onda prikazuje rezultat na-ekranu (A hint is provided in Appendix G, "Answers.")
8. ON YOUR OWN: Napišite program koji printa vaš **C** izvorni kood. Koristite preusmjeravanje da unesete izvorni file, i koristite **fprintf()** da radi printanje.
9. ON YOUR OWN: Modifikujte program iz vježbe **8** da stavlja liniju brojeva na početak listinga kada se printa. (A hint is provided in Appendix G.)
10. ON YOUR OWN: Napišite “tipkajuće” program koji prihvata unos sa tastature, eho-iše ga na ekran, i onda reproducira ovaj unos (input) na printer. Program treba da broji linije i pomjera papir u printeru na novu stranicu ako je potrebno. Koristite funkcijsku tipku za terminiranje programa.

LEKCIJA 15 → Pointers: Beyond the Basics ←

→→→→ Pointer na Pointere

Kao što ste naučili Dana 9, **pointer** je numerička varijabla sa vrijednošću koja je adresa neke druge varijable. Vi deklarišete pointer koristeći indirektni operator (*****).
Na primjer, deklariacija:

```
int *ptr;
```

deklariše pointer nazvan **ptr**, koji može pokazivati na varijablu tipa **int**. Onda možete koristiti adresa-od operator (**&**) da učinite da pointer pokazuje na neku određenu varijablu odgovarajućeg tipa. Pretpostavljajući da je **x** deklarisan kao varijabla tipa **int**, iskaz:

```
ptr = &x;
```

pridružuje adresu od **x** na **ptr** i čini **ptr** pointer na **x** (da **ptr** pokazuje na **x**). Ponovo, koristeći indirektni operator, vi možete pristupiti pokazuje-na varijabli koristeći njen pointer.
Oba sljedeća iskaza pridružuju vrijednost **12** varijabli **x**:

```
x = 12;
*ptr = 12;
```

Zato što je pointer sam po sebi numerička vrijednost, on je smješten u memoriji vašeg kompjutera na određenoj adresi. Tako da, vi možete kreirati **pointer na pointer**, varijablu čija je vrijednost adresa od pointera.
Evo kako:

```
int x = 12;                    /* x is a type int variable. */
int *ptr = &x;                 /* ptr is a pointer to x. */
int **ptr_to_ptr = &ptr;       /* ptr_to_ptr is a pointer to a */
                               /* pointer to type int. */
```

Primjetitie upotrebu duplog indirektnog operatora (******) kada se deklariše pointer na pointer. Vi također možete koristiti dupli indirektni operator kada pristupate pokazuje-na (pointed-to) varijabli sa pointerom na pointer. Tako da, iskaz:

```
**ptr_to_ptr = 12;
```

pridružuje vrijednost **12** varijabli **x**, i iskaz:

```
printf("%d", **ptr_to_ptr);
```

prikazuje vrijednost **x**-a na-ekran. Ako zabunom koristite jedan indirektni operator, dobićete greške. Iskaz:

```
*ptr_to_ptr = 12;
```

pridružuje vrijednost **12 ptr**-u, što rezultira da ptr pokazuje-na šta god da je smješteno na adresi **12**. Ovo je očigledno **greška**:

Deklaracija i korištenje pointera na pointer se zove višestruka indirekcija (multiple indirection).

Ne postoji pravi limit za nivo mogućih višestrukih indirekcija → možete imati pointer na pointer na pointer do beskonačnosti, ali rijetko ima koristi od prelaženja preko dva nivoa; komplexnost je prizivanje greške.

→ Za šta možete koristiti pointer na pointer???
Najčešća upotreba su niizovi pointera, koji se rade kasnije u ovom poglavlju. Listing **19.5** Dana 19 prezentuje primjer upotrebe višestrukih indirekcija.

→→→ Pointeri i Višedimenzionalni Nizovi

Ime niza bez zagrada je pointer na prvi element tog niza. Kao rezultat, lakše je koristiti pointer notaciju kada pristupate određenim tipovima niizova.

→ Šta je višedimenzionalnim niizovima???
Zapamtite da se višedimanzionalni niiz deklariše sa jednim setom zagrada za svaku dimenziju.
Na primjer, sljedeći iskaz deklariše dvo-dimenzionalni niz koji sadrži osam varijabli tipa **int**:

```
int multi[2][4];
```

Možete zamisliti (vizualizirati) da niz ima redove i kolone → u ovom slučaju, dva reda i četiri kolone.
Možete zamisliti da je **multi** (više) dvo-elementni niz, da je svaki od ova dva elementa niz sa četiri **int**egera.

Evo kako interpretirati komponente deklaracije:
1. Deklariši niz nazvan **multi** (više).
2. Niz **multi** sadrži dva elementa.
3. Svaki od ova dva elementa sadrži četiri elementa.
4. Svaki od ova četiri elementa su tipa **int**.

Sada, vratimo se temi imena niizova kao pointera. (Ipak je ovo poglavlje o pointerima!).
Kao i kod jednodimenzionalnih nizova, ime višedimenzionalnog niza, bez zagrada, je pointer na prvi element niiza.
Nastavljajući sa ovim primjerom, **multi** je pointer na prvi element dvo-dimenzionalnog niza koji je deklarisan kao **int multi [2][4]**.
Šta je tačno prvi element od multi???
To nije varijabla **multi[0][0]** tipa **int**, kao što možda mislite. Zapamtite da je **multi** niz od nizova, tako da je prvi element **multi[0]**, koji je niz od četiri varijable tipa **int** (jedan od dva takva niza sadržana u **multi**).

Sad, ako je multi[0] takođe niz, da li on pokazuje na nešto???
Da, naravno! **multi[0]** pokazuje na svoj prvi element, **multi[0][0]**. Možda se pitate zašto je **multi[0]** pointer. Zapamtite da je ime niza bez zagrada pointer na prvi element tog niza.
Pojam **multi[0]** je ime od niza **multi[0][0]** bez zadnjeg para zagrada, tako da je kvalifikovan kao pointer.

Možda vam pomogne ako zapamtite pravila za bilo koji **n** – dimenzionalni niiz koji se koristi u koodu:
- Ime niiza nakon čega slijedi **n** parova zagrada (svaki par sadrži prigodni index, naravno) procjenjuje (evaluates) se kao niiz podataka (tj., podataka smještenih u specifičnom niizu elemenata).
- Ime niza nakon čega slijedi manje parova zagrada od **n** procjenjuje se kao pointer na element niiza.

Tako, u primjeru, **multi** se procjenjuje kao pointer, **multi[0]** se procjenjuje kao pointer, i **multi[0][0]** se procjenjuje kao niiz podataka.

Sad pogledajte na šta sva tri pointera stvarno pokazuju. Listing **15.1** deklariše dvo-dimenzionalni niiz; slično onim koje ste koristili u primjerima; i onda printa vrijednosti od pridruženih pointera. Takođe printa adrese od prvih elemenata niza.

Listing 15.1. Odnos između više-dimensionalnog niza i pointera.

```
1:    /* Demonstrira pointere i visedimensionalne nizove. */
2:
3:    #include <stdio.h>
4:
5:    int multi[2][4];
6:
7:    main()
8:    {
9:        printf("\nmulti = %u", multi);
10:       printf("\nmulti[0] = %u", multi[0]);
11:       printf("\n&multi[0][0] = %u\n", &multi[0][0]);
13:       return(0);
12:   }
multi = 1328
multi[0] = 1328
&multi[0][0] = 1328
```

ANALIZA: Stvarna vrijednost možda neće biti **1328** na vašem sistemu, ali će tri vrijednosti biti iste. Adresa od niza **multi** je ista kao i adresa od niza **multi[0]**, i obadva su jednaka adresi od prvog **integer**a u niizu, **multi[0][0]**.

Ako sva tri pointera imaju istu vrijednost, koja je praktična razlika između njih u pojmu vašeg programa? Kompajler zna na šta pointer pokazuje. Da budemo precizniji, kompajler zna **veličinu** predmeta na koji pokazuje pointer.

Koje su veličine elemenata koje vi koristite? Listing **15.2** koristi operator **sizeof()** da prikaže veličine, u byte-ima, tih elemenata.

Listing 15.2. Određivanje veličina elemenata.

```
1: /* Demonstrira veličine od elemenata višedimenziog niza. */
2:
3: #include <stdio.h>
4:
5: int multi[2][4];
6:
7: main()
8: {
9:     printf("\nThe size of multi = %u", sizeof(multi));
10:    printf("\nThe size of multi[0] = %u", sizeof(multi[0]));
11:    printf("\nThe size of multi[0][0] = %u\n", sizeof(multi[0][0]));
12:    return(0);
13: }
```

Izlaz ovog programa (pretpostavljajući da vaš kompajler koristi dvo-byte-ne **integer**e) je kako slijedi:

```
The size of multi = 16
The size of multi[0] = 8
The size of multi[0][0] = 2
```

ANALIZA: Ako pokrenete 32-bit-ni operativni sistem, kao npr. IBMov OS/2, vaš izlaz će biti 32, 16 i 4. Ovo je zato što tip **int** sadrži četiri byte-a na ovim sistemima.

Razmislite o ovim veličinama. Niz **multi** sadrži dva niza, svaki od ovih nizova sadrži četiri **integer**a. Svaki **integer** zahtjeva dva byte-a prostora. Sa ukupno osam **integer**-a, veličina od 16 byte-a ima smisla.

multi[0] je niz koji sadrži četiri **integer**a. Svaki **integer** uzima dva byte-a, tako da veličina za **multi[0]** od 8 byte-a ima smisla.

multi[0][0] je **integer**, tako da je njegova veličina, naravno, dva byte-a.

C kompajler zna veličinu objekta na koji se pokazuje, i pointer aritmetika uzima ovu veličinu u obzir (u računu).
Kada se pointer inkrementira, on se inkrementira za veličinu objekta na koji on pokazuje.

Kada ovo primjenite na primjer, **multi** je pointer na četvoro-elementni **integer** niz sa veličinom od **8**. Ako inkrementirate **multi**, njegova vrijednost bi se trebala povećati za **8** (veličina od četvoro-elementnog niza).
Tako, ako **multi** pokazuje na **multi[0]**, **(multi + 1)** bi trebao pokazivati na **multi[1**].

Listing **15.3** testira ovu teoriju.
Listing 15.3. Pointer aritmetika sa više-dimensionalnim nizovima.

```
1: /*  Demonstrira pointer aritmetiku sa pointerima */
2: /*  na više-dimensional nizove. */
3:
4: #include <stdio.h>
5:
6: int multi[2][4];
7:
8: main()
9: {
10:     printf("\nThe value of (multi) = %u", multi);
11:     printf("\nThe value of (multi + 1) = %u", (multi+1));
12:     printf("\nThe address of multi[1] = %u\n", &multi[1]);
13:     return(0);
14: }
            The value of (multi) = 1376
            The value of (multi + 1) = 1384
            The address of multi[1] = 1384
```

ANALIZA: Precizna vrijednost može biti različita na vašem sistemu, ali su odnosi isti.
Inkrementirajući **multi** za **1**, povećava njegovu vrijednost za **8** (ili za 16 na 32-bit-nim sistemima) i pravi da pokazuje na sljedeći element niza, **multi[1]**.

U ovom primjeru vidjeli ste da je **multi** pointer na **multi[0]**. Takođe ste vidjeli da je **multi[0]** i sam pointer (na **multi[0][0]**). Tako da, **multi** je pointer na pointer.
Da koristite **multi** izraz da pristupite podacima niza, morate koristiti **duplu indirekciju**. Da printate vrijednost smještene u **multi[0][0]**, možete koristiti bilo koji od sljedeća tri iskaza:

```
printf("%d", multi[0][0]);
printf("%d", *multi[0]);
printf("%d", **multi);
```

Ovaj koncept se primjenjuje na nizove koji imaju tri ili više dimenzija.

Kada radite sa više-dimanzionalnim nizovima, imajte ovu misao na pameti:
Niz sa **n** dimenzija ima elemente koji su nizovi sa **n-1** dimenzijama. Kad **n** postane **1**, elementi niza su varijable tipa podataka navedenom na početku linije deklaracije niza.

Do sada ste koristili imena nizova koji su pointer konstante i koji ne mogu biti promjenjeni. Kako bi deklarisali pointer varijablu koja pokazuje na element od više-dimenzionalnog niza?
Nastavimo sa prethodnim primjerom, koji deklariše dvo-dimenzionalni niz, kako slijedi:

```
int multi[2][4];
```

Da deklarišete pointer varijablu **ptr**, koja može pokazivati na element od **multi** (tj., može pokazivati na četvoro-elementni **integer** niz), možete napisati:

```
int (*ptr)[4];
```

Onda možete učiniti da **ptr** pokazuje na prvi element od **multi** pišući:

```
ptr = multi;
```

Možda se pitate zašto su neophodne zagrade u pointer deklaraciji.
Zagrade (**[ ]**) imaju veće značenje (prioritet) nego *****. Ako ste napisali:

```
int *ptr[4];
```

vi biste deklarisali niz od četiri pointera na tip **int**. Svakako, vi možete deklarisati i koristiti nizove pointera. Ipak, to nije što ste želili.

→ Kako mogu koristiti pointere na elemente više-dimenzionalnih nizova?
Kao i sa jedno-dimenzionalnim nizovima, pointeri se moraju koristiti da prosljeđuju niz funkciji. Ovo je ilustrovano za više-dimenzionalni niz u Listingu **15.4**, koji koristi dvije metode za prosljeđivanje više-dimenzionalnog niza funkciji.

Listing 15.4. Prosljeđivanje više-dimensionalnog niza funkciji koristeći pointer.

```
1:  /* Demonstrira prosljedjivanje pointera sa vise-dimanzionalnog */
2:  /* niza funkciji. */
3:
4:  #include <stdio.h>
5:
6:  void printarray_1(int (*ptr)[4]);
7:  void printarray_2(int (*ptr)[4], int n);
8:
9:  main()
10: {
11:    int  multi[3][4] = { { 1, 2, 3, 4 },
12:                         { 5, 6, 7, 8 },
13:                         { 9, 10, 11, 12 } };
14:
15:    /* ptr je pointer na niz od 4 int-a. */
16:
17:    int (*ptr)[4], count;
18:
19:    /* Podesi da ptr pokazuje na prvi element od multi. */
20:
21:    ptr = multi;
22:
23:    /* Sa svakom petljom, ptr se inkrementira da pokazuje na sljedeci */
24:    /* element (tj., sljedeci 4-elementni integer niz) od multi. */
25:
26:    for (count = 0; count < 3; count++)
27:        printarray_1(ptr++);
28:
29:    puts("\n\nPress Enter...");
30:    getchar();
31:    printarray_2(multi, 3);
32:    printf("\n");
33:    return(0);
34: }
35
36: void printarray_1(int (*ptr)[4])
37: {
```

```
38: /* Printa elemente od jednog cetvoro-elementnog integer niza. */
39: /* p je pointer na tip int. Morate koristiti tip cast */
40: /* da napravite p jednako sa adresom u ptr-u. */
41:
42:      int *p, count;
43:      p = (int *)ptr;
44:
45:      for (count = 0; count < 4; count++)
46:          printf("\n%d", *p++);
47: }
48:
49: void printarray_2(int (*ptr)[4], int n)
50: {
51: /* Prints the elements of an n by four-element integer array. */
52:
53:      int *p, count;
54:      p = (int *)ptr;
55:
56:      for (count = 0; count < (4 * n); count++)
57:          printf("\n%d", *p++);
58: }
```

```
1
2
3
4
5
6
7
8
9
10
11
12
Press Enter...
1
2
3
4
5
6
7
8
9
10
11
12
```

ANALIZA: U linijama **11** do **13**, program deklariše i inicijalizira niz **integer**a, **multi[3][4**].
Linije **6** i **7** su prototipi za funkcije **printarray_1()** i **printarray_2()**, koje printaju sadržaj niza.

Funkciji **printarray_1()** (linije **36** do **47**) se prosljeđuje samo jedan argument, pointer na niz od četiri **integer**a. Ova funkcija printa sva četiri elementa od niza. Prvi put, **main()** poziva **printarray_1()** u liniji **27**, prosljeđuje pointer na na prvi element (prvi četvoro-elementni **integer** niz) u **multi**. Onda zove funkciju još dva puta, inkrementira pointer svaki put da pokazuje na drugi, i onda na treći, element od **multi**. Nakon sva tri poziva, **12 integer**a u **multi** se prikazuju na-ekran.

Druga funkcija, **printarray_2()**, koristi drugačiji pristup. i njoj se prosljeđuje pointer na niz od četiri **integer**a, ali, u dodatku, prosljeđuje se **integer** varijabla koja navodi broj elemenata (broj od niza sa četiri **integer**a) koji je sadržan u više-dimenzionalnom nizu. Sa jednim pozivom iz linije **31**, **printarray_2()** prikazuje cijeli sadržaj od **multi**.

Obadvije funkcije koriste pointer notaciju da idu kroz pojedinačne **integer**e u nizu. Notacija **(int *)ptr** u obadvije funkcije (linije **43** i **54**) možda nije jasna.

(int *) je **typecast**, koji privremeno mijenja varijablin tip podataka iz njenog deklarisanog tipa podataka u novi. **typecast** je potreban kada pridružujete vrijednost od **ptr**-a ka **p**-u, zato što su oni pointeri na različite tipove (**p** je pointer na tip **int**, dok je **ptr** pointer na niz od četiri **integer**a). **C** ne dozvoljava pridruživanje vrijednosti od jednog pointera pointeru koji je raličitog tipa. **typecast** govori kompajleru, **"Samo za ovaj iskaz, tretiraj ptr kao pointer na tip int."** Dana 20, pokriva **typecast** detaljnije.

NE zaboravite da koristite dupli indirektni operator (******) kada deklarišete pointer na pointer.
NE zaboravite da se pointer inkrementira za veličinu pointerovog tipa (obično za onu na koju pokazuje pointer).
NE zaboravite da koristite zagrade kada deklarišete pointer na nizove. Da deklarišete pointer na karakterni niz, koristite ovaj format:

```
char (*letters)[26];
```

Da deklarišete niz pointera na karaktere, koristite ovaj format:

```
char *letters[26];
```

→→→→ Nizovi pointeera

Sjetite se iz Dana 8, da je niz skup smještajnih lokacija podataka koje imaju isti tip podataka i na njih se odnosi sa istim imenom. Zato što su pointeri jedan od **C**-ovih tipova podataka, vi možete deklarisati i koristiti nizove pointeera. Ovaj tip konstruisanja programa može biti veoma snažan u nekim situacijama.

Možda je najčešća upotreba nizova pointeera, sa stringovima. String, kao što znate iz Dana 10, je sekvenca karaktera smještenih u memoriji. Početak stringa je označen sa pointerom na prvi karakter (pointer na tip **char**), a kraj stringa je označen sa **null** karakterom.
Deklarišući i inicijalizirajući niz pointeera na tip **char**, možete pristupiti i manipulisati s velikim brojem stringova koristeći niz pointeera. Svaki element u ovom nizu pokazuje na različit string, i peetljajući kroz niz, vi možete pristupiti svakom od njih u pokretu.

→→ Stringovi i Pointeri: -A Review-

Ovo je dobro vrijeme da se ponovi nešto materijala iz Dana 10, a što se tiče alokacije i inicijalizacije stringa.
Jedan način da alocirate i inicijalizirate string je da deklarišete niz tipa **char**, kako slijedi:

```
char message[] = "This is the message.";
```

Možete postiči istu stvar, deklarišući pointer na tip **char**:

```
char *message = "This is the message.";
```

Obadvije deklaracije su ekvivalentne. U svakom slučaju, kompajler alocira dovoljno prostora da drži string zajedno sa terminirajućim **null** karakterom, i izraz **message** je pointer na početak stringa.
Ali šta je sa sljedeće dvije deklaracije???

```
char message1[20];
char *message2;
```

Prva linija deklariše niiz tipa **char** koji je **20** karaktera dugačak, sa **message1** kao pointerom na prvu poziciju niza. Iako je prostor za niz alociran, on nije inicijaliziran, i sadržaj niza je neodređen.
Druga linija deklariše **message2**, pointer na tip **char**. Nikakav prostor za string nije alociran u ovom iskazu → samo prostor da drži pointer. Ako želite da kreirate string i onda imate **message2** da pokazuje na njega, vi morate prvo alocirati prostor za string. Dana 10 ste naučili kako da koristite **malloc()** memerijsko alokacijsku funkciju za ovu svrhu. Zapamtite da bilo koji string mora imati prostor

alociran za njega, bilo za vrijeme kopilacije u deklaraciji ili za vrijeme izvršenja sa **malloc()** ili nekom memorijsko alokacijskom funkcijom koja se odnosi na ovo.

→→→ Niiz Pointera na Tip Char

Sad kad ste završili ponavljanje, **kako deklarišete niz pointera (array of pointers)???**
Sljedeći iskaz deklariše niz od **10** pointera na tip **char**:

```
char *message[10];
```

Svaki element niza **message[]** je samostalni pointer na tip **char**. Kao što možete pretpostaviti, možete kombinovati deklaraciju sa inicijalizacijom i alokacijom smještajnog prostora za stringove:

```
char *message[10] = { "one", "two", "three" };
```

Ova deklaracija radi sljedeće:

- Ona alocira **10**-elementni niz nazvan **message**; svaki element od **message** je pointer na tip **char**.
- Ona alocira prostor negdje u memoriji (gdje, briga vas) i smješta tri inicijalizacijska stringa, svaki sa terminirajućim **null** karakterom.
- Ona inicijalizira **message[0]** da pokazuje na prvi karakter od stringa **"one"**, **message[1]** da pokazuje na prvi karakter od stringa **"two"**, i **message[2]** da pokazuje na prvi karakter od stringa **"three"**.

Listing **15.5** je primjer korištenja niizova pointera.

Listing 15.5. Inicijalizacija i upotreba nizova pointera na tip char.

```
1: /* Inicijaliziranje niza pointera na tip char. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     char *message[8] = { "Four", "score", "and", "seven",
8:                          "years", "ago,", "our", "forefathers" };
9:   int count;
10:
11:     for (count = 0; count < 8; count++)
12:         printf("%s ", message[count]);
13:     printf("\n");
14:     return(0);
15: }
      Four score and seven years ago, our forefathers
```

ANALIZA: Ovaj program deklariše niz od osam pointera na tip **char** i inicijalizira ih da pokazuju na osam stringova (linije **7** i **8**). On onda koristi **for** petlju u linijama **11** i **12** da prikaže svaki element niza na-ekran.

Vjerovatno vidite kako je manipulisanje nizova pointera lakše od manipulacije samih stringova. Prednost je očigledna u komplikovanijim programima, kao što su oni prezentovani kasnije u ovom poglavlju.
Prednost je najveća kada koristite funkcije. Puno je lakše prosljediti niz pointera funkciji nego prosljediti nekoliko stringova. Ovo može da se ilustruje prepisujući (rewriting) program u Listingu **15.5** tako da on koristi funkciju da prikaže stringove. Modifikovan program je prikazan u Listingu **15.6**.

Listing 15.6. Prosljeđivanje niza pointera funkciji.

```
1: /* Prosljedjivanje niza pointera funkciji. */
2:
3: #include <stdio.h>
4:
5: void print_strings(char *p[], int n);
6:
7: main()
8: {
9:     char *message[8] = { "Four", "score", "and", "seven",
10:                        "years", "ago,", "our", "forefathers" };
11:
12:     print_strings(message, 8);
13:     return(0);
14: }
15:
16: void print_strings(char *p[], int n)
17: {
18:     int count;
19:
20:     for (count = 0; count < n; count++)
21:         printf("%s ", p[count]);
22:    printf("\n");
23:  }
       Four score and seven years ago, our forefathers
```

ANALIZA: Gledajući liniju **16**, vidite da funkcija **print_strings()** uzima dva argumenta. Jedan je niz pointera na tip **char**, i drugi je broj elemenata u niizu. Tako da **print_strings()** može biti korištena da printa stringove na koje pokazuje niz pointera.

Pointer na pointer → Listing **15.6** deklariše niz pointera, i ime niza je pointer na njegov prvi element (od niza). Kada prosljedite niz funkciji, vi prosljeđujete pointer (ime niza) pointeru (prvi element niza).

Primjer

Sad je vrijeme za malo komplikovaniji primjer.
Listing **15.7** koristi mnogo programskih vještina koje ste naučili, uključujući nizove pointera. Ovaj program prihvata liniju ulaza sa tastature, alocirajući prostor za svaku liniju koja je unešena i drži bilješku o linijama u smislu niza pointera na tip **char**.
Kada signalizirate kraj unosa, unošeći praznu liniju, program sortira stringove po abecedi i prikazuje ih na-ekran.

Ako pišete program od početka (from scratch), pristupili biste dizajnu ovog programa iz strukturno programerske perspektive. Prvo, napravite listu stvari koje program mora da uradi:

1. Prihvata linije ulaza (input) sa tastature jedan-po-jedan (one at a time) dok se ne unese prazna linija.
2. Sortira linije texta po abecednosm redosljedu.
3. Prikazuje sortirane linije na-ekran.

Spisak predlaže da program mora imati bar tri funkcije: jednu za prihvat ulaza (unosa), jednu za sortiranje linija, i jednu da prikaže linije. Sad možete dizajnirati svaku funkciju posebno.
Za šta vam treba ulazna funkcija → nazvana **get_lines()** →da radi?

Ponovo, napravite spisak (listu):

1. Drži bilješku o unesenom broju linija, i vraća tu vrijednost pozivajućem programu nakon što se unesu sve linije.
2. Ne dozvoli unos više od dozvoljenog maximalnog broja linija.
3. Alociraj smještajni prostor za svaku liniju.
4. Drži bilješku o svim linijama smještajući pointere na stringove u niiz.
5. Vrati se pozivajućem programu kada se unese prazna linija.

Sad razmislite o drugoj funkciji, ona koja sortira linije.
Može se nazvati **sort()**. (Stvarno oreginalno, zar ne?)
Tehnika za sortiranje koja se koristi je jednostavna, brute-force (čista-sila) metoda koja uporeduje susjedne stringove i zamjenjuje ih ako je drugi string manji od prvog stringa. Preciznije, funkcija uporeduje dva stringa čiji su pointeri susjedni u nizu pointera i zamjenjuje dva pointera ako je potrebno.

Da budete sigurni da je sortiranje gotovo, morate ići kroz niz, od početka do kraja, upoređujući svaki par i zamjenjujući ih ako je potrebno. Za niz od **n** elemenata, morate proći kroz niz **n-1** puta. Zašto je ovo potrebno?
Zato što znamo kako radi **bubble sort** ☺.

Primjetite da je ovo vrlo neefikasan i neelegantan metod sortiranja. Ipak, lakše ga je implementirati i razumjeti, i adekvatniji je za kratke liste koje sortira ovaj program primjer.

Finalna funkcija prikazuje sortirane linije na-ekran. To je, u stvari, već napisano u Listingu **15.6**, i potrebna je samo manja modifikacija da ja koristite u Listingu **15.7**.

Listing 15.7. Program koji čita linije texta sa tastature, sortira ih po abecedi, i prikazuje sortiranu listu.

```
1:  /* Unesi listu stringova sa tastature, sortiraj ih, */
2:  /* i onda ih prikaži na ekran. */
3:  #include <stdlib.h>
4:  #include <stdio.h>
5:  #include <string.h>
6:
7:  #define MAXLINES 25
8:
9:  int get_lines(char *lines[]);
10: void sort(char *p[], int n);
11: void print_strings(char *p[], int n);
12:
13: char *lines[MAXLINES];
14:
15: main()
16: {
17:     int number_of_lines;
18:
19:     /* Čitaj linije sa tastature. */
20:
21:     number_of_lines = get_lines(lines);
22:
23:     if ( number_of_lines < 0 )
24:     {
25:         puts(" Memory allocation error");
26:         exit(-1);
27:     }
28:
29:     sort(lines, number_of_lines);
30:     print_strings(lines, number_of_lines);
```

```
31:     return(0);
32: }
33:
34: int get_lines(char *lines[])
35: {
36:     int n = 0;
37:     char buffer[80];  /* Privremeno smjestanje za svaku liniju. */
38:
39:     puts("Enter one line at time; enter a blank when done.");
40:
41:     while ((n < MAXLINES) && (gets(buffer) != 0) &&
42:            (buffer[0] != `\0'))
43:     {
44:         if ((lines[n] = (char *)malloc(strlen(buffer)+1)) == NULL)
45:             return -1;
46:         strcpy( lines[n++], buffer );
47:     }
48:     return n;
49:
50: } /* Kraj od get_lines() */
51:
52: void sort(char *p[], int n)
53: {
54:     int a, b;
55:     char *x;
56:
57:     for (a = 1; a < n; a++)
58:     {
59:         for (b = 0; b < n-1; b++)
60:         {
61:              if (strcmp(p[b], p[b+1]) > 0)
62:             {
63:                  x = p[b];
64:                  p[b] = p[b+1];
65:                  p[b+1] = x;
66:             }
67:         }
68:     }
69: }
70:
71: void print_strings(char *p[], int n)
72: {
73:     int count;
74:
75:     for (count = 0; count < n; count++)
76:         printf("%s\n ", p[count]);
77: }
```

```
Enter one line at time; enter a blank when done.
dog
apple
zoo
program
merry
apple
dog
merry
program
zoo
```

ANALIZA: Nekoliko novih bibliotečnih funkcija je korišteno za različite tipove manipulacije stringovima. One su objašnjene ovdje ukratko, a detaljno Dana 17.
File zaglavlja **STRING.H** mora biti uključen u programu koji koristi ove funckije.

U **get_lines()** funkciji, ulaz je kontrolisan sa **while** iskazom u linijama **41** i **42**, koje čitaju kako slijedi (kondenzovane ovdje u jednu liniju):

```
while ((n < MAXLINES) && (gets(buffer) != 0) && (buffer[0] != `\0'))
```

Uslov koji je testiran od strane **while** petlje ima tri dijela:
Prvi dio → **n<MAXLINES** → osigurava da maximalan broj linija nije unesen, još.
Drugi dio → **gets(buffer) !=0** → poziva **gets()** bibliotečnu funkciju da čita liniju sa tastature u buffer i verifikuje da se **e**nd-**o**f-**l**ine ili neka druga greška, nije desila.
Treći dio → **buffer[0] != '\0'** → verifikuje (potvrđuje) da prvi karakter koji je upravo unesen nije **null** karakter, koji bi dao signal da je unesena prazna linija.

Ako ijedan od ova tri uslova nije zadovoljen, **while** petlja se terminira, i izvršenje se vraća na pozivajući program, sa brojem unešenih linija, kao povratnom vrijedosti.

Ako su sva tri uslova zadovoljena, sljedeći **if** iskaz u liniji **44** se izvršava:

```
if ((lines[n] = (char *)malloc(strlen(buffer)+1)) == NULL)
```

Ovaj iskaz poziva **malloc()** da alocira prostor za string koji je upravo unesen.
→→→ **strlen()** funkcija vraća dužinu stringa prosljeđenog kao argument; vrijednost je inkrementirana za **1**, tako da **malloc()** alocira prostor za string plus njegov terminirajući **null** karakter.

Bibliotečna funkcija **malloc()**, možda se sjećate, vraća pointer. Iskaz pridružuje vrijednost pointera, koji je vraćen od **malloc()**-a, odgovarajućem elementu niza pointera. Ako **malloc()** vrati **NULL**, tada **if** petlja vraća izvršenje pozivajućem programu sa povratnom vrijednošću od **–1**. Kood u **main()**-u testira povratnu vrijednost **get_lines()** i provjerava da li je vrijednost, koja je vraćana, manja od **0**; linije **23** do **27** javljaju grešku memorijske alokacije i terminiraju program.

Ako je memorija uspješno alocirana, program koristi →→→ **strcpy()** funkciju, u liniji **46**, da kopira string iz privremene smještajne lokacije (buffer-a) u smještajni prostor koji je upravo alociran pomoću **malloc()**. Onda se **while** petlja ponavlja, uzimajući drugu liniju ulaza.

Jednom kad se izvršenje vrati iz **get_lines()** na **main()**, sljedeće se postiže (pretpostavljajući da se nije desila greška memorijske alokacije):

- Čita se broj linija texta sa tastature i smješta u memoriju kao **null**-terminirajući stringovi.
- Niz **lines[]** sadrži pointere na sve stringove. Redosljed pointera u nizu je redosljed u kojem su stringovi unešeni.
- Varijabla **number_of_lines** drži broj linija koje su unešene.

Sad je vrijeme za sortiranje.

→ zapamtite da vi nećete stvarno da premještate stringove okolo, samo redosljed pointera u nizu **lines[]**.

Pogledajte kood u funkciji **sort()**. On sadrži jednu **for** petlju, ugnijezđenu jedna u drugu (linije **57** do **68**). Vanjska petlja se izvršava **number_of_lines – 1** puta. Svaki put kad se izvrši vanjska petlja, unutrašnja petlja ide kroz niz pointera, upoređujući (string **n**) sa (string **n+1**) za od **n=0** do **n=number_of_lines – 1**. Poređenje se dešava sa bibliotečnom funkcijom →→→ **strcmp()** ←←← u liniji **61**, kojoj se prosljeđuju pointeri na dva stringa.

Funkcija **strcmp()** vraća jedno od sljedećeg:

- Vrijednost **veću od nula** ako je prvi string "veći od" drugog stringa.
- **Nula** ako su dva stringa indetična.
- Vrijednost **manju od nula** ako je prvi string "manji od" drugog stringa.

U programu, povratna vrijednost, od **strcmp()**, koja je veća od nule znači da je prvi string "veći od" drugog stringa, i oni moraju da se zamjene (tj., njihovi pointeri u **lines[]** moraju da se zamjene). Ovo se radi koristeći privremenu varijablu **x**.

Linije **63** do **65** obavljaju zamjenu.

Kada se izvršenje programa vrati od **sort()**, pointeri u **lines[]** su poredani pravilno: Pointer na "najmanji" string je u **lines[0]**, pointer na "sljedeći najmanji" je u **lines[1]**, itd.
Pretpostavimo, na primjer, da ste unijeli sljedećih pet linija, u ovom redosljedu:

```
dog
apple
zoo
program
merry
```

Na kraju, program poziva funkciju **print_strings()** da prikaže sortiranu listu stringova na-ekran.

→→→ Pointeri na Funkcije

Pointeri na funkcije obežbjeđuju još jedan način pozivanja funkcija.
"**Stani malo paša**", možda ćete pomisliti. "**kako možete imati pointer na funkciju? Zar pointer ne drži adresu gdje je varijabla smještena?"**

Paaa, da i ne. Tačno je pointer drži adresu, ali to ne mora biti adresa gdje je varijabla smještena. Kada se vaš program pokrene, kood za svaku funkciju je učitan (loaded) u memoriju s početkom na specifičnoj adresi.
→ Pointer na funkciju drži početnu adresu funkcije → njenu ulaznu tačku (its entry point).

→ Zašto koristiti pointer na funkciju???
Kao što sam rekao ranije, to obezbjeđuje flexibilniji način pozivanja funkcije. Dopušta da program izabere, između nekoliko funkcija, onu koja je najpogodnija za trenutne okolnosti.

→→ Deklariacija Pointera na Funkciju

Kao i ostali pointeri, vi morate deklarisati pointer na funkciju. Opšta forma deklaracije je kako slijedi:

```
type (*ptr_to_func)(parameter_list);
```

Ovaj iskaz deklariše **ptr_to_func** kao pointer na funkciju koja vraća **type** i prosljeđuje parametre u **parameter_list**.
Evo par konkretnijih primjera:

```
int (*func1)(int x);
void (*func2)(double y, double z);
char (*func3)(char *p[]);
void (*func4)();
```

- Prva linija deklariše **func1** kao pointer na funkciju koja uzima jedan argument tipa **int i** vraća tip **int**.
- Druga linija deklariše **func2** kao pointer na funkciju koja uzima dva argumenta tipa **double** i ima **void** povratnu vrijednost (nema povratne vrijednosti).
- Treća linija deklariše **func3** kao pointer na funkciju koja uzima niz pointera na tip **char** kao svoj argument i vraća tip **char**.
- Posljednja linija deklariše **func4** kao pointer na funkciju koja ne uzima nikakve argumente i ima **void** povratnu vrijednost.

Zašto su vam potrebne zagrade oko imena pointera??? Zašto ne možete napisati, za prvi primjer:

```
int *func1(int x);
```

Razlog za ovo ima veze sa prioritetom indirektnog operatora, ***.**
On ima relativno nizak prioritet, niži nego zagrade koje okružuju listu parametara. Deklaracija, koja upravo data, bez prvog seta zagrada, deklariše **func1** kao funkciju koja vraća pointer na tip **int**. (Funkcije koje vraćaju pointere su pokrivene Dana 18).

Kada deklarišete pointer na funkciju, uvijek pamtite da uključite set zagrada oko imena pointera i indirektnog operatora, ili ćete uletiti u belaj.

→→→ Initcijaliziranje i upotreba Pointera na Funkciju

Pointer na funkciju ne samo da mora biti deklarisan, već i inicijaliziran da pokazuje na nešto. To "nešto" je, naravno, funkcija. Ne postoji ništa posebno o funkciji na koju se pokazuje. Jedina potreba je da tip (**type**) njene povratne vrijednosti i parametarska lista (**parameter_list**) odgovaraju povratnom tipu i parametarskoj listi u deklaraciji pointera.

Na primjer, sljedeći kood deklariše i definiše funkciju i pointer na tu funkciju:

```
float square(float x);     /* Prototip funkcije.  */
float (*p)(float x);       /* Deklaracija pointera. */
float square(float x)      /* Deklaracija funkcije. */
{
return x * x;
}
```

Zato što funkcija **square()** i pointer **p** imaju iste parametre i povratnu vrijdenost, vi možete inicijalizirati **p** da pokazuje na **square**, kako slijedi:

```
p = square;
```

Onda možete pozvati funkciju, koristeći pointer, kako slijedi:

```
answer = p(x);
```

Tako je jednostavno.

Za pravi primjer, kompajlirajte i pokrenite Listing **15.8**, koji deklariše i inicijalizira pointer na funkciju i onda poziva funkciju dva puta, koristeći ime funkcije prvi put i pointer drugi put. Obadva poziva proizvode isti rezultat.

Listing 15.8. Korištenje pointera na funkciju za poziv funkcije.

```
1:  /* Demonstracija deklaracije i upotrebe pointera na funkciju.*/
2:
3:  #include <stdio.h>
4:
5:  /* Prototip funkcije. */
6:
7:  double square(double x);
8:
9:  /* Deklaracija pointera. */
10:
11: double (*p)(double x);
12:
13: main()
14: {
15:     /* Inicijalizacija p da pokazuje na square(). */
16:
17:     p = square;
18:
19:     /* Poziv  square()-a dva puta. */
20:     printf("%f  %f\n", square(6.6), p(6.6));
21:     return(0);
22: }
23:
24: double square(double x)
25: {
26:     return x * x;
27: }
            43.559999  43.559999
```

NAPOMENA: Preciznost vrijednost može prouzrokovati da se neki brojevi ne prikažu kao tačne vrijednosti koje su unešene. Na primjer, tačan odgovor, **43.56**, može izgledati kao **43.559999**.

Linija **7** deklariše funkciju **square()**, i linija **11** deklariše pointer **p** na funkciju koja sadrži **double** argument i povratnu **double** vrijednost, koji odgovara deklaraciji **square()**-a.
Linija **17** postavlja pointer **p** jednak sa **square**. Primjetite da zagrade nisu korištene sa **square** ili **p**.
Linija **20** printa povratne vrijednosti od poziva na **square()** i **p()**.

Ime funkcije bez zagrada je pointer na funkciju (zvuči slučno kao i situacija sa nizovima, zar ne? ha?).

→ Koja je korist od deklaracije i korištenja odvojenog pointera na funkciju???
Paa, ime funkcije je samo po sebi pointer konstanta i ne može biti promjenjena (ponovo, paralelno sa nizovima). Pointer varijabla, u kontrastu, MOŽE biti promjenjena. Naročito, može biti napravljena da pokazuje na različite funkcije, ako se potreba poveća.

Listing **15.9** poziva funkciju, prosljeđujući **integer** argument. Zavisno ad vrijednosti argumenta, funkcija inicijalizira pointer da pokazuje na jednu od tri ostale funkcije i onda koristi pointer da pozove odgovarajuću funkciju. Svaka od ove tri funkcije prikazuje posebnu poruku na-ekranu.

Listing 15.9. Korištenje pointera na funkciju za pozivanje različitih funkcija zavisno od programskih okolnosti.

```
1:  /* Koristenje pointera za poziv razlicitih funkcija. */
2:
3:  #include <stdio.h>
4:
5:  /* Prototip funkcije. */
```

```
6:
7:  void func1(int x);
8:  void one(void);
9:  void two(void);
10: void other(void);
11:
12: main()
13: {
14:      int a;
15:
16:     for (;;)
17:     {
18:         puts("\nEnter an integer between 1 and 10, 0 to exit: ");
19:          scanf("%d", &a);
20:
21:          if (a == 0)
22:             break;
23:         func1(a);
24:     }
25:     return(0);
26: }
27:
28: void func1(int x)
29: {
30:     /* Pointer od funkcije. */
31:
32:     void (*ptr)(void);
33:
34:     if (x == 1)
35:         ptr = one;
36:     else if (x == 2)
37:         ptr = two;
38:     else
39:         ptr = other;
40:
41:     ptr();
42: }
43:
44: void one(void)
45: {
46:     puts("You entered 1.");
47: }
48:
49: void two(void)
50: {
51:     puts("You entered 2.");
52: }
53:
54: void other(void)
55: {
56:     puts("You entered something other than 1 or 2.");
57: }
        Enter an integer between 1 and 10, 0 to exit:
        2
        You entered 2.
        Enter an integer between 1 and 10, 0 to exit:
        9
        You entered something other than 1 or 2.
        Enter an integer between 1 and 10, 0 to exit:
        0
```

ANALIZA: Ovaj program zapošljava beskonačnu petlju u liniji **16** da nastavi izvršenje dok se ne unese vrijednost **0**. Kada se unese vrijednost nenula, ona se prosljeđuje do **func1()**. Primjetite da linija **32**, u **func1()**, sadrži deklaraciju za pointer **ptr** na funkciju. Pošto je deklarisan unutar funkcije, to čini **ptr** lokalnim za **func1()**, što je prigodno jer nijedan drugi dio programa ne treba da joj pristupa.
func() onda koristi ovu vrijednost da postavi **ptr** jednak sa prigodnom funkcijom (linije **34** do **39**).
Linija **41** onda čini jedan poziv na **ptr()**, koji poziva prigodnu funkciju.

Naravno, ovaj program je samo za svrhe ilustracije. Lakše ste mogli obaviti iste rezultate koristeći pointer na funkciju.

Sad možete naučiti drugi način kako da koristite pointere da pozivate različite funkcije: prosljeđujući pointer kao argument do funkcije.

Listing **15.10** je revizija Listinga **15.9**.

Listing 15.10. Prosljeđivanje pointera funkciji kao argument.

```
1:  /* Prosljedjivanje pointera funkciji kao argument. */
2:
3:  #include <stdio.h>
4:
5:  /* Prototipi funkcija. Funkcija func1() uzima svoj */
6:  /* argument pointer na funkciju koje ne uzima  */
7:  /* argumente i nema povratnu vrijednost. */
8:
9:  void func1(void (*p)(void));
10: void one(void);
11: void two(void);
12: void other(void);
13:
14: main()
15: {
16:     /* Pointer na funkciju. */
18:     void (*ptr)(void);
19:     int  a;
20:
21:     for (;;)
22:     {
23:         puts("\nEnter an integer between 1 and 10, 0 to exit: ");
24:         scanf("%d", &a);
25:
26:         if (a == 0)
27:             break;
28:         else if (a == 1)
29:             ptr = one;
30:         else if (a == 2)
31:             ptr = two;
32:         else
33:             ptr = other;
34:         func1(ptr);
35:     }
36:     return(0);
37: }
38:
39: void func1(void (*p)(void))
40: {
41:     p();
42: }
43:
44: void one(void)
45: {
```

```
46:     puts("You entered 1.");
47: }
48:
49: void two(void)
50: {
51:     puts("You entered 2.");
52: }
53:
54: void other(void)
55: {
56:     puts("You entered something other than 1 or 2.");
57: }
        Enter an integer between 1 and 10, 0 to exit:
        2
        You entered 2.
        Enter an integer between 1 and 10, 0 to exit:
        11
        You entered something other than 1 or 2.
        Enter an integer between 1 and 10, 0 to exit:
        0
```

ANALIZA: Primjetite razliku između Listinga **15.9** i Listinga **15.10**. deklaracija pointera na funkciju je premještena u liniju **18** u **main()**-u, gdje je potrebna.
Kood u **main()**-u sad inicijalizira pointer da pokazuje na tačnu funkciju, zavisno od vrijednosti koje korisnik unese (linije **26** do **33**), i onda prosljeđuje inicijalizirani pointer ka **func1()**.
func1() ne služi pravoj svrsi u Listingu **15.10**; sve što radi je poziv funkciji na koju pokazuje **ptr**.
Ponovo, ovaj je program ilustracija samo. Isti se princip može koristiti u programima stvarnog-svijeta, kao što je primjer u sljedećem dijelu.

Jedna programerska situcaija kada možete koristiti pointere na funkcije, je kada je potrebno **sortiranje**. Ponekad vi želite da koristite različita pravila za sortiranje.
Na primjer, možda želite da jednom sortirate u abecednom redosljedu, a drugi put u suprotnom smjeru abecednog redosljeda. Koristeći pointere na funkcije, vaš program može pozvati pravu sortirajuću funkciju. Preciznije, uobičajeno se poziva različita upoređujuća funkcija.

Pogledajte ponovo Listing **15.7**. U **sort()** funkciji, stvarni raspored sortiranja je odlučen sa vrijednošću koja je vraćena od **strcmp()** bibliotečne funkcije, koja govori programu da li je dati string “manji od” ili “veći od” drugog stringa.
Šta ako biste napisali dvije upoređujuće funkcije → jedna koja sortira abecedno (gdje je **A** manje od **Z**), i druga koja sortira u suprotnom abecednom smjeru (gdje je **Z** manje od **A**)?

Program može pitati korisnika koji redosljed želi i, koristeći pointere, sortirajuća funkcija može pozvati ispravnu upoređujuću funkciju.

Listing **15.11** modifikuje Listing **15.7** da ubaci ovaj dodatak (incorporate this feature):
Listing 15.11. Upotreba pointera na funkcije za kontrolu sortirajućeg redosljeda.

```
1:  /* Unesi listu stringova sa tastature, sortirajuci ih */
2:  /* u rastucem i opadajucem redosljedu, i onda ih prikazi */
3:  /* na - ekran. */
4:  #include <stdlib.h>
5:  #include <stdio.h>
6:  #include <string.h>
7:
8:  #define MAXLINES 25
9:
10:  int get_lines(char *lines[]);
11:  void sort(char *p[], int n, int sort_type);
12:  void print_strings(char *p[], int n);
13:  int alpha(char *p1, char *p2);
```

```
14:  int reverse(char *p1, char *p2);
15:
16:  char *lines[MAXLINES];
17:
18:  main()
19:  {
20:     int number_of_lines, sort_type;
21:
22:     /* Ucitaj linije sa tastature. */
23:
24:     number_of_lines = get_lines(lines);
25:
26:     if ( number_of_lines < 0 )
27:     {
28:        puts("Memory allocation error");
29:        exit(-1);
30:     }
31:
32:     puts("Enter 0 for reverse order sort, 1 for alphabetical:" );
33:     scanf("%d", &sort_type);
34:
35:     sort(lines, number_of_lines, sort_type);
36:     print_strings(lines, number_of_lines);
37:     return(0);
38:  }
39:
40:  int get_lines(char *lines[])
41:  {
42:      int n = 0;
43:      char buffer[80];  /* Temporary storage for each line. */
44:
45:      puts("Enter one line at time; enter a blank when done.");
46:
47:      while (n < MAXLINES && gets(buffer) != 0 && buffer[0] != `\0')
48:      {
49:          if ((lines[n] = (char *)malloc(strlen(buffer)+1)) == NULL)
50:          return -1;
51:          strcpy( lines[n++], buffer );
52:      }
53:      return n;
54:
55:  } /* Kraj od get_lines() */
56:
57:  void sort(char *p[], int n, int sort_type)
58:  {
59:      int a, b;
60:      char *x;
61:
62:      /* Pointer na funkciju.  */
63:
64:      int (*compare)(char *s1, char *s2);
65:
66:      /* Inicijaliziraj pointer da pokazuje na pravilnu uporedjujucu */
67:      /* funkciju zavisno od argumenta  sort_type. */
68:
69:      compare = (sort_type) ? reverse : alpha;
70:
71:      for (a = 1; a < n; a++)
72:      {
73:          for (b = 0; b < n-1; b++)
74:          {
```

```
75:              if (compare(p[b], p[b+1]) > 0)
76:              {
77:                  x = p[b];
78:                  p[b] = p[b+1];
79:                  p[b+1] = x;
80:              }
81:          }
82:      }
83:  }  /* kraj od sort() */
84:
85:  void print_strings(char *p[], int n)
86:  {
87:      int count;
88:
89:      for (count = 0; count < n; count++)
90:          printf("%s\n ", p[count]);
91:  }
92:
93:  int alpha(char *p1, char *p2)
94:  /* Alphabetical (abecedno) poredjenje. */
95:  {
96:      return(strcmp(p2, p1));
97:  }
98:
99:  int reverse(char *p1, char *p2)
100: /* Obratno abecedno (alphabetical) poredjenje. */
101: {
102:      return(strcmp(p1, p2));
103: }
       Enter one line at time; enter a blank when done.
       Roses are red
       Violets are blue
       C has been around,
       But it is new to you!
       Enter 0 for reverse order sort, 1 for alphabetical:
       0
       Violets are blue
       Roses are red
       C has been around,
       But it is new to you!
```

ANALIZA: Linije **32** i **33** u **main()**-u promptaju korisnika za željeni sortirajući redosljed. Vrijednost unešena se smješta u **sort_type**. Vrijednost se prosljeđuje **sort()** funkciji zajedno sa ostalim informacijama opisanim za Listing **15.7**.
sort() funkcija sadrži par izmjena. Linija **64** deklariše pointer na funkciju nazvanu **compare()**, jednu od dvije nove funkcije dodate u listing na osnovu vrijednosti od **sort_type**.
Dvije nove funkcije su **alpha()** i **reverse()**.
alpha() koristi **strcmp()** bibliotečnu funkciju baš kako je korištena i u Listingu **15.7**;
reverse() ne. **reverse()** prebacuje (switches) prosljeđene parametre tako da se uradi obrnuti-redosljed sort.

KORISTITE strukturno programiranje
NE zaboravite da koristite zagrade kada deklarišete pointere na funkcije. Evo kako deklarišete pointer na funkciju koja ne uzima argumente i vraća karakter:

```
char (*func)();
```

Evo kako deklarišete funkciju koja vraća pointer na karakter:

```
char *func();
```

INICIJALIZIRAJTE pointer prije nego što ga koristite.
NE koristite funkcijski pointer koji je bio deklarisan sa različitim povratnim tipom ili drugačijim argumentom od onog koji vam je potreban.

→→→→→ Uvezane liste (uvezane (Linked Lists) liste)

Uvezana lista je moćna metoda za smještanje podataka koja može lako biti implementirana u **C**-u. Zašto pokrivamo uvezane liste u ovom poglavlju o pointerima??? Zato što, kako ćete ubrzo vidjeti, pointeri su centralni u uvezanim listama.

Postoji nekoliko vrsta uvezanih liista, uključujući jednostruko-uvezane liste, dvostruko-uvezane liste, i binarna drva.

Svaki tip je pogodan za određeni tip smještajnih podataka. Jedna stvar koju ove liste imaju kao zajedničku je da su veze između predmeta podataka definisane sa informacijama koje su sadržane u samim predmetima (items), u formi pointera.
Ovo je različito od nizova, u kojima su veze između predmeta podataka rezultiraju od izgleda (layout) i smještanja (storage) niza. Ovaj dio poglavlja objašnjava najfundamentalnije vrste uvezanih lista; jednostruko-uvezane-liste (kojima ćemo se obraćati jednostavno kao **uvezana lista**).

→→→→ Osnovi Uvezane Liste

Svaki podatkovni predmet u uvezanoj listi je sadržan u strukturi.
Struktura sadrži podatke elemente koji su potrebni da čuvaju podatke koji se smještaju; ovo zavisi od potreba specifičnog progama. S dodatkom, postoji još jedan podatkovni element → **pointer**. Ovaj pointer obezbjeđuje vezu u uvezanoj listi (povezanoj (ulančanoj (linked)) listi). Evo jednostavan primjerćić:

```
struct person {
char name[20];
struct person *next;
};
```

Ovaj kood definiše strukturu nazvanu **person**. Za podatke, **person** sadrži samo **10**-elementni niz karaktera. Uopšte, vi ne biste koristili uvezanu listu za tako jednostavne podatke, ali ovo će poslužiti za primjer. Struktura **person** takođe sadrži pointer na tip **person** → drugim riječima, pointer na drugu strukturu istog tipa. Ovo znači da svaka struktura od tipa **person** ne samo da može sadržavati dio (chunk) podatka, već može i pokazivati na drugu strukturu **person**-a.

Posljednji element u uvezanoj listi je indentifikovan kao pointer element kojem je pridružena vrijednost **NULL**.

NAPOMENA: Strukturama koje ostvaruju vezu (link) sa uvezanom listom se može obraćati sa linkovi (veze), čvorovi (nodes), ili elementi uvezane liste.

Vidjeli ste kako se identifikuje posljednji link u uvezanoj listi. **Šta je sa prvim linkom?**
On je definisan pomoću specijanog pointera (ne strukture) nazvan **head pointer**. **Head pointer** uvijek pokazuje na prvi element u uvezanoj listi. Prvi element sadrži pointer na drugi element, drugi element sadrži pointer na treći, itd., sve dok ne naiđe element čiji je pointer **NULL**.
Ako je cijela lista prazna (ne sadži veze (linkove)), **head pointer** (pointer glava) se postavlja na **NULL**.

NAPOMENA: head pointer je pointer na prvi element u uvezanoj listi. **Head pointer**-u se ponekad obraća sa prvi element pointer ili top pointer (gornji pointer).

→→ Korištenje uvezanih lista

Kada radite sa uvezanim listama, vi možete **dodavati**, **brisati**, ili **modifikovati** elemente ili veze (linkove).
Modifikovanje elemenata ne predstavlja neki veliki izazov; ipak, dodavanje i brisanje elemenata može.

Elementi u listi su povezani pointerima. Većina posla dodavanja i brisanja elemenata se sastoji u manipulaciji ovih pointera.

Elementi se mogu dodavati na **početku**, **sredini**, ili **na kraju** uvezane liste; ovo odlučuje kako se moraju mijenjati pointeri.

Kasnije u ovom poglavlju ćete naći jednostavne demonstracije uvezane liste, takođe sa više komplexnim radnim programom. Prije nego što počnemo sa detaljćićima, ipak, dobra je ideja da ispitamo neke od akcija koje moramo da radimo sa uvezanim listama. Za ovu svrhu, nastavićemo da koristimo strukturu **person**, koja je predstavljena ranije.

→ Uvod

Prije nego što možete početi uvezanu listu, treba da definišete strukturu podataka koji će biti korišteni u listi, i takođe treba da deklarišete **head pointer**.
Pošto lista počinje prazna, **head pointer** bi trebao biti inicijaliziran na **NULL**. Takođe ćete trebati dodatni pointer na tip vaše liste struktura, za dodavanje slogova (možda će vam biti potrebno više od jednog pointera, kako ćete vidjeti ubrzo).

Evo kako ovo radite:

```
struct person {
  char name[20];
  struct person *next;
};
struct person *new;
struct person *head;
head = NULL;
```

→→ Dodavanje elementa na početak liste

Ako je **head pointer NULL**, lista je prazna, i novi element će biti njen jedini član. Ako **head pointer** nije **NULL**, lista već sadrži jedan ili više elemenata. U svakom slučaju, ipak, procedura za dodavanje novog elementa na početaj liste je ista:

1. Kreirajte instancu vaše strukture, alocirajući memorijski prostor koristeći **malloc()**.
2. Podesite sljedeći pointer novog elementa na trenutnu vrijednost od **head pointera**. Ovo će biti **NULL** ako je lista prazna, ili adresa od trenutno prvog elementa u drugom slučaju.
3. Napravite da **head pointer** pokazuje na novi element.

Evo i kood, kako da uradite ovaj zadatak:

```
new = (person*)malloc(sizeof(struct person));
new->next = head;
head = new
```

UPOZORENJE: Vrlo je važno da prebacujete (switch) pointere u ispravnom redosljedu.
Ako prepridružite (reassign) head pointer prvo, izgubićete listu!

malloc() se koristi da alocira memoriju za novi element. Kako se novi element dodaje, samo se memorija koja je potrebna za to alocira.
→→→ **calloc()** funkcija se takođe može koristiti. Trebate biti na oprezu sa razlikama između ove dvije funkcije. Glavna razlika je da će **calloc()** inicijalizirati novi element; dok **malloc()** neće.

UPOZORENJE: malloc() u prethodnom dijelu kooda nije osigurao da je memorija alocirana. Treba uvijek da provjerite povratnu vrijednost od memorijsko alokacijske funkcije.

SAVJET: Kad je moguće, inicijalizirajte pointere na **NULL** kada ih deklarišete. **Nikad ne ostavljajte pointer neinicijaliziran.**

→→ Dodavanje elementa na kraj liste

Da dodate element na kraj uvezane liste, treba da počnete kod **head pointera** i idete kroz listu, dok ne pronađete posljednji element. Nakon što ga pronađete, slijedite ove korake:

1. Kreirajte instancu od vaše strukture, alocirajući memorijski prostor koristeći **malloc()**.
2. Podesite sljedeći pointer u posljednjem elementu da pokazuje na novi element (čija je adresa vraćena od **malloc()**).
3. Podesite sljedeći pointer u novom elementu na **NULL** da signalizira da je on posljednji predmet u listi.

Evo kooda:

```
person *current;
...
current = head;
while (current->next != NULL)
    current = current->next;
new = (person*)malloc(sizeof(struct person));
current->next = new;
new->next = NULL;
```

→→ Dodavanje elementa u sredinu liste

Kada radite sa uvezanim listama, većinu vremena ćete dodavati elemente negdje u sredinu liste. Tačno, gdje je smješten novi element, zavisi od toga kako održavate vašu listu → na primjer, da li je sortirana na jedan ili više podatkovnih elemenata. Ovaj proces, onda, zahtjeva da vi prvo alocirate poziciju u listi gdje će doći novi element, i onda ga dodate.
Evo koraka koje trebate pratiti:

1. U listi, locirajte (nađite) postojeći element, nakon kojeg će se smjestiti novi element. Nazovimo ga element **marker** (postojeći element).
2. Kreirajte instancu od vaše strukture, alocirajući memorijski prostor koristeći **malloc()**.
3. Podesite sljedeći pointer, od elementa **marker**, da pokazuje na novi element (čija je adresa vraćena od **malloc()**).
4. Podesite sljedeći pointer, od novog elementa, da pokazuje na element na koji je prije pokazivao element **marker**.

Evo kako bi kood mogao da izgleda:

```
person *marker;
/* Code here to set marker to point to the desired list location.
*/
...
new = (LINK)malloc(sizeof(PERSON));
new->next = marker->next;
marker->next = new;
```

→→ Brisanje elementa iz liste

Brisanje elementa iz liste je jednostavna stvar manipulacije pointera. Tačan proces zavisi od toga gdje je u listi smješten element:

- **Da izbrišete prvi element**, podesite **head pointer** da pokazuje na drugi element u listi.
- **Da izbrišete posljednji element**, podesite sljedeći (next) pointer od sljedećeg-do-posljednjeg (next-to-last) elementa, na **NULL**.
- **Da izbrišete bilo koji drugi element**, podesite sljedeći pointer od elementa, prije onoga kojeg hoćete da izbrišete, da pokazuje na element poslije onoga koji se briše.

Evo kooda, za **brisanje prvog elementa** u uvezanoj listi:

```
head = head->next;
```

Ovaj kood **briše posljednji element** u listi:

```
person *current1, *current2;
current1 = head;
current2= current1->next;
while (current2->next != NULL)
{
    current1 = current2;
    current2= current1->next;
}
current1->next = null;
if (head == current1)
    head = null;
```

Konačno, sljedeći kood **briše element iz unutrašnjosti** liste:

```
person *current1, *current2;
/* Code goes here to have current1 point to the */
/* element just before the one to be deleted. */
current2 = current1->next;
current1->next = current2->next;
```

Poslijde svake od ovih procedura, izbrisani element još uvijek postoji u memoriji, ali je uklonjen iz liste zato što nema pointera u listi da pokazuje na njega.

U stvarnim programima, vi želite da povratite memoriju koja je zauzeta sa “izbrisanim” elementom. Ovo se obavlja sa →→→ **free()** ←←← funkcijom (Dana 20).

→→ Demonstracija jednostavno uvezane liste

Listing **15.12** demonstrira osnove upotrebe uvezane liste. Ovaj program je samo za svrhe demonstracije, zato što ne prihvata unos od korisnika i ne radi ništa korisno do toga što prikazuje kood za najjednostavnije poslove uvezane liste.

Ovaj program radi sljedeće:

1. On definiše strukturu i potrebne pointere za listu.
2. On dodaje prvi element na listu.
3. On dodaje element na kraj liste.
4. On dodaje element u sredinu liste.
5. On prikazuje sadržaj liste na-ekran.

Listing 15.12. Osnove uvezane liste.

```
1:  /* Demonstrira osnove koristenja */
2:  /* uvezane liste. */
3:
4:  #include <stdlib.h>
5:  #include <stdio.h>
6:  #include <string.h>
7:
8:  /* The list data structure. */
9:  struct data {
10:     char name[20];
11:     struct data *next;
12:     };
13:
14: /* Define typedefs for the structure */
15: /* and a pointer to it. */
16: typedef struct data PERSON;
17: typedef PERSON *LINK;
18:
19: main()
20: {
21: /* Head, new, and current element pointers. */
22: LINK head = NULL;
23: LINK new = NULL;
24: LINK current = NULL;
25:
26: /* Dodaj prvi element liste. Mi ne pretpostavljamo */
27: /* da je lista prazna, iako ce u ovom */
28: /* demo programu uvijek biti. */
29:
30: new = (LINK)malloc(sizeof(PERSON));
31: new->next = head;
32: head = new;
33: strcpy(new->name, "Abigail");
34:
35: /* Dodaj element na kraj liste. */
36: /* Pretpostavljamo da lista sadrzi bar jedan element. */
37:
38: current = head;
39: while (current->next != NULL)
40: {
41:     current = current->next;
42: }
43:
44: new = (LINK)malloc(sizeof(PERSON));
```

```
45: current->next = new;
46: new->next = NULL;
47: strcpy(new->name, "Catherine");
48:
49: /* Dodaj drugi element na drugu poziciju u listi. */
50: new = (LINK)malloc(sizeof(PERSON));
51: new->next = head->next;
52: head->next = new;
53: strcpy(new->name, "Beatrice");
54:
55: /* Printaj sve predmete podataka po redosljedu (data items). */
56: current = head;
57: while (current != NULL)
58: {
59:     printf("\n%s", current->name);
60:     current = current->next;
61: }
62:
63: printf("\n");
64: return(0);
65: }
Abigail
Beatrice
Catherine
```

ANALIZA: Vjerovatno možete sami shvatiti dio kooda.
Linije **9** do **12** deklarišu strukturu **data** za listu.
Linije **16** i **17** definišu **typedef**, za obadvije data strukture i za pointer na strukturu **data**. Precizno govoreći, ovo nije neophodno, ali pojednostavljuje koodiranje tako što vam dozvoljava da napišete **PERSON** na mjesto strukture **struct data** i **LINK** na mjesto strukture **struct data ***.

Linije **22** do **24** deklarišu **head pointer** i nekoliko drugih pointera koji će biti korišteni kada se bude manipulisala lista. Svi ovi pointeri su inicijalizirani na **NULL**.

Linije **30** do **33** dodaju novi link na početak liste. Linija **30** alocira novu strukturu **data**. Primjetite da se uspješna operacija **malloc()**-a pretpostavlja → nešto što nikad ne biste radili u pravom programu!

Linija **31** postavlja (sets) sljedeći pointer u ovoj novoj strukturi da pokazuje na ono šta **head pointer** sadrži. Zašto jednostavno ne pridružiti **NULL** ovom pointeru? To radi samo ako znate da je lista prazna. Kao što je napisano, kood će raditi, čak i ako lista već sadrži neke elemente. Novi prvi element će završiti pokazujući na element koji je nekad bio prvi, što je ono što ste želili.

Linija **32** čini da **head pointer** pokazuje na novi slog (new record), i linija **33** smješta nešto podataka u slog.

Dodavanje elementa na kraj liste je nešto komplikovanije. Iako u ovom slučaju vi znate da lista sadrži samo jedan element. Ovo ne možete pretpostaviti u pravom (realnom) programu. Tako da, je neophodno da se peetlja kroz listu, s početkom sa prvim elementom, dok ne pronađete posljednji element (što je indicirano tako što sljedeći pointer pokazuje na **NULL**). Tada znate da ste našli kraj liste. Ovaj posao se obavlja u linijama **38** do **42**. Jednom kada ste našli posljednji element, jednostavna je stvar da alocirate novu podatkovnu strukturu (**data** structure), da stari posljednji element pokazuje na nju, i da podesite sljedeći pointer novog elementa na **NULL**, jer je sad on posljednji element u listi. Ovo je urađeno u linijama **44** do **47**.

Sljedeći zadatak je da se doda novi element u sredinu liste → u ovom slučaju, na drugu poziciju. Nakon što se alocira nova **data** struktura (linija **50**), sljedeći pointer od novog elementa se podesi da pokazuje na element, koji je bio drugi i sad je treći u listi (linija **51**), i sljedeći pointer prvog elementa se podesi da pokazuje na novi element (linija **52**).

Konačno, program printa sve slogove u uvezanoj listi. Ovo je jednostavna stvar počinjanja sa elementom na koji pokazuje **head pointer** i onda napreduje kroz listu dok se ne pronađe posljednji element, koji je indiciran kao **NULL** pointer. Linije **56** do **61** obavljaju ovaj zadatak.

→→→ Implementacija uvezane liste

Sad kad ste vidjeli načine kako se dodaju linkovi (veze) u listi, vrijeme je da ih vidimo u akciji. Listing **15.13** je relativno dug program koji koristi uvezanu listu da smjesti listu od pet karaktera. Karakteri se smještaju u memoriju koristeći uvezanu listu. Ovi karakteri lako mogu biti imena, adrese, ili neki drugi podaci. Da bi primjer bio što jednostavniji, u svaki link se smješta samo jedan karakter.

Ovaj program uvezane liste čini komplikovanim to da on sortira linkove u trenutku kad se oni dodaju. Naravno, ovo je ono što čini ovaj program vrijednim. Svaki link se dodaje na početak, sredinu, ili na kraj, zavisno od njegove vrijednosti. Link je uvijek sortiran. Ako želite da napišete program koji jednostavno dodaje linkove na kraj, logika bi bila puno jednostavnija. Ipak, program bi bio manje upotrebljiv.

Listing 15.13. Implementacija uvezane liste karaktera.

```
1:  /*=======================================================*
2:   * Program:  list1513.c                                  *
3:   * Knjiga:     Teach Yourself C in 21 Days                 *
4:   * Svrha:  Implementacija uvezane liste                *
5:   *=======================================================*/
6:  #include <stdio.h>
7:  #include <stdlib.h>
8:
9:  #ifndef NULL
10: #define NULL 0
11: #endif
12:
13: /* List podatkovna struktura */
14: struct list
15: {
16:   int    ch;     /* koristi se int da drzi char */
17:   struct list *next_rec;
18: };
19:
20: /* Typedef-ovi za strukturu i pointer. */
21: typedef struct list LIST;
22: typedef LIST *LISTPTR;
23:
24: /* Prototipi funkcije. */
25: LISTPTR add_to_list( int, LISTPTR );
26: void show_list(LISTPTR);
27: void free_memory_list(LISTPTR);
28:
29: int main( void )
30: {
31:    LISTPTR first = NULL;  /* head pointer */
32:    int i = 0;
33:    int ch;
34:    char trash[256];       /* za ciscenje  stdin buffer-a. */
35:
36:    while ( i++ < 5 )     /* sagradi listu na osnovu 5 datih predmeta */
37:    {
38:       ch = 0;
39:       printf("\nEnter character %d, ", i);
40:
```

```
41:       do
42:       {
43:           printf("\nMust be a to z: ");
44:           ch = getc(stdin);  /* uzmi sljedeci karakteru u buffer  */
45:           gets(trash);       /* ukloni smece iz buffer-a */
46:       } while( (ch < `a' || ch > `z') && (ch < `A' || ch > `Z'));
47:
48:       first = add_to_list( ch, first );
49:    }
50:
51:    show_list( first );           /* Rijesava se (Dumps) cijele liste */
52:    free_memory_list( first );    /* Oslobodi svu memoriju  */
53:    return(0);
54: }
55:
56: /*=======================================================*
57:  * Funkcija: add_to_list()
58:  * Svrha   : Insertuje novi link u listu
59:  * Unos    : int ch = character za smjeshtanje
60:  *           LISTPTR first = address od oreginalnog head pointera
61:  * Vratcha : Adresu od head pointera (prvo)
62:  *=======================================================*/
63:
64: LISTPTR add_to_list( int ch, LISTPTR first )
65: {
66:    LISTPTR new_rec = NULL;       /* Drzi adresu od novog rec (sloga) */
67:    LISTPTR tmp_rec = NULL;       /* Drzi tmp (privremeni) pointer    */
68:    LISTPTR prev_rec = NULL;
69:
70:    /* Alociranje memorije. */
71:    new_rec = (LISTPTR)malloc(sizeof(LIST));
72:    if (!new_rec)      /* Nije u mogucnosti da alocira memoriju */
73:    {
74:       printf("\nUnable to allocate memory!\n");
75:       exit(1);
76:    }
77:
78:    /* Podesi novi linkov podatak (set new link's data) */
79:    new_rec->ch = ch;
80:    new_rec->next_rec = NULL;
81:
82:    if (first == NULL)   /* dodavanje prvog linka na listu */
83:    {
84:        first = new_rec;
85:        new_rec->next_rec = NULL;  /* suvishan ali siguran */
86:    }
87:    else    /* ne prvi slog (record) */
88:    {
89:       /* pogledaj da li ide prije prvog linka */
90:       if ( new_rec->ch < first->ch)
91:       {
92:          new_rec->next_rec = first;
93:          first = new_rec;
94:       }
95:       else   /* dodaje se u sredinu ili kraj */
96:       {
97:          tmp_rec = first->next_rec;
98:          prev_rec = first;
99:
100:          /* Provjera da vidi gdje je dodat link. */
101:
```

```
102:          if ( tmp_rec == NULL )
103:          {
104:              /* dodajemo drugi slog (record) na kraj */
105:              prev_rec->next_rec = new_rec;
106:          }
107:          else
108:          {
109:             /* provjera da se vidi da li se dodaje u sredinu */
110:             while (( tmp_rec->next_rec != NULL))
111:             {
112:                if( new_rec->ch < tmp_rec->ch )
113:                {
114:                   new_rec->next_rec = tmp_rec;
115:                   if (new_rec->next_rec != prev_rec->next_rec)
116:                   {
117:                      printf("ERROR");
118:                      getc(stdin);
119:                      exit(0);
120:                   }
121:                   prev_rec->next_rec = new_rec;
122:                   break;   /* link is added; exit while */
123:                }
124:                else
125:                {
126:                  tmp_rec = tmp_rec->next_rec;
127:                  prev_rec = prev_rec->next_rec;
128:                }
129:             }
130:
131:             /* provjera da se vidi da li se dodaje na kraj */
132:             if (tmp_rec->next_rec == NULL)
133:             {
134:                if (new_rec->ch < tmp_rec->ch ) /* 1 b4 end */
135:                {
136:                   new_rec->next_rec = tmp_rec;
137:                   prev_rec->next_rec = new_rec;
138:                }
139:                else  /* at the end */
140:                {
141:                   tmp_rec->next_rec = new_rec;
142:                   new_rec->next_rec = NULL;  /* suvisan */
143:                }
144:             }
145:          }
146:       }
147:    }
148:    return(first);
149: }
150:
151: /*=======================================================*
152:  * Funkcija : show_list
153:  * Svrha    : Prikazuje informacije koje se trenutno nalaze u listi
154:  *=======================================================*/
155:
156: void show_list( LISTPTR first )
157: {
158:    LISTPTR cur_ptr;
159:    int counter = 1;
160:
161:    printf("\n\nRec addr  Position  Data  Next Rec addr\n");
162:    printf("========  ========  ====  =============\n");
```

```
163:
164:    cur_ptr = first;
165:    while (cur_ptr != NULL )
166:    {
167:       printf("  %X   ", cur_ptr );
168:       printf("     %2i       %c", counter++, cur_ptr->ch);
169:       printf("      %X   \n",cur_ptr->next_rec);
170:       cur_ptr = cur_ptr->next_rec;
171:    }
172: }
173:
174: /*=========================================================*
175:  * Funkcija : free_memory_list
176:  * Svrha    : Oslobadja svu memoriju skupljenu za listu
177:  *=========================================================*/
178:
179: void free_memory_list(LISTPTR first)
180: {
181:    LISTPTR cur_ptr, next_rec;
182:    cur_ptr = first;                /* Pochni na pochetku  */
183:
184:    while (cur_ptr != NULL)       /* Idi dok (while) nije kraj liste */
185:    {
186:       next_rec = cur_ptr->next_rec; /* Uzmi adresu novog sloga */
187:       free(cur_ptr);                /* Oslobodi trenutni slog  */
188:       cur_ptr = next_rec;           /* Prilagodi trenutni slog */
189:    }
190: }
```

```
Enter character 1,
Must be a to z: q
Enter character 2,
Must be a to z: b
Enter character 3,
Must be a to z: z
Enter character 4,
Must be a to z: c
Enter character 5,
Must be a to z: a
Rec addr  Position  Data  Next Rec addr
========  ========  ====  =============
C3A          1       a      C22
C22          2       b      C32
C32          3       c      C1A
C1A          4       q      C2A
C2A          5       z      0
```

NAPOMENA: Vaš izlaz će vjerovatno pokazivati drugačije vrijednost adresa.

ANALIZA: Ovaj program demonstrira dodavanje veze na uvezanu listu. Nije najlakši način za razumijevanje; ipak, ako prođete kroz njega, vidjet ćete da je on kombinacija tri metode dodavanja veze (linka) o kojima smo pričali ranije. Ovaj listing može biti korišten za dodavanje linka na početak, sredinu, ili na kraj uvezane liste. S dodatkom, ovaj listing uzima u obzir specijalne slučajeve o dodavanju prvog linka (onaj koji se dodaje na početak) i drugog linka (onaj koji se dodaje u sredinu).

SAVJET: Najlakši način da u potpunosti shvatite ovaj listing je da prođete kroz njega liniju-po-liniju u vašem debageru kompajlera i da čitate sljedeće analize. Gledajući logiku koja se izvršava, bolje ćete shvatiti listing.

Linije **9** do **11** provjeravaju da vide da li je vrijednost **NULL** već definisana. Ako nije, linija **10** definiše je da ona bude **0**.
Linije **14** do **22** definišu strukturu za uvezanu listu i takođe deklariše tip definicija, koj će rad sa strukturama i pointerima učiniti lakšim.

main() funkcija bi trebala biti lagana za praćenje. **Head pointer** koji se prvi poziva, je deklarisan u liniji **31**. Primjetite da je inicijaliziran na **NULL**. Zapamtite da nikad ne ostavljate pointer neinicijaliziranim.
Linije **36** do **49** sadrže **while** petlju koja se koristi da uzme pet karaktera od korisnika. **Unutar** ove vanjske **while** petlje, koja se ponavlja pet puta, koristi se **do…while** petlja, da osigura da je svaki karakter koji je unešen slovo. →→→ **isalpha()** funkcija bi se takođe lako mogla koristiti.

Nakon što se dobije nešto podataka, **add_to_list()** se poziva. Pointer na početak liste i podatak koji se dodaje u listu, se prosljeđuju funkciji.

main() funkcija završava sa: pozivanjem **show_list()**, da prikaže podatke liste i onda **free_memory_list()**-om da oslobodi svu memoriju koja je alocirana da drži linkove u listi.
Obadvije ove funkcije operišu na sličan način. Svaka počinje na početku od uvezane liste koristeći **head pointer** prvo. **while** petlja je korištena da ide od jednog linka do sljedećeg, koristeći vrijednost **next_ptr**-a. Kada je **next_ptr** jednak sa **NULL**, dostignut je kraj uvezane liste, i funkcija se vraća.

Najvažnije (i najkomplikovanija) funkcija u listingu je **add_to_list()** u linijama **56** do **149**.
Linije **66** do **68** deklarišu tri pointera koji će se koristiti da pokazuju na tri različita linka.
new_rec pointer će pokazivati na novi link koji treba da se doda.
tmp_rec pointer će pokazivati na trenutni link u listi koja se procjenjuje.
Ako postoji više od jednog linka u listi, **prev_rec** pointer će se korstiti da pokazuje na prethodni link koji je procjenjen.

Linija **71** alocira memoriju za novi link koji se dodaje. **new_rec** pointer je postavljen na vrijednost koja je vraćena od **malloc()**. Ako memorija ne može da se alocira, linije **74** i **75** printaju poruku greške i izlaze iz programa. Ako je memorija alocirana uspješno, program se nastavlja.

Linija **79** postavlja podatke u strukturi, na podatke prosljeđene do ove funkcije. Ovo se jednostavno sastoji iz pridruživanja karaktera koji je prosljeđen funkciji, **ch**, na polje karaktera novog sloga (**new_rec->ch**). U komplexnijim programima, ovo bi moglo biti pridruživanje ne nekoliko polja. Linija **80** postavlja **next_rec**, u novom slogu, na **NULL**, tako da ne pokazuje na neku nasumičnu lokaciju.

Linija **82** počinje “**dodaj novi link**” logiku, provjeravajući da vidi da li ima nekih linkova u listi. Ako je link koji je dodaje prvi link u listi, što je indicirano od strane prvog **head pointera** postavljenog na **NULL**, **head pointer** se jednostavno podesi (setuje) da bude jednak novom pointeru, i završili ste.

Ako ovaj link nije prvi, funkcija nastavlja unutar **else**-a u liniji **87**. Linija **90** provjerava da vidi da li novi link ide na početak liste. Kao što ste trebali zapamtiti, ovo je jedan od tri slučaja za dodavanje linka. Ako link ide prvi, linija **92** postavi **next_rec** pointer, u novom linku, da pokazuje na prethodno “prvi” link. Linija **93** onda podesi **head pointer**, prvi, da pokazuje na novi link. Ovo rezultira da se novi link dodaje na početak liste.

Ako novi link nije prvi link koji se dodaje u praznu listu, i ako je dodaje na prvu poziciju u već postojećoj listi, vi znate da on mora biti u sredini ili na kraju liste. Linije **97** i **98** postavljaju **tmp_rec** i **prev_rec** pointere koji su deklarisani ranije.
Pointer **tmp_rec** se postavlja na adresu od drugog linka u listi, i **prev_rec** se postavlja na prvi link u listi.

Primjetite da ako postoji samo jedan link u listi, **tmp_rec** će biti jednak sa **NULL**. Ovo je zato što je **tmp_rec** postavljen na prvi link, koji će biti jednak sa **NULL**. Linija **102** provjerava za ovaj specijalni slučaj. Ako je **tmp_rec NULL**, vi znate da je ovo dodavanje drugog linka u listu. Zato jer znate da novi link ne dolazi prije prvog linka, on može ići samo na kraj. Da postignete ovo, vi jednostavno postavite **prev_rec->next_ptr** na novi link, i gotovi ste.

Ako **tmp_rec** pointer nije **NULL**, vi znate da već imate više od dva linka u vašoj listi. **while** iskaz u linijama **110** do **129** peetlja kroz ostatak linkova da utvrdi gdje će se smjestiti novi link. Linija **112** provjerava da vidi da li je podatkovna vrijednost novog linka manja od linka na kojeg se trenutno (currently) pokazuje. Ako jeste, vi znate da je ovo gdje hoćete da smjestite novi link. Ako su novi podaci veći od tekućih (current) podataka linka, vi treba da pogledate na sljedeći link u listi. Linije **126** i **127** postavljaju pointere **tmp_rec** i **next_rec** na sljedeće linkove.

Ako je karakter "manji od" karaktera tekućeg linka, vi biste pratili logiku predstavljenu ranije u ovom poglavlju za dodavanje u sredinu u uvezanoj listi. Ovaj proces može se vidjeti u linijama od **114** do **122**. U liniji **114**, sljedeći pointer novog linka je postavljen na jednako sa adresom tekućeg linka (**tmp_rec**). Linija **121** postavlja prethodni linkov sljedeći pointer da pokazuje na novi link. Nakon ovoga, gotovi ste. Kood koristi **break** iskaz da izađe iz **while** petlje.

NAPOMENA: Linije **115** do **120** sadrže debagirajući kood koji je ostavljen u listingu kako bi ga vi mogli vidjeti. Ove linije mogu biti uklonjene; ipak, sve dok program radi ispravno, one nikad neće biti pozvane. Nakon što se sljedeći pointer novog linka postavi na tekući pointer (after the new link's next pointer is set to the current pointer), to bi trebalo biti jednako sa sljedećim pointerom prehodnog linka, što takođe pokazuje na tekući slog (record). Ako oni nisu jednaki, nešto nije u redu!!!

Prethodna logika se brine o linkovima koji se dodaju u sredinu liste. Ako je dostignut kraj liste, **while** petlja u linijama **110** do **129** će završiti bez dodavanja novog linka. Linije **132** i **144** se brinu o dodavanju novog linka na kraj.

Ako je dostignut posljednji link u listi, **tmp_rec->next_rec** će jednačiti sa **NULL**. Linija **132** provjerava za ovaj uslov. Linije **134** provjerava da vidi da li link ide prije ili poslije posljednjeg linka. Ako ide poslije posljednjeg linka, onda, **next_rec** od posljednjeg linka se postavlja na novi link (linija **132**), i sljedeći pointer novog linka se postavlja na **NULL** (linija **142**).

→ Poboljšanje Listinga 15.13

Uvezane liste nisu najlakša stvar za naučiti. Kako ćete vidjeti iz Listinga **15.13** ipak, da su one odličan način smještanja podataka u sortirajućem redosljedu. Zato što je lako dodati novi podatkovni predmet bilo gdje u uvezanu listu, kood za čuvanje lista podatkovnih predmeta u sortiranom redosljedu sa uvazanom listom, je puno jednostavniji nego što bi bio da ste koristili, recimo, nizove. Ovaj listing se može lako konvertovati da sortira imena, telefonske brojeve, ili bilo koje druge podatke. S dodatkom, iako je ovaj listing sortiran u rastućem redosljedu (**A** do **Z**), vrlo je lako mogao biti sortiran i u opadajućem redosljedu (**Z** do **A**).

→→ Brisanje iz uvezane liste

Sposobnost da dodajete informacije u uvezanu listu je esencijalna, ali će biti vremena kada ćete želiti da uklonite informacije. **Brisanje linkova**, ili elemenata, je slično sa njihovim dodavanjem. Možete izbrisati link sa **početka**, **sredine**, ili **kraja** uvezane liste. U svakom slučaju, prigodni se pointer mora prilagoditi. Takođe, memorija koja je korištena od izbrisanog linka, mora biti oslobođena.

NAPOMENA: Ne zaboravite da oslobodite memoriju kada brišete linkove.

NE zaboravite da oslobodite bilo koju memoriju alociranu za linkove kada ih brišete.
RAZUMITE razliku između **calloc()** i **malloc()**. Najvažnije, zapamtite da **malloc()** ne inicijalizira alociranu memoriju → **calloc()** da.

Sažetak

Ovo je poglavlje pokrilo neke od najnapredniji upotreba pointera. Kao što ste vjerovatno primjetili do sada, pointeri su centralni dio **C** jezika. **C** programi koji ne koriste pointere su rijetki. Vidjeli ste kako se koriste **pointeri na pointere** i kako niizovi pointera mogu biti veoma korisni kada radite sa stringovima. Takođe ste naučili kako **C** tretira **više-dimenzionalne nizove** kao **nizove nizova**, i vidjeli ste kako koristiti pointere sa takvim nizovima. Naučili ste kako da deklarišete i koristite **pointere na funkcije**, vrlo važan i flexibilan programerski alat. Konačno, naučili ste kako da implementirate **uvezane liste**, snažan i flexibilan metod smještanja podataka.

Ovo je bilo dugačko i zaposleno poglavlje. Iako su neke od tema bile komplikovane, bile su i uzbudljive. Sa ovim poglavljem, vi ulazite u neke od najsofisticiranijih sposobnosti **C** jezika. Snaga i flexibilnost su među mnogim razlozima zašto je **C** popularan jezik.

P&O

P Da li postoji razlika izmežu pointera na string i pointera na niz karaktera?
O Ne. String se može smatrati kao niz karaktera.

P Da li postoje drugi slučajevi kada su funkcijski pointeri korisni?
O Da. Pointeri na funkcije se takođe koriste sa meni-ima. Na osnovu vrijednosti vraćene od meni-a, pointer se postavlja na prigodnu funkciju.

P Navedi dvije glavne prednosti uvezanih lista.
O Prva: Veličina uvezane liste može biti povećana i smanjena dok se program izvršava, i ne mora da bude predefinisana kada pišete kood.
Druga: Lako je držati uvezanu listu u sortirajućem redosljedu, zato što elementi mogu lako biti dodavani i brisani bilo gdje u listi.

Vježbe:

1. Napišite deklaraciju za pointer na funkciju koja uzima **integer** kao svoj argument i vraća varijablu tipa **float**.
2. Napišite deklariaciju niza pointera na funkcije. Sve funkcije bi trebale uzimati karakterni string kao parametar i vratiti **integer**. Za šta bi se takav niz mogao koristiti?
3. Napišite iskaz da deklariše niz od **10** pointera na tip **char**.
4. BUG BUSTER: Da li nešto nije u redu sa sljedećim koodom?

```
int x[3][12];
int *ptr[12];
ptr = x;
```

5. Napišite strukturu koja će da se koristi u jednostruko-uvezanoj-listi. Ova bi struktura trebala držati imena i adrese vaših prijatelja.
6. Napišite program koji deklariše **12*12** niz od karaktera. Smjestite **X**-ove u svaki drugi element. Koristite pointer na niz da printate vrijednosti na-ekran u formatu mreže.
7. Napišite program koji koristi pointere na varijable tipa **double** da prihvata **10** brojeva od korisnika, sortira ih, i printa ih na-ekran. (Hint: Pogledaj Listing **15.10**).
8. Modifikuj program u vježbi **7** da dozvoli korisniku da navede koji sort raspored želi (opadajući ili rastući).

LEKCIJA 16: → Using Disk Files ←

Većina programa koje pišete će koristiti disk file-ove za jednu ili neku drugu svrhu: smještanje podataka, konfiguracija informacija, itd.

→→ Stream-ovi i Disk File-ovi

Kao što ste naučili Dana 14, **C** obavlja sav ulaz i izlaz (input i output), uključujući disk file-ove, sa streamovima. Vidjeli ste kako se koriste **C**-ovi predefinisani streamovi koji su spojeni na neki uređaj kao što je tastatura, ekran, i (u DOS sistemima) printer.
Disk file stream-ovi rade u suštini na isti način. Ovo je jedna od prednosti od stream ulaz/izlaz tehnika, za korištenje stream-a, da se može koristiti sa malo ili nikako izmjena, sa drugim stream-ovima.
Glavna razlika sa disk file streamovima je da vaš program mora explicitno kreirati stream sa navedenim (specifičnim) disk file-om.

→→ Tipovi Disk File-ova

Dana 14, vidjeli ste da **C**-ovi streamovi dolaze s dva ukusa: **textualni** i **binarni**.
Vi možete povezati bilo koji tip stream-a sa file-om, i važno je da razumijete razliku s ciljem da koristite ispravan mode za vaše file-ove.

→ **Textualni stream**-ovi se povezuju sa text-mode file-ovima. **Text-mode file-ovi** se sastoje od sekvence linija. Svaka linija sadrži nula ili više karaktera i završava sa jednim ili više karaktera da signaliziraju **e**nd-**o**f-**l**ine. Maximalna dužina linije je **255** karaktera. Važno je da zapamtite da "linija" nije **C** string; nema terminirajući **NULL** karakter (**\0**). Kada koristite text-mode stream, prenos se dešava između **C**-ove novalinija (**\n**) i bilo kojeg karaktera (ili više njih) koje koristi operativni sistem da označi **e**nd-**o**f-**l**ine na disk file-u. Na DOS sistemima, to je **c**arriage-**r**eturn **l**ine**f**eed (**CR-LF**) kombinacija. Kada se podaci zapisuju u text-mode file, svaki **\n** se prevodi do **CR-LF**; kada se podaci čitaju sa disk file-a, svaki **CR-LF** se prevodi na **\n**. Na UNIX sistemima, ne radi se nikakvo prevođenje → karakter novalinija ostaje nepromjenjen.

→ **Binarni stream**-ovi se povezuju sa **binarni-mode file-ovima**. Neki ili svi podaci se zapisuju i čitaju nepromijenjeni, bez razdvajanja u linijama i bez korištenja **e**nd-**o**f-**l**ine karaktera.
NULL i **e**nd-**o**f-**l**ine karakteri nemaju neko posebno značenje i tretiraju se kao bilo koji drugi byte podataka.

Neke file ulaz/izlaz funkcije su ograničene na jedan file mode, dok druge funkcije mogu koristiti bilo koji mode. Ovo poglavlje vas uči koji mode da koristite sa kojom funkcijom.

→→ Imena file-ova (Filenames)

Svaki disk file ima ime, i morate koristiti imena file-ova kada radite sa disk file-ovima.
Imena file-ova se smještaju kao stringovi, kao i bilo koji drugi textualni podaci. Pravila, za šta je prihvatljivo za imena file-ova i šta se ne razlikuje od jednog do drugog operativnog sistema.
U DOS i windows 3.x, kompletno ime file-a se sastoji od imena i extenzije koja ima od jednog do tri karaktera.
Suprotno tome, Win 95 i Win NT operativni sistemi, kao i većina UNIX sistema, dozvoljavaju imena file-ova do **256** karaktera dugačke.

Operativni sistemi se također razlikuju u karakterima koji su dozvoljeni u imenima file-ove. U Win 95, npr., sljedeći karakteri nisu dozvoljeni:

/ \ : * ? " < > |

Morate biti na oprezu o pravilima za imena file-ova, zavisno od toga na kojem operativnom sistemu pišete.

Ime file-a u **C** programu može sadržavati informaciju puta (path). Put (path) navodi drive i/ili direktorij (ili folder) gdje je file lociran. Ako navedete ime file-a bez puta (path-a), biće pretpostavljeno da se file nalazi na bilokojoj loakciji koja je trenutno predstavljena operativnom sistemu kao default-na. Dobra je programerska praxa da uvijek navedete put (path) informaciju kao dio imena vašeg file-a.

Na PC-ovima, backslash se koristi da razdvoji imena direktorija u putu (path). Na primjer, za DOS i Windows, ime:

```
c:\data\list.txt
```

se odnosi na file nazvan **LIST.TXT** u direktoriju **\DATA** na drive-u **C**. Zapamtite da backslash karakter ima posebno značenje u **C**-u kada je u stringu. Da predstavite saam backslash karakter, morate ga prethoditi sa još jednim backslash karakterom. Tako da u **C** programu, vi biste predstavili ime file-a kako slijedi:

```
char *filename = "c:\\data\\list.txt";
```

Ako unesete ime file-a koristeći tastaturu, ipak, unesite samo jedan backslash.

Ne koriste svi sistemi backslash kao separator direktorija. Na primjer, UNIX koristi forward slash (/).

→→ Otvaranje File-a

Proces kreiranja stream-a, koji je povezan na disk file, se zove **otvaranje file-a** (opening a file). Kada otvorite file, on postaje dostupan za čitanje (što znaši da se podaci unose (input) iz file-a u program), pisanje (što znači da su podaci iz file-a snimljeni u file-u), ili obadvoje. Kada završite s upotrebom file-a, morate ga zatvoriti. Zatvaranje file-a je pokriveno kasnije u ovom poglavlju.

Da otvorite file, vi koristite →→→ **fopen()** ←←← bibloteču funkciju.
Prototip od **fopen()**-a je lociran u **STDIO.H** i čita kako slijedi:

```
FILE *fopen(const char *filename, const char *mode);
```

Ovaj prototip vam govori da **fopen()** vraća pointer na tip **FILE**, koji je struktura deklarisana u **STDIO.H**. Članovi od **FILE** strukture se koriste od strane programa u raznim operacijama za pristup file-u, ali vi ne treba da budete zabrinuti za njih. Ipak, za svaki file koji želite da otvorite, vi morate deklarisati pointer na tip **FILE**. Kada pozovete **fopen()**, ta funkcija kreira instancu od strukture **FILE** i vraća pointer na tu strukturu. Vi koristite ovaj pointer u svim kasnijim operacijama na file-u.
Ako **fopen()** ne uspije (fails), ona vraća **NULL**. Takav neuspjeh (failure) može biti prouzrokovan, na primjer, hardverskom greškom ili pokušajem da se otvori file na disketi koja nije bila formatirana.

Argument ***filename*** je ime file-a koji treba da bude otvoren. Kako je rečeno ranije, ***filename*** može – i treba – sadržavati naveden put (path). Argument ***filename*** može biti literalni string zatvoren u znake navoda ili pointer na string varijablu.

Argument **mode** navodi mode (način (metodu)) na koji će se otvoriti file. U ovom kontextu, **mode** kontroliše da li je file binarni ili textualni i da li je za čitanje, pisanje, ili obadvoje. Dozvoljene vrijednosti za **mode** su nabrojane u Tabeli **16.1**.

Tabela 16.1. Vrijednosti od mode-a za fopen() funkciju.

mode	Značenje
r	Otvara file za **čitanje**. Ako file **ne** postoji, **fopen()** vraća **NULL**.
w	Otvara file za **pisanje**. Ako file, navedenog imena, **ne** postoji, on se kreira. Ako file navedenog imena **postoji**, on se briše bez upozorenja, i novi, prazan file se kreira.
a	Otvara file za **dodavanje** (for appending). Ako file, navedenog imena **ne** postoji, on se kreira. Ako file **postoji**, novi podaci se dodaju na kraj file-a (to the end of the file).
r+	Otvara file za **čitanje** i **pisanje**. Ako file, navedenog imena, **ne** postoji, on se kreira. Ako file, navedenog imena, **postoji**, novi podaci se dodaju na početak file-a, prebrišući (overwriting) postojeće podatke.
w+	Otvara file za **čitanje** i **pisanje**. Ako file, navedenog imena, **ne** postoji, on se kreira. Ako file, navedenog imena, **postoji**, on se prepisuje (overwritten).
a+	Otvara file za **čitanje** i **dodavanje** (reading and appending). Ako file, navedenog imena **ne** postoji, on se kreira. Ako file, navedenog imena **postoji**, novi podaci se dodaju na kraj file-a.

→ Default-ni file mode je text.
Da otvorite file u binarnom mode-u, vi dodate **b** na mode argument. Tako da, **mode** argument od **a** bi otvorio text-mode file za dodavanje, dok bi **ab** otvorio binarni-mode file za dodavanje.

Zapamtite da **fopen()** vraća **NULL** ako se desi greška. Uslov greške koji može prouzrokovati povratnu vrijednost **NULL** uključuje sljedeće:

- Korištenje nepravilnog imena file-a.
- Pokušaj da se otvori file na disku koji nije spreman (vrata drive-a nisu zatvorena ili disk nije formatiran, na primjer).
- Pokušaj da se otvori file u nepostojećem direktoriju ili na nepostojećem disk drive-u.
- Pokušaj da se otvori nepostojeći file u **r** mode-u.

Kad god koristite **fopen()**, vi treba da testirate postojanje greške. Ne postoji način da se tačno kaže koja se greška desila, ali možete prikazati poruku korisniku i pokušati ponovo otvoriti file, ili možete prekinuti program. Većina **C** kompajlera uključuje **ne-ANSI** extenzije koja vam dozvoljavaju da dobijete informacije o prirodi greške.

Listing 16.1 demonstrira **fopen()**.
Listing 16.1. Upotreba fopen() da otvori disk file na različine mode-ove (načine).

```
1:  /* Demonstrira fopen() funkciju. */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:      FILE *fp;
8:      char filename[40], mode[4];
9:
10:     while (1)
11:     {
12:
13:         /* Unesi (input) ime file-a i mode. */
14:
15:         printf("\nEnter a filename: ");
16:         gets(filename);
17:         printf("\nEnter a mode (max 3 characters): ");
18:         gets(mode);
```

```
19:
20:         /* Pokushaj da otvorish file. */
21:
22:         if ( (fp = fopen( filename, mode )) != NULL )
23:         {
24:             printf("\nSuccessful opening %s in mode %s.\n",
25:                     filename, mode);
26:             fclose(fp);
27:             puts("Enter x to exit, any other to continue.");
28:             if ( (getc(stdin)) == `x')
29:                 break;
30:             else
31:                 continue;
32:         }
33:         else
34:         {
35:             fprintf(stderr, "\nError opening file %s in mode %s.\n",
36:                     filename, mode);
37:             puts("Enter x to exit, any other to try again.");
38:             if ( (getc(stdin)) == `x')
39:                 break;
40:             else
41:                 continue;
42:         }
43:     }
44: }
            Enter a filename: junk.txt
            Enter a mode (max 3 characters): w
            Successful opening junk.txt in mode w.
            Enter x to exit, any other to continue.
            j
            Enter a filename: morejunk.txt
            Enter a mode (max 3 characters): r
            Error opening morejunk.txt in mode r.
            Enter x to exit, any other to try again.
            x
```

ANALIZA: Ovaj program vas prompta za oboje, ime file-a i mode specifikator, u linijama **15** do **18**. Nakon što dobije imena, linija **22** pokušava da otvori file i pridruži njgov pointer na **fp**. Kao primjer dobre programerske praxe, **if** iskaz u liniji **22** provjerava da vidi da pointer otvorenog file-a nije **NULL**. Ako **fp** nije jednak **NULL**, printaju se: poruka koja govori da je file uspješno otvoren i da korisnik može nastaviti.
Ako je file pointer **NULL**, drugi (**else**) uslov **if** petlje se izvršava. **else** uslov u linijama **33** do **42** printa poruku koja govori da se desio problem. Ona onda prompta korisnika da odluči da će program nastaviti.

Vi možete experimentisati sa drugačijim imenima i mode-ovima da vidite koji vam daju grešku. U izlazu, upravo prikazanom, možete vidjeti da pokušaj otvaranja **MOREJUNK.TXT** u mode-u **r** rezultira sa greškom, jer file nije postojao na disku. Ako se greška desi, dat vam je izbor od unošenja informacija ili izlaženja iz programa. Da isforsirate grešku, vi možete unijeti nepravilno ime file-a kao npr., **[]**.

→→→ Pisanje i čitanje podataka file-a

Program koji koristi disk file može pisati podatke na file, čitati podatke sa file-a, ili kombinaciju ovo dvoje. **Možete pisati podatke na disk file na tri načina:**

- Možete koristiti **formatiran izlaz** da snimite formatirane podatke u file. Trebate koristiti formatiran izlaz samo sa text-mode file-ovima. Primarna upotreba formatiranog izlaza je da kreira file-ove koje sadrže text-ualne i numeričke podatke koji će da se čitaju od drugog programa kao što su spreadsheets ili baze podataka. Vi rijetko, ako ikad, koristite formatiran izlaz da kreirate file da se čita ponovo od strane **C** programa.
- Možete koristiti **karakterni izlaz** da snimite jedinstvene (single) karaktere ili linije karaktera u file. Iako je tehnički moguće koristiti karakterni izlaz sa binarnim-mode file-ovima, to može biti zaguljeno. Trebate ograničiti karakterni-mode izlaz na textualne file-ove. Glavno korištenje karakternog izlaza je da se snime textualni (ali ne numerički) podaci u formi koja se može čitati od **C**-a, kao i od drugih programa kao što je word procesor.
- Možete koristiti **direkni izlaz** da snimite sadržaj dijela (sekcije) memorije direktno na disk file. Ovaj metod je za binarne file-ove samo. Direktni izlaz je najbolji način da se snime podaci za kasniju upotrebu od strane **C** programa.

Kada želite da čitate podatke sa file-a, vi imate iste tri opcije: formatiran ulaz, karakterni izlaz, ili direktni izlaz.
Tip ulaza (inputa) koji koristite u nekom naročitom slučaju, zavisi, skoro u potpunosti, od prirode file-a koji se čita. Uopšteno, vi ćete čitati podatke u istom mode-u na kojem su oni snimljeni, ali ovo nije neophodno. Ipak, čitanje file-a u mode-u različitom od onoga koji je zapisan zahtjeva temeljno znanje **C**-a i file formata.

Prethodni opisi tri tipa file ulaza i izlaza sugeriraju zadatke koji najbolje leže za svaki tip izlaza. Ovo ni na koji način nije skup strogih pravila. **C** jezik je veoma flexibilan (jedna od njegovih prednosti!), tako da mudar programer može napraviti bilo koji tip file izlaza, koji odgovara skoro svakim potrebama. Kao programeru početniku, može biti lakše ako pratite ove naputke (guidelines), bar inicijalno.

→→ Ulaz i Izlaz Formatiranog File-a

Ulaz/izlaz formatiranog file-a radi sa textualnim i numeričkim podacima koji su formatirani ne specifičan način. To je dorektno analogno sa formatiranom tastaturom ulazom i ekranskim izlazom, koji su urađeni sa **printf()** i **scanf()** funkcijama, opsianim Dana 14. Prvo ćemo pričati o formatiranom izlazu, a nakon toga ulazu.

→ Izlaz Formatiranog File-a

Izlaz formatiranog file-a je urađen sa bibliotečnom funkcijom **fprintf()**.
Prototip za **fprintf()** je file zaglavlja **STDIO.H**, i on čita kako slijedi:

```
int fprintf(FILE *fp, char *fmt, ...);
```

Prvi argument je pointer na tip **FILE**. Da pišete podatke na neki naročit disk file, vi prosljeđujete pointer koji je vraćen kada ste otvorili file sa **fopen()**.

Drugi argument je format string. Naučili ste o format stringovima u diskusiji o **printf()**, Dana 14. Format string kojeg koristi **fprintf()** slijedi tačno ista pravila kao i **printf()**.

Posljednji argument je **….** **Šta ovo znači?** U prototipu funkcije, elipse (**ellipses**) predstavljaju broj argumenata varijable. Drugim riječima, s dodatkom sa file pointerom i argumentima format stringa, **fprintf()** uzima nulu, jedan, ili više dodatnih argumenata. Ovo je kao i kod **printf()**. Ovi argumenti su imena od varijabli koji će ići na izlaz na neki navedeni stream.

Zapamtite, **fprintf()** radi kao i **printf()**, s razlikom što šalje svoj izlaz na stream naveden u listi argumenata. U stvari, ako navedete stream argument **stdout**, **fprintf()** je indentičan sa **printf()**.

Listing 16.2 demonstrira upotrebu **fprintf()**.

Listing 16.2. Eqivalencija od fprintf() formatiranog izlaza na obadvoje, file i stdout.

```
1:  /* Demonstrira  fprintf()  funkciju. */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:
5:  void clear_kb(void);
6:
7:  main()
8:  {
9:      FILE *fp;
10:     float data[5];
11:     int count;
12:     char filename[20];
13:
14:     puts("Enter 5 floating-point numerical values.");
15:
16:     for (count = 0; count < 5; count++)
17:         scanf("%f", &data[count]);
18:
19:     /* Uzmi ime file-a i otvori file. Prvo ocisti stdin */
20:     /* od bilo kojih dodatnih karaktera. */
21:
22:     clear_kb();
23:
24:     puts("Enter a name for the file.");
25:     gets(filename);
26:
27:     if ( (fp = fopen(filename, "w")) == NULL)
28:     {
29:         fprintf(stderr, "Error opening file %s.", filename);
30:         exit(1);
31:     }
32:
33:     /* Pishi numerishke podatke u file i u stdout. */
34:
35:     for (count = 0; count < 5; count++)
36:     {
37:         fprintf(fp, "\ndata[%d] = %f", count, data[count]);
38:         fprintf(stdout, "\ndata[%d] = %f", count, data[count]);
39:     }
40:     fclose(fp);
41:     printf("\n");
42:     return(0);
43: }
44:
45: void clear_kb(void)
46: /* Chistii stdin od bilo kojih karaktera koji chekaju. */
47: {
48:     char junk[80];
49:     gets(junk);
50: }
            Enter 5 floating-point numerical values.
            3.14159
            9.99
            1.50
            3.
```

```
1000.0001
Enter a name for the file.
numbers.txt
data[0] = 3.141590
data[1] = 9.990000
data[2] = 1.500000
data[3] = 3.000000
data[4] = 1000.000122
```

ANALIZA: Možda se pitate zašto program prikazuje **1000.000122** kada je unešena vrijednost **1000.0001**. Ovo nije greška u programu. Normalna je posljedica da **C** smješta brojeve interno. Neke **float**ing-point vrijednosti ne mogu biti smještene tačno precizno, tako da male nepravilnosti, kao ova, nekad rezultiraju.

Ovaj program koristi **fprintf()** u linijama **37** i **38** da šalje nešto formatiranog texta i numeričke podatke do **stdout** i do disk file-a, čija ste imena vi naveli. Jedina razlika između dvije linije je prvi argument → tj., stream na koji se šalju podaci.
Nakon pokretanja programa, koristite vaš editor da pogledate sadržaj file-a **NUMBERS.TXT** (ili bilo koje ime koje ste mu pridružili), koji će biti u istom direktorijumu kao i programski file-ovi. Vidjet ćete da je text u file-u tačna kopija texta koji je prikazan na-ekran.

Primjetite da Listing **16.1** koristi **clear_kb()** funkciju o kojoj smo pričali Dana 14. Ovo je neophodno da se iz **stdin** uklone bilo koji dodatni karakteri koji mogu ostati od poziva **scanf()**. Da niste očistili **stdin**, ovi dodatni karakteri (naročito, novalinija) se čitaju od strane **gets()** koja unosi ime file-a, i rezultat je file greška kreacija.

→→ Ulaz Formatiranog file-a

Za ulaz formatiranog file-a, koristite **fscanf()** bibliotečnu funkciju, koja se koristi kao **scanf()** (Dana 14), s razlikom da ulaz dolazi sa navedenog stream-a, umjesto sa **stdin**-a.
Prototip za **fscanf()** je:

```
int fscanf(FILE *fp, const char *fmt, ...);
```

Argument **fp** je pointer na tip **FILE** vraćen od **fopen()**, i **fmt** je pointer na format string koji navodi kako će **fscanf()** čitati ulaz.
Komponente format stringa su iste kao za **scanf()**. Konačno, elipse (**ellipses**) (**…**) indiciraju jednu ili više dodatnih argumenata, adrese varijabli gdje **fscanf()** će da pridruži ulaz.

Prije nego što počnete sa **fscanf()**, možda želite da ponovite dio sa **scanf()**-om Dana 14. Funkcija **fscanf()** radi upravo isto kao i **scanf()**, s razlikom da se karakteri uzimaju sa navedenog stream-a, umjesti sa **stdin** stream-a.

Da demonstrirate **fscanf()**, trebate textualni file koji sadrži neke brojeve ili stringove u formatu koji može biti čitan od funkcije. Koristite vaš editor da kreirate file nazvan **INPUT.TXT**, i unesite pet **float**ing-point brojeva sa nešto razmaka između njih (blanko ili novalinija). Na primjer, vaš file može izgledati ovako:

```
123.45     87.001
100.02
0.00456    1.0005
```

Sad, kompajlirajte i pokrenite Listing **16.3**.

Listing 16.3. Upotreba fscanf()-a za čitanje formatiranih podataka sa disk file-a (to read formatted data from a disk file).

```
1:  /* Chitanje podataka formatiranog file-a sa fscanf(). */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:      float f1, f2, f3, f4, f5;
8:      FILE *fp;
9:
10:     if ( (fp = fopen("INPUT.TXT", "r")) == NULL)
11:     {
12:         fprintf(stderr, "Error opening file.\n");
13:         exit(1);
14:     }
15:
16:     fscanf(fp, "%f %f %f %f %f", &f1, &f2, &f3, &f4, &f5);
17:     printf("The values are %f, %f, %f, %f, and %f\n.",
18:             f1, f2, f3, f4, f5);
19:
20:     fclose(fp);
21:      return(0);
22: }
The values are 123.45, 87.0001, 100.02, 0.00456, and 1.0005.
```

> **NAPOMENA:** Preciznost vrijednost može prouzrokovati neke brojeve da se ne prikazuju na tačan način koji ste unijeli. Na primjer, **100.02** može izgledati kao **100.0199**.

ANALIZA: Ovaj program čita pet vrijednosti sa file-a koji ste kreirali i onda ih prikazuje na-ekran. **fopen()** poziv u liniji **10** otvara file za mode čitanja. Takođe provjerava da vidi da li je file otvoren pravilno.
Ako file nije otvoren, poruka greške se prikazuje u liniji **12**, i program izlazi (exits (linija **13**)).
Linija **16** demonstrira upotrebu **fscanf()** funkcije. Sa izuzetkom od provg parametra, **fscanf()** je identična sa **scanf()**.
Prvi parametar pokazuje na file kojeg želite da vaš program čita.
Možete dalje experimentisati sa **fscanf()**, kreirajući ulazne file-ove sa vašim programerskim editorom i vidjeti kako **fscanf()** čita podatke.

→→→ Karakterni ulaz i izlaz (Character Input and Output)

Kada se koristi sa disk file-ovima, pojam **character I/O** (karakterni U/I) se odnosi na jedan (single) karakter, kao i na liniju karaktera. Zapamtite, da je linija sekvenca od nula ili više karaktera terminiranih sa karakterom novalinija.
Koristite character-ni I/O, sa text-mode file-ovima.

Sljedeći dio opisuje karakterni U/I (character I/O) funkcije, i onda ćete vidjeti demonstraciju programa.

→→ Karakterni ulaz (Character Input)

Postoje tri funkcije za karakterni ulaz: **getc()** i **fgetc()** za jedne (single) karaktere, i **fgets()** za linije.

→→→→ getc() i fgetc() Funkcije

Funkcije **getc()** i **fgetc()** su identične i mogu se koristiti naizmjenično. One unose (input) jedan (single) karakter sa navedenog stream-a.

→→→ Evo protoipa za **getc()**, koji je u **STDIO.H**:

```
int getc(FILE *fp);
```

Argument **fp** je pointer vraćen od **fopen()** kada je file otvoren.
Funkcija vraća karakter koji je unešen (input) ili **EOF** u slučaju greške.

Vidjeli ste **getc()** u ranijim programima za unos karaktera sa tastature. Ovo je još jedan primjer flexibilnosti **C**-ovim stream-ova → ista funkcija može biti korištena za unos (ulaz (input)) sa tastature ili file-a.

Ako **getc()** i **fgetc()** vrate jedan karakter, zašto su prototipisani da vrate tip **int**?
Razlog je taj što, kada čitate file-ove, morate biti u mogućnosti da čitate u **e**nd-**o**f-**l**ine **marker**-u, koji na nekim sistemima nije tipa **char**, nego tipa **int**. Vidjet ćete **getc()** u akciji kasnije, u Listingu **16.10**.

→→→ fgets() Funkcija

Da čitate liniju karaktera iz file-a, koristite **fgets()** bibliotečnu funkciju.
→→→ Prototip je:

```
char *fgets(char *str, int n, FILE *fp);
```

Argument **str** je pointer na buffer u koji je smješta ulaz (unos (input)), **n** je maximalan broj karaktera koji se unose, i **fp** je pointer na tip **FILE** koji je vraćen od **fopen()** kada je file otvoren.

Kad se pozove, **fgets()** čita karaktere od **fp** u memoriju, počinjući na lokaciji na koju pokazuje **str**.

Karakteri se čitaju dok ne naiđe novalinija ili dok je ne pročita **n-1** karaktera, šta god da se desi prvo. Postavljajući da je **n** jednako sa brojem byte-ova alociranim za buffer **str**, vi sprječavate unos (input) sa prepisivanje (overwriting) memorije van alociranog prostora. (**n-1** je da dozvoli prostor za terminirajući **\0** koji **gets()** dodaje na kraj stringa). Ako je uspješno, **fgets()** vraća **str**.
Mogu se desiti dva tipa grešaka, kako je pokazano sa povratnom vrijednošću **NULL**:

- Ako se pročitna greška, ili se naiđe na **OEF**, prije nego što se bilo koji karakter pridruži **str**-u, **NULL** se vraća, i memorija, na koju pokazuje **str**, je nepromjenjena.
- Ako se pročitna greška, ili se naiđe na **EOF**, nakon što se jedan ili više karaktera pridruži **str**-u, **NULL** se vraća, i memorija, na koju pokazuje **str**, sadrži smeće.

Možete vidjeti da **fgets()** ne mora unijeti (input) cijelu liniju (tj., sve do sljedećeg novalinija karaktera). Ako se **n-1** karaktera pročita prije nego se susretne novalinija, **fgets()** se zaustavlja. Sljedeća operacija čitanja iz file-a počinje gdje je posljednja napustila. Da budete sigurni da **fgets()** čita cijele stringove, zaustavljajući se samo na novalinija, uvjerite se da je veličina vašeg ulaznog buffer-a i odgovarajuća vrijednost od **n**, prosljeđena do **fgets()**, dovoljno velika.

→→→→ Karakterni izlaz (Character Output)

Treba da znate o dvije karakterne izlazne funkcije: **putc()** i **fputs()**.

→→→ putc() Funkcija

Bibliotečna funkcija **putc()** piše (writes) jedan (single) karakter na navedeni stream.
Njen prototip u **STDIO.H** čita:

```
int putc(int ch, FILE *fp);
```

Argument **ch** je karakter ka izlazu. Kao i sa ostalim karakternim funkcijama, formalno se poziva tip **int**, ali samo se koriste niže-rangiran byte (lower-order byte).
Argument **fp** je pointer pridružen sa file-om (pointer vraćen od **fopen()** kada je file otvoren).
Funkcija **putc()** vraća karakter koji je upravo napisan, ako je uspješno ili **EOF** ako se desi greška.

Simbolična konstanta **EOF** je definisana u **STDIO.H**, i ona ima vrijednost **–1**. Zato jer nikakvi "praavi" karakteri nemaju tu numeričku vrijednost, **EOF** može biti korišten kao **indikator greške** (**samo sa text-mode file-ovima**).

→→→→ fputs() Funkcija

Da napišete liniju karaktera na stream, koristite bibliotečnu funkciju **fputs()**. Ova funkcija radi kao i **puts()**, pokrivena Dana 14. Jedina razlika je da sa **fputs()** vi možete navesti (specificirati) izlazni stream. Također, **fputs()** ne dodaje novalinija na kraju stringa; ako želite da da, morato to explicitno uključiti.
→→→ Njen prototip u **STDIO.H** je:

```
char fputs(char *str, FILE *fp);
```

Argument **str** je pointer na **null**-terminiran string koji se piše, i **fp** je pointer na tip **FILE**, vraćen od **fopen()**, kada je file otvoren.
String, na koji pokazuje **str**, se piše u file, minus njegov terminirajući karakter **\0**.
Funkcija **fputs()** vraća nenegativnu vrijednost ako je uspješna i **EOF** u slučaju greške.

→→→→ Direktni ulaz i izlaz file-a (Direct File Input and Output)

Vi koristite direktni file I/O (U/I) često kada snimate podatke, koji će biti pročitani kasnije od strane istog ili nekog drugog **C** programa.
Direktni I/O se koristi samo sa file-ovima u binarnom-mode-u.
Sa direktnim izlazom, blokovi podataka se pišu iz memorije na disk.
Direktni ulaz obrće proces: Blok podataka se čitaju sa diska u memoriju.
Na primjer, jedan poziv diretkno-izlazne funkcije može pisati cijeli niz tipa **double** na disk, i jedan poziv direktno-ulazne funkcije može čitati cijeli niz sa diska nazad u memoriju.
Direktne I/O (U/I) funkcije su →→ **fread()** ←←← i →→→ **fwrite()** ←←←.

→→→ fwrite() Funkcija

fwrite() bibliotečna funkcija piše blok podataka iz memorije u binarni-mode file.

Njen prototip u **STDIO.H** je:

```
int fwrite(void *buf, int size, int count, FILE *fp);
```

Argument **buf** je pointer na oblast (region) memorije koja drži podatke koji će se pisati u file.
Tip pointera je **void**; može biti pointer na bilo što.

Argument **size** navodi veličinu, u byte-ovima, individualnih podatkovnih predmeta, i **count** navodi broj predmeta koji će se pisati.

Na primjer, ako želite da spasite (snimite (save)) **100**-elementni **integer** niz, **size** bi bila **2** (zato jer svaki **int** zauzima **2** byte-a) i **count** bi bio **100** (zato jer niz sadrži **100** elemenata). Da dobijete **size** argument, možete koristiti **sizeof()** operator.

Argument **fp** je, naravno, pointer na tip **FILE**, vraćen od **fopen()**-a kada je otvoren file.
fwrite() funkcija vraća broj predmeta koji su uspješno zapisani (written);
ako je vraćena vrijednost manja od **count**, to znači da se desila greška.

Da provjerite postojanje greške, obično programirate **fwrite()** kako slijedi:

```
if( (fwrite(buf, size, count, fp)) != count)
fprintf(stderr, "Error writing to file.");
```

Evo par primjera upotrebe **fwrite()**-a. Da napišete (write) jednu varijablu **x** tipa **double** u file, koristite sljedeće:

```
fwrite(&x, sizeof(double), 1, fp);
```

Da napišete niz **data[]** od **50** struktura tipa **address** u file, imate dva izbora:

```
fwrite(data, sizeof(address), 50, fp);
fwrite(data, sizeof(data), 1, fp);
```

Prva metoda piše niz kao **50** elemenata, gdje svaki element ima veličinu od jedne strukture tipa **address**.
Druga metoda tretira niz kao jedan element.
Ove dvije metode obavljaju sasvim istu stvar.

Sljedeći dio objašnjava **fread()** i zatim predstavlja program demonstrirajući **fread()** i **fwrite()**.

→→→ fread() Funkcija

fread() bibliotečna funkcija čita blok podataka sa binarnog-mode-a file-a u memoriju.
Njen prototip u **STDIO.H** je:

```
int fread(void *buf, int size, int count, FILE *fp);
```

Argument **buf** je pointer na oblast (region) memorije koja prihvata (prima) podatke pročitane iz file-a.
Kao i sa **fwrite()**, tip pointera je **void**.

Argument **size** navodi veličinu, u byte-ovima, individualnih podatkovnih predmeta koji se čitaju, i **count** navodi broj predmeta za čitanje.
Primjetite kako su ovi argumenti paralelni sa onim koje koristi **fwrite()**. Ponovo, **sizeof()** operator se tipično koristi da obezbjedi **size** argument.

Argument **fp** je (kao i uvijek) pointer na tip **FILE** koji je vraćen od **fopen()** kada je file otvoren.
fread() funkcija vraća broj pročitanih predmeta (items read); ovo može biti manje od **count** ako je dostignut **e**nd-**o**f-**f**ile ili se desila greška.

Listing **16.4** demonstrira upotrebu **fwrite()** i **fread()**.
Listing 16.4. Korištenje fwrite() i fread() za direktni pristup file-u.

```
1:  /* Direct file I/O sa fwrite() i fread(). */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:
5:  #define SIZE 20
6:
7:  main()
```

```
8:  {
9:      int count, array1[SIZE], array2[SIZE];
10:     FILE *fp;
11:
12:     /* Inicijaliziraj array1[]. */
13:
14:     for (count = 0; count < SIZE; count++)
15:         array1[count] = 2 * count;
16:
17:     /* Otvori binary mode file. */
18:
19:     if ( (fp = fopen("direct.txt", "wb")) == NULL)
20:     {
21:         fprintf(stderr, "Error opening file.");
22:         exit(1);
23:     }
24:     /* Snimi (Save) array1[] u file. */
25:
26:     if (fwrite(array1, sizeof(int), SIZE, fp) != SIZE)
27:     {
28:         fprintf(stderr, "Error writing to file.");
29:         exit(1);
30:     }
31:
32:     fclose(fp);
33:
34:     /* Sad otvori isti file za chitanje u binarnom-mode-u. */
35:
36:     if ( (fp = fopen("direct.txt", "rb")) == NULL)
37:     {
38:         fprintf(stderr, "Error opening file.");
39:         exit(1);
40:     }
41:
42:     /* Chitaj podatke u array2[]. */
43:
44:     if (fread(array2, sizeof(int), SIZE, fp) != SIZE)
45:     {
46:         fprintf(stderr, "Error reading file.");
47:         exit(1);
48:     }
49:
50:     fclose(fp);
51:
52:     /* Sad prikazi obadva niza da se pokaze da su isti. */
53:
54:     for (count = 0; count < SIZE; count++)
55:         printf("%d\t%d\n", array1[count], array2[count]);
56:     return(0);
57: }
            0       0
            2       2
            4       4
            6       6
            8       8
            10      10
            12      12
            14      14
            16      16
            18      18
            20      20
```

```
22      22
24      24
26      26
28      28
30      30
32      32
34      34
36      36
38      38
```

ANALIZA: Program inicijalizira niz u linijama **14** i **15**. On onda koristi **fwrite()** u liniji **26** da snimi niz na disk. Program koristi **fread()** u liniji **44** da čita podatke u različit (different) niz. Konačno, on prikazuje oba niza na-ekran da pokaže da oni sad drže iste podatke (linije **54** i **55**).

Kada snimite podatke sa **fwrite()**, ne može puno stvari krenuti loše, osim nekih tipova disk grešaka. Sa **fread()**, morate, ipak, biti oprezni. Što se tiče **fread()**, podaci na disku su samo sekvence byte-ova. Funkcija nema načina kako da zna šta ona predstavlja.

Na primjer, na **16**-bitnim sistemima, blok od **100** byte-ova bi mogao biti **100** varijabli **char**, **50** varijabli **int**, **25** varijabli **long**, ili **25** varijabli **float**. Ako zatražite da **fread()** čita taj blok u memoriju, on poslušno to i čini. Ipak, ako je blok snimljen sa niza tipa **int**, i vi ga obnovite u niz tipa **float**, ne dešava se greška, ali dobijate čudne rezultate.

Kada pišete programe, morate biti sigurni da se **fread()** koristi pravilno, čitajući podatke u varijable i nizove prigodnog tipa. Primjetite da u Listingu **16.4**, svi pozivi na **fopen()**, **fwrite()**, i **fread()** se provjeravaju da se uvjerimo da rade pravilno.

→→→ File Buffer-ovanje: Zatvaranje i Ispiranje file-ova (Closing and Flushing Files)

Kada završite s upotrebom ovog file-a, vi ga trebate zatvoriti koristeći →→→ **fclose()** ←←← funkciju.
Vidjeli ste **fclose()** upotrebu u programima koji su predstavljeni ranije u ovom poglavlju.

Njen prototip je:

```
int fclose(FILE *fp);
```

Argument **fp** je **FILE** pointer pridružen stream-u; **fclose()** vraća **0** ako je uspješno ili **–1** ako je greška. Kada zatvorite file, file-ov buffer se ispušta (flushed) (napisan u file).

Također možete zatvoriti sve otvorene stream-ove, osim standardnih (**stdin**, **stdout**, **stdprn**, **stderr**, i **stdaux**) koristeći →→→ **fcloseall()** ←←← funkciju.

Njen prototip je:

```
int fcloseall(void);
```

Ova funkcija također ispušta (flushes) bilo koje stream-ove buffer-e i vraća broj zatvorenih stream-ova.

Kada se program terminira (bilo dostizanjem kraja **main()**-a ili izvršavanjem **exit()** funkcije), svi stream-ovi se automatski ispiraju (flushed) i zatvaraju. Ipak, dobra je ideja da stream-ove zatvorite explicitno → naročito one koji su povazani sa disk file-ovima → odmah nakon što završite sa njima. Razlog ima veze sa stream buffer-ima.

Kada vi kreirate stream povezan sa disk file-om, buffer je automatski kreiran i pridružen sa stream-om. **Buffer** je blok memorije koji se koristi za privremeno smještanje podataka koji se pišu ili čitaju sa file-a. Buffer-i su potrebni, jer su **disk drive-ovi blok-orijentisani uređaji**, što znači da rade najefikasnije kada se podaci čitaju i pišu u blokovima neke određene veličine. Veličina idealnog bloka zavisi od vrste hardvera kojeg koristite. Tipično od nekoliko **100** do nekoliko **1000 byte**-ova. Ipak, ne trebate se brinuti za tačne veličine blokova.

Buffer, pridružen sa file stream-om, služi kao interface između stream-a (koji je krakter-orjentisan) i disk hardware-a (koji je blok-orjentisan). Dok vaš program piše podatke na stream, podaci su snimljeni u buffer dok se buffer ne napuni, i onda se cijeli sadržaj buffer-a piše, kao blok, na disk. Analogni proces se dešava kada čitate podatke sa diska.
Kreiranje i operacije sa bufferom radi operativni sistem i u potpunosti su automatske. (**C** nudi neke funkcije za manipulaciju buffer-om, ali su one van dometa ove knjige).

U praxi, ove operacije buffer-a znače da, tokom izvršenja programa, se podaci koje vaš program piše na disk, još uvijek mogu nalaziti u buffer-u, ne na disku. Ako vaš program zakaže, ako postoji veliki kvar, ili se desi neki drugi problem, podaci koji su još uvijek u buffer-u mogu biti izgubljeni, i nećete znati šta se nalazi u disk file-u.

Vi možete ispustiti (flush) stream-ove buffer-e bez njihovog zatvaranja koristeći:
→→→ **fflush()** ←←← ili →→→ **flushall()** ←←← bibliotečne funkcije.

Koristite **fflush()** kada želite da se file-ov buffer zapiše na disk dok još koristite file.

Koristite **flushall()** da isperete (flush) sve buffer-e svih otvorenih stream-ova.

Prototip za ove dvije funkcije slijedi:

```
int fflush(FILE *fp);
int flushall(void);
```

Argument **fp** je **FILE** pointer, vraćen od **fopen()** kada je file otvoren. Ako je file otvoren za pisanje, **fflush()** piše njegov buffer na disk. Ako je file otvoren za čitanje, buffer se čisti.
Funkcija **fflush()** vraća **0** nakon uspjeha ili **EOF** u slučaju greške.
Funkcija **flushall()** vraća broj otvorenih stream-ova.

OTVORITE file prije nego što pokušate da čitate ili pišete na njega.
NE pretpostavljajte da je pristup file-u okay. Uvijek provjerite nakon čitanja, pisanja ili otvaranja, da budete sigurni da je funkcija radila kako treba.
KORISTITE sizeof() operator sa **fwrite()** i **fread()** funkcijama.
ZATVORITE sve file-ove koje ste otvorili.
NE koristite **fcloseall()**, osim ako imate razlog da zatvorite sve stream-ove.

→→→ Sequencijalni nasuprot Nasumičnom pristupu file-a (Random File Access)

Svaki otvoren file ima **pozicioni indikator** koji je pridružen njemu.
Pozicioni indikator navodi gdje se rade operacije čitanja i pisanja u file-u. Pozicija je uvijek data u formi byte-ova od početka file-a.
Kada je novi file otvoren, pozicioni indikator je uvijek na početku file-a, pozicija **0**.
(Zato što je file novi i ima dužinu **0**, ne postoji druga lokacija koju bi indicirao).
Kada je otvoren postojeći file, pozicioni indikator je na kraju file-a ako je file otvoren u dodaj-mode-u (append mode), ili na početku file-a ako je file otvoren u nekom drugom mode-u.
To mu je početna pozicija. Dalje se on kreće sekvencijalno (npr. pišemo 10 byte-a podataka (0 – 9) i tu mu je trenutna pozicija (9); daljnje pisanje (čitanje) počinje odatle).

Kad vam je potrebno više kontrole, koristite **C** bibliotečne funkcije koje vam dopuštaju da odredite i promjenite vrijednost pozicionog indikatora file-a.
Kontrolišući pozicioni indikator, vi možete obavljati nasumičan pristup file-u. Pod pojmom, nasumičan (random), se podrazumjeva da možete čitati podatke sa, ili pisati podatke na, bilo koju poziciju u file-u bez čitanja ili pisanja svih prethodnih podataka.

→→→ ftell() ←←← i →→→ rewind() ←←← Funkcije

Da postavite pozicioni indikator na **početak file-a,** koristite bibliotečnu funkciju
→→→ rewind() ←←←.

Njen prototip, u **STDIO.H**, je:

```
void rewind(FILE *fp);
```

Argument **fp** je **FILE** pointer koji je povezan sa stream-om. Nakon što se pozove **rewind()**, pozicioni indikator file-a je postavljen na početak file-a (byte **0**). Koristite **rewind()** ako ste pročitali nešto podataka sa file-a i želite da počnete čitanje iz početka file-a ponovo, bez zatvaranja i ponovnog otvaranja file-a.

Da **odredite vrijednost** pozicionog indikatora file-a, koristite →→→ **ftell()** ←←←.
Prptotip ove funkcije, koji se nalazi u **STDIO.H**, čita:

```
long ftell(FILE *fp);
```

Argument **fp** je **FILE** pointer koji je vraćen od **fopen()** kada je file otvoren.

Funkcija **ftell()** vraća tip **long** koji daje tekuću poziciju, u byte-ovima, od početka file-a (prvi byte je na poziciji **0**). Ako se desi greška, **ftell()** vraća **–1 L** (**-1** tipa **long**).

Da dobijete osjećaj za operacije sa **rewind()** i **ftell()**, pogledajte Listing **16.5**.

Listing 16.5. Upotreba ftell() i rewind().

```
1:  /* Demonstrira ftell() i rewind(). */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:
5:  #define BUFLEN 6
6:
7:  char msg[] = "abcdefghijklmnopqrstuvwxyz";
8:
9:  main()
10: {
11:     FILE *fp;
12:     char buf[BUFLEN];
13:
14:     if ( (fp = fopen("TEXT.TXT", "w")) == NULL)
15:     {
16:         fprintf(stderr, "Error opening file.");
17:         exit(1);
18:     }
19:
20:     if (fputs(msg, fp) == EOF)
21:     {
22:         fprintf(stderr, "Error writing to file.");
23:         exit(1);
24:     }
25:
26:     fclose(fp);
27:
28:     /* Sad otvori file za chitanje. */
29:
30:     if ( (fp = fopen("TEXT.TXT", "r")) == NULL)
31:     {
32:         fprintf(stderr, "Error opening file.");
33:         exit(1);
```

```
34:     }
35:     printf("\nImmediately after opening, position = %ld", ftell(fp));
36:
37:     /* Chitaj u  5 karaktera. */
38:
39:     fgets(buf, BUFLEN, fp);
40:     printf("\nAfter reading in %s, position = %ld", buf, ftell(fp));
41:
42:     /* Chitaj u sljedecih 5 karaktera. */
43:
44:     fgets(buf, BUFLEN, fp);
45:     printf("\n\nThe next 5 characters are %s, and position now = %ld",
46:             buf, ftell(fp));
47:
48:     /* Rewind  stream. */
49:
50:     rewind(fp);
51:
52:     printf("\n\nAfter rewinding, the position is back at %ld",
53:             ftell(fp));
54:
55:     /* Chitaj u 5 karaktera. */
56:
57:     fgets(buf, BUFLEN, fp);
58:     printf("\nand reading starts at the beginning again: %s\n", buf);
59:     fclose(fp);
60:     return(0);
61: }
        Immediately after opening, position = 0
        After reading in abcde, position = 5
        The next 5 characters are fghij, and position now = 10
        After rewinding, the position is back at 0
        and reading starts at the beginning again: abcde
```

ANALIZA: Ovaj program piše string, **msg**, u file nazvan **TEXT.TXT**.
Poruka (message) sadrži **26** slova alfabeta, po redu (in order).
Linije **14** do **18** otvaraju **TEXT.TXT** za pisanje i testiraju da provjere da je file otvoren uspješno.
Linije **20** do **24** pišu **msg** u file koristeći **fputs()** i provjeravaju da se uvjere da je pisanje bilo uspješno.
Linija **26** zatvara file sa **fclose()**, završavajući proces kreiranja file-a za upotrebu od ostatka programa.

Linije **30** do **34** ponovo otvaraju file, samo ovaj put za čitanje.
Linija **35** printa povratnu vrijednost od **ftell()**. Primjetite da je ova pozicija na početku file-a.
Linija **39** obavlja **gets()** da čita pet karaktera. Pet karaktera i pozicija novog file-a se printaju u liniji **40**.
Primjetite da **ftell()** vraća tačan offset (pomjeraj).
Linija **50** poziva **rewind()** da vrati pointer nazad na početak file-a, prije nego linija **52** printa poziciju file-a ponovo. Ovo bi vam trebalo potvditi da **rewind()** resetuje poziciju.
Dodatno čitanje u liniji **57** dalje potvrđuje da se program zaista vratio na početak file-a.
Linija **59** zatvara file prije završetka programa.

→→→ fseek() Funkciju

Preciznija kontrola nad stream-ovim indikatorom pozicije je moguća sa →→→ **fseek()** ←←← bibliotečnom funkcijom.
Koristeći **fseek()**, možete postaviti indikator pozicije **bilo gdje** u file-u.

Prototip funkcije, u **STDIO.H**, je:

```
int fseek(FILE *fp, long offset, int origin);
```

Argument **fp** je **FILE** pointer pridružen file-u. Distanca, na koju će indikator pozicije biti pomjeren, je dat sa **offset**-om, u byte-ovima.
Argument **origin** navodi (specificira) pomjerajnu odnosnu početnu tačku.
Mogu biti tri vrijednosti za **origin**, sa simboličnim kostantama definisanim u **IO.H**, kako je pokazano u Tabeli **16.2**.

Tabela 16.2. Moguće origin vrijednosti za fseek().

Konstanta	Vrijednost	Opis
SEEK_SET	**0**	Pomjera indikatorove offset byte-ove sa **početka** file-a.
SEEK_CUR	**1**	Pomjera indikatorove offset byte-ove sa njegove **tekuće** pozicije.
SEEK_END	**2**	Pomjera indikatorove offset byte-ove sa **kraja** file-a.

Funkcija **fseek()** vraća **0**, ako je indikator uspješno pomjeren ili **nenula** ako se desila greška.

Listing **16.6** koristi **fseek()** za **nasumičan** pristup file-u.
Listing 16.6. Nasumičan pristup file-u sa fseek().

```
1:  /* Nasumican pristup file-u sa fseek(). */
2:
3:  #include <stdlib.h>
4:  #include <stdio.h>
5:
6:  #define MAX 50
7:
8:  main()
9:  {
10:     FILE *fp;
11:     int data, count, array[MAX];
12:     long offset;
13:
14:     /* Initicijaliziraj niz. */
15:
16:     for (count = 0; count < MAX; count++)
17:         array[count] = count * 10;
18:
19:     /* Otvori binarni file za pisanje. */
20:
21:     if ( (fp = fopen("RANDOM.DAT", "wb")) == NULL)
22:     {
23:         fprintf(stderr, "\nError opening file.");
24:         exit(1);
25:     }
26:
27:     /* Upisi niz u file, onda ga zatvori. */
28:
29:     if ( (fwrite(array, sizeof(int), MAX, fp)) != MAX)
30:     {
31:         fprintf(stderr, "\nError writing data to file.");
32:         exit(1);
33:     }
34:
35:     fclose(fp);
36:
37:     /* Otvori file za citanje. */
38:
39:     if ( (fp = fopen("RANDOM.DAT", "rb")) == NULL)
40:     {
41:         fprintf(stderr, "\nError opening file.");
```

```
42:         exit(1);
43:     }
44:
45:     /* Pitaj korisnika koji element da se pricita. Unesi element */
46:     /* i prikazi ga, izlaz kada je uneseno -1. */
47:
48:     while (1)
49:     {
50:         printf("\nEnter element to read, 0-%d, -1 to quit: ",MAX-1);
51:         scanf("%ld", &offset);
52:
53:         if (offset < 0)
54:             break;
55:         else if (offset > MAX-1)
56:             continue;
57:
58:         /* Pomjeri indikator pozicije na navedeni element. */
59:
60:         if ( (fseek(fp, (offset*sizeof(int)), SEEK_SET)) != 0)
61:         {
62:             fprintf(stderr, "\nError using fseek().");
63:             exit(1);
64:         }
65:
66:         /* Procitaj u jednom integer-u. */
67:
68:         fread(&data, sizeof(int), 1, fp);
69:
70:         printf("\nElement %ld has value %d.", offset, data);
71:     }
72:
73:     fclose(fp);
74:     return(0);
75: }
```

```
Enter element to read, 0-49, -1 to quit: 5
Element 5 has value 50.
Enter element to read, 0-49, -1 to quit: 6
Element 6 has value 60.
Enter element to read, 0-49, -1 to quit: 49
Element 49 has value 490.
Enter element to read, 0-49, -1 to quit: 1
Element 1 has value 10.
Enter element to read, 0-49, -1 to quit: 0
Element 0 has value 0.
Enter element to read, 0-49, -1 to quit: -1
```

ANALIZA: Linije **14** do **35** su slične Listingu **16.5**.
Linije **16** i **17** inicijaliziraju niz nazvan **data** sa **50** varijabli tipa **integer**. Vrijednost smještena u svaki element niza je jednaka sa index-om puta **10**. Onda se niz zapisuje u binarni file. Nazvan **RANDOM.DAT**.
Vi znate da je binarni, jer je file otvoren sa "**wb**" mode-om u liniji **21**.

Linija **39** ponovo otvara file u binarnom mode-u za čitanje prije nego što ode u beskonačnu **while** petlju.
while petlja prompt-a korisnika da unese broj elemenata niza koje oni žele da čitaju. Primjetite da linije **53** do **56** provjeravaju da vide da je uneseni element u opsegu file-a.

→ Da li vam C dopušta da čitate element koji je izvan kraja file-a???
DA. Kao da idete iza kraja niza, sa vrijednostima, **C** vam dopušta da čitate nakon kraja file-a. Ako čitate nakon kraja (ili prije početka), vaši rezultati se nepredvidivi.

Najbolje je uvijek provjeriti šta radite (kao što linije **53** do **56** u ovom listingu rade).

Nakon što unesete element za nalaženje, linija **60** skače na odgovarajući offset sa pozivom na **fseek()**. Zato što se **SEEK_SET** koristi, potraga (seek) se radi od početka file-a. Primjetite da distanca u file-u nije samo offset, nego offset pomnožen sa veličinom elementa koji se čita.
Linija **68** onda čita vrijednost, i linija **70** ga printa.

→→→→ Detektovanje (otkrivanje) kraja file-a (Detecting the End of a File)

Nekad znate tačno koliko je file dugačak, tako da nema potrebe da budete u mogućnosti da otkrijete (detect) kraj file-a (**e**nd **o**f **f**ile).
Na primjer, ako koristite **fwrite()** da snimite **100**-elementni **integer** niz, vi znate da je file **200** byte-a dugačak (pretpostavljajući **2**-byte-ne **integer**e). U drugim slučajevima, ipak, vi ne znate koliko je file dugačak, ali želite da čitate podatke sa file-a, počinjući na početku i nastavljajući do kraja. Postoje dva načina kako da detektujete kraj file-a (**e**nd-**o**f-**f**ile):

Kada čitate iz text-mode file-a, karakter-po-karakter, vi možete potražiti kraj-file-a (**e**nd-**o**f-**f**ile) karakter. Simbolična konstanta **EOF** je definisana u **STDIO.H** kao **–1**, vrijednost koja je nikad korištena od strane "realnog" karaktera.
Kada karakterna ulazna funkcija pročita **EOF** sa **text**-mode stream-a, možete biti sigurni da ste dostigli kraj file-a.
Na primjer, možete napisati sljedeće:

```
while ( (c = fgetc( fp )) != EOF )
```

Sa **binarnim**-mode stream-om, vi **NE** možete detektovati kraj-file-a gledajući na **–1**, zato što byte podatka u binarnom stream-u može imati tu vrijednost, što bi rezultiralo sa preranim krajem ulaza (input-a). Umjesto toga, vi možete koristiti bibliotečnu funkciju →→→ **feof()** ←←←, koja može biti korištena i za **binarni** i za **text**-mode file-ove:

```
int feof(FILE *fp);
```

Argument **fp** je **FILE** pointer, vraćen od **fopen()** kada je file otvoren.
Funkcija **feof()** vraća **0** ako kraj file-a **fp** nije dostignut, ili **nenula** vrijednost ako je dostignut kraj-file-a (**e**nd-**o**f-**f**ile).

Ako poziv na **feof()** detektuje kraj-file-a, nikakve daljnje operacije nisu dozvoljene sve dok se ne odradi **rewind()**, **fseek()** se poziva, ili se file zatvori i ponovo otvori.

Listing **16.7** demonstrira upotrebu **feof()**-a. Kada ste promptani za ime file-a, unesite ime bilo kojeg text file-a – na primjer, jednog od vaših **C**-ovih file-ova sa izvornim koodom, ili file zaglavlja kao što je **STDIO.H**.
Samo budite sigurni da je file u tekućem direktoriju, ili unesite put (path) kao dio imena file-a. Program čita file jednu po jednu liniju, prikazujući svaku liniju na **stdout**, dok **feof()** ne detektuje kraj-file-a.

Listing 16.7. Korištenje feof() da detektuje kraj-file-a (end of a file).

```
1:  /* Detektovanje end-of-file-a. */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:
5:  #define BUFSIZE 100
6:
7:  main()
8:  {
9:      char buf[BUFSIZE];
10:     char filename[60];
11:     FILE *fp;
12:
13:     puts("Enter name of text file to display: ");
14:     gets(filename);
```

```
15:
16:     /* Otvori file za chitanje. */
17:     if ( (fp = fopen(filename, "r")) == NULL)
18:     {
19:         fprintf(stderr, "Error opening file.");
20:         exit(1);
21:     }
22:
23:     /* Ako kraj file-a nije dostignut, chitaj liniju i prikazi je. */
24:
25:     while ( !feof(fp) )
26:     {
27:         fgets(buf, BUFSIZE, fp);
28:         printf("%s",buf);
29:     }
30:
31:     fclose(fp);
32:     return(0);
33: }
            Enter name of text file to display:
            hello.c
            #include <stdio.h>
            main()
            {
                printf("Hello, world.");
                return(0);
            }
```

ANALIZA: while petlja u ovom programu (linije **25** do **29**) je tipična **while**, korištena u komplexnijim programima da obavlja sekvencijalno procesiranje. Sve dok nije dostignut kraj-file-a, kood unutar **while** iskaza (linije **27** i **28**) nastavlja da se izvršava ponavljano (repeatedly). Kada poziv do **feof()** vrati **nenula** vrijednost, petlja završava, file je zatvoren, i program završava.

PROVJERITE vašu poziciju unutar file-a tako da ne čitate nakon kraja ili prije početka file-a.
KORISTITE ili **rewind()** ili **fseek(fp, SEEK_SET, 0)** da resetujete poziciju file-a na početak file-a.
KORISTITE feof() da provjeri kraj-file-a kada radite sa binarnim file-ovima.
NE koristite **EOF** sa binarnim file-ovima.

→→→→ Funkcije za upravljanje file-om

Pojam upravljanje file-a se odnosi na rad sa **postojećim** file-ovima → ne sa čitanjem ili pisanjem na njih, već brisanjem, reimenovanjem, ili njihovim kopiranjem.

C-ova standardna biblioteka sadrži funkcije za brisanje i reimenovanje file-ova, i također možete napisati vaš file-kopirajući program.

→→→ Brisanje file-a (Deleting a File)

Da izbrišete file, vi koristite bibliotečnu funkciju →→→ **remove()** ←←←.

Njen prototip je u **STDIO.H**, kako slijedi:

```
int remove( const char *filename );
```

Varijabla ***filename** je pointer na ime file-a koji se briše. Navedeni file ne mora biti otvoren.
Ako file postoji, on je izbrisan (baš kao što ste koristili **DEL** komandu u DOS promptu ili **rm** komandu u UNIX-u), i **remove()** vraća **0**.
Ako file ne postoji, ako je read-only (samo za čitanje), ako nemate dovoljna pristupna prava, ili ako se neka greška desi, **remove()** vraća **–1**.

Listing **16.8** demonstrira korištenje **remove()**-a. Budite oprezni: ako vi **remove()** file, on je izgubljen zauvijek:

Listing 16.8. Korištenje remove() funkcije za brisanje disk file-a.

```
1:  /* Demonstrira remove() funkciju. */
2:
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:      char filename[80];
8:
9:      printf("Enter the filename to delete: ");
10:     gets(filename);
11:
12:     if ( remove(filename) == 0)
13:         printf("The file %s has been deleted.\n", filename);
14:     else
15:         fprintf(stderr, "Error deleting the file %s.\n", filename);
16:     return(0);
17: }
            Enter the filename to delete: *.bak
            Error deleting the file *.bak.
            Enter the filename to delete: list1414.bak
            The file list1414.bak has been deleted.
```

ANALIZA: Ovaj program prompta korisnika u liniji **9** za ime file-a koji će biti izbrisan.
Linija **12** onda poziva **remove()** da izbriše uneseni file.
Ako je vraćena vrijednost **0**, file je uklonjen (izbrisan), i poruka se prikazuje koja potvrđuje ovu činjenicu.
Ako je vraćena vrijednost **nenula**, desila se greška, i file nije uklonjen (izbrisan).

→→→ Reimenovanje file-a (Renaming a File)

→→→ **rename()** ←←← funkcija mijenja ime postojećeg disk file-a.

Prototip funkcije, u **STDIO.H**, je kako slijedi:

```
int rename( const char *oldname, const char *newname );
```

Ime file-a, na koji pokazuje **oldname** i **newname** slijedi pravila koja su data ranije u ovom poglavlju. Jedina restrikcija (ograničenje) je da oba imena koraju da se odnose na isti disk drive; vi ne možete preimenovati file na različitim disk drive-ovima.

Funkcija **rename()** vraća **0** ako je uspješno, ili **–1** ako se desila greška.

Greška može biti prouzrokovana u sljedećim uslovima (između ostalih):

- File ***oldname*** ne postoji.
- File sa imenom ***newname*** već postoji.
- Vi pokušavate da preimenujete na drugi disk.

Listing **16.9** demonstrira korištenje **rename()**-a.

Listing 16.9. Korištenje rename() da promjenite ime disk file-a.

```
1:  /* Upotreba rename() da promjenite ime file-a. */
2:
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:      char oldname[80], newname[80];
8:
9:      printf("Enter current filename: ");
10:     gets(oldname);
11:     printf("Enter new name for file: ");
12:     gets(newname);
13:
14:     if ( rename( oldname, newname ) == 0 )
15:         printf("%s has been renamed %s.\n", oldname, newname);
16:     else
17:         fprintf(stderr, "An error has occurred renaming %s.\n",
oldname);
18:     return(0);
19: }
            Enter current filename: list1609.c
            Enter new name for file: rename.c
            list1609.c has been renamed rename.c.
```

ANALIZA: Listing **16.9** pokazuje kako moćan **C** može biti. Sa samo **18** linija kooda, ovaj program zamjenjuje komandu operativnog sistema, i puno je prijateljskija funkcija.
Linija **9** prompta za ime file-a koji će biti preimenovan.
Linija **11** prompta za novo ime file-a.
Poziv na **rename()** funkciju je umotana u **if** iskaz u liniji **14**.
if iskaz provjerava, da su uvjeri, da je reimenovanje file-a obavljeno korektno. Ako jeste, linija **15** printa potvrdnu poruku; u drugom slučaju, linija **17** printa poruku koja kaže da se desila greška.

→→→ Kopiranje file-a (Copying a File)

Često je potrebno da se napravi kopija file-a → istovjetan duplikat sa drugačijim imenom (ili sa istim imenom na različitom drive-u ili direktoriju). U DOS-u, ovo radite sa **COPY** komandom, i drugi operativni sistemi imaju ekvivalente.

→ Kako kopirate file u C-u???
Ne postoji biliotečna funkcija, tako da treba sami da napišete svoju.

Ovo može zvučati malo komplikovano, ali je u stvari sasvim jednostavna zahvaljujući **C**-ovoj upotrebi stream-ova za ulaz i izlaz.
Evo koraka koje treba da pratite:

1. Otvorite izvorni file za čitanje u **binarnom** mode-u. (Koristeći **binarni** mode osigurava da funkcija može kopirati sve vrste file-ova, ne samo **text**ualne file-ove).
2. Otvorite odredišni file za pisanje u **binarnom** mode-u.
3. Čitaj karakter iz izvornog file-a. Zapamtite, kada je prvi file otvoren, pointer je na početku file-a, tako da nema potrebe da explicitno pozicionirate pointer.
4. Ako funkcija **feof()** indicira da ste dostigli kraj-file-a u izvornom file-u, završili ste i možete zatvoriti oba file-a i vratiti se pozivajućem programu.
5. Ako niste dostigli kraj-file-a, piši karakter u odredišni file, i onda peetljaj nazad na korak **3**.

Listing **16.10** sadrži funkciju, **copy_file()**, kojoj su prosljeđena imena izvornog i odredišnog file-a, i onda obavlja operaciju kopiranja kako je rečeno u prethodnim koracima.
Ako se desi greška pri otvaranju nekog od file-ova, funkcija ne pokušava operaciju kopiranja i vraća **–1** pozivajućem programu.
Kada je operacija kopiranja završena, program zatvara oba file-a i vraća **0**.

Listing 16.10. Funkcija koja kopira file.

```
1:  /* Kopiranje file-a. */
2:
3:  #include <stdio.h>
4:
5:  int file_copy( char *oldname, char *newname );
6:
7:  main()
8:  {
9:      char source[80], destination[80];
10:
11:     /* Uzmi izvorna i odredishna imena. */
12:
13:     printf("\nEnter source file: ");
14:     gets(source);
15:     printf("\nEnter destination file: ");
16:     gets(destination);
17:
18:     if ( file_copy( source, destination ) == 0 )
19:         puts("Copy operation successful");
20:     else
21:         fprintf(stderr, "Error during copy operation");
22:     return(0);
23: }
24: int file_copy( char *oldname, char *newname )
25: {
26:     FILE *fold, *fnew;
27:     int c;
28:
29:     /* Otvori izvorni file za chitanje u binarnom mode-u. */
30:
31:     if ( ( fold = fopen( oldname, "rb" ) ) == NULL )
32:         return -1;
33:
34:     /* Otvori odredishni file za pisanje u binarnom mode-u. */
35:
36:     if ( ( fnew = fopen( newname, "wb" ) ) == NULL  )
37:     {
38:         fclose ( fold );
39:         return -1;
40:     }
41:
42:     /* Chitaj jedan byte (at a time) sa izvora; ako kraj-file-a */
43:     /* nije dostignut, pishi byte na */
44:     /* odredishte. */
45:
46:     while (1)
47:     {
48:         c = fgetc( fold );
49:
50:         if ( !feof( fold ) )
51:             fputc( c, fnew );
52:         else
53:             break;
```

```
54:     }
55:
56:     fclose ( fnew );
57:     fclose ( fold );
58:
59:     return 0;
60: }
       Enter source file: list1610.c
       Enter destination file: tmpfile.c
       Copy operation successful
```

ANALIZA: Funkcija **copy_file()** radi perfektno dobro, dopuštajući vam da kopirate bilo šta od malog textualnog file-a do velikih programskih file-ova.
Ipak, ona ima ograničenja. Ako odredišni file već postoji, funkcija ga briše bez pitanja.
Dobra programerska vježba će vam biti da modifkujete **copy_file()** funkciju da provjerite da li odredišni file već postoji, i onda pitate korisnika da li stari file treba biti prepisan (overwritten).

main() u Listing **16.10** bi trebao izgledati poznato. Skoro je identičan **main()**-u u Listingu **16.9**, sa izuzetkom u liniji **14**. Umjesto **rename()**, ova funkcija koristi **copy()**.

Zato što **C** nema funkciju kopiranja, linije **24** do **60** kreiraju funkciju kopiranja.
Linije **31** i **32** otvaraju izvorni file, **fold**, u binarnom čitajućem mode-u.
Linije **36** do **40** otvaraju odredišni file, **fnew**, u binarnom čitajućem mode-u.
Primjetite da linija **38** zatvara izvorni file, **fold**.
Linija **50** provjerava da vidi da li je kraj-file-a marker pročitan. Ako je dostignut kraj-file-a, **break** iskaz se izvršava s ciljem da izađe iz **while** petlje. Ako kraj-file-a nije dostignut, karakter je piše na odredišni file, **fnew**.
Linije **56** i **57** zatvaraju dva file-a prije vraćanja u **main()**.

→→→→ Korištenje privremenih file-ova (Using Temporary Files)

Neki programi koriste jedan ili više privremenih file-ova tokom izvršenja.
Privremeni (temporary) **file** je file koji je kreiran od strane programa, korišten za neku svrhu tokom izvršenja programa, i onda izbrisan prije terminiranja programa.

Kada kreirate privremeni file, vas baš briga koje je njegovo ime, zato što se on briše. Sve što je potrebno je da koristite ime koje već nije u upotrebi od strane nekog drugog file-a.
C-ova standardna biblioteka uključuje funkciju →→→ **tmpnam()** ←←← koja kreira validno ime file-a, koje ne dolazi u konflikt sa ostalim postojećim file-om.

Njen prototip u **STDIO.H**, je kako slijedi:

```
ar *tmpnam(char *s);
```

Argument **s** mora biti pointer na buffer dovoljno velik da drži ime file-a.
Vi također možete prosljediti **null** pointer (**NULL**), u čijem je slučaju privremeno ime smješteno u buffer unutar **tmpnam()**, i funkcija vraća pointer na taj buffer.

Listing **16.11** demonstrira obje metode korištenja **tmpnam()**-a za kreiranje privremenih imena file-ova.
Listing 16.11. Korištenje tmpnam() za kreiranje privremenih imena file-ova.

```
1:  /* Demonstracija provremenih (temporary filenames) imena file-ova. */
2:
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:      char buffer[10], *c;
```

```
8:
9:      /* Uzmi privremeno ime u definisani buffer. */
10:
11:     tmpnam(buffer);
12:
13:     /* Uzmi drugo ime, ovaj put u funkcijski */
14:     /* interni buffer. */
15:
16:     c = tmpnam(NULL);
17:
18:     /* Prikazi imena. */
19:
20:     printf("Temporary name 1: %s", buffer);
21:     printf("\nTemporary name 2: %s\n", c);
22: }
            Temporary name 1: TMP1.$$$
            Temporary name 2: TMP2.$$$
```

ANALIZA: Privremena imena generisana na vašem sistemu će vjerovatno biti drugačija od ovih. Ovaj program samo generiše i printa imena privremenih file-ova; on u stvari ne kreira nikakve file-ove.

Linija **11** smješta privremeno ime u karakterni niz, **buffer**.
Linija **16** pridružuje karakterni pointer na ime vraćeno od **tmonam()** na **c**.
Vaš program bi morao koristiti generisano ime da otvori privremeni file i onda izbriše file prije nego što se izvršenje programa terminira.

Sljedeći fregment kooda ilustruje:

```
char tempname[80];
FILE *tmpfile;
tmpnam(tempname);
tmpfile = fopen(tempname, "w");  /* Koristite prigodan mode */
fclose(tmpfile);
remove(tempname);
```

NE brišite file koji bi vam možda trebao ponovo.
NE pokušavajte da reimenujete file-ove kroz (across) drive-ove.
NE zaboravite da uklonite (izbrišete) privremene file-ove koje kreirate. Oni se ne brišu automatski.

Sažetak (Summary)

U ovom poglavlju ste naučili kako **C** koristi disk file-ove.
C tretira disk file-ove kao stream (sekvenca karaktera), baš kao i predefinisani stream-ovi koje ste naučili Dana 14. Stream, pridružen disk file-u mora biti otvoren prije nego što se može koristiti, i morate ga zatvoriti nakon upotrebe. Stream disk file-a je otvoren ili u text ili u binarnom mode-u.

Nakon što je disk file otvoren, vi možete čitati podatke sa file-a u vaš program, pisati podatke iz programa u file, ili obadvoje. Postoje tri uopštena U/I-na tipa file-ova: **formatirani**, **karakterni**, i **direktni**. Svaki tip U/I je najbolje korišten za određene tipove smještanja podataka i potraživanja zadataka.

Svaki otvoreni disk file ima pozicioni indikator file-a kojem je pridružen. Ovaj indikator navodi (specificira) poziciju u file-u, mjeren kao broj byte-ova od početka file-a, gdje se kasnije dešavaju operacije čitanja i pisanja. Sa nekim tipovima pritupa file-u, pozicioni indikator je update-ovan

automatski, i vi ne morate brinuti oko toga. Za nasumičan (random) pristup file-u, **C** standardna biblioteka obezbjeđuje funkcije za manipulaciju pozicionim indikatorom.

Konačno (I na kraju), **C** obezbjeđuje neke elementarne (osnovne) upravljačke funkcije (menagement functions), koje vam dopuštaju da brišete ili reimenujete disk file-ove. U ovom poglavlju, vi ste razvili vlastitu funkciju za kopiranje file-a.

P&O
P Mogu li koristiti drive-ove i puteve (drives and paths) sa imenima file-ova kada koristim remove(), rename(), fopen(), i druge file funkcije???
O DA. Vi možete koristiti puno ime file-a sa putem i drive-om ili saamo ime file-a. Ako koristite saamo ime file-a, funkcija gleda (looks) na file u tekućem direktoriju. Zapamtite, kada koristite byckslah(\), vi treba da koristite escape sequence. Takođe zapamtite da UNIX koristi forward slash (*/*) kao separator direktorija.

P Da li mogu čitati nakon kraja-file-a???
O DA. Vi takođe možete čitati prije početka file-a. Rezultati od takvog čitanja mogu biti katastrofalni. Čitanje file-ova je kao i rad sa nizovima. Vi gledate na offset unutar memorije. Ako koristite **fseek()**, vi trebate provjeriti da se uvjerite da ne idete preko kraja-file-a.

P Šta se desi ako ne zatvorim file???
O Dobra je programerska praxa da zatvorite bilo koji file koji otvorite. Po default-u, file bi trebao da se zatvori nakon što program izađe (exits); ipak, nikad se ne trebate osloniti na ovo. Ako file nije zatvoren, možda nećete biti u mogućnosti da mu pristupite kasnije, zato što će operativni sistem misliti da je file već u upotrebi.

P Da li mogu čitati file sekvencijalno sa nasumično-pristupnom funkcijom???
O Kada čitate file sekvencijalno, nema potrebe da koristite funkcije kao što je **fseek()**. Zato što je file pointer postavljen na posljednju poziciju koju zauzima, on je uvijek tamo gdje ga želite za sekvencijalno čitanje. Vi možete koristiti **fseek()** za čitanje file-a sekvencijalno; ipak, ne dobijate ništa s tim.

Vježbe:

1. Napišite kood da zatvorite sve file stream-ove.
2. Pokažite dva različita načina da resetujete file-ov pozicioni pointer na početak file-a.
3. BUG BUSTER: Da li nešto nije u redu sa sljedećim?

```
FILE *fp;
int c;
if ( ( fp = fopen( oldname, "rb" ) ) == NULL )
     return -1;
while (( c = fgetc( fp)) != EOF )
     fprintf( stdout, "%c", c );
fclose ( fp );
```

4. Napišite program koji prikazuje file na-ekran.
5. Napišite program koji otvara file i printa ga na printer (**stdprn**). Program bi trebao printati samo **55** linija po strani.
6. Modifikujte program u vježbi **5** da printa zaglavlja (headings) na svakoj strani. Zaglavlja bi trebala sadržavati imena file-ova i broj stranice.
7. Napišite program koji otvara file i broji broj karaktera. Program bi takođe trebao printati broj karaktera kada završi.
8. Napišite program koji otvara postojeći textualni file i kopira ga u novi textualni file sa svim malim slovima pretvorenim u velika slova, i svim ostalim karakterima neprojenjenim.
9. Napišite program koji otvara bilo koji disk file, čita ga u **128**-byte-ne blokove, i prikazuje sadržaj svakog bloka na-ekran u obadva, hexadecimalnom i ASCII formatu.
10. Napišite funkciju koja otvara novi privremeni file sa navedenim mode-om. Svi privremeni file-ovi kreirani od ove funkcije bi trebli automatski biti zatvoreni i izbrisani kada se program terminira. (Hint: Koristite **atexit()** bibliotečnu funkciju).

LEKCIJA 17: → Manipulating Strings ←

Textualni podaci, koje **C** smješta u stringove, su bitan dio puno programa. Do sada, vi ste naučili kako **C** program smješta stringove i kako da unesete ili prosljedite (input / output) stringove. **C** takođe daje razne funkcije za ostale tipove manipulacije stringovima.
Danas ćete naučiti:

- Kako da odredite dužinu stringa
- Kako da kopirate i združujete stringove
- O funkcijama koje upoređuju stringove
- Kako da tražite stringove
- Kako da konvertujete stringove
- Kako da testirate karaktere

→→→ Dužina i smještanje stringa

String je sekvenca karaktera, na čiji početak pokazuje pointer i čiji je kraj markiran sa **null** karakterom **\0**.
Nekad vam je potrebno da znate dužinu stringa (broj karaktera između početka i kraja stringa). Ova dužina se dobija pomoću bibliotečne funkcije →→→ **strlen()** ←←←.

Njen prototip, u **STRING.H**, je:

```
size_t strlen(char *str);
```

Možda vas zbunjuje **size_t** povratni tip. Ovaj tip je definisan u **STRING.H** kao **unsigned**, tako da funkcija **strlen()** vraća kao **unsigned integer**.
size_t tip se koristi sa puno string funkcija. Samo zapamtite da on znači **unsigned**.

Argument koji je prosljeđen do **strlen**, je pointer na string od čijeg želite da znate dužinu. Funkcija **strlen()** vraća broj karaktera između **str** i sljedećeg **null** karaktera, ne računajući **null** karakter.

Listing **17.1** demonstrira **strlen()**.
Listing 17.1. Korištenje strlen() funkcije za određivanje dužine stringa.

```
1:  /* Upotreba strlen() funkcije. */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  main()
7:  {
8:      size_t length;
9:      char buf[80];
10:
11:     while (1)
12:     {
13:         puts("\nEnter a line of text; a blank line terminates.");
14:         gets(buf);
15:
16:         length = strlen(buf);
17:
18:         if (length != 0)
19:             printf("\nThat line is %u characters long.", length);
20:         else
21:             break;
22:     }
23:      return(0);
24: }
        Enter a line of text; a blank line terminates.
```

```
Just do it!
That line is 11 characters long.
Enter a line of text; a blank line terminates.
```

ANALIZA: Ovaj program daje malo više od demonstracije upotrebe **strlen()**.
Linije **13** i **14** prikazuju poruku i dobijaju string nazvan **buf**.
Linija **16** koristi **strlen()** da pridruži dužinu od **buf** varijabli **length**.
Linija **18** provjerava da li je string prazan, provjeravajući dužinu od **0**. Ako string nije prazan, linija **19** printa veličinu stringa.

→→→ Kopiranje stringova

C-ova biblioteka ima tri funkcije za kopiranje stringova. Zbog načina na koji **C** radi sa stringovima, vi jednostavno možete pridružiti jedan string drugom, kao što možete u nekim kompjuterskim jezicima. Vi morate kopirati izvorni string sa njegove lokacije u memoriji u memorijsku lokaciju odredišnog stringa.

String-kopirajuće funkcije su →→→ **strcpy()**, **strncpy()**, i **strdup()** ←←←.

Sve string-kopirajuće funkcije zahtjevaju **STRING.H** file zaglavlja.

→→→ strcpy() ←←← Funkcija

Bibliotečna funkcija →→→ **strcpy()** ←←← kopira cijeli string na drugu memorijsku lokaciju.
Njen prototip je kako slijedi:

```
char *strcpy( char *destination, char *source );
```

Funkcija **strcpy()** kopira string (uključujući terminirajući **null** karakter **\0**) na koji pokazuje izvor na lokaciju na koju pokazuje odredište. Povratna vrijednost je pointer na novi string, odredište.

Kada koristite **strcpy()**, vi morate prvo alocirati smještajni prostor za odredišni string.
Funkcija nema načina kako da zna da li odredište pokazuje na alocirani prostor. Ako prostor nije alociran, funkcija prepisuje **strlen**(izvor) byte-e memorije, s početkom u odredištu; ovo može prouzrokovati nepredvidive probleme.

Upotreba **strcpy()** je ilustrovana u Listingu **17.2**.

NAPOMENA: Kada program koristi **malloc()** da alocira memoriju, kao u Listingu **17.2**, dobra programerska praxa zahtjeva korištenje **free()** funkcije da oslobodi memoriju kada program završi s njom. (o **free()** detaljnije Dana 20).

Listing 17.2. Prije korištenja strcpy(), vi morate alocirati smještajni prostor za odredišni string.

```
1:  /* Demonstrira  strcpy(). */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  char source[] = "The source string.";
7:
8:  main()
9:  {
10:     char dest1[80];
11:     char *dest2, *dest3;
12:
13:     printf("\nsource: %s", source );
```

```
14:
15:     /* Kopiranje na  dest1 je okay, zato shto dest1 pokazuje na  */
16:     /* 80 byte-a alociranog prostora. */
17:
18:     strcpy(dest1, source);
19:     printf("\ndest1:  %s", dest1);
20:
21:     /* Da kopirate dest2, morate alocirati prostor. */
22:
23:     dest2 = (char *)malloc(strlen(source) +1);
24:     strcpy(dest2, source);
25:     printf("\ndest2:  %s\n", dest2);
26:
27:     /* Kopiranje bez alociranja odredishnog prostora je ne-ne. */
28:     /* Sljedece bi moglo prouzrokovali ozbiljne probleme. */
29:
30:     /* strcpy(dest3, source); */
31:     return(0);
32: }
            source: The source string.
            dest1:  The source string.
            dest2:  The source string.
```

ANALIZA: Ovaj program demonstrira kopiranje stringova i na karakterne nizove, kao što je **dest1** (deklarisan u liniji **10**) i na karakterne pointere, kao što je **dest2** (deklarisan sa **dest3** u liniji **11**). Linija **13** printa oreginalni izvorni string. Ovaj string se onda kopira na **dest1** sa **strcpy()** u liniji **18**. Linija **24** kopira izvor na **dest2**.
I **dest1** i **dest2** se printaju da pokažu da je funkcija bila uspješna.
Primjetite da linija **23** alocira dovoljnu količinu prostora za **dest2** sa **malloc()** funkcijom. Ako vi kopirate string na karakterni pointer pri nealociranoj memoriji, dobićete nepredvidive rezultate.

→→→ strncpy() ←←← Funkcija

→→→ **strncpy()** ←←← funkcija je slična sa **strcpy()**, izuzev što vam **strncpy()** dopušta da navedete koliko karaktera da kopirate.
Njen prototip je:

```
char *strncpy(char *destination, char *source, size_t n);
```

Argument **destination** (odredište) i **source** (izvor) su pointeri na odredište i izvor stringova. Funkcija kopira, najviše, prvih **n** karaktera iz izvora na odredište. Ako je izvor kraći od **n** karaktera, dovoljno **null** karaktera se dodaje na kraj izvora, što čini ukupno **n** karaktera kopiranih na odredište. Ako je izvor duži od **n** karaktera, ne dodaju se nikakvi terminirajući **\0** karakteri na odredište. Povratna vrijednost funkcije je odredište.

Listing **17.3** demonstrira upotrebu **strncpy()**.

Listing 17.3. strncpy() funkcija.

```
1:  /* Koristenje  strncpy() funkcije. */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  char dest[] = ".........................";
7:  char source[] = "abcdefghijklmnopqrstuvwxyz";
8:
9:  main()
10: {
11:     size_t n;
```

```
12:
13:     while (1)
14:     {
15:         puts("Enter the number of characters to copy (1-26)");
16:         scanf("%d", &n);
17:
18:         if (n > 0 && n< 27)
19:             break;
20:     }
21:
22:     printf("\nBefore strncpy destination = %s", dest);
23:
24:     strncpy(dest, source, n);
25:
26:     printf("\nAfter strncpy destination = %s\n", dest);
27:     return(0);
28: }
       Enter the number of characters to copy (1-26)
       15
       Before strncpy destination = ........................
       After strncpy destination = abcdefghijklmno...........
```

ANALIZA: S dodatkom demonstracije **strncpy()** funkcije, ovaj program također ilustruje efektivan način da se osigura da je samo ispravna informacija unesena od strane korisnika.
Linije **13** do **20** sadrže **while** petlju koja prompta korisnika za broj između **1** i **26**.
Petlja se nastavlja dok se ne unese validna vrijednost, tako da program ne može da nastavi dok korisnik ne unese validnu vrijednost.
Kada je broj između **1** i **26** unesen, linija **22** printa oreginalnu vrijednost od **dest**, i linija **26** printa konačnu (finalnu) vrijednost od **dest**.

UPOZORENJE: Budite sigurni da broj karaktera kopiranih ne pređe alociranu veličinu odredišta.

→→→ strdup() ←←← Funkcija

Bibliotečna funkcija **strdup()** je slična sa **strcpy()**, izuzev što **strdup()** obavlja vlastitu alokaciju memorije za odredišni string sa pozivom na **malloc()**.
Ona radi ono što ste uradili u Listingu **17.2**, alocira prostor sa **malloc()** i onda poziva **strcpy()**.

Prototip za **strdup()** je:

```
char *strdup( char *source );
```

Argument **source** (izvor) je pointer na izvorni string. Funkcija vraća pointer na odredišni string → prostor alociran od **malloc()** ← ili **NULL** ako potrebna memorija nije mogla biti alocirana.

Listing **17.4** demonstrira korištenje **strdup()**.
Primjetite da **strdup()** nije **ANSI**-standardna funkcija. Ona je uključena u **C** biblioteke od Microsoft-a, Borland-a i Symantec-a, ali ne mora biti prisutna (ili može biti različita) u ostalim **C** kompajlerima.

Listing 17.4. Korištenje strdup() za kopiranje stringa sa automatskom alokacijom memorije.

```
1:  /* strdup() function. */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  char source[] = "The source string.";
```

```
7:
8:  main()
9:  {
10:     char *dest;
11:
12:     if ( (dest = strdup(source)) == NULL)
13:     {
14:         fprintf(stderr, "Error allocating memory.");
15:         exit(1);
16:     }
17:
18:     printf("The destination = %s\n", dest);
19:     return(0);
20: }
               The destination = The source string.
```

ANALIZA: U ovom listingu, **strdup()** alocira prigodnu memoriju za **dest**.
Ona onda pravi kopiju prosljeđenog stringa, **source** (izvora).
Linija **18** printa dupliciran string.

→→→→ Concatenating Strings (spajanje stringova)

Ako niste upoznati sa pojmom ***concatenation,*** možda se pitate, "Šta je to ba???" i "Da li je legalna???".
Paa, on znači sjedinjavanje dva stringa → **da uzme jedan string na kraj drugog** ← i, u većini slučajeva, jeste legalna.

C-ova strandardna biblioteka sadrži dvije string concatenation funkcije →→→ **strcat()** i **strncat** ←←← obadvije zahtajvaju **STRING.H** file zaglavlja.

→→→ strcat() ←←← Funkcija

Prototip za **strcat()** je:

```
char *strcat(char *str1, char *str2);
```

Funkcija dodaje kopiju od **str2** na kraj od **str1**, pomjerajući terminirajući **null** karakter na kraj novog stringa.
Vi morate alocirati dovoljno prostora za **str1** da drži rezultirajući string.
Povratna vrijednost od **strcat()** je pointer na **str1**.

Listing **17.5** demonstrira **strcat()**.
Listing 17.5. Korištenje strcat() da se concatenate (spoje) stringovi.

```
1:  /*  strcat() function. */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  char str1[27] = "a";
7:  char str2[2];
8:
9:  main()
10: {
11:     int n;
12:
13:     /* Stavi null karakter na kraj od str2[]. */
14:
```

```
15:     str2[1] = `\0';
16:
17:     for (n = 98; n< 123; n++)
18:     {
19:         str2[0] = n;
20:         strcat(str1, str2);
21:         puts(str1);
22:     }
23:      return(0);
24: }
```

```
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
abcdefghijkl
abcdefghijklm
abcdefghijklmn
abcdefghijklmno
abcdefghijklmnop
abcdefghijklmnopq
abcdefghijklmnopqr
abcdefghijklmnopqrs
abcdefghijklmnopqrst
abcdefghijklmnopqrstu
abcdefghijklmnopqrstuv
abcdefghijklmnopqrstuvw
abcdefghijklmnopqrstuvwx
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxyz
```

ANALIZA: ASCII kood za slova od **b** do **z** su **98** do **122**. Ovaj program koristi ove **ASCII** koodove u svojoj demonstraciji **strcat()**-a.
for petlja u linijama **17** do **22** pridružuje ove vrijednosti po prolazu (in turn) do **str2[0]**.
Zato što je **str2[1]** već **null** karakter (linija **15**), efekat je da se pridruže stringovi "**b**", i "**c**", i tako dalje do **str2**.
Svaki od ovih stringova se concatenated (spaja) sa **str1** (linija **20**), i onda se **str1** prikazuje na-ekran (linija **21**).

→→→ strncat() ←←← Funkcija

Bibliotečna funkcija **strncat()** takođe obavlja string concetenation-aciju, ali vam dozvoljava da navedete koliko karaktera iz izvornog stringa se dodaje na kraj odredišnog stringa.

Prototip je:

```
char *strncat(char *str1, char *str2, size_t n);
```

Ako **str2** sadrži više od **n** karaktera, onda se prih **n** karaktera dodaje (zalijepi) na kraj od **str1**.
Ako **str2** sadrži manje od **n** karaktera, onda se sav **str2** dodaje (zalijepi) na kraj od **str1**.
U svakom slučaju, terminirajući **null** karakter se dodaje na kraj rezultirajućeg stringa.
Vi morate alocirati dovoljno prostora za **str1** da drži rezultirajući string. Funkcija vraća pointer na **str1**.

Listing **17.6** koristi **strncat()** da producira isti izlaz kao i Listing **17.5**.

Listing 17.6. Upotreba strncat() funkcije da concatenate string-ove.

```
1:  /* strncat() funkcija. */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  char str2[] = "abcdefghijklmnopqrstuvwxyz";
7:
8:  main()
9:  {
10:     char str1[27];
11:     int n;
12:
13:     for (n=1; n< 27; n++)
14:     {
15:         strcpy(str1, "");
16:         strncat(str1, str2, n);
17:         puts(str1);
18:     }
19: }
```

```
a
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
abcdefghijkl
abcdefghijklm
abcdefghijklmn
abcdefghijklmno
abcdefghijklmnop
abcdefghijklmnopq
abcdefghijklmnopqr
abcdefghijklmnopqrs
abcdefghijklmnopqrst
abcdefghijklmnopqrstu
abcdefghijklmnopqrstuv
abcdefghijklmnopqrstuvw
abcdefghijklmnopqrstuvwx
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxyz
```

ANALIZA: Možda se pitate koja je svrha linije **15**, **strcpy(str1, “”);**.
Ova linija kopira na **str1**, prazan string koji se sastoji od samo **null** karaktera. Rezultat je da je prvi karakter u **str1** – **str1[0]** – postavljen (jednako sa) na **0** (**null** karakter). Ista stvar bi se mogla uraditi sa iskazima **str1[0] = 0;** ili **str1[0] = ‘\0’;**.

→→→ Upoređujući stringovi (Comparing Strings)

Stringovi se upoređuju da se odredi da li su jednaki ili nisu jednaki.
Ako oni nisu jednaki, jedan string je “**veći od**” ili “**manji od**” drugog. Određivanje “veći” i “manji” se obavlja sa **ASCII** koodovima karaktera.
U slučaju slova, ovo je ekvivalentno sa alfabetskim redosljedom, sa jednom izuzetnom razlikom da su sva velika slova “manja od” malih slova. Ovo je istinito (true) zato što velika slova imaju **ASCII** koodove od **65** do **90** za **A** do **Z**, dok mala slova **a** do **z** se predstavljaju od **97** do **122**. Tako da se, “**GOOROO**” ne bi smatrao da je manji od “**guru**” po ovim **C**-ovim funkcijama.

ANSI C biblioteka sadrži funkcije za dva tipa uspoređivanja (komparacije) stringova: uspoređivanje dva cijela stringa, i uspoređivanje određenog broja karaktera u dva stringa.

→→→ Uspoređivanje (komparacija) dva cijela (kompletna (entire)) stringa

Funkcija →→→ **strcmp()** ←←← uspoređuje dva stringa, karakter po karakter.

Njen prototip je:

```
int strcmp(char *str1, char *str2);
```

Argumenti **str1** i **str2** su pointeri na stringove koji se uspoređuju.
Povratne vrijednosti funkcije su date u Tabeli **17.1**.
Listing **17.7** demonstrira **strcmp()**.
Table 17.1. Povratne vrijednosti od strcmp().

Povratna (vraćena) vrijednost	Značenje
< 0	str1 is less than str2.
0	str1 is equal to str2.
> 0	str1 is greater than str2.

Listing 17.7. Korištenje strcmp() za uspoređivanje stringova.

```
1:  /*  strcmp() funkcija. */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  main()
7:  {
8:      char str1[80], str2[80];
9:      int x;
10:
11:     while (1)
12:     {
13:
14:         /* Unesi dva stringa (Input two strings). */
15:
16:         printf("\n\nInput the first string, a blank to exit: ");
17:         gets(str1);
18:
19:         if ( strlen(str1) == 0 )
20:             break;
21:
22:         printf("\nInput the second string: ");
23:         gets(str2);
24:
25:         /* Uporedi ih i prikazi rezultat. */
26:
27:         x = strcmp(str1, str2);
28:
29:         printf("\nstrcmp(%s,%s) returns %d", str1, str2, x);
30:     }
31:     return(0);
32: }
        Input the first string, a blank to exit: First string
        Input the second string: Second string
```

```
strcmp(First string,Second string) returns -1
Input the first string, a blank to exit: test string
Input the second string: test string
strcmp(test string,test string) returns 0
Input the first string, a blank to exit: zebra
Input the second string: aardvark
strcmp(zebra,aardvark) returns 1
Input the first string, a blank to exit:
```

NAPOMENA: Na nekim UNIX sistemima, upoređujuće (komparacijske) funkcije stringova, ne vraćaju obavezno **–1** kada stringovi nisu isti. One će, ipak, uvijek vratiti nenula vrijednost za nejednake stringove.

ANALIZA: Ovaj program demonstrira **strcmp()**, promptajući korisnika za dva stringa (linije **16**, **17**, **22**, i 23) i prikazujući rezultate vraćene od **strcmp()**-a u liniji **29**.

Experimentišite sa ovim programom da dobijete osjećaj kako **strcmp()** upoređuje stringove. Pokušajte unijeti dva stringa koja su indentična, osim u veličini slova (case) kao što je **Grujic** i **GRUJIC**. Vidjet ćete da je **strcmp()** osjetljiv na veličinu slova (case-sensititve), što znači da program uzima u obzir velika i mala slova kao različita.

→→→ Upoređivanje parcijalnih (djelimičnih) stringova (Comparing Partial Strings)

Bibliotečna funkcija →→→ **strncmp()** ←←← upoređuje navedeni broj karaktera jednog stringa sa drugim stringom.

Njen prototip je:

```
int strncmp(char *str1, char *str2, size_t n);
```

Funkcija **strncmp()** upoređuje **n** karaktera od **str2** sa **str1**.
Upoređivanje (komparacija) nastavlja dok se ne uporedi **n** karaktera ili se ne dostigne kraj **str1**.
Metod upoređivanja i povratne vrijednosti su iste kao i kod **strcmp()**.
Upoređivanje je osjetljivo na veličinu slova (case-sensitive).

Listing **17.8** demonstrira **strncmp()**.

Listing 17.8. Upoređivanje dijelova stringova sa strncmp().

```
1:  /* strncmp() funkcija. */
2:
3:  #include <stdio.h>
4:  #include[Sigma]>=tring.h>
5:
6:  char str1[] = "The first string.";
7:  char str2[] = "The second string.";
8:
9:  main()
10: {
11:     size_t n, x;
12:
13:     puts(str1);
14:     puts(str2);
15:
16:     while (1)
17:     {
18:         puts("\n\nEnter number of characters to compare, 0 to exit.");
19:         scanf("%d", &n);
```

```
20:
21:         if (n <= 0)
22:             break;
23:
24:         x = strncmp(str1, str2, n);
25:
26:         printf("\nComparing %d characters, strncmp() returns %d.", n,
x);
27:     }
28:     return(0);
29: }
            The first string.
            The second string.
            Enter number of characters to compare, 0 to exit.
            3
            Comparing 3 characters, strncmp() returns .©]
            Enter number of characters to compare, 0 to exit.
            6
            Comparing 6 characters, strncmp() returns -1.
            Enter number of characters to compare, 0 to exit.
            0
```

ANALIZA: Ovaj program upoređuje dva stringa definisana u linijama **6** i **7**.
Linije **13** i **14** printaju stringove na ekran, tako da korisnik može vidjeti koji su.
Program izvršava **while** petlju u linijama **16** do **27** tako da se mogu uraditi višestruka poređenja.
Ako korisnik pita da uporedi nula karaktera u linijama **18** i **19**, program se prekida (breaks) u liniji **22**; u drugom slučaju, **strncmp()** se izvršava u liniji **24**, i rezultat se printa u liniji **26**.

→→→ Upoređivanje (komparacija) dva stringa s ignorisanjem veličine slova (Comparing Two Strings While Ignoring Case)

Nažalost, **C**-ova **ANSI** biblioteka ne uključuje nikakve funkcije za funkcije koje nisu osjetljive na veličinu slova. Srećom, većina **C** kompajlera obezbjeđuje svoje “in-house” (domaće) funkcije za ovaj zadatak.

Symantec koristi funkciju **strcmpl()**.
Microsoft koristi funkciju **_stricmp()**.
Borland ima dvije funkcije → **strcmpi()** i **stricmp()**.

S ovom funkcijom **Grujic** i **GRUJIC** se upoređuju kao jednaki.
Modifikujte liniju **27** u Listingu **17.7** da koristite prigodnu upoređujuću funkciju koja nije osjetljiva na veličinu slova za vaš kompajler, i pokušajte program ponovo.

→→→ Traženje stringova (Searching Strings)

C-ova biblioteka sadrži broj (nekoliko) funkcija koje traže stringove. Drugim riječima, ove funkcije odlučuju da li se jedan string pojavljuje u drugom stringu i, ako da, gdje.
Vi možete izabrati od šest string-pronalazećih funkcija, od kojih svaka zahtjeva **STRING.H** file zaglavlja.

→→→ strchr() ←←← Funkcija

strchr() funkcija pronalazi pojavljivanje navedenog karaktera u stringu.
Njen prototip je:

```
char *strchr(char *str, int ch);
```

Funkcija **strchr()** traži **str** s lijeva na desno dok se ne nađe karakter **ch** ili se ne nađe **null** terminirajući karakter.
Ako je **ch** pronađen, pointer na njega je vraćen. Ako ne, vraća se **NULL**.

Kada **strchr()** pronađe karakter, ona vraća pointer na taj karakter. Znajući da je **str** pointer na prvi karakter u stringu, vi možete dobiti poziciju pronađenog karaktera oduzimajući **str** od vrijednosti vraćenog pointera od **strchr()**.
Listing **17.9** ilustruje ovo.
Zapamtite da je prvi karakter u stringu na poziciji **0**.
Kao većina **C**-ovih string funkcija, **strchr()** je osjetljiva na veličinu slova.
Na primjer, prijavio bi da karakter **G** nije pronađen u stringu **gooRoo**.

Listing 17.9. Korištenje strchr() za pronalaženje jednog karaktera u stringu.

```
1:  /* Pronalazenje jednog karaktera sa strchr(). */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  main()
7:  {
8:      char *loc, buf[80];
9:      int ch;
10:
11:     /* Unesi string i karakter. */
12:
13:     printf("Enter the string to be searched: ");
14:     gets(buf);
15:     printf("Enter the character to search for: ");
16:     ch = getchar();
17:
18:     /* Obavi pretragu. */
19:
20:     loc = strchr(buf, ch);
21:
22:     if ( loc == NULL )
23:         printf("The character %c was not found.", ch);
24:     else
25:         printf("The character %c was found at position %d.\n",
26:                 ch, loc-buf);
27:     return(0);
28: }
            Enter the string to be searched: How now Brown Cow?
            Enter the character to search for: C
            The character C was found at position 14.
```

ANALIZA: Ovaj program koristi **strchr()** u liniji **20** da pronađe karakter unutar stringa.
strchr() vraća pointer na lokaciju (mjesto) gdje je pronađen prvi karakter, ili **NULL** ako karakter nije pronađen.
Linija **22** provjerava da li je vrijednost od **loc NULL** i printa prigodnu poruku.
Kao što je upravo pomenuto, pozicija karaktera unutar stringa je određena oduzimanjem string pointera od vraćene vrijednosti funkcije.

→→→ strrchr() ←←← Funkcija

Bibliotečna funkcija **strrchr()** je indentična sa **strchr()**, osim što ona pronalazi (traži) string za posljednje pojavljivanje navedenog karaktera u stringu.

Njen prototip je:

```
char *strrchr(char *str, int ch);
```

Funkcija **strrchr()** vraća pointer na zadnje pojavljivanje od **ch** i **NULL** ako ga ne pronađe. Da vidite kako ova funkcija radi, modifikujte liniju **20** u Listingu **17.9** da koristite **strrchr()** umjesto **strchr()**.

→→→ strcspn() ←←← Funkcija

Bibliotečna funkcija **strcspn()** traži jedan string za prvo pojavljivanje bilo kojih od karaktera u drugom stringu.

Njen prototip je:

```
size_t strcspn(char *str1, char *str2);
```

Funkcija **strcspn()** počinje pretragu sa prvim karakterom od **str1**, gledajući na bilo koji od pojedinačnih karaktera koji su sadržani u **str2**. Ovo je važno zapamtiti. Funkcija ne gleda na string **str2**. već samo karaktere koje on sadrži.

Ako funkcija pronađe poklapanje (match), ona vraća offset (pomak) sa početka od **str1**, gdje je poklapajući karakter lociran.

Ako funkcija ne pronađe poklapanje, **strcspn()** vraća vrijednost od **strlen(str1)**.
Ovo indicira da je prvo poklapanje (match) bilo **null** karakter koji terminira string.

Listing **17.10** vam pokazuje kako da koristite **strcspn()**.

Listing 17.10. Pronalaženje skupa karaktera sa strcspn().

```
1:  /* Pronalazenje (trazenje) sa strcspn(). */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  main()
7:  {
8:      char  buf1[80], buf2[80];
9:      size_t loc;
10:
11:     /* Unesi stringove (Input the strings). */
12:
13:     printf("Enter the string to be searched: ");
14:     gets(buf1);
15:     printf("Enter the string containing target characters: ");
16:     gets(buf2);
17:
18:     /* Obavi pronalazenje (Perform the search). */
19:
20:     loc = strcspn(buf1, buf2);
21:
22:     if ( loc ==  strlen(buf1) )
23:         printf("No match was found.");
24:     else
25:         printf("The first match was found at position %d.\n", loc);
26:     return(0);
27: }
```

```
Enter the string to be searched: How now Brown Cow?
Enter the string containing target characters: Cat
The first match was found at position 14.
```

ANALIZA: Ovaj listing je sličan Listingu **17.10**. Umjesto da traži prvo pojavljivanje jednog karaktera, on traži prvo pojavljivanje bilo kojeg karaktera unesenog u drugom stringu.
Program poziva **strcspn()** u liniji **20** sa **buf1** i **buf2**. Ako je neki od karaktera iz **buf2** u **buf1**, **strcspn()** vraća offset (pomak) sa početka od **buf1** na lokaciju prvog pojavljivanja.
Linija **22** provjerava povratnu vrijednost da odluči da li je **NULL**.
Ako je vrijednost **NULL**, nikakvi karakteri nisu pronađeni, i odgovarajuća poruka se prikazuje u liniji **23**.
Ako je vrijednost pronađena, poruka se prikazuje govoreći poziciju karaktera u stringu.

→→→ strspn() ←←← Funkcija

Ova funkcija se odnosi na prethodnu, **strcspn()**, kao što sljedeći paragraf objašnjava.
Njen prototip je:

```
size_t strspn(char *str1, char *str2);
```

Funkcija **strspn()** traži **str1**, uspoređujući ga karakter po karakter sa karakterima sadržanim u **str2**. Ona vraća poziciju od prvog karaktera u **str1** koji ne odgovara karakteru u **str2**. Drugim riječima, **strspn()** vraća dužinu inicijalnog segmenta od **str1** koji se sastoji u potpunosti od karaktera pronađenim u **str2**.
Vraćena vrijednost je **0** ako je ne pronađu poklapajući (match) karakteri.
Listing **17.11** demonstrira **strspn()**.

Listing 17.11. Pronalaženje prvog nepoklapajućeg karaktera sa strspn().

```
1: /* Trazenje sa  strspn(). */
2:
3: #include <stdio.h>
4: #include <string.h>
5:
6: main()
7: {
8:     char  buf1[80], buf2[80];
9:     size_t loc;
10:
11:     /* Unesi stringove. */
12:
13:     printf("Enter the string to be searched: ");
14:     gets(buf1);
15:     printf("Enter the string containing target characters: ");
16:     gets(buf2);
17:
18:     /* Obavi pronalazenje. */
19:
20:     loc = strspn(buf1, buf2);
21:
22:     if ( loc ==  0 )
23:         printf("No match was found.\n");
24:     else
25:         printf("Characters match up to position %d.\n", loc-1);
26:
27: }
```

```
Enter the string to be searched: How now Brown Cow?
Enter the string containing target characters: How now what?
Characters match up to position 7.
```

ANALIZA: Ovaj program je indentičan sa prethodnim primjerom, s izuzetkom što on poziva **strspn()** umjesto **strcspn()** u liniji **20**.
Funkcija vraća offset (pomak) u **buf1**, gdje je pronađen prvi karakter koji nije u **buf2**.
Linije **22** do **25** procjenjuju povratnu vrijednost i printaju prigodnu poruku.

→→→ strpbrk() ←←← Funkcija

Bibliotečna funkcija **strpbrk()** je slična sa **strcspn()**, tražeći jedan string za prvo pojavljivanje bilo kojeg karaktera koji je sadržan u drugom stringu. Razlikuje se u tome što ona ne uključuje terminirajući **null** karakter u svojoj pretrazi.

Prototip funkcije je:

```
char *strpbrk(char *str1, char *str2);
```

Funkcija **strpbrk()** vraća pointer na prvi karakter u **str1** koji odgovara (match) bilo kojem od karaktera u **str2**.
Ako ona ne pronađe poklapanje, funkcija vraća **NULL**.
Kao šro je prethodno objašnjeno za funkciju **strchr()**, vi možete dobiti offset prvog poklapanja u **str1** oduzimajući pointer **str1** sa pointerom vraćenim od **strpbrk()** (naravno, ako nije **NULL**).
Na primjer, zamjenite **strcspn()** u liniji **20** u Listingu **17.10** sa **strbrk()**.

→→→ strstr() ←←← Funkcija

Posljednja, i vjerovatno najkorisnija, **C**-ova string-pronalazeća funkcija je **strstr()**.
Ova funkcija traži prvo pojavljivanje jednog stringa unutar drugog, i traži cijeli string, a ne pojedinačne karaktere unutar stringa.

Njen prototip je:

```
char *strstr(char *str1, char *str2);
```

Funkcija **strstr()** vraća pointer na prvo pojavljivanje **str2** unutar **str1**.
Ako ne pronađe poklapanje, funkcija vraća **NULL**. Ako je dužina od **str2 0**, funkcija vraća **str1**.
Kada **strstr()** pronađe poklapanje, vi možete dobiti offset (pomak) od **str2** unutar **str1** sa oduzimanjem pointera, kako je objašnjeno ranije za **strchr()**.
Poklapajuće procedure koje **strstr()** koristi su osjetljive na veličinu slova.

Listing **17.12** demonstrira kako se koristi **strstr()**.

Listing 17.12. Korištenje strstr() za pretragu jednog stringa unutar drugog.

```
1:  /* Trazenje sa  strstr(). */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  main()
7:  {
8:      char *loc, buf1[80], buf2[80];
9:
10:     /* Unesi stringove. */
11:
12:     printf("Enter the string to be searched: ");
13:     gets(buf1);
14:     printf("Enter the target string: ");
15:     gets(buf2);
16:
17:     /* Obavi pronalazenje. */
```

```
18:
19:     loc = strstr(buf1, buf2);
20:
21:     if ( loc ==  NULL )
22:         printf("No match was found.\n");
23:     else
24:         printf("%s was found at position %d.\n", buf2, loc-buf1);
25:     return(0);
26: }
        Enter the string to be searched: How now brown cow?
        Enter the target string: cow
        Cow was found at position 14.
```

ANALIZA: Ova funkcija obezbjeđuje alternativni način za pretraživanje stringa. U ovom slučaju možete tražiti cijeli string unutar nekog drugog stringa.
Linije **12** do **15** promptaju za dva stringa.
Linija **19** koristi **strstr()** da traži drugi string, **buf2**, unutar prvog stringa, **buf1**.
Pointer na prvo pojavljivanje je vraćen, ili **NULL** je vraćena ako string nije pronađen.
Linije **21** do **24** procjenjuju vraćenu vrijednost, **loc**, i printaju odgovarajuću poruku.

ZAPAMTITE da za većinu string funkcija, postoje ekvivalentne funkcije koje vam dozvoljavaju da navedete broj karaktera za manipulaciju. Funkcije koje dozvoljavaju navođenje broja karaktera se obično nazivaju →→ **strnxxx()** ←←, gdje je **xxx** specifičan za funkciju.

NE zaboravite da je **C** osjetljiv na veličinu slova. **A** i **a** su različiti.

→→→ Konverzije stringa (String Conversions)

Puno **C**-ovih biblioteka sadrži dvije funkcije koje mogu promjeniti veličinu (case) karakteera unutar stringa. Ove funkcije nisu **ANSI** standard, tako da mogu biti različite ili čak nepostojeće na vašem kompajleru.
Zato jer mogu biti prilično korisne, one su uključene ovdje.
Njihovi prototipi, u **STRING.H**, su kako slijedi za Microsoft **C** kompajler (ako koristite drugačiji kompajler, one bi trebale biti slične):

```
char *strlwr(char *str);
char *strupr(char *str);
```

Funkcija →→→ **strlwr()** ←←← konvertuje sva karakterna slova u **str** iz velikih slova u mala slova;
→→→ **strupr()** ←←← radi obratno, konvertujući sve karaktere iz **str** u velika slova.
Neslova karakteri su netaknuti.
Obje funkcije vraćaju **str**.
Primjetite da nijedna funkcija ne kreira novi string, već modifikuje postojeći string na mjesto.
Listing **17.13** demonstrira ove funkcije.
Zapamtite da za kompajlerianje programa, koji koristi ne-**ANSI** funkcije, će vam možda biti potrebno da kažete vašem kompajleru da ne nameće **ANSI** standard.

Listing 17.13. Konvertovanje (case) karaktera u stringu sa strlwr() i strupr().

```
1:  /* Karakterno konverzione funkcije strlwr() i strupr(). */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  main()
7:  {
8:      char buf[80];
9:
10:     while (1)
11:     {
12:         puts("Enter a line of text, a blank to exit.");
13:         gets(buf);
14:
15:         if ( strlen(buf) == 0 )
16:             break;
17:
18:         puts(strlwr(buf));
19:         puts(strupr(buf));
20:     }
21:     return(0);
22: }
        Enter a line of text, a blank to exit.
        Bradley L. Jones
        bradley l. jones
        BRADLEY L. JONES
        Enter a line of text, a blank to exit.
```

ANALIZA: Ovaj listing prompta za string u liniji **12**. On onda provjerava da se uvjeri da string nije prazan (linija **15**).
Linija **18** printa string nakon konverzije u **sva mala** slova.
Linija **19** printa nakon konverzije u **sva velika** slova.

Ove funkcije su dio Symantec, Microsoft, i Borland C biblioteka. Prije njihove upotrebe provjerite bibliotečne reference. Što se portabilnosti tiče, izbjegavajte ne-**ANSI** funkcije kao što su ove.

→→→→ Raznovrsne string funkcije (Miscellaneous String Functions)

Ovaj dio pokriva nekoliko string funkcija koje ne spadaju u nijednu kategoriju. Sve one zahtjevaju file zaglavlja **STRING.H**.

→→→ strrev() ←←← Funkcija

Funkcija **strrev()** obrće redosljed svih karaktera u stringu.
Njen prototip je:

```
char *strrev(char *str);
```

Redosljed svih karaktera u **str** je obrnut, sa terminirajućim **null** karakterom, koji ostaje, na kraju. Funkcija vraća **str**. Nakon što su **strset()** i **strnset()** definisane u nastavku, **strrev()** je demonstrirana u Listingu **17.14**.

→→→ strset() ←←← i →→→ strnset() ←←← Funkcije

Kao i prethodne funkcije, **strrev()**, **strset()** i **strnset()** nisu dio **ANSI C** standardne biblioteke. Ove funkcije mijenjaju sve karaktere (**strset()**), ili navedeni broj karaktera (**strnset()**), u stringu na navedeni karakter.

Prototipi su:

```
char *strset(char *str, int ch);
char *strnset(char *str, int ch, size_t n);
```

Funkcija **strset()** mijenja sve karaktere u **str** na **ch** izuzev **null** terminirajućeg karaktera.
Funkcija **strnset()** mijenja prvih **n** karaktera od **str** na **ch**.

Ako **n>=strlen(str)**, **strnset()** mijenja sve karaktere u **str**.

Listing **17.14** demonstrira sve tri funkcije.

Listing 17.14. Demonstracija od strrev(), strnset(), i strset().

```
1:  /* Demonstrira  strrev(), strset(), i strnset(). */
2:  #include <stdio.h>
3:  #include <string.h>
4:
5:  char str[] = "This is the test string.";
6:
7:  main()
8:  {
9:      printf("\nThe original string: %s", str);
10:     printf("\nCalling strrev(): %s", strrev(str));
11:     printf("\nCalling strrev() again: %s", strrev(str));
12:     printf("\nCalling strnset(): %s", strnset(str, `!', 5));
13:     printf("\nCalling strset(): %s", strset(str, `!'));
14:     return(0);
15: }
```

```
The original string: This is the test string.
Calling strrev(): .gnirts tset eht si sihT
Calling strrev() again: This is the test string.
Calling strnset(): !!!!!is the test string.
Calling strset(): !!!!!!!!!!!!!!!!!!!!!!
```

ANALIZA: Ovaj program demonstira tri različite string funkcije. Demonstracije su urađene tako da printaju vrijednosti stringa, **str**.

Iako ove funkcije nisi **ANSI** standard, one su uključene u Symantec, Microsoft, i Borland **C** kompajlerove bibliotečne funkcije.

→→ Konverzije String-u-Broj

Nekad ćete rebati konvertovati predstavljanje stringa brojeva u stvarnu numeričku varijablu.
Na primjer, string "**123**" može biti konvertovan u varijablu tipa **int** sa vrijednosti **123**.

Tri funkcije mogu biti korištene za konvertovanje stringa u broj.
One su objašnjene u sljedećim dijelovima; njihovi prototipi su u →→→ **STDLIB.H** ←←←.

→→→ atoi() ←←← Funkcija

Bibliotečna funkcija **atoi()** konvertuje string u **integer**.
Njen prototip je:

```
int atoi(char *ptr);
```

Funkcija **atoi()** konvertuje string na koji pokazuje **ptr** u **integer**. Osim cifara, string može sadržavati vodeći bijeli-znak i **+** ili **–** znak.

Konverzija počinje na početku stringa i nastavlja dok se ne naiđe na nekonvertibilan (na primjer, slovo ili znak punktacije) karakter.
Rezultirajući **integer** je vraćen na pozivajući program.
Ako ne pronađe konvertibilan karakter, **atoi()** vraća **0**.

Tabela **17.2** navodi neke primjere:
Table 17.2. String-u-Broj konverzije sa atoi().

String	Vrijednost vraćena od atio()
"157"	157
"-1.6"	-1
"+50x"	50
"twelve"	0
"x506"	0

String mora počinjati sa brojem i ovo je (by the way) string-u-**integer** konverzija. Ako string ne počinje sa brojem, **atio()** vraća **0**.

→→→ atol() ←←← Funkcija

Bibliotečna funkcija **atol()** radi kao i **atoi()**, s razlikom da vraća tip **long**.

Prototip funkcije je:

```
long atol(char *ptr);
```

Vrijednost vraćena od **atol()** će biti ista kako je pokazano u Tabeli **17.2** za **atoi()**, s razlikom da će povratna vrijednost biti tipa **long**, umjesto tipa **int**.

→→→ atof() ←←← Funkcija

Funkcija **atof()** konvertuje string u tip **double**.
Prototip je:

```
double atof(char *str);
```

Argument **str** pokazuje na string koji se konvertuje. Ovaj string može sadržavati vodeći bijeli-znak i **+** ili **–** karakter. Broj može sadržavati cifre od **0** do **9**, decimalnu tačku, i indikator exponenta **E** ili **e**. Ako nema konvertibilnih karaktera, **atof()** vraća **0**.

Tabela **17.3** navodi neke primjere korištenja **atof()**-a.
Table 17.3. String-u-Broj konverzije sa atof().

String	Vrijednost vraćena od atof()
"**12**"	12.000000
"**-0.123**"	-0.123000
"**123E+3**"	123000.000000
"**123.1e-5**"	0.001231

Listing **17.15** vam dozvoljava da unesete vaše stringove za konverzije.
Listing 17.15. Korištenje atof() za konvertovanje stringova u numeričke varijable tipa double.

```
1:  /* Demonstracija atof(). */
2:
3:  #include <string.h>
4:  #include <stdio.h>
5:  #include <stdlib.h>
6:
7:  main()
8:  {
9:      char buf[80];
10:     double d;
11:
12:     while (1)
13:     {
14:         printf("\nEnter the string to convert (blank to exit):     ");
15:         gets(buf);
16:
17:         if ( strlen(buf) == 0 )
18:             break;
19:
20:         d = atof( buf );
21:
22:         printf("The converted value is %f.", d);
23:     }
24:     return(0);
25: }
        Enter the string to convert (blank to exit):     1009.12
        The converted value is 1009.120000.
        Enter the string to convert (blank to exit):     abc
        The converted value is 0.000000.
        Enter the string to convert (blank to exit):     3
        The converted value is 3.000000.
        Enter the string to convert (blank to exit):
```

ANALIZA: while petlja u linijama **12** do **23** dopušta vam da nastavljate izvršenje programa dok ne unesete praznu liniju.
Linije **14** i **15** promptaju za vrijednost.
Linija **17** provjerava da li je unesena prazna linija. Ako jeste, program prekida (breaks) **while** petlju i završava.
Linija **20** zove **atof()**, konvertujući unesenu vrijednost (**buf**) u tip **double**, **d**.
Linija **22** printa konačan rezultat.

→→→ Funkcije za testiranje karaktera (Character Test Functions)

File zaglavlja →→→ **CTYPE.H** ←←← sadrži prototipe za broj funkcija koje testiraju karaktere, vraćajući **TRUE** ili **FALSE** zavisno od toga da li se nailazi na karakter pod određenim uslovima.

Na primjer, da li je slovo ili broj?
→→→ **isxxxx()** ←←← funkcije su u stvari **makroi**, definisani u **CTYPE.H** (o makroima Dana 21).

Svi **isxxxx()** makroi imaju isti prototip:

```
int isXXXX(int ch);
```

U prethodnoj liniji, **ch** je karakter koji se testira. Povratna vrijednost je **TRUE** (nenula) kao se naiđe na uslov ili **FALSE** (nula) ako ne.

Tabela **17.3** navodi kompletan skup (set) od **isxxxx()** makroa.

Tabela 17.4. isxxxx() makroi.

Makro	Akcija
isalnum()	Vraća **TRUE** ako je **ch** slovo ili broj.
isalpha()	Vraća **TRUE** ako je **ch** slovo.
isascii()	Vraća **TRUE** ako je **ch** standardni **ASCII** karakter (između **0** i **127**).
iscntrl()	Vraća **TRUE** ako je **ch** kontrolni karakter.
isdigit()	Vraća **TRUE** ako je **ch** broj (digit).
isgraph()	Vraća **TRUE** ako je **ch** printajući karakter (svi osim space).
islower()	Vraća **TRUE** ako je **ch** malo slovo.
isprint()	Vraća **TRUE** ako je **ch** printajući karakter (uključujući space).
ispunct()	Vraća **TRUE** ako je **ch** karakter punktacije.
isspace()	Vraća **TRUE** ako je **ch** bijeli-znak karakter (space, tab, vertical tab, line feed, form feed, ili carriage return).
isupper()	Vraća **TRUE** ako je **ch** veliko slovo.
isxdigit()	Vraća **TRUE** ako je **ch** hexadecimalni broj (digit (od **0** do **9**, **a** do **f**, **A** do **F**)).

Možete raditi puno interesantnih stvari sa makroima za testiranje karaktera. Jedan primjer je funkcija **get_int()**, pokazana u Listingu **17.16**. Ova funkcija unosi **integer** sa **stdin** i vraća ga kao varijablu tipa **int**. Funkcija preskače vodeći (prvi) bijeli-znak (white space) i vraća **0** ako prvi nebijeli karakter nije numerički karakter.

Listing 17.16. Korištenje isxxxx() makroa za implementaciju funkcije koja unosi integer.

```
1:  /* Upotreba makroa za testiranje karaktera za kreiranje integera  */
2:  /* ulazne funkcije (input function). */
3:
4:  #include <stdio.h>
5:  #include <ctype.h>
6:
7:  int get_int(void);
8:
9:  main()
10: {
11:     int x;
12:     x =  get_int();
13:
14:     printf("You entered %d.\n", x);
```

```
15: }
16:
17: int get_int(void)
18: {
19:     int ch, i, sign = 1;
20:
21:     /* Preskoci bilo koji vodeci bijeli space. */
22:
23:     while ( isspace(ch = getchar()) )
24:         ;
25:
26:     /* Ako je prvi karakter nenumericki, onda unget */
27:     /* karakter i vrati 0. */
28:
29:     if (ch != `-' && ch != `+' && !isdigit(ch) && ch != EOF)
30:     {
31:         ungetc(ch, stdin);
32:         return 0;
33:     }
34:
35:     /* Ako je prvi karakter znak minus, postavi */
36:     /* znak prigodno (sign accordingly). */
37:
38:     if (ch == `-')
39:         sign = -1;
40:
41:     /* Ako je prvi karakter plus ili minus znak, */
42:     /* uzmi (get) sljedeći karakter. */
43:
44:     if (ch == `+' || ch == `-')
45:         ch = getchar();
46:
47:     /* Citaj karaktere dok se ne unese nebroj. Pridruzi vrijednosti, */
48:     /* pomnozene sa prigodnom vrijednosti dignutom na 10, sa i. */
49:
50:     for (i = 0; isdigit(ch); ch = getchar() )
51:         i = 10 * i + (ch - `0');
52:
53:     /* Make result negative if sign is negative. */
54:
55:     i *= sign;
56:
57:     /* Ako EOF nije susretnut, nebroj karakter mora */
58:     /* da je prochitan, tako da ga unget. */
59:
60:     if (ch != EOF)
61:         ungetc(ch, stdin);
62:
63:     /* Vrati unesenu vrijednost. */
64:
65:     return i;
66: }
```

-100
You entered -100.
abc3.145
You entered 0.
9 9 9
You entered 9.
2.5
You entered 2.

ANALIZA: Ovaj program koristi bibliotečnu funkciju **ungetc()** u linijama **31** i **61** (Dan 14). Zapamtite da ova funkcija "ungets", ili vraća, karakter na navedeni stream. Ovaj vraćeni karakter je prvi unos sljedeći put kada program čita karakter sa stream-a. Ovo je neophodno zato što, kada funkcija **get_int()** pročita nenumerički karakter sa **stdin**, vi želite da stavite karakter nazad u slučaju da program treba da ga pročita kasnije.

U ovom programu, **main()** je jednostavan. **Integer** varijabla **x**, je deklarisana (linija **11**), pridružena joj je vrijednost od **get_int()** funkcije (linija **12**), i printana na-ekran (linija **14**). **get_int()** funkcija čini ostatak programa.

get_int() nije tako jednostavna. Da uklonite vodeći bijeli space koji može biti unesen, linija **23** peetlja sa **while** komandom.
isspace() makro testira karakter, **ch**, dobijen sa **getchar()** funkcijom. Ako je **ch** space, dobavlja se još jedan karakter, dok se ne primi nebijelispace karakter.
Linija **29** provjerava da li se karakter upotrebljiv. Linija **28** može biti pročitana kao: "**Ako karakterni unos nije negativan znak, plus znak, broj, ili kraj-file-a(ova)**." Ako je ovo true, **ungetc()** se koristi u liniji **31** da stavi karakter nazad, i funkcija se vraća u **main()**. Ako je karakter upotrebljiv, izvršenje se nastavlja.

Linije **38** do **45** rade sa znakom broja.
Linija **38** provjerava da vidi da li je uneseni karakter negativan znak. Ako jeste, varijabla (**sign**) se postavlja na **–1**. **sign** je iskorišten da napravi konačni (finalni) broj bilo pozitivan ili negativan (linija **55**). Zato što su pozitivni brojevi po defaultu, nakon što se pobrinete za negativne znakove, vi ste skoro spremni na nastavak. Ako je znak unesen, program treba da uzme (get) drugi karakter. Linije **44** i **45** se brinu za ovo.

Srce funkcije je **for** petlja u linijama **50** i **51**, koja nastavlja da uzima karaktere sve dok su dobijeni karakteri cifre (digits). Linija **51** može u početku biti malo zbunjujuća. Ova linija se brine o pojedinačnim unesenim karakterima i pretvara ih u broj. Oduzimajući karakter **'0'** od vašeg broja, mijenja broj karaktera u realan broj. (Zapamtite **ASCII** vrijednosti). Kada se dobije ispravna karakterna vrijednost, brojevi se množe da odgovarajućim exponentom od **10** (proper power of **10**).
for petlja nastavlja se dok se ne unese na necifarski broj. U toj tački, linija **55** primjenjuje znak (sign) na broj, čineći ga kompletnim.

Prije vraćanja (povratka), program treba da radi malo čišćenja. Ako zadnji broj nije kraj-file-a, on treba da bude stavljen nazad u slučaju da je potreban negdje drugo. Linija **61** radi ovo prije nego što se linija **65** vrati.

NE miješajte karaktere sa brojevima (numbers). Lako je zaboraviti da karater **"1"** nije ista stvar kao i broj **1**.

Sažetak

Ovo je poglavlje pokazalo različite načine kako možete manipulisati stringovima. Koristeći **C**-ove standardne bibliotečne funkcije (takođe i bilo koje kompajler-navedene funkcije), vi možete kopirati, concatenate-irati (spajati), upoređivati, ili tražiti stringove. Ovo su sve potrebni poslovi u većini programerskih poslova. Standardna biblioteka takođe sadrži funkcije za konvertovanje case-a (velika, mala slova) karakteera u stringovima i za konvertovanje stringova u brojeve. Konačno, **C** obezbjeđuje razne karakterno-testirajuće funkcije ili, preciznije, makroe koji obavljaju razne testove na pojedinačnim karakterima. Koristeći ove makroe za testiranje karaktera, vi možete kreirati vaše ulazne funkcije (your own custom input functions).

P&O

P Da li strcat() ignoriše vučene space-ove kada radi concatenation???
O Ne. **strcat()** gleda na space samo kao na još jedan karakter.

P Da li mogu da konvertujem brojeve u stringove???
O Da. Vi možete napisati funkciju sličnu onoj u Listingu **17.16**, ili možete pogledati na bibliotečnu referencu (Library Reference) za dostupne funkcije. Neke dostupne funkcije uključuju **itoa()**, **ltoa()**, i **ultoa()**. **sprint(f)** također može biti korišten.

Vježbe

1. Koje vrijednosti vraćaju testne funkcije?
2. Šta bi **atoi()** funkcija vratila ako su joj prosljeđene sljedeće vrijednosti?
- **a.** "65"
- **b.** "81.23"
- **c.** "-34.2"
- **d.** "ten"
- **e.** "+12hundred"
- **f.** "negative100"

3. Šta bi **atof()** funkcija vratila ako joj je prosljeđeno sljedeće???
- **a.** "65"
- **b.** "81.23"
- **c.** "-34.2"
- **d.** "ten"
- **e.** "+12hundred"
- **f.** "1e+3"

4. BUG BUSTER: Da li nešto nije u redu sa sljedećim??

```
char *string1, string2;
string1 = "Hello World";
strcpy( string2, string1);
printf( "%s %s", string1, string2 );
```

5. Napišite program koji prompta za korisnikovo prezime i ime. Zatim smjesti ime u novi string kao prvi inicijal, period, prostor (space), prezime. Na primjer, ako je uneseno Semso, Bonapparta, smjesti B. Bonapparta. Prikaži novo ime na ekran.
8. Napišite funkciju koja odlučuje broj puta koji je jedan string pojavljuje unutar drugog.
9. Napišite program koji traži text file za ponavljanja od korisničko-navedenih meetnih (target) stringova i onda raportuje broj linija gdje je meta pronađena. Na primjer, ako pretražujete jedan od vaših **C** file-ova izvornih koodova za string **"printf()"**, program bi trebao izlistati sve linije gdje je **printf()** funkcija pozivana od strane programa.
10. Listing **17.16** demonstrira funkciju koja unosi integer sa **stdin**-a. Napišite funkciju **get_float()** koja unosi **float**ing-point vrijednost sa **stdin**-a.

Univerzitet na Dolac Malti
GooRoo's fakultet - Sarajevo

C – handmade script
(inteni materijal)

Sarajevo – željezničar, mart 2004. godine

Uvod

Kratki istorijski razvoj:

1. **1950 godine**
obiljžavaju ga jezici kao: **Fortran** (numerička računanja), **ALGOL** (numerička računanja), **COBOL** (procesiranje podataka i pravljenje poslovnih aplikacija)

2.rane **1960**
LISP- na matematičkim principima, oslobođen von Neuman arihktekure, samo jedan tip podataka program se pisao kao lista.
APL-bogat sa operatorima, pogotovu na nizovima

3. **sredinom 1960**
BASIC-sredinom 1960, ima sličnu sintaksu kao Fortran, limitiran je kontrolnim strukturama i strukturama podataka.
Ovaj jezik nije uveo novi longvistički koncept, ali je prvi uveo raspoložive alate za podršku sa interaktivnim radom sa interpreterom. Visual Basic obezbjeđuje razvojne mogućnosti pod Windows okruženjem.

4. **70 godine**
Pascal-71 godine obezbeđuje konstrukcije za modularni dizajn, opšta namjena jezika i edukativne svrhe.
C-72 za sistemsko programiranje, autor **C**-a je Denis Ritchie i nastao je u Belovim laboratirijama, pri izradi operativnog sistema **Unix** (90% napisano u **C**-u),

1989 . ANSI

Modula-2 i sistemsko i modularno programiranje
Prolog nekonvencionalni jezik **pro**gramming in **log**ic programski jezik se koristi za vještačku intelegenciju.

5. **80 godine objektna orjentacija**

POJEDINAČNO O JEZICIMA

• **FORTRAN** (**FOR**mula **TRAN**slator)
razvijen je u IBM-u između 1954. i 1957. za znanstvene i inženjerske aplikacije koje zahtjevaju kompleksne matematičke proračune.
FORTRAN je još u širokoj upotrebi, posebno u inženjerskim aplikacijama

• **COBOL** (**CO**mmon **B**usiness **O**riented **L**anguage) razvijen je 1959.
COBOL se prvenstveno koristio za komercijalne aplikacije koje su zahtijevale preciznu i efikasnu manipulaciju sa velikim količinama podataka.
Dosta softvera u svijetu je još uvijek u COBOL-u

Basic i Visual Basic
• Visual Basic je evoluirao iz BASIC-a (**B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode)
• BASIC su sredinom 1960-tih razvili profesori John Kemeny i Thomas Kurtz sa Darmouth Koledža kao jezik za pisanje jednostavnih programa
• Osnovna namjena BASIC-a je bila učenje programiranja
• Razvojem Microsoft Windows grafičkog korisničkog sučelja (GUI), u kasnim 1980-tim i ranim 1990-tim, prirodna evolucija BASIC-a bila je Visual Basic, koji je napravljen u Microsoft korporaciji 1991.
• Do pojave Visual Basica, razvoj Windows orjentisanih aplikacija bio je težak i mukotrpan proces

• **Javu** je razvio Sun Microsystems i izdana je 1995. Java se bazira na C-u i C++- u te objedinjuje mnoge osobine ostalih objektno orjentiranih jezika

• Microsoft verzija Jave se zove **Visual J++.** Vjeruje se da će Java i Visual J++
biti najznačajniji dugotrajni konkurenti Visual Basic-u

Power Builder, razvijen u Powersoft korporaciji, i **Delphi**, razvijen u Borland International-u, su jezici koji konkuriraju Visual Basicu ali imaju mnogo manje tržište

Pascal nastao 1971. programski jezik
Kreirao ga je profesor Nicklaus Wirth.
Pascal je dizajniran za učenje strukturiranog
programiranja u akademskim ustanovama i ubrzo je postao najpoželjniji programski jezik na mnogim visokim školama

Nažalost jezik se nije zbog svojih nedostataka mogao upotrebljavati u komercijalne svrhe te je sedamdesetih godina i u ranim osamdesetima razvijen programski jezik **ADA** pod sponzorstvom SAD-ovog ministarstva obrane

• Jedno bitno svojstvo **ADA**-e je višezadaćnost (**multitasking**) ili višenitnost (**multithreading**) kako se naziva u drugim jezicima

Programski jezik C
-Autor Dennis Ritchie (Bell Telephone Laboratories)
- Razvijen sedamdesetih godina
- Standardiziran 1989. ANSI-C, (C90 standard). Novi standard izašao 1999, (C99 standard)
C je jedan od najpopularnijih jezika za implementaciju sistema u industriji, u početku se koristio za razvoj UNIX operativnog sistema .
-opća namjena jednostavan za učenje modularan, brz, omogućava optimizaciju koda

C++
nastavak C-a je razvio Bjarne Stroustrup u ranim 1980-tim u Bell laboratorijima.
C++ pruža brojne mogućnosti što unaprijeđuje C jezik i dodaje mogućnosti za tzv. objektno orjentirano programiranje (OOP)

→ UVOD II

Algoritam zapisan formalnim jezikom koji "razumije" računar naziva se **program** a jezik zapisivanja **programski jezik**.

Osnovne logičke strukture
- sekvencija ili slijed
- iteracija ili ponavljanje
- selekcija ili odabir
- skokovi

Sistemski pristup bazira se na efikasnim tehnikama programiranja:
strukturiranom, modularnom i objektnom programiranju.

U svim tehnikama **zadatak** se posmatra kao sistem formiran od podzadatka kao podsistema.

Metode programiranja:
(1) **Proceduralno programiranje** : linijska struktura
(2) **Proceduralno-strukturirani model** :hijerarhijska struktura
(3) **Objektni pristup programiranju**

- (neproceduralno programiranje
- programiranje je vođeno događajima
- program se ne odvija po unaprijed utvrđenom slijedu
- programom se upravlja pomoću niza događaja (miš, tipka, menija-izbornik)
- operacije moraju biti dostupne istovremeno
- program se rastavlja na niz zatvorenih cjelina
- radi se s objektima (objedinjuju operacije i podatke)

Osnovni alati:

-algoritam : općenito rješenje problema
-pseudokod : rješenje u formi programa na 'govornom jeziku'
-**P**(rogram) **D**(efinition) **L**(anguage) rješenje : rješenje u formi programa na 'engleskom'
- apstrakcija

Programski jezici:

Česta podjela: podijeliti u tri osnovna tipa:

1. **Mašinski jezici**
2. **Asemblerski jezici**
3. **Viši programski jezici**

Mašinski jezik (izvršni program, instrukcije u binarnom kodu)
Asemblerski jezik (izvorni program, jezik prilagoden arhitekturi računara)
Viši programski jezici
- Neovisnost o arhitekturi računara
-Veća efikasnost programiranja
-Prilagođenost pojedinim specifičnim zadacima
primjer: C, FORTRAN, BASIC, LISP, C++, Java, Perl

Tip podataka je skup vrijednosti koje imaju zajedničke karakteristike.
Vrijednost(**Value**) – nedefinisani osnovni koncept
Literal – specificirana vrijednost se označava u programu kao literal i predstavlja sekvencu simbola
Predstavljanje (Representation) vrijednost se unutar računara predstavlja kao specifičan string bita. Npr. vrijednost označena sa 'x' može se predstaviti kao string od 8 bitova 01111000.
Varijabla (promjenjiva) je ime dodjeljeno memorijskim ćelijama ili ćelijama koje čuvaju predstavljenu vrijednost specificiranog tipa.
Konstanta je ime dodjeljeno memorijskim ćelijama koje zadržavaju predstavljenu vrijednost specificiranog tipa.
Vrijednost konstante se ne može mijenjati tokom izvršavanja programa.
Objekt – je varijabla ili konstanta
Varijabla se označava **imenom**.
Izrazi su sintaksne konstrukcije koje omogućavaju programeru kombinaciju vrijednosti i operatora da bi izračunali novu vrijednost.
Tip **int**eger je podskup skupa cijelih brojeva.

U C-u operator "**/**" se koristi za vraćanje količnika a drugi operator **%.**
Npr. značenje operatora **/** i **%** u **C**-u zavise od implementcije, zbog toga program koji koristi ove operatore nije prenosiv.
Integer vrijednosti se smještaju direktno u **memorijsku** (cijelu ili djelimičnu) **riječ**.

U **C** realne varijable se deklarišu sa **float** ključnom riječi.

Varijable sa duplom preciznošću koriste 64 bita dok neki računari i kompajleri podržavaju i duže tipove.
Float sa duplom preciznošću nazivaju se **double** u C-u.

Definisane su i funkcije tipa:

trunc(x)-daje cijeli dio realnog broja x,
round(x)-vrši zaokruživanje cijelog broja,

Promjenjivoj karakternog tipa moguće je dodjeliti **tačno jedan znak** iz datog skupa znakova.
Logički operatori u **C**-u su:

! negacija
&& logičko i
|| logičko ili

Relacijski operatori u **C**-u su:

>veće **>=**veće ili jednako
< manje **<=** manje ili jednako

Operatori jednakosti manji prioritet od gornjih operatora:

== jednakosti
!= različito

Relacijski operatori imaju niži prioritet od aritmetičkih operatora.

Podtip je ograničenje na postojeći tip.

Integer i prebrojivi tipovi imaju rang ograničenja.

Konstante predstavljaju veličine koje ne mijenjaju vrijednost tokom izvršavanja programa.
Logički operacije sa bitovima :

& bitni AND
| bitni OR
^ bitni ekskluzivni OR
<< pomjeranje lijevo
>> pomjeranje desno
~ komplement

Kontrolni iskazi se koriste za promjenu redosljeda izvršavanja.

Loop iskazi su veoma skoženi iskaz.
Pelja ima:

-ulaznu tačku,
-**sekvencu iskaza** i
-**jednu ili više izlaznih tačaka**.

Continue iskaz prosljeđuje kontrolu na slijedeću iteraciju 'do', 'while' ili 'for' petlje u kojoj se pojavio zaobilazeći ostale iskaze u tijelu 'do', 'while' ili 'for' petlje.
Unutar 'do' ili 'while' petlje sljedeća iteracija počinje sa ponovnim ispitivanjem uslova petlje, a unutar 'for' petlje slijedeća iteracija počinje sa evaluacijom kontrolne linije te petlje.

Break iskaz prekida izvršenje 'do', 'while', 'switch' ili 'for' iskaza u kojima se pojavljuje.
Kontrola se prenosi na iskaz koji slijedi iza prekinutog iskaza.

Ako se 'break' iskaz pojavi van 'do', 'while', 'switch' ili 'for' iskaza pojaviće se greška.

Podprogram je programska jedinica koja se sastoji od:
deklaracije varijabli i iskaza koji se mogu pozivati iz raznih dijelova programa.
Podprogrami (**procedure, funkcije, subrutine**) su prvo korišteni da omoguće višestruku upotrebu programskih segmenata.
Noviji pogled shvata **podprograme kao osnovne elemente programske strukture** i preferira da se svaki programski segment koji obrađuje neki zadatak smjesti u odvojene podprograme.
Podprogram se sastoji od:

- **deklaracije** koja definiše **interfejs** sa podprogramom.
- **lokalne deklaracije** koje su dostupne samo unutar tijela podprograma.
- **sekvence izvršivih iskaza**

Deklaracija podprograma uključuje:

ime podprograma,
listu parametara ako postoje i
tip povratne vrijednosti ako postoji.

Lokalne deklaracije su dostupne samo unutar tijela podprograma.
Lokalne deklaracije i izvršivi iskazi formiraju **tijelo podprograma**.

Podprogrami koji vraćaju vrijednost nazivaju se **funkcije**, a oni koje to ne rade nazivaju se **procedure**.

→ **C** nema odvojenu sintaksu za procedure umjesto toga pišemo funkciju koja vraća **void** što znači da nema vrijednosti.

void proc(int a, float b) ;

Ovakva funkcija ima iste osobine kao procedura u drugim jezicima.

Procedura se u nekim jezicima poziva sa **call** iskazom. Npr. **call proc(x,y)** dok u nekim jezicima uključujući i **C** piše se ime procedure iza kojeg slijede aktuelni parametri. **proc(x,y)**

U **C**-u tip funkcije prethodi njenu deklaraciju:

int func(int a, float b) ;

dok u nekim jezicima ćete naći sintaksu oblika:

function Func(A:Integer; B:Float) return Integer;

Pošto su svi **C** podprogrami **funkcije**, često se funkcije koriste u ne-matematičkoj situaciji tipa: ulazno – izlaznih podprograma.

Podaci se predaju u podprogram u formi sekvence vrijednosti koja se naziva **parametri**. Koncept je preuzet iz matematike gdje se funkciji daje sekvenca argumenata.

Ovdje postoje dva pojma koja se moraju jasno shvatiti i razgraničiti.
-**Formalni parametar** je deklaracija koja se pojavljuje u deklaraciji podprograma. Računanje u tijelu podprograma je na ovim parametrima.
-**Aktuelni parametar** je vrijednost koju program iz kojeg se poziva šalje podprogramu.

Predaja parametara podprogramu :
copy-in semantics ili **call-by-value**
copy-out semantika

Međutim **copy-based** parametar passig mehanizam **ne može riješiti** efektivno prouzrokovan sa velikim parametrima.
Rješenje za ovo **je** poznato kao **call-by-reference** ili **referencna semantika** (tj. poziv po referenci).
Sastoji se u predaji adrese aktuelnog parametra i pristupa parametru indirektno.

C ima samo jedan parametar passing mehnizam a to je **copy-in**:
U želji da se postigne referenca ili copy-out semantika **C** programer mora koristiti **pointere.**
Read-only pristup parametru može se koristiti specificirajući ga sa **const**.
const se koristi za pojašnjavanje čitljivosti značenja parametara za čitaoca programa, i da zaustavi potencijalne greške.
Drugi potencijalni problem sa parametrima u **C**-u je da **nizovi** nemogu biti parametri.
Ako se nizovi moraju predati, predaje se adresa prvog elementa niza.
Po konvenciji **ime niza** je automatski konstruisano da bude pointer na prvi element.

Blok je programska struktura koja se sastoji od deklaracija i izvršivih iskaza.

Slična definicija je data za tijelo podprograma i preciznije je da je tijelo podprograma blok.
Blokovi i procedure mogu se ugnježdavati jedne u druge.

Blok struktura se prvo definisala u Algolu koji je uključivao i procedure i neimenovane blokove.
Pascal sadrži ugnježdene procedure ali ne i neimenovane blokove, **C sadži neimenovane blokove ali ne ugnježdene procedure.**

Neimenovani blokovi obično se koriste za restrikciju vidljivosti varijabli (deklarisanje samo tamo gdje je potrebno, umjesto na početku podprograma).
Ugnježdene procedure se koriste za grupisanje iskaza koji se izvršavaju više puta unutar podprograma a referiraju na lokalne varijable koje ne mogu biti vidljive od strane podprograma.

1.Područje (scope) varijable je segment programa unutar kojeg je definisana.
Često se naziva i **oblast važenja**.
Oblast važenja nekog podatka može biti:
-lokalno,
-globalno ograničena i
-globalno neograničena.
2.Vidljivost (Visibility) - Varijabla je vidljiva unutar podsegmenta svog područja ako joj se može direktno pristupiti po imenu.

3.Vrijeme života (Lifetime) varijable je interval za vrijeme izvršavanja podprograma kada je memorija dodjeljena varijabli.

Može se uočiti da je vrijeme života **dinamička osobina** i zavisna od run-time izvršavanjaa programa, dok scope i visibility je vezana za **statički programski tekst**.

Svaka deklaracija se veže sa tri osobine:
Scope varijable počinje tačkom deklaracije i završava na kraju bloka u kojem je definisana.
Scope **Global** varijable **uključuje cijeli program**, dok scope **Local** je **limitiran na jednu proceduru**.
Svakoj varijabli može se direktno pristupiti u okviru scopa.
Vrijeme života varijable je od početka izvršavanja bloka do kraja njegovog izvršavanja.

Global egzistira za vrijeme izvršavanja cijelog programa.
Takve varijable se nazivaju i **statičke (static),** jednom kada se alocira prostor on živi do kraja programa.
Lokalna varijabla ima dva vremena života koja odgovaraju 2 poziva lokale procedure.
Svaki put kada se kreira, alocira se različito područje za ovu varijablu.
Lokalne varijable se nazivaju **automatske (automatic)** jer se automatski alociraju kada se procedura pozove (pri ulasku u blok) i oslobodi kada se procedura vraća.

Konceptualno limit nivoa ugnježdenja ne postoji, ali mnogi kompajleri postavljaju taj limit.

scope varijable je od tačke deklaracije do kraja bloka, a vidljivost je ista osim ako nije skrivena sa unutrašnjom deklaracijom.

Prednosti blok strukture je da obezjedi lagan i efektivan metod dekompozicije procedure. Blok strukture su veoma važne kada se vrše kompleksna računanja.

Većina imperativnih programskih jezika vrši iteraciju, to su **petlje**. međutim **rekurzija je** matematički koncept koje je često nepodržan od strane programskih jezika.
Koje osobine se zahtjevaju da bila podržana rekurzija?

- -Pošto se iste sekvence mašinskih instrukcija koriste za izvršavanje svakog poziva funkcije kod ne treba modifikovati (Ako je program smješten u ROM po definiciji se ne može modifikovati.)
- -Za vrijeme run-time mora biti moguće alocirati proizvoljan broj memorijskih ćelija za parametre i lokalne varijable.(Način alokacije memorije podržan je sa stack arihtekturom.)

Ali zašto koristiti rekurziju? Mnogi algoritmi su elegantniji i čitljiviji koristeći rekurziju dok je iterativno rješenje teže za programiranje i podložno bugovima.

Kako se stek koristi za implementaciju programskog jezika?
Stek se koristi za smještanje informacija vezanih za poziv procedure, uključujući lokalne varijable i parametre koji se automatski alociraju na ulazu u proceduru i oslobađa nakon izlaska iz procedure.

Razlog što je stek odgovarajuća struktura podataka je što procedure nastaju i završavaju u LIFO redosljedu i svaki akcesibilni podatak koji pripada proceduri.

Klasa memorije jednog podatka određuju vrijeme njegovog postojanja u programu.

Podaci programskog jezika **C** pripadaju jednoj od 4 klase memorije:
auto,
register,
static,
extern

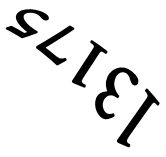
Ovi su novi pojmovi koje vežemo sa dosad uvedenim:

1.oblast važenja:
-**lokalni podaci** - za podatak kažemo da je lokalan, ukoliko se on nalazi unutar jednog funkcijskog bloka ili bloka naredni i ako je unutar tog bloka deklarisan
-globalni podaci - globalno ograničeni podaci-poznat u cijelom izvornom programu u kome je deklarisan i može se koristiti unutar ovog izvornog programa)

2.ciklus trajanja:
statički- smješteni u memoriji za svo vrijeme izvršavanja i
automatski - smješteni u memoriji samo za vrijeme izvršavanja bloka kojem pripadaju.

Memorijska klasa auto

Ova klasa se često naziva i lokalna.
Sve varijable deklarisane unutar bloka ili funkcija su po defaultu automatske.
Kada se funkcija pozove ona se smješta na stek i podiže se sa steka kada se funkcija završi.
Unutar funkcije je ekvivalentno:
int bra, brb; i auto int bra, brb;
float brc, brd ; i auto float brc, brd ;



Memorijska klasa static

a)unutra funkcije

b) Svaki podatak klase memorije **static** koji je deklarisan unutar jedne funkcije ima **lokalnu oblast važenja** i **statički ciklus trajanja**.
Ovi podaci su slični podaci klase auto uz jednu bitnu razliku: **static podatak zadržava svoju vrijednost i poslije izvršavanja funkcije u kojoj je deklarisan**.
Iako **podatak static klase** zadržava svoju vrijednost u memoriji on je **poznat samo lokalno**, tj.unutar funkcije u kojoj je deklarisan.

Memorijska klasa register

Automatske varijable i formalni parametri funkcije su varijable koje mogu koristiti **register** atribut.
Ako se varijable deklarišu kao **register** varijable tada:
-smještaju se u register računara ako je moguće
-tip mora biti **int**, **char**, **pointer**
Razlog zbog postojanja ove klase je u **brzini pristupa**. Upisivanje i očitavanje vrijednosti smještenih u registrima je nekoliko puta brže od upisivanja i učitavanja vrijednosti smještenih u memoriji.

Klasa memorije extern

Podatak koji pripada klasi **extern** ima:
globalno neograničenu oblast važenja i statički ciklus trajanja.
To znači da se varijabla može koristi u cijelom izvornom programu i u svim izvornim programima.

Nizovi se koriste za pretvaranje podataka realnog svijeta u matematički model vektora i matrica.
Slogovi se koriste u poslovnom procesiranju podataka za modeliranje različitih podataka u jednu jedinicu.

Kao i za svaki tip mi moramo opisati skup **vrijednosti sastavljenog tipa podataka i operacija na ovoj vrijednosti**.

Diskusiju ćemo početi sa slogovima (u **C**-u se nazivaju **strukture**) ←.
Slog je sastavljen od skupa vrijednosti **različitog tipa** koje se nazivaju u zavisnosti od jezika **komponente**, **članovi**, **polja**.

U **C**-u se nazivaju **članovi** a u Pascalu polja.

Unutar deklaracije svako polje ima**: ime i tip**.

C dozvoljava inicijalizaciju slogova prilikom deklaracije.
Pristup pojedinim slogovima se vrši tako što se navedena **varijabla tipa strukture** zatim → **.** ← iza koje slijedi **ime** polja.

Selektirano polje sloga je normalna varijabla tipa selektiranog polja, i sve operacije koje odgovaraju tom tipu su primjenjive.

Ako je za polje potreban prostor u memoriji koji nije cjelobrojno djeljiv sa veličinom riječi, kompajler stavlja slog tako da osigura da svako polje je na početku riječi, jer pristup koji nije na noivou riječi je manje efektivan.

Za računanje broja bajta koje zauzima struktura u memoriji koristi se **sizeof** operator.

Niz je tip podataka sastavlen od više elemenata (slog) koji su **istog tipa**.

Ova polja (nazivaju se **elementi** ili komponente) nisu obilježeni sa različitim identifikatorom nego sa pozicijom unutar niza.

Snaga niza kao tipa podatka dolazi od sposobnosti efikasnog pristupa elementu korištenjem **index-a**.

Pošto su svi elementi istog tipa moguće je računati **lokaciju specifičnog elementa** množenjem indeksa sa veličinom elementa.
Korištenjem indeksa lako je pretraživati niz za specifični element ili sortirati ili preurediti elemente.

U nekim jezicima indeks može biti svaki tip sa kojim je moguće vršiti brojanje; to je integer tip i prebrojivi tip (uključujući karakterni i logički tip).

C ograničava indeksni tip na integer; prilikom deklaracije specificiramo koliko komponenti želimo:
Komponente niza mogu biti bilo kojeg tipa:

Slogovi i nizovi se mogu ugnježdavati u želji konstrukcije kompleksih struktura.
Najčešća greška koja dovodi do složenih bugova **je prekoračenje limita indeksa**:
Nizovi se predstavljaju u memoriji smještajući njihove elemente u sekvencu u memoriju.
C programeri povećavaju efektivnost programa sa nizovima ekspilicitnim pristupom nizu korištenjem **pointera umjesto indeksa**.

Multi dimenzionalni nizovi su veoma neifikasni jer svaka dodatna dimenzija zahtjeva dodatno množenje za računanje indeksa.

Izuzev Fortana svi jezici smještaju dvodimenzionalan niz kao sekvencu nizova.

Postoje dva načina definisanja višedimenzionalnih nizova direktno **i kao** kompleksna struktura.
Mi ćemo se ograničiti na nizove gdje dodavanje veće dimenzije je neposredno.
Inicijalizacija za lokalno definisane tabele, učitavanje, ispisivanje uglavnom se vrši pomoću for petlje.

Smještanje u memoriji je slično za dvodimenzionalne nizove i niz nizova.

Ako želimo postići bolje perfomanse u **C**-u programu možemo ignorisati dvo-dimenzionalnu strukturu niza i predstaviti da je jednodimenzionalan niz.

Ovo nije preporučljivo i treba pažljivo dokumentovati za korištenje.

String je niz karaktera, ali dodatni suport je zavisan od programske konvencije.

Unija predstavlja složeni tip podataka koji je veoma sličan strukturi.
Deklaracija unije se razlikuje od deklaracije strukture u tome što se umjesto rezervisane riječi **struct** koristi rezervisana riječ **union**.
Osim ove razlike postoji i jedna funkcionalna razlika : **svi elementi unije zauzimaju isti memorijski prostor**.
To znači da se svi podaci koji su deklarisani u jednoj uniji smještaju na istoj adresi memorije.
Dužina unije odgovara elementu koji zauzima najviše memorijskog prostora.
Kod definicije i deklaracije unije važi sve što smo već rekli za strukture.
Unije mogu da sadrže strukture i obrnuto.
Kao i struktura, unija ne može sadržati samu sebe, ali može sadržati pokazivač koji pokazuje na nju.
Sve operacije koje su dozvoljene kod struktura su dozvoljene i kod unija.
To znači da se pristup elementima unije vrši korištenjem operatora **'.'** i **'-->'**
Unije se koriste za opisivanje različitih varijanti podataka koje se mogu pojaviti u strukturama.

Programski jezik C omogućuje deklarisanje i pristup jednom ili više bitova koji se nalaze unutar jedne mašinske riječi.
Takvi objekti se nazivaju **polja bitova** i njih je moguće deklarisati samo unutar struktura.
Polje bitova je uvijek sastavljeno od međusobno povezanih bitova.
Pošto se polje bitova uvijek mora nalaziti unutar jedne mašinske riječi veličina konst ne smije biti veća od broja bitova mašinske riječi.
Ukoliko je broj preostalih bitova mašinske riječi nedovoljan za smještanje narednog polja to polje se smješta u sljedeću mašinsku riječ a ostatak prethodne ostaje neiskorišten.

Svako polje bitova može biti **imenovano ili neimenovano.**
Imenovano polje bitova mora imati jednoznačno ime koje se koristi za pristup polju.
Neimenovano polje bitova određuje broj bitova koje treba preskočiti prije nego što se izvrši smještanje sljedećeg polja.
Neimenovano polje dužine nula ima specijalnu namjenu. Ono prouzrokuje smještanje narednog deklarisanog polja na početak sljedeće mašinske riječi.

Adresu jednog polja bitova nije moguće odrediti jer se ono nalazi unutar mašinske riječi.
To ujedno znači da se adresni operator **&** i pokazivači ne mogu primjenjivati na njih.
Polja bitova se obično koriste za opisivanje registara i za deklaraciju indikatora računara.

Pointeri:
Varijabla - konvenijalna notacija za označavanje adrese memorijske lokacije.
Ime varijable je **statičko** i određuje se za vrijeme kompilacije programa:različita imena određuje različite lokacije.
Vrijednost pointer tipa je adresa.
Pointer varijabla sadrži adresu druge varijable ili konstante.

Objekat na koji se pokazuje naziva se **označeni objekat** i njemu pristupamo **indirektno**.
Unarni operator & daje adresu jednog objekta .
Opreand **&** može biti primjenjen samo na varijable i elemente polja.
Konstrukcije tipa: **&(x+1) i &3 su ilegalne.**
Unarni operator ***** tretira svoj operand kao adresu i pristupa istoj da pozove sadržaj.
Pointeri se mogu pojaviti u izrazima.
Unarni operatori ***** i **&** su prioritetniji od aritmetičkih operatora.
Poziv pointera se može javiti na lijevoj strani dodjeljivanja.
Ako px pokazuje na x onda

***px=0 postavlja x na 0 i**
*px+=1 povećava x za 1 isto kao i (*px)++

Zagrade su potrebne u zadnjem primjeru bez njih izraz bi povećao **px** a ne ono što pokazuje , jer su unarni operatori kao ***** i **++** procjenjeni s desna na lijevo.
Pošto su pointeri varijable sa njima se može manipulisati kao i sa drugim varijablama.

Unarni operator & vraća adresu od operanda koji ga slijedi.
Važno je biti svjestan razlike između konstantnog pointera i pointera na konstantni označeni objekat. Čineći objekat konstantnim ne štitimo označeni objekat od modifikacije.

Indirektni pristup podacima preko pointera zahtjeva dodatne instrukcije u mašinskom kodu: Uporedićemo **direktno pridruživanje sa indirektnim pridruživanjem** ;
npr:

```
int i,j ;
int *p=&i ;
int *q=&j ;
i=j ;      /* direktno pridruživanje */
*p=*q;   /*indirektno pridruživanje */
```

U ranim 70 godinama kada se dizajnirao **C** i Pascal računari su obično imali 16K ili 32K glavne memorije, i adrese su smještale na 16 bita.
Sada kada su PC računari i radne stanice sa više megabajta memorije pointeri se smješaju na 32 bita.
Ustvari, zbog toga što se upravljanje memorijom bazira na keširanju i tzv. paging-u pristup podacima je brži nego pristup nizovima u neprekidnim lokacijama.

Jezici koji imaju samo copy-in mehanizam predaje paramtera funkciji, koriste predaju pointera na vrijednosti u slučaju kada se mora promijenti argument funkcije.
C ime samo jedan mehanizam predaje parametara a to je copy-in mehanizam.
U želji da se postigne referenca ili copy-out semantika C programer mora koristiti pointere.
Primjer predaje parametra po referenci je funkcija koja treba zamijeniti dva elementa.
U C-u nizovi i strukture i ne mogu biti parametri funkcije.
Ako se **nizovi** moraju predati **predaje se adresa prvog elementa niza**,
Parametri mogu biti bilo kojeg tipa uključujući nizove, slogove i druge kompleksne podatke.

C nudi osim pridruživanja pointera, izjednačavanja i dereferenciranja i mogućnost konstrukcije pointere na implicitne sekvencijalne adrese.

I aritmetika je moguća na pointer vrijednostima.
U nekim programskim jezicima uključujući i **C** postoji jak odnos između pointera i polja.
Bilo koja operacija koja može biti postignuta sa dopisivanjem indeksa polju, također može biti napravljena pomoću pointera.

Pointer verzija će biti općeniti brža , ali teža za shvatiti.

Niz funkciji se predaje tako što se preda početna adresa niza.
Unutar pozvane funkcije njegov argument je variajbla i pošto je ime polja pointer varijabla sadrži adresu.

Kao formalni parametri u definiciji funkcije:
char niz[] ; i **char *niz ;**
su potpuno ekvivalentni , tako da se unutar funkcije može manipulirati ili notacijom polja ili pointera,
može čak koristiti obadvije vrste operacija.

Imperativni programski jezici imaju eksplicitne izraze ili iskaze za **alociranje** i **dealociranje** memorije.
C koristi **malloc** koji je opasan jer ne postoji provjera da li je područje memorije korektno za veličinu označenog tipa. Preporučuje se korištenje **sizeof**-a.

Za oslobađanje memorije koristi se : free(p) ;

Memorija upravljana od strane **malloc** i **free** je stek, ili LIFO memoija

Operacije na strukturama su :
- pristupanje pojedinih elemenata korištenjem operatora:
  “.” i “->”
- korištenje adrese strukture korištenjem adresnog operatora **&**
- određivanje veličine strukture korištenjem operatora **sizeof**
- dodjeljivanje svih elemenata jedne strukture drugoj

Već smo upoznali pristup elementima strukture preko operatora ‘**.**’ (**direktni** pristup)

Elementima strukture može se pristupiti i preko **pokazivača**.
Ovaj način pristupa naziva se **indirektni** i vrši se korištenjem operatora “**-->**”.

Operator “**-->**” je binarni operator čiji lijevi operand predstavlja **pokazivač koji pokazuje na neku strukturu**, dok je desni operand element te strukture.
Pokazivači predstavljaju idealno sredstvo za vršenje različitih operacija nad strukturama.

Naredbu sa operatorom “**-->**” je na veoma jednostavan način moguće transformisati u ekvivalentne naredbe sa operatorom “**.**” i obrnuto.

Operator . ima veći prioritet od operatora *.

Dodavanjem jedinice pokazivaču koji pokazuje na neku strukturu se vrši povećanjem njegove adrese za dužinu cjelokupne strukture. To znači da povećanje pokazivača za neku konstantnu vrijednost ima smisla samo ako on pokazuje na strukturu koja je element neke tabele.

U većini programskih jezika postoje **rekurzivne strukture** tj. strukture koje sadrže same sebe se definišu korištenjem pokazivača.

Rekurzivne strukture omogućuju rješavanje čitavog niza problema. Npr. korištenjem rekurzivnih struktura je moguće konstruisati **ulančane liste** ili **stabla** pri čemu broj elemenata odnosno čvorova ne mora biti unaprijed poznat.

U **C**-u jedine operacije koje se mogu izvršiti nad **strukturama** su uzimanje njezine adrese **&** i **pristup** jednom od njezinih članova.
To implicira da strukture ne mogu biti dodjeljene ili kopirane kao jedinice i ne mogu biti predane ili vraćene od funkcija.

Ponteri na strukture nemaju ta ograničenja pa se **predaja struktura funkciji** vrši **preko pointera**.
Drugi način je naravno predaja komponenti odvojeno...

LEKCIJA 1&2&3: Getting Started with C, The Components of a C Program, Storing Data: Variables and Constants

→ Izgled C Programa:

```
preprocesorske direktive
globalne deklaracije
main()
{
  lokalne varijable na funkciji main ;
  iskazi pridruženi sa funkcijom main ;
}
f1()
{
  lokalne varijable na funkciji 1 ;
  iskazi pridruženi sa funkcijom 1 ;
}
f2()
{
  lokalne varijable na funkciji 3 ;
  iskazi pridruženi sa funkcijom 2 ;
}
.
.
etc
```

Sve preprocesorske direktive počinju sa **#** i moraju počinjati u prvoj koloni.

Funkcija je nezavisni dio programskog kooda koji obavlja neki zadatak i kojoj je dodjeljeno ime.
Pozivajući ime funkcije, vaš program može izvršiti kood u funkciji.
Također, program može slati informacije, zvane argumente, funkciji, i funkcija može vratiti informacije glavnom dijelu programa.
Dva tipa funkcija u **C**-u su **bibliotečne funkcije** (library functions), koje su dio **C** kompajlerovog paketa, i **korisničko-definisane** (user-defined functions) **funkcije**, koje vi, programer kreirate.

Jedina komponenta koja je potrebna svakom **C** programu je **main()** funkcija. U zagradama su “statements” (iskazi) koje sadrže glavno tijelo programa. U normalnim okolnostima, izvršenje programa počinje sa prvom “statement” (iskazom) u main()-u i prekida se sa zadnjim “statement” (iskazom) u main()-u.

Varijabla je ime pridruženo memorijskoj lokaciji gdje su podaci. Program koristi varijable da smjesti svakakve podatke prilikom njegovog izvršenja. U **C**-u, varijabla mora biti deklarisana prije nego što se može upotrijebiti. Deklaracija varijable informiše kompajler o njenom imenu i tipu podataka koje ona sadrži.

Prototip funkcije obezbjeđuje **C** kompajler sa imenom i argumentima funkcije sadržane u programu. Mora se pojaviti prije nego što se koristi funkcija. prototpi funkcije je različit od deklaracije funkcije, koja sadrže stvarne “statements”(iskaze) koje prave funkciju.

Stvarni posao **C** programa se izvršava pomoću “statements” (**iskaza**). **C** “statements” (iskazi) prikazuju informacije on-screen, čitaj ulaz sa tastature, izvrši matematičke operacije, zovi funkcije, čitaj file-ove sa diska, i mnoge druge koje su potrebne programu da se izvrši.
Printf() “statement” (iskaz) je bibliotečna (library) funkcija koja prikazuje informacije na ekranu (on-screen). Printf() iskaz može prikazati običnu tekst poruku ili poruku i vrijednost jedne ili više programskih varijabli.

Scanf() iskaz je također bibliotečna funkcija. Ona čita podatke sa tastature i pridružuje te podatke jednoj ili više programskoj/ih varijabli.

Listing 2.1. MULTIPLY.C.

```
1:  /* Program za računanje proizvoda dva broja. */
2:  #include <stdio.h>
3:
4:  int a,b,c;
5:
6:  int product(int x, int y);
7:
8:  main()
9:  {
10:     /* Input the first number */
11:     printf("Enter a number between 1 and 100: ");
12:     scanf("%d", &a);
13:
14:     /* Input the second number */
15:     printf("Enter another number between 1 and 100: ");
16:     scanf("%d", &b);
17:
18:     /* Calculate and display the product */
19:     c = product(a, b);
20:     printf ("%d times %d = %d\n", a, b, c);
21:
22:     return 0;
23: }
24:
25: /* Function returns the product of its two arguments (user-defined
function)*/
26: int product(int x, int y)
27: {
28:     return (x * y);
29: }
```

```
Enter a number between 1 and 100: 35
Enter another number between 1 and 100: 23
35 times 23 = 805
```

Programski iskaz u liniji **19** zove funkciju s imenom **product()**. Drugim riječima, izvršava programske iskaze koje se sadrže u funkciji **product()**. On još šalje i argumente **a** i **b** funkciji. Nakon izvršenja iskaza u **product()**, **product()** vraća vrijednost programu. Ova vrijednost je smještena u varijabli s imenom **c**.

return:
Linije **22** i **28** sadrže "**return**" iskaze. "**return**" iskaz (statement) u liniji **28** je dio funkcije **product()**. Ona računa proizvod varijabli **x** i **y** i vraća vrijednost programskom iskazu koji se zove **product()**.
"**return**" iskaz u liniji **22** vraća vrijednost **0** operativnom sistemu upravo prije nego što se program završi.

Koriistite zagrade (**{}**) da zatvorite (završite) programske linije koje tvore svaku **C** funkciju → uključujući i **main()** funkciju. Grupa od jednog ili više iskaza zatvorenih zagradama se zovu **blok**.

Svaki **byte** memorije ima jedinstvenu **adresu** kojom je identifikovan → adresu kojom se takođe razlikuje od svih ostalih byte-ova u memoriji. Adrese se pridružuju memorijskim lokacijama po redu, počevši od **0** i povećavajući do limita sistema.
Za većinu komapjlera, **dužina varijable** u **C**-u može biti do **31 karakter** (i više, ali kompajler gleda samo prvih 31 karaktera njenog imena).

Typedef ključna riječ se koristi da se kreira novo ime za već postojeći tip podatka(data). U stvari, **typedef** kreira sinonim.
Npr. iskaz:

```
typedef int integer;
```

Kreira integer kao sinonim za **int**. onda možete koristiti **integer** da definišete varijable tipa **int**, kao u ovom primjeru:

```
integer count;
```

Primjetite da **typedef** ne kreira novi tip podataka, već samo dopušta da koristite drugo ime za prije definisani tip podatka. Najčešća upotreba **typedef**-a se tiče stanja tipa podataka.

Kada deklarišete varijablu, naredite kompajleru da podesi prostor za smještanje te varijable. Vrijednost smještena u taj prostor → vrijednost varijable → nije definisana.
Može biti nula, ili neka druga, random, "smeće" vrijednost. Prije nego što koristite varijablu, morate je uvijek **inicijalizirati** sa nekom poznatom vrijednošću. Ovo možete uraditi nezavisno o deklaraciji varijable koristeći iskaz pridruživanja (**assignment statement**), kao u sljedećem primjeru:

```
int count;   /* podesi sa strane prostor za count */
count = 0;   /* smjesti 0 u count */
```

NE koristite varijable koje nisu inicijalizirane. Rezultat može biti neočekivan.

C ima dvije vrste konstanti, svaka sa svojom specifičnom upotrebom.
Literalna konstanta (**literal constant**) je vrijednost koja se direktno unosi u izvorni kood (source code) kad god je potrebna.
Evo dva primjera:

```
int count = 20;
float tax_rate = 0.28;
```

Prisustvo ili odsustvo decimalne tačke razlikuje **float**ing-point konstante od integer konstanti.
Literalne konstante napisane sa decimalnom tačkom su **float**ing-point konstante i predstavljene su **C** kompajleru sa brojem duple preciznosti (**double**-point number). **Float**ing-point konstante mogu biti napisane u standardnoj decimalnoj notaciji, kao što je prikazano na sljedećim primjerima:

```
123.456
0.019
100.
```

Primjetite da treća konstanta, **100.**, je napisana sa decimalnom tačkom iako je **int**eger (nema dio iza tačke (frakciju)). Decimalna tačka uzrokuje da je **C** kompajler tretira konstantu kao vrijednost sa duplom preciznosti (**double**-point value). Bez decimalne tačke, bila bi tretirana kao **int**eger konstanta.

Konstanta koja počinje sa bilo kojom cifrom **osim 0** (nule) se interpretira kao **int**eger.
Konstanta koja počinje **sa 0** se interpretira kao **oktalni int**eger (8-brojni bazni sistem).
Konstanta koja počinje **sa 0x ili 0X** se interpretira kao **heksadecimalna konstanta** (16-brojni bazni sistem).

Simbolična konstanta (**simbolic constant**) je konstanta koja se predstavlja sa imenom (simbolom) u vašem programu. Kao i literal konstanta, simbolična konstanta se ne može mijenjati. Kad god vam je potrebna vrijednost konstante u vašem programu, koriistite njeno ime kao što koristite i ime varijable. Stvarna vrijednost simbolične konstante treba biti unesena samo jednom, kada se prvi put definiše.

```
circumference = 3.14159 * (2 * radius);
area = 3.14159 * (radius)*(radius);
```

Prvi od ovih iskaza znači "**Pomnoži 2 puta vrijednost smještenu u varijabli radius, pa onda pomnoži rezultat sa 3.14159. Na kraju, pridruži rezultat varijabli koja se zove circumference**".
Druga prednost simboličnih konstanti postaje očigledna kada trebate promijeniti vrijednost konstante (samo jednom (prilikom definisanja)).
C ima dvije metode za definisanje simboličnih konstatni: **#define** direktivu i **const** ključnu riječ.

#define direktiva se koristi kako slijedi:

```
#define IMEKONSTANTE literal
```

ovo kreira konstantu s imenom **IMEKONSTANTE** sa vrijedošću **literal**. Literal predstavlja literalnu konstantu, kao što je opisano ranije. **IMEKONSTANTE** prati ista pravila opisana kao za ime varijable. Po konvenciji, imena konstanti se pišu velikim slovima.

Za prethodni primjer, potrebna **#define** direktiva će biti

```
#define PI 3.14159
```

Primjetite da **#define** linije ne završavaju sa **;**.
#define može biti smješteno bilo gdje u izvornom koodu (source code), ali imaju efekta samo za dio kooda koji slijedi iza **#define** direktive.
Najčešće, programeri grupišu **#define** zajedno, blizu početka file-a i **prije** starta **main()** –a.

Varijabla deklarisana da bude konstanta ne može biti modifikovana (mijenjana) prilikom izvršenja programa → već samo inicijalizovana za vrijeme deklaracije. Evo par primjera:

```
const int count = 100;
const float pi = 3.14159;
const long debt = 12000000, float tax_rate = 0.21;
```

const utiče na sve varijable u deklaracionoj liniji. U zadnjoj liniji, **debt** i **tax_rate** su simbolične konstante. Ako vaš program pokuša da modifikuje const varijablu, kompajler generiše poruku greške, kako je pokazano:

```
const int count = 100;
count = 200;        /* Ne kompajlira! Ne može da pridruži ili promijeni*/
                            /* vrijednost konstante. */
```

Koje su praktične razlike između simboličnih konstanti kreiranih sa **#define** direktivom i onim kreiranim sa **const** ključnom riječi? Razlika mora biti sa pokazivačima i vidljivosti varijable (pointers and variable scope).

Ako pridružite **–1** na **unsigned int** varijablu koja je dugačka **2** byte-a, kompajler će smjestiti najveći mogući broj u varijablu (**65535**).

LEKCIJA 4: → Statements, Expressions, and Operators ←

```
printf(
"Hello, world!"
);
```

Ovo, ipak, je nelegalno:

```
printf("Hello,
world!");
```

Da prekinete konstantnu liniju literalnog stringa, morate koristiti backslash karakter (\) tačno prije prekida. Tako da je sljedeće legalno:

```
printf("Hello,\
world!");
```

Ako napišete samo → **;** ← u liniji, tako kreirate **null-iskaz** → iskaz koji ne izvršava nikakvu akciju.

U **C**-u, **izraz** (**expression**) je sve što računa numeričke vrijednosti.
C izrazi dolaze u svim nivoima komplexnosti.
Najjednostavniji **C** izraz se sastoji od jednog predmeta: jednostavna varijabla, literal konstanta, ili simbolična konstanta. Evo četiri izraza:

Izraz	Opis
PI	Simbolična konstanta (definisana u programu)
20	Literalna konstanta
rate	Varijabla
-1.25	Još jedna literalna konstanta

Komplexni izrazi se sastoje od jednostavnijih izraza spojenih operatorima.
C izrazi postaju još interesantniji. Pogledajte sljedeća pridruživanja iskazima (statements):

```
x = a + 10;
```

Ovaj iskaz procjenjuje izraz **a + 10** i pridružuje rezultat u **x**.
S dodatkom, cijeli iskaz **x=a+10** je sam po sebi izraz koji procjenjuje vrijednost varijable na lijevoj stranu od znaka jednako.
Tako, možete pisati iskaze kao što slijedi, koji će pridruživati vrijednost izraza **a+10** obadvijema varijablama, **x** i **y**:

```
y = x = a + 10;
```

takođe možete pisati iskaze kao što je ovaj:

```
x = 6 + (y = 4 + 5);
```

rezultat ovog iskaza je da **y** ima vrijednost **9**, i **x** ima vrijednost **15**. Primjetite upotrebu zagrada, koje su potrebne da bi se ovaj iskaz kompajlirao.

C operatori spadaju u nekoliko kategorija:

- Operatori pridruživanja
- Matematicki operatori
- Relacioni operatori
- Logicki operatori

→ **Operator pridruživanja** je znak jednako (**=**).

x=y; "pridruži vrijednost **y**-a **x**-u."
varijabla = izraz(expression);

kada se izvrši, izraz se procjenjuje, i vrijednost rezultata se pridružuje varijabli.
C-ovi matematički operatori vrše matematičke operacije kao što su sabiranje i oduzimanje. **C** ima dva unarna matematička operatora i pet binarnih matematičkih operatora.

→ **C-ovi unarni matematički operatori:**

Operator	Simbol	Šta radi	Primjeri
Increment	++	Increments the operand by one	++x, x++
Decrement	--	Decrements the operand by one	--x, x--

```
++x;
--y;
```

su ekvivalenti sljedećih iskaza:

```
x = x + 1;
y = y - 1;
```

pogledajte ova dva iskaza (statements):

```
x = 10;
y = x++;
```

nakon što se izvrše ovi iskazi, **x** ima vrijednost **11**, a **y** ima vrijednost **10**. vrijednost **x**-a je pridružena **y**-u, pa je onda **x** inkrementiran (uvećan za **1**). U suprotnom, sljedeći iskaz rezultira da i **x** i **y** imaju vrijednost **11**. **x** je inkrementiran, pa je onda njegova vrijednost pridružena **y**-u.

```
x = 10;
y = ++x;
```

→ **C-ovi binarni matematički operatori:**

Operator	Symbol	Action	Primjer
Sabiranje	+	Sabira dva operanda	x + y
Oduzimanje	-	Oduzima drugi operand od prvog operanda	x - y
Množenje	*	Množi dva operanda	x * y
Dijeljenje	/	Dijeli prvi operand sa drugim operandom	x / y
Modul	%	Daje ostatak kada se prvi operand podijeli sa drugim operandom	x % y

Peti operator, **modul**, može biti novi. **Modul** vraća ostatak kada se prvi operand podijeli sa drugim operandom. Npr. **11 modul 4 = 3** (to je, 4 ide u 11 dva puta, sa ostatkom 3). Evo par primjera:

```
100 modulus 9 equals 1
10 modulus 5 equals 0
40 modulus 6 equals 4
```

→ Prethođenje (prednjačenje) C-ovih matematičkih operatora.

Operators	Relative Precedence
++ --	1
*** / %**	2
+ -	3

C-ovi relacioni operatori se koriste kada se porede izrazi, postavljajući pitanje kao što je, "**Da li je x veće od 100?**" ili "**Da li je y jednako 0**". Izraz koji sadrži relacioni operator procjenjuje da li je **true** (1) ili **false** (0).

→ C-ovi relacioni operatori

Operator	Symbol	Question Asked	Example
Equal	**==**	Is operand 1 equal to operand 2?	**x == y**
Greater than	**>**	Is operand 1 greater than operand 2?	**x > y**
Less than	**<**	Is operand 1 less than operand 2?	**x < y**
Greater than or equal to	**>=**	Is operand 1 greater than or equal to operand 2?	**x >= y**
Less than or equal to	**<=**	Is operand 1 less than or equal to operand 2?	**x <= y**
Not equal	**!=**	Is operand 1 not equal to operand 2?	**x != y**

→ If iskaz (statement)

If iskaz (statement) je jedan od **C**-ovih programsko kontrolnih iskaza.
U svojoj osnovnoj formi, **if** iskaz procjenjuje izraz i naređuje programsko izvršenje zavisno od rezultata procjene. Forma **If** iskaza je kako slijedi:

```
if (expression)
    statement;
```

ili

```
if (izraz)
        iskaz;
```

Ako izraz procjeni **true**, iskaz se izvršava. U suprotnom (**false**), iskaz se ne izvršava. U svakom slučaju, izvršavanje se dalje prebacuje na kod koji slijedi iza **if** iskaza.
If iskaz može kontrolisati izvršenje više iskaza pomoću upotrebe složenog iskaza, ili bloka.

Blok je grupa od dva ili više iskaza izmežu zagrada **{}**.
Blok može biti korišten bilo gdje gdje se koristi i jedan iskaz. Tako da možete pisati **if** iskaz ovako kako slijedi:

```
if (expression)
{
    statement1;
    statement2;
    /* dodatni kood ide ovdje */
    statementn;
}
```

NE pravite grešku da stavljate znak → **;** ← na kraju **if** iskaza. **If** iskaz treba završiti sa uslovnim iskazom koji ga slijedi.
statement1 (iskaz1) se izvršava bez obrzira da li je ili nije **x jednako 2**, zato što se svaka linija procjenjuje kao zaseban iskaz, a ne zajedno :

```
if( x == 2);          /* ; ne pripada (tačka – zarez)!  */
statement1;
```

U vašem programiranju, primjetićete da se **if** iskazi najčešće koriste sa relacionim izrazima, u drugim riječima, "**Izvršite sljedeći iskaz(e) ako i samo ako je uslov true**". Evo primjera:

```
if (x > y)
    y = x;
```

Ovaj kod pridružuje vrijednost **x**-a **y**-u samo ako je **x** veći od **y**-a. Ako **x** nije veće od **y**-a, nikakvo pridruživanje neće nastupiti.
If iskaz može opciono sadržavati i **else** član.
else član se uključuje kako slijedi:

```
/* nested if statement (ako postoji više "else"-ova) (ovdje ne, ali inače ☺) /*
          if (expression)
              statement1;
          else
              statement2;
```

Ako je izraz (expression) **true**, **statement1** se izvršava.
Ako je **izraz false**, izvršava se **statement2**.
Oda iskaza: **statement1** i **statement2** mogu biti složeni iskazi ili blokovi.

→ Prvo, svi relacioni operatori imaju manji prioritet nego matematički operatori.
Tako da ako napišete, **2** se sabira sa **x**, i rezultat se upoređuje sa **y**:

```
if (x + 2 > y)
```

ovo je ekvivalentno sa sljedećom linijom, koja je dobar primjer upotrebe zagrada radi jasnoće:

```
if ((x + 2) > y)
```

iako nisu potrebne **C** kompajleru, zagrade **(x+2)** jasno označavaju da je suma **x** i **2** koja se upoređuje sa **y**-om.

Takođe postoji prethođenje sa dva-nivoa kod relacionih operatora, kao što slijedi u tabeli:

→ Redosljed prethođenja C relacionih operatora.

Operators	Relative Precedence
< <= > >=	1
!= ==	2

Tako da, ako napišete:

```
x == y > z
```

je isto što i:

```
x == (y > z)
```

zato što **C** procjenjuje izraze **y>z**, rezultirajući sa vrijednostima **0** ili **1**. Sljedeće, **C** odlučuje da li je **x** jednako sa **1** ili **0** sadržano u prvom koraku. Rijetko ćete, ako ikad, koristiti ove kratke kontrukcije, ali trebate znati za njih.

NE stavljajte **znak pridruživanja** u **if** iskaze. Ovo može zbuniti one koji čitaju vaš kood. Mogu misliti da je to greška i promjeniti vaše pridruživanje u logičko jednak iskaz.
NE koristite "**nije jednako sa**" (not equal to) operator **(!=)** u **if** iskazu koji sadrži **else.** Skoro je uvijek jasnije da koristite **"jednako sa"** (equal to) operator **(==)** sa **else**-om.

Npr. Sljedeći kood:

```
if ( x != 5 )
   statement1;
else
   statement2;
```

Bi bio bolje napisan kao:

```
if (x == 5 )
   statement2;
else
   statement1;
```

Nekad morate pitati više od jednog relacionog pitanja odjednom. Npr. **"Ako je 7:00 sati i vikend i nisam na odmoru, zazvoni alarm."**
C-ovi logički operatori vam dopuštaju da kombinujete dva ili više relaciona izraza (expressions) u jedan izraz koji će procijenjivati da li je **true** ili **false**.

→ 3 C-ova logička operatora

Operator	Simbol	Primjer
AND	**&&**	***exp1* && *exp2***
OR	**||**	***exp1* || *exp2***
NOT	**!**	**!*exp1***

→ Upotreba C-ovih logičkih operatora:

Izraz	Na šta se procjenjuje
(***exp1* && *exp2***)	True (1) samo ako su oba, i exp1 i exp2 true; false (0) u drugom slučaju
(***exp1* || *exp2***)	True (1) ako je ijedan od **exp1** ili **exp2** is true; false (0) samo ako su oba false
(**!*exp1***)	False (0) ako je **exp1** true; true (1) ako je **exp1** false

Možete primjetiti da izrazi koji koriste logičke operatore se procjenjuju na **true** ili **false**, zavisno od **true/false** vrijednosti njihovih operanada (ili operanda).

→ Stvarni primjeri u koodu za C-ove logičke operatore:

Izraz	Na šta se procjenjuje
(5 == 5) && (6 != 2)	True (1), zato što su oba operanda true.
(5 > 1) || (6 < 1)	True (1), zato što je jedan operand true
(2 == 1) && (5 == 5)	False (0), zato što je jedan operand false
!(5 == 4)	True (1), zato što je operand false

Možete kreirati izraze koji koriste više logičkih operatora. Npr. Ako postavljate pitanje "**Dal li je x jednako 2,3 ili 4?.**" Napisali biste:

```
(x == 2) || (x == 3) || (x == 4)
```

Logički operatori često obezbjeđuju više od jednog načina da postavite pitanje. Ako je **x** integer varijabla, prethodno pitanje može takođe biti napisano na jedan od sljedećih načina:

```
(x > 1) && (x < 5)
(x >= 2) && (x <= 4)
```

Želite da napišete logičke izraze koji tvore tri samostalna poređenja:

1. **Da li je a manje od b?**
2. **Da li je a manje od c?**
3. **Da li je c manje od d?**

Želite da cijeli logički izraz procijeni **true** ako je uslov **3 true** i ako je bilo koji od uslova **1** ili uslov **2 true**.
Možete napisati:

```
a < b || a < c && c < d
```

Ipak, ovo **neće** raditi ono što ste željeli.
Zato što **operator && ima veći prioritet nego operator ||,** prethodni izraz je ekvivalentan sa:

```
a < b || (a < c && c < d)
```

i procjenjuje (evaluates) **true** ako je **(a<b) true**, bilo da su relacije **(a<c)** i **(c<d) true**.
Trebate napisati:

```
(a < b || a < c) && c < d
```

koje forsira **||** da bude pocijenjeno prije **&&**.
Npr., ako želite da povećate vrijednost **x**-a za **5**, ili, dodate **5 x**-u i pridružite vrijednost **x**-u. možete napisati:

```
x = x + 5;
```

Koristeći složeni operator pridruživanja, koji možete gledati kao skraćeni metod pridruživanja, mogli ste napisati:

```
x += 5;
```

Više uopšteno, složeni **operator pridruživanja** ima sljedeću sintaxu (gdje **op** predstavlja binarni operator):

```
exp1 op= exp2
```

Ovo je ekvivalentno kao da biste napisali:

```
exp1 = exp1 op exp2;
```

→ Primjeri složenih operatora pridruživanja:

Kada napišete ovo…	To je ekvivalentno sa…
x *= y	x = x * y
y -= z + 1	y = y - z + 1
a /= b	a = a / b
x += y / 8	x = x + y / 8
y %= 3	y = y % 3

Uslovni (kodicionalni) **operator** u **C**-u je jedini ternarni operator, što znači da koristi tri operanda. Njegova sintaxa je:

```
exp1 ? exp2 : exp3;
```

Ako se **exp1** procjeni na **true** (tj. nenula), cijeli izraz se procijenjuje na vrijednost **exp2**. Ako **exp1** procijeni **false** (tj. nula), cijeli izraz se procijenjuje na vrijednost **exp3**. Npr., ako sljedeći iskaz pridruži vrijednost **1 x**-u ako je **y true** i pridruži **100 x**-u ako je **y** false:

```
x = y ? 1 : 100;
```

Tako i, da napravite **z** jednako sa većim od **x** i **y**, možete napisati:

```
z = (x > y) ? x : y;
```

Možda ste primjetili da uslovni operator funkcioniše nešto kao **if** iskaz. Prethodni iskaz bi se mogao napisati i kao:

```
if (x > y)
     z = x;
else
     z = y;
```

Uslovni operator može biti korišten na mjestima gdje ne možete napisati **if** iskaz, kao npr. u jednom **printf( )** iskazu:

```
printf( "The larger value is %d", ((x > y) ? x : y) );
```

U određenim situacijama, **zarez** djeluje kao operator više nego kao separator. Možete formirati izraz odvajajući dva podizraza sa zarezom. Rezultat je sljedeći:

- Oba izraza se procijenjuju (upoređuju), samo što se lijevi izraz procjenjuje prvi.
- Cijeli izraz procjenjuje u odnosu na vrijednost desnog izraza.

Npr., sljedeći iskaz pridružuje vrijednost **b** **x**-u, zatim inkrementira **a**, pa onda inkrementira **b**:

```
x = (a++ , b++);
```

Zato što je **++** operator korišten u postfix metodi, vrijednost **b**-a se, prije nego što se inkrementira, pridruži **x**-u. Korištenje zagrada je neophodno zato što operator **zarez** ima manji prioritet, čak i manji nego operator pridruživanja.
Najčešća upotreba **zarez** operatora je u **for** iskazima.

→ Prioritet operatora:

Nivo	Operatori
1	**() [] -> .**
2	**! ~ ++ -- * *(indirection)* & *(address-of)* (*type*)**
-	**sizeof + *(unary)* - *(unary)***
3	*** *(multiplication)* / %**
4	**+ -**
5	**<< >>**
6	**< <= > >=**
7	**== !=**
8	**& *(bitwise* AND*)***
9	**^**
10	**|**
11	**&&**
12	**||**
13	**?:**
14	**= += -= *= /= %= &= ^= |= <<= >>=**
15	**,**
( ) is the function operator; **[ ]** is the array operator.	

Zapamtite da je **izraz** sve što procjenjuje numeričku (brojnu) vrijednost. **Komplexni izrazi** (complex expressions) mogu sadržavati manje izraze, koji se zovu **podizrazi** (**subexpressions**).

Lekcija 5: → Functions: The Basics ←

Definisanje funkcije
Prvo definicija: Funkcija je imenovani, nezavisni dio **C** kooda koji obavlja određeni zadatak i opcionalno vraća vrijednost programu koji je pozvao. Sad, pogledajmo dijelove ove definicije:
Funkcija je imenovana. Svaka funkcija ime jedinstveno ime. Koristeći to ime u nekom drugom dijelu programa, možete izvršiti iskaze koji se sadrže u toj funkcji. Ovo se zove pozivanje funkcije (*calling the function*). Funkcija može biti pozvana iz unutrašnjosti neke druge funkcije.
Funkcija je nezavisna. Funkcija može izvršavati svoj zadatak bez miješanja sa drugim dijelovima programa.
Funkcija obavlja određeni zadatak. Ovo je jednostavan dio definicije. Zadatak je diskretan posao koji vaš program mora uraditi kao dio svoje sveukupne operacije, kao što je slanje linije texta printeru, sortiranje niza u numeričkom redosljedu, ili izračunavanje trećeg korjena.
Funkcija može vratiti vrijednost pozivajućem programu. Kad vaš program pozove funkciju, iskazi koje ona sadrži se izvršavaju. Ako želite, ovi iskazi mogu vratiti informacije nazad pozivajućem programu.

→ Ilustracija funkcije:

Listing **5.1** sadrži korisničko definisane funkciju (user-defined function).
Listing 5.1. Program koji koristi funkciju da izračuna kuub broja.

```
1:   /* Demonstrira jednostavnu funkciju */
2:   #include <stdio.h>
3:
4:   long cube(long x);
5:
6:   long input, answer;
7:
8:   main()
9:   {
10:     printf("Unesite integer vrijednost: ");
11:     scanf("%d", &input);
12:     answer = cube(input);
13:     /* Napomena: %ld je specifikator konverzije za */
14:     /* long integer */
15:     printf("\nKuub od %ld je %ld.\n", input, answer);
16:
17:     return 0;
18:  }
19:
20:  /* Funkcija: cube() – Kuub vrijednosti od varijable */
21:  long cube(long x)
22:  {
23:     long x_cubed;
24:
25:     x_cubed = x * x * x;
26:     return x_cubed;
27:  }
      Unesite integer vrijednost: 100
      Kuub od 100 je 1000000.
      Unesite integer vrijednost: 9
      Kuub od 9 je 729.
      Unesite integer vrijednost: 3
      Kuub od 3 je 27.
```

NAPOMENA: sljedeća analiza se fokusira na komponente programa koji se odnose direktno na funkciju. (nego da se objašnjava kompletan program)

ANALIZA: Linija **4** sadrži **prototip funkcije**, model za funkciju koja će se pojaviti kasnije u programu. Prototip funkcije sadrži **ime funkcije**, **listu varijabli koje joj moraju biti prosljeđene**, i **tip varijabli koje vraća**, ako ih vraća.

Gledajući u liniju **4**, možemo reći da je funkcija nazvana **cube**, da zahtjeva varijablu tipa **long**, i da će vratiti vrijednost tipa **long**.

Varijable koje se dostavljaju funkciji se zovu **argumenti**, i zatvoreni su zagradama iza imena funkcije. U ovom primjeru, argument funkcije je **long x**.

Ključna riječ prije imena funkcije indicira tip varijable koji funkcija vraća. U ovom slučaju, vraća se varijabla tipa **long**.
Linija **12** poziva funkciju **cube** i prosljeđuje varijablin unos (input) do argumenta funkcije. Vrijednost koju vraća funkcija se pridružuje varijabli **answer**.
Primjetite da su i unos (**input**) i **answer** deklarisani u liniji **6** kao **long** varijable, čuvajući se sa prototipom funkcije u liniji **4**.
Funkcija se sama po sebi zove definicija funkcije (**function definition**). U ovom slučaju, zove se **cube** i sadržana je u linijama od **21** do **27**.
Kao i prototip, definicija funkcije ima nekoliko dijelova. Funkcija počinje sa zaglavljem – linija **21**. Zaglavlje funkcije (**function header**) je na početku funkcije, i daje funkciji ime (u ovom slučaju ime je **cube**). Zaglavlje takođe daje funkcijski tip koji vraća i opisuje njene argumente. Primjetite da je zaglavlje funkcije identično kao i protorip funkcije (minus znak → **;** ←).

Tijelo funkcije, linije **22** do **27**, je zatvoreno u zagradama. Tijelo sadrži iskaze, kao u liniji **25**, koji se izvršavaju kad god se funkcija poziva. Linija **23** je deklaracija varijable koja izgleda kao deklaracija koju ste vidjeli prije, sa jednom razlikom: lokalna je. **Lokalne varijable** su deklarisane u tijelu funkcije. i na kraju, funkcija završava sa **return** iskazom, u liniji **26**, koji signalizira kraj funkcije. **return** iskaz takođe prosljeđuje vrijednost nazad ka pozivajućem programu. U ovom slučaju, vrijednost varijable **x_cubed** se vraća.

Ako uporedite strukturu **cube()** funkcije sa **main()** funkcijom, vidjet ćete da su iste. **main()** je takođe funkcija. koristili ste funkcije **printf()** i **scanf()**. Iako su **printf()** i **scanf()** bibliotečne funkcije (nasuprot korisničko-definisanim funkcijama (user-defined function)), one su funkcije koje mogu uzeti argumente i vratiti vrijednosti kao i funkcije koje vi kreirate.

→→→ Kako funkcije rade:

C program ne izvršava iskaze u funkciji sve dok se funkcija ne pozove od strane nekog drugog dijela programa. Kada se funkcija pozove, program može slati funkciji informacije u formi jednog ili više argumenata.
Argument je programski podatak koji je potreban funkciji da uradi svoj zadatak. Tada se izvrše iskazi u funkciji, obavljejući zadatke za koje je dizajnirana da uradi. Kada se funkcijski iskazi izvrše, izvršenje se prosljeđuje nazad na istu lokaciju u programu koja je pozvala funkciju. Funkcija može slati informacije nazad programu u formi povratne vrijednosti (return value).
Funkcija može biti pozvana onoliko puta koliko je potrebno, i funkcije mogu biti pozvane u bilo kojem redosljedu.

→→ Funkcije

→ Prototip funkcije:

```
return_type function_name( arg-type name-1,...,arg-type name-n);
```

→ Definisanje funkcije (Function Definition):

```
return_type function_name( arg-type name-1,...,arg-type name-n)
{
    /* iskazi (statements); */
}
```

Prototip funkcije obezbjeđuje kompajleru opis funkcije koja će biti definisana kasnije u programu. Prototip uključuje povratni tip (return tape) indicirajući tip varijable koji će funkcija vratiti. Takođe uključuje ime funkcije, koji će opisati šta funkcija radi. Prototip takođe sadrži tip varijabli argumenata (arg type) koji će biti prosljeđeni funkciji. Opcionalno, može sadržavati imena varijabli koje će biti prosljeđene. Prototip uvijek treba završavati sa tačka zarez-om **;**.

Definicija funkcije (function definition) je stvarna funkcija.
Definicija sadrži kood koji će biti izvršen. Prva linija definicije funkcije se zove zaglavlje funkcije. Nakon zaglavlja je tijelo funkcije, koje sadrži iskaze koje će funkcija uraditi. Tijelo funkcije treba da počinje sa otvorenom zagradom i završi sa zatvorenom zagradom.
Ako je funkcijski povratni tip bilo šta drugo osim **void**, **return** iskaz treba biti uključen, vraćajući vrijednost koja odgovara povratnom tipu (return type).

→ Primjeri prototipa funkcije (Function Prototype Examples):

```
double squared( double number );
void print_report( int report_number );
int get_menu_choice( void );
```

→ Primjeri definicije funkcije (Function Definition Examples):

```
double squared( double number )          /* zaglavlje funkcije  */
{                                        /* otvarajuća zagrada  */
    return(  number * number );          /* tijelo funkcije     */
}                                        /* zatvarajuća zagrada */
void print_report( int report_number )
{
    if( report_number == 1 )
        puts( "Printing Report 1" );
    else
        puts( "Not printing Report 1" );
```

→ Prednosti strukturalnog programiranja

Zašto je strukturalno programiranje tako divno? Postoje dva važna razloga:
- lakše je napisati struktuiran program, zato što su problemi komplexnog programiranja razbijeni na nekoliko jednostavnijih zadataka (tasks). Svaki zadatak je izvršava od strane funkcije čiji kood i varijable su izolovane od ostalog dijela programa.
- lakše je debagovati struktuiran program. Ako vaš program ima **bug** (nešto što uzrokuje da ne radi kako treba), struktuirani dizajn omogućava lakše pronalaženje tog problema u nekom dijelu kooda (određena funkcija).

Prednost struktuiranog programiranja je i u vremenu koje uštedite. Lakše je koristiti već postojeću funkciju u nekom drugom programu nego pisati svaki put novu (istu) iz početka.

→ <u>Planiranje struktuiranog programa:</u>
Ako ćete pisati struktuiran program, morate malo planirati. Ovo planiranje treba da se radi prije nego što napišete i jednu liniju kooda, obično sa olovkom i papirom. Vaš plan treba biti lista specifičnih zadataka koje obavlja vaš program. Počinjući sa globalnom idejom programskih funkcija. Ako planirate program koji će da se bavi vašim listama imena i adresa, šta vaš program treba da radi? Evo nekoliko očiglednih stvari:

- unesi novo ime i adrese
- modifikuj postojeće unose
- sortiraj unose po prezimenu
- printaj natpise za slanje pošte.

Sa ovom listom, podijelili ste program u 4 glavna zadatka, svaki koji može biti pridružen funkciji. Sad možete otići korak dalje, dijeljeći ove zadatke u podzadatke. Npr. “unesi novo ime i adrese” može biti podijeljeno u sljedeće podzadatke:

- Čitaj postojeću listu adresa sa diska
- Izbaci (prompt) korisnika za jean ili više novih unosa
- Dodaj nove podatke listi
- Snimi update-ovanu listu na disk

Tako i "modifikuj postojeće unose" zadatak se može podpodijeliti kako slijedi:

- Čitaj postojeću listu adresa sa diska
- Modifikuj (izmjeni (promjeni)) jedan ili više unosa.
- Snimi update-ovanu listu na disk

Možda ste primjetili da ove dvije liste imaju dva zajedništa podzadatka → jedan koji se bavi sa čitanjem sa i snimanjem na disk. Možete napisati jednu funkciju koju će pozivati obadvije funkcije (you know what i mean).

Ovaj metod programiranja rezultira u hijerarhijskoj ili slojevitoj programskoj strukturi.
Struktuiran program je organizovan hijerarhijski.

Kada pratite ovaj planirajući pristup, brzo možete napraviti listu diskretnih zadataka koje vaš program mora da uradi. Onda se možete boriti sa jednim po jednim zadatkom (task), što vam omogućava posvećivanje pažnje na jedan relativno jednostavan zadatak. Kada je ta funkcija napisana i radi kako treba, možete preći na sljedeći zadatak. Prije nego što primjetite, vaš program počinje da poprima oblik.

Puno **C** programa ima malu količinu koda u glavnom tijelu programa, tj. u **main()**-u. Teret (oblik) programskog kooda se nalazi u funkcijama. U **main()**-u, sve što možete naći su par desetina linija kooda koje usmjeravaju izvršenje programa među funkcijama. Često, meni (menu) je predstavljen osobi koja koristi program. Izvršenje programa se grana prema izboru korisnika. Svaka grana meni-a koristi različitu funkciju.

PLANIRAJTE prije pisanja kooda. Određujući strukturu vašeg programa unaprijed na vrijeme, možete uštediti vrijeme na pisanju kooda i na njegovom debagiranju.
NE pokušavajte da sve uradite u jednoj funkciji. Jedna funkcija treba da obavlja jedan zadatak, kao što je čitanje informacija iz datoteke (file-a).

→→→ Pisanje funkcije

→→ Zaglavlje funkcije:

Prva linija svake funkcije je zaglavlje funkcije, koja ima tri komponente, koja svaka služi određenoj funkciji.

→ Povratni tip funkcije (The Function Return Type):

Povratni tip funkcije specificira tip podatka koji funkcija vraća pozivajućem programu. Povratni tip može biti bilo koji od **C**-ovih podatkovnih tipova: char, int, long, float, ili double.
Takođe možete definisati funkciju koja ne vraća vrijednost koristeći povratni tip **void.** Evo par primjera:

```
int    func1(...)          /* Vraća tip int.   */
float  func2(...)         /* Vraća tip float. */
void   func3(...)         /* Ne vraća ništa.  */
```

→ Ime funkcije:

Možete nazvati funkciju kako god želite, sve dok se pridržavate pravila za imena **C** varijabli.
Ime funkcije mora biti jedinstveno (ne pridruženo bilo kojoj drugoj funkciji ili varijabli). Dobra je ideja da pridružite ime koje odražava šta funkcija radi.

→ Lista parametara:

Većia parametara koristi argumente, čije se vrijednosti prosljeđuju funkciji kada je pozvana. Funkcija treba da zna koje vrste argumenata da očekuje → tip podatka za svaki agrument. Možete proslijediti funkciji bilo koji od **C**-ovih tipova podataka. Tip argumenta se obezbjeđuje u zaglavlju funkcije od strane liste parametara.
Za svaki argument koji je prosljeđen funkciji, lista parametara mora sadržavati jedan unos (entry). Ovaj unos specificira tip podatka i ime parametra.

Npr.

```
long cube(long x)
```

Lista parametara sadrži **long x**, specificirajući da ova funkcija uzima jedan tipa **long** argument, predstavljen parametrom **x**. Ako postoji više od jednog parametra, svaki mora biti razdvojen sa zarezom.

→ Zaglavlje funkcije:

```
void func1(int x, float y, char z)
```

specificira funkciju sa tri argumenta: tip **int** nazvan **x**, tip **float** nazvan **y**, i tip **char** nazvan **z**. Neke funkcije ne uzimaju argumente, u čijem slučaju lista parametara treba da sadrži **void**, kao:

```
void func2(void)
```

NAPOMENA: Ne stavljajte znak → **;** ← na kraju zaglavlja funkcije. Ako ovo greškom uradite, kompajler će generisati poruku greške.

Nekad nastaje konfuzija oko **razlike između parametara i argumenta**.

→ **Parametar** je unos (entry) u zaglavlju funkcije, služi kao "čuvač-mjesta" za argument. Funkcijski parametri su fiksni, ne mijenjaju se tokom izvršenja programa.

→ **Argument** je stvarna vrijednost proslijeđena funkciji od strane pozivajućeg programa. Svaki put kada se funkcija poziva, mogu joj biti proslijeđeni drugačiji argumenti. U **C**-u, funkciji mora biti proslijeđen isti broj i tip argumenata svaki put kada se zove, ali vrijednost argumenata može biti različita (drugačija). U funkciji, argumentu se pristupa koristeći korespodirajuće ime parametra.

Listing 5.2. Razlika između argumenta i parametra

```
1:  /* Ilustrira razliku između argumenta i parametra. */
2:
3:  #include <stdio.h>
4:
5:  float x = 3.5, y = 65.11, z;
6:
7:  float half_of(float k);
8:
9:  main()
10: {
11:     /* U ovom pozivu, x je argument do half_of(). */
12:     z = half_of(x);
13:     printf("The value of z = %f\n", z);
14:
15:     /* U ovom pozivu, y je argument do half_of(). */
16:     z = half_of(y);
17:     printf("The value of z = %f\n", z);
18:
19:     return 0;
20: }
21:
22: float half_of(float k)
23: {
24:     /* k je parametar. Svaki put kada se half_of() pozove, */
25:     /* k ima vrijednost koja je prosljedjena kao argument. */
26:
27:     return (k/2);
28: }
        The value of z = 1.750000
        The value of z = 32.555000
```

→ Svaki put kada se funkcija zove, argumenti su proslijeđeni funkcijskim parametrima!!!! ←

ANALIZA: Gledajući u Listing **5.2**, možete vidjeti da je **half_of()** prototip funkcije deklarisan na liniji **7**.
Linije **12** i **16** zovu **half_of()**, i linije **22** do **28** sadrže stvarnu funkciju.
Linije **12** i **16** salju različite argumente **half_of()**-u.
Linija **12** šalje **x**, koji sadrži vrijednost **3.5,** i linija **16** šalje **y**, koji sadrži vrijednost **65.11**.
Kada se program starta, on printa tačne brojeve za svaku.
Vrijednosti u **x**-u i **y**-u se prosljeđuju u argument **k half_of()**-a. Ovo je kao kopiranje vrijednosti od **x** do **k**, pa onda od **y** do **x**. **half_of()** onda vraća ovu vrijednost poslije djeljenja sa **2** (linija **27**).

KORISTITE ime funkcije da biste opisali namjeru funkcije.
NE prosljeđujte vrijednosti funkciji koje joj nisu potrebne.
NE pokušavajte da prosljeđujete nekoliko (ili više) argumenata funkciji nego što je parametara.
U **C** programima, broj argumenata prosljeđenih mora odgovarati broju parametara.

→ Tijelo funkcije:
Tijelo funkcije je zatvoreno u zagradama, i ono slijedi odmah nakon zaglavlja funkcije. Ovdje se obavlja stvarni posao. Kada je funkcija pozvana, izvršenje počinje na početku tijela funkcije i prekida (vraća pozivajućem programu) kada naiđe **return** iskaz ili kada izvršenje dođe do zatvorene zagrade.

→ Lokalne varijable:
Možete deklarisati varijable u tijelu funkcije. One se zovu lokalne varijable. One su **privatne** za tu funkciju i razlikuju se od varijabli istog imena deklarisane negdje drugo u programu.
Lokalna varijabla se deklariše kao bilo koja druga varijabla. Može biti inicijalizirana nakon što se deklariše.
Možete deklarisati bilo koji tip **C**-ove varijable u funkciji. Evo primjera 3 lokalne varijable koje su deklarisane unutar funkcije:

```
int func1(int y)
{
    int a, b = 10;
    float rate;
    double cost = 12.55;
    /* kood funkcije ide ovdje... */
}
```

Varijable **a**, **b**, **rate** i **cost**, mogu biti korištene od strane kooda u funkciji. Primjetite da se funkcijski parametri smatraju varijabliskim deklaracijama, tako da varijable, ako ih ima, u listi parametra funkcije su također dostupne.

→ Tri pravila upravljaju upotrebu varijabli u funkciji:

- da koristite varijablu u funkciji, morate je deklarisati u zaglavlju finkcije ili u tijelu funkcije (osim globalnih varijabli)
- s ciljem da funkcija dobije vrijednost od pozivajućeg programa, vrijednost mora biti prosljeđena kao argument
- s ciljem da pozivajući program dobije vrijednost od funkcije, vrijednost mora biti eksplicitno vraćena od funkcije

Ova pravila se ne primjenjuju striktno. (zasad je tako ☺)

→ Funkcijski iskazi (statements)
Praktično nema ograničenja u iskazima unutar funkcije. Jedina stvar koju ne možete uraditi unutar funkcije je definisati drugu finkciju. Možete koristiti sve **C** iskaze, uključujući petlje, **if** iskaze, i iskaze pridruživanja.
Možete pozvati bibliotečne funkcije i druge korisničko-definisane funkcije.
C ne ograničava dužinu funkcija, ali iz praktičnih razloga trebate držati vaše funkcije relativno malim.

→ Vraćanje vrijednosti (return)

Da biste vratili vrijednost od funkcije, koristite ključnu riječ **return**, slijedeći **C** izraz. Kada izvršenje dođe do **return** iskaza, izraz se procjenjuje, izraz prosljeđuje vrijednost nazad ka pozivajućem programu. Vraćena vrijednost funkcije je vrijednost izraza.
Razmotrite sljedeću funkciju:

```
int func1(int var)
{
    int x;
    /* Kood funkcije ide ovdje... */
    return x;
}
```

Kada se funkcija pozove, iskazi u tijelu funkcije se izvršavaju do **return** iskaza.
return prekida funkciju i vraća vrijednost **x**-a pozivajućem programu (calling program). Izraz koji slijedi ključnu riječ **return** može biti bilo koji validan **C** izraz.

Funkcija može sadržavati nekoliko **return** iskaza. Prvi **return** koji se izvršava je jedini koji ima nekog efekta. Nekoliko **return** iskaza može biti efikasan način da vratite vrijednost od funkcije, kako je demonstrirano u

Listing-u 5.4.:

```
1:  /* Demonstracija korištenja više return iskaza u funkciji. */
2:
3:  #include <stdio.h>
4:
5:  int x, y, z;
6:
7:  int larger_of( int , int );
8:
9:  main()
10: {
11:     puts("Unesite dvije integer vrijednosti: ");
12:     scanf("%d%d", &x, &y);
13:
14:     z = larger_of(x,y);
15:
16:     printf("\nVeca vrijednost je: %d.", z);
17:
18:     return 0;
19: }
20:
21: int larger_of( int a, int b)
22: {
23:     if (a > b)
24:         return a;
25:     else
26:         return b;
27: }
```

```
Unesite dvije integer vrijednosti:
200 300
Veca vrijednost je: 300.
Unesite dvije integer vrijednosti:
300
200
Veca vrijednost je: 300.
```

Linija **11** u ovom programu je prvi put viđena: **puts()** → znači stavi string (**put string**) → to je jednostavna funkcija koja pokazuje string na standardnom izlazu, obično ekranu (monitoru).

→ Zapamtite da vrijednost koje vraća funkcija ima tip koji je specificiran u zaglavlju funkcije i prototipu funkcije. Vrijednost vraćena od strane funkcije mora biti istog tipa, ili kompajler generiše grešku. ←

NAPOMENA: Strukturalno programiranje sugeriše da imate samo jedan unos i jedan izlaz u funkciji. Ovo znači da trebate pokušati da imate samo jedan **return** iskaz u vašoj funkciji. Ponekad je program lakše čitati i održavati sa više **return** iskaza. U takvim slučajevima, održivost (maintainability) treba imati prednost.

→→ Prototip funkcije:

Program mora sadržavati prototip za svaku funkciju koja se koristi.
Šta je prototip funkcije i zašto je potreban?
Možete vidjeti iz ranijih primjera da je prototip funkcije identičan sa zaglavljem funkcije, sa dodatim tačka-zarezom na kraju. Kao i zaglavlje funkcije, prototip funkcije uključuje informacije o funkcijskom **return** tip (tipu koji funkcija vraća), ime i parametre. Posao prototipa je da kaže kompajleru o tipu koji funkcija vraća, imenu i parametrima. Sa ovom informacijom, kompajler može provjeriti svaki put pozive funkcije vašeg izvornog kooda (source code) i potvrditi da prosljeđujete tačan broj i tip argumenata funkciji i da pravilno koristite return vrijednost. Ako postoji razlika (nepoklapanje (mismatch)), kompajler generiše poruku greške.

Govoreći striktno, prototip funkcije ne mora tačno odgovarati zaglavlju funkcije. Imena parametara mogu biti različita, sve dok su istog tipa, broja, i u istom redosljedu. Nema razloga da se zaglavlje i prototip ne poklapaju; držeći iz identičnim čini izvorni kood lakšim za razumljevanje. Poklapanje ova dva takođe čini pisanje programa lakšim. Kada završite definisanje feunkcije, koristite cut-and-paste feature na vašem editoru da kopirate zaglavlje funkcije i kreirate prototip. Uvjerite se da dodate tačka-zarez (**;**) na kraju.

Gdje trebaju prototipi funkcije biti smješteni u izvornom koodu? Trebaju biti smješteni prije početka **main()**-a ili prije nego što se definiše prva funkcija. radi čitljivosti, najbolje je grupisati sve prototipe na jednom mjestu.

NE pokušavajte vraćati (return) vrijednost koja ima različit tip od tipa funkcije.
KORISTITE lokalne varijable kad god je to moguće.
NE dozvolite da funkcije postanu prevelike. Ako postanu, razbijte ih na manje, razdvojene zadatke.
OGRANIČITE limit svake funkcije na samo jedan zadatak.
NEmojte imati više **return** iskaza ako nisu potrebni. Imajte samo jedan **return** kad god je to moguće; ipak, nekad je korištenje više **return** iskaza lakše i jasnije.

→→ Prosljeđivanje argumenta funkciji:

Da argument proslijednite funkciji, nabrojite ih u zagradi nakon imena funkcije. Broj i tip svakog argumenta mora odgovarati parametrima u zaglavlju funkcije i prototipu.
Npr., ako je funkcija definisana da uzme dva tipa **int**, argumenta, morate proslijediti tačno dva **int** argumenta → ni manje → ni više → i ne drugog tipa. Ako pokušate da prosljedite funkciji netačan broj i/ili tip(ova) argumen(a)ta, kopajler će to otkriti, na osnovu informacija u prototipu funkcije.

Ako funkcija uzima više argumenata, nabrojani argumenti u funkciji se pridružuju funkcijskim parametrima po redosljedu: prvi argument prvom parametru, drugi argument drugom parametru, itd.

Svaki argument može biti pravilan **C** izraz: konstanta, varijabla, matematički ili logički izraz, ili čak neka druga funkcija (ona sa **return** vrijednosti).

Npr., ako su **half()**, **square()**, i **third()** funkcije koje vraćaju vrijednost (return), možete napisati:

```
x = half(third(square(half(y))));
```

Program prvo poziva **half()**, prosljeđujući joj **y** kao argument. Kada se izvršenje vrati od **half()**, program poziva **square()**, prosljeđujući **half()**-ovu **return** vrijednost kao argument. Sljedeće, **third()** se poziva sa **square()**-ovom povratnom (return) vrijednosti kao argumentom. Onda, **half()** se poziva ponovo, ovaj put sa **third()**-ovom povratnom vrijednosti kao argumentom. i na kraju, **half()**-ova vrijednost koju vraća, se pridružuje varijabli **x**.

Slijedi ekvivalentan dio kooda:

```
a = half(y);
b = square(a);
c = third(b);
x = half(c);
```

Pozivanje funkcije:
Postoje dva načina da se pozove funkcija. Svaka funkcija se može pozvati jednostavno koristeći njeno ime i nabrojane argumente, saame u iskazu, kao što slijedi na primjeru. Ako funkcija ima **return** vrijednost (koju vraća), ona se odbacuje.

```
wait(12);
```

Drugi metod može biti korišten samo sa funkcijama koje imaju vrijednosti koje vraćaju. Zato što ove funkcije procjenjuju na vrijednost (tj., njihovu **return** vrijednost), one su validni **C** izrazi i mogu biti korišteni svugdje gdje se **C**-ov izraz može koristiti. Već ste vidjeli izraz sa povratnom vrijednosti korišten kao desna strana iskaza pridruživanja. Evo još par primjera:

U sljedećem primjeru, **half_of()** je parametar funkcije:

```
printf("Half of %d is %d.", x, half_of(x));
```

Prvo, funkcija **half_of()** se poziva sa vrijednosti **x**, onda se poziva **printf()** koristeći vrijednost **x**-a i **half_of(x)**.

U ovom drugom primjeru, više funkcija se koristi u izrazu:

```
y = half_of(x) + half_of(z);
```

Iako je **half_of()** korišten dva puta, drugi poziv je mogao biti bilo koja druga funkcija. Sljedeći kood pokazuje isti iskaz, ali ne sve u jednoj liniji:

```
a = half_of(x);
b = half_of(z);
y = a + b;
```

Posljednja dva primjera pokazuju efikasan način da koristite **return** vrijednosti funkcije. Ovdje, funkcija se koristi sa **if** iskazom:

```
if ( half_of(x) > 10 )
{
    /* statements; */          /* ovo mogu biti bilo koji iskazi! */
}
```

Ako povratna vrijednost funkcije odgovara kriterijima (u ovom slučaju, ako **half_of()** vrati vrijednost veću od **10**), if iskaz je **true**, i njegovi iskazi se izvršavaju. Ako povratna (vraćena) return) vrijednost ne odgovara kriterijima, **if** iskazi se ne izvršavaju.

Sljedeći primjer je još bolji:

```
if ( do_a_process() != OKAY )
{
    /* statements; */          /* do error routine */
}
```

Ponovo, nisam dao tačne iskaze, niti je **do_a_process()** stvarna funkcija, ipak, ovo je važan primjer koji provjerava povratnu (return) vrijednost procesa da vidi da li se izvršava kako treba. Ako ne, iskazi vode računa o bilo kojim greškama ili čišćenju. Ovo je uobičajeno korišteno sa pristupnim informacijama u file-ovima, komparaciji vrijednosti, i alokaciji memorije.
Ako pokušate da koristite funkciju sa **void** povratnom (return) vrijednosti kao izrazom, kompajler generiše grešku.

ISKORISTITE prednost mogućnosti da stavljate funkcije u izraze.
NE pravite individualne iskaze zbunjivajući stavljanjem funo funkcija u nju. Trebate stavljati funkcije u vaš iskaz samo ako one ne čine kood dodatno zbuljujućim.

→→ Rekurzija:

Pojam rekurzije se odnosi na situaciju u kojoj funkcija poziva samu sebe direktno ili indirektno.
Indirektna rekurzija se javlja kada jedna funkcija poziva drugu funkciju koja kasnije zove prvu funkciju.
C dozvoljava rekurzivne funkcije, i one mogu biti korisne u nekim situacijama.

Npr, rekurzija može biti korištena za izračunavanje faktorijela nekog broja.
Faktorijel broja **x** se piše **x!** i računa se kao:

```
x! = x * (x-1) * (x-2) * (x-3) * ... * (2) * 1
```

Ipak **x!** možete računati i kao:

```
x! = x * (x-1)!
```

Još jedan korak dalje, **(x-1)!** možete izračunati koriteći istu proceduru:

```
(x-1)! = (x-1) * (x-2)!
```

Možete nastaviti računati rekurzivno dok ne dođete do vrijednosti **1**, u čijem slučaju ste završili. Program u **Listing-u 5.5** koristi rekuzivnu funkciju da izračuna faktorijal. Zato što program koristi unsigned integers, ograničen je na vrijednost unosa **8**; faktorijal od **9** i veći su van dozvoljenog opsega za integere.

Listing 5.5. Upotreba rekurzivne funkcije za izračunavanje faktorijaala.

```
1:   /* Demonstrira rekurzivnu funkciju. Izračunava */
2:   /* faktorijal broja. */
3:
4:   #include <stdio.h>
5:
6:   unsigned int f, x;
7:   unsigned int factorial(unsigned int a);
8:
9:   main()
10:  {
11:      puts("Enter an integer value between 1 and 8: ");
12:      scanf("%d", &x);
13:
14:      if( x > 8 || x < 1)
15:      {
16:          printf("Only values from 1 to 8 are acceptable!");
17:      }
18:      else
19:      {
20:          f = factorial(x);
21:          printf("%u factorial equals %u\n", x, f);
22:      }
23:
24:      return 0;
25:  }
26:
27:  unsigned int factorial(unsigned int a)
28:  {
29:      if (a == 1)
30:          return 1;
31:      else
32:      {
33:          a *= factorial(a-1);
34:          return a;
```

```
35:       }
36:   }
          Enter an integer value between 1 and 8:
          6
          6 factorial equals 720
```

ANALIZA: Prva polovina programa je kao i drugi programi.
Linija **4** → linija zaglavlja se ubacuje za **input/output** rutine.
Linija **6** deklariše par **unsigned int**eger vrijednosti.
Linija **7** je prototip funkcije za faktorijal funkciju.
Primjetite da uzima **unsigned int** kao svoj parametar i vraća **unsigned int**.
Linije od **9** do **25** su **main()** funkcije.
Linije **11** i **12** printaju poruku postavljući upit za vrijednost od **1** do **8** i onda prima unešenu vrijednost.
Linije **14** do **22** pokazuju interesantan **if** iskaz. Zato što vrijednost veća od **8** uzrokuje problem, ovaj **if** iskaz provjerava vrijednost. Ako je veća od **8**, printa se poruka greške; u drugom slučaju, program računa faktorial na liniji **20** i printa rezultat u liniji **21**. Kad znate da bi mogao biti problem, kao što je ograničenje veličine broja, dodajte kood da otkrije problem i spriječi ga.
Naša rekurzivna funkcija, **factorial()**, je locirana na liniji **27** do **36**.
Vrijednost koja se prosljeđuje se pridružuje **a**-u.
Na liniji **29**, vrijednost od **a** se provjerava. Ako je **1**, program vraća vrijednost od **1**. Ako vrijednost nije **1**, **a** se postavlja na jednako sa samim sobom puta faktorijal**(a-1)**.
Program poziva faktorijal funkciju ponovo, ali ovaj put vrijednost od **a** je **(a-1)**. Ako **(a-1)** nije jednako **1**, **factorial()** se poziva ponovo sa **((a-1)-1)**, što je isto kao i **(a-2)**. Ovaj proces se ponavlja sve dok iskaz na liniji **29** ne bude **true**. Ako je vrijednost faktorijala **3**, faktorijal se procjenjuje kako slijedi:

```
3 * (3-1) * ((3-1)-1)
```

Broj faktorijala je potreban u mnogim statističkim izračunavanjima.

Rekurzija je samo petlja, sa jednom razlikom od klasičnih petlji (loops). Sa rekurzijom, svaki put se poziva rekurzivna funkcija, novi set varijabli se kreira. Ovo nije tako sa ostalim petljama (loop) koje ćemo učiti kasnije.

SHVATITE i radite sa rekurzijom prije nego što je koristite.
NE koristite rekurziju ako će biti nekoliko iteracija. (**iteracija** je ponavljanje programskih iskaza). Rekurzija koristi mnogo resursa, zato što funkcija mora da pamti gdje je.

Možda se pitate u vašem izvornom koodu staljate vaše definicije funkcije. Za sada, one treba da idu u isti file izvornog kooda kao i **main()** i nakon završetka **main()**-a.

Stavljajte vaše prototipe funkcija prije main()**-a i vaše definicije funkcija poslije** main()**-a**

Q Da li main() treba da bude prva funkcija u programu?
A NE. To je standard u **C**-u da je **main()** funkcija prva funkcija koja se izvršava, ipak, ona može biti smještena bilo gdje u vašem izvornom koodu (file-u). Većina ljudi je smješta kao prvu, tako da se nalazi lako.

LEKCIJA 6: → Basic Program Control ←

→→→ Nizovi (Arrays): osnove

for iskaz i **nizovi** su usko povezani u **C**-u, tako da je teško definisati jedno bez objašnjenja drugog.

→ **Niz** (array) je indexna grupa od lokacija spremanja podataka (data storage locations) koje imaju isto ime i razlikuju se jedan od drugog po indexu (ili subscript (index)) → broj koji slijedi iza imena varijable, zatvoren zagradama.
Kao druge **C** varijable, **niz** mora biti deklarisan. Deklaracija **niza** porazumjeva i tip podataka i veličinu **niza** (broj elemenata u nizu). Npr., sljedeći iskaz deklariše **niz** nazvan (s imenom) data tipa **int** i ima **1000** elemenata:

```
int data[1000];
```

Elementima se pristupa pomoću indexa kao **data[0]** do **data[999]**. Prvi element je **data[0]**, a ne **data[1]**. U drugim jezicima, kao što je BASIC, prvi element niza je **1**; što nije tačno u **C**-u.

Svaki element ovog **niza** je ekvivalentan sa normalnom **int**eger varijablom i može se koristiti na isti način. Index **niza** može biti neka **C** varijabla, kao u ovom primjeru:

```
int data[1000];
int count;
count = 100;
data[count] = 12;     /* Isto kao i  data[100] = 12 */
```

NE deklarišite nizove sa indeximа većim nego što vam je potrebno (troši se mem.-a)
NE zaboravite da u **C**-u, nizovi su referencirani sa indexom **0**, a ne **1**.

→→→ FOR iskaz

for iskaz je kostrukcija **C** programiranja koja izvršava blok od jedan ili više iskaza određeni broj puta. Nekada se zove petlja zato što programsko izvršenje tipično peetlja kroz iskaze više od jednom.
for iskaz ima sljedeću strukturu:

```
for ( initial; condition; increment )
    statement;
```

initial (inicijaliziraj), ***condition*** (uslov), i ***increment*** su sve **C**-ovi izrazi, i iskaz je jedan ili sastavljeni **C** iskaz. Kada se prilikom programog izvršenja susretne sa **for** iskazom, dešava se sljedeće:

1. izraz "**initial**" (inicijaliziraj) se procjenjuje. **Initial** je uobičajeno iskaz pridruživanja koji podešava varijable na određenu vrijednost
2. izraz "**condition**" (stanje) se procjenjuje. **Condition** je tipično relacioni izraz.
3. ako je "condition" procjeni na **false** (tj, nula), for iskaz se prekida i izvršenje se prosljeđuje dalje do prvog iskaza koji slijedi iza iskaza.
4. ako je "condition" procjenjen na **true** (tj, nenula), **C** iskaz(i) u iskazu se izvršavaju.
5. procjenjuje se inkrement izraza, i izvršavanje se vraća na korak **2**.

Listing 6.1 koristi **for** iskaz da printa brojeve od **1** do **20**. Možete vidjeti da je rezultirajući kood više kompaktan nego što bi bio pri upotrebi odvojenih **printf()** iskaza za svaku od **20** vrijednosti.

Listing 6.1. Jednostavan for iskaz.

```
1:   /* demonstrira jednostavan for iskaz */
2:
3:   #include <stdio.h>
4:
5:   int count;
```

```
6:
7:   main()
8:   {
9:       /* Printaj brojeve 1 do 20 */
10:
11:      for (count = 1; count <= 20; count++)
12:          printf("%d\n", count);
13:
14:      return 0;
15:  }
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

ANALIZA:
Linija **2** uključuje standardni input/output (ulaz/izlaz) file zaglavlja. Linija **5** deklariše tip **int** varijable, nazvanu **count**, koja će biti korištena u **for** petlji.
Linije **11** i **12** su **for** petlja. Kada se dođe do **for** iskaza, inicijalni iskaz se izvršava prvo. U ovom primjeru, inicijalni iskaz je **count=1**. Ovo inicijalizira **count** tako da može biti korišteno od ostatka petlje.

Drugi korak u izvršenju ovog **for** iskaza je procjena uslova **count <= 20**. Zato što je **count** upravo inicijaliziran na **1**, znate da je manje od **20**, tako da se iskaz u **for** komandi, **printf()**, izvršava.
Nakon izvršenja **printf()** funkcije, inkrement izraza, **count++**, se procjenjuje. Ovo dodaje **1 count**-u, tako da je sad **2**. Sada program pravi petlju nazad i provjerava uslov ponovo.
Ako je **true**, **printf()** se ponovo izvršava, inkrement se dodaje **count**-u (čineći ga **3**), i provjerava se uslov. Ova petlja nastavlja dok se uslov ne procjeni na **false**, gdje program izlazi iz petlje i nastavlja na sljedeću liniju (liniju **14**), koja vraća **0** prije kraja programa.

For iskaz se koristi često, kao u prethodnom primjeru, da "**count up**" inkrementirajući brojač od jedne vrijednosti do druge. Takođe možete koristiti "**count down**", dekrementirajući brojačku (counter) varijablu.

```
for (count = 100; count > 0; count--)
```

Možete "računati po" (count by) vrijednost različitu od **1**, kao u ovom primjeru:

```
for (count = 0; count < 1000; count += 5)
```

For iskaz je vrlo flexibilan. Npr. možete zanemariti izraz inicijalizacije ako je test varijable inicijaliziran prije u programu. (svejedno morate koristiti tačka-zarez separator, kako je pokazano):

```
count = 1;
for ( ; count < 1000; count++)
```

Izraz inicijalizacije ne mora biti baš inicijalizacija, on može biti bilo koji validan **C** izraz. Šta god da je, izvršava se jednom kada se dođe do **for** iskaza. Npr., sljedeći primjer printa "Now sorting the array…":

```
count = 1;
for (printf("Now sorting the array...") ; count < 1000; count++)
    /* Sorting statements here */
```

Da printate brojeve od **0** do **99**, na primjer, možete napisati:

```
for (count = 0; count < 100; )
    printf("%d", count++);
```

Testni izraz koji terminira (prekida) petlju može biti bilo koji **C**-ov izraz. Sve dok procjenjuje na **true** (nenula), **for** iskaz nastavlja da se izvršava. Možete koristiti **C**-ove logičke operatore da konstruišete komplexne testne izraze.
Npr., sljedeći **for** iskaz printa elemente niza nazvanog **array[ ]**, zaustavljajući se kada se svi elementi isprintaju ili se naiđe na element sa vrijednošću **0**:

```
for (count = 0; count < 1000 && array[count] != 0; count++)
    printf("%d", array[count]);
```

Možete još pojednostaviti ovu **for** petlju:

```
for (count = 0; count < 1000 && array[count]; )
    printf("%d", array[count++]);
```

Možete staviti **null** iskaz iza **for** iskaza, dozvoljavajući da se sav posao uradi u samom **for** iskazu. Zapamtite, **null iskaz je samo tačka-zarez (;) u liniji**. Npr., da inicijalizirate sve elemente **1000**-elementnog niza na vrijednost **50**, možete napisati:

```
for (count = 0; count < 1000; array[count++] = 50)
    ;
```

U ovom for iskazu, **50** je pridruženo svakom članu niza od strane inkrementirajućeg dijela iskaza.
Možete kreirati izraz odvajajući dva podizraza (subexpressions) sa zarez operatorom. Dva podizraza se procjenjuju (s lijeva na desno), i cijeli izraz procjenjuje vrijednost desnog podizraza. Koristeći **zarez** operator, možete napraviti da svaki dio **for** iskaza obavlja višestruke dužnosti.

Zamislite da imate dva **1000**-elementna niza, **a[ ]** i **b[ ]**. Želite da kopirate sadržaj **a[ ]** u **b[ ]** u obrnutom redosljedu tako da nakon operacije kopiranja, **b[0] = a[999]**, **b[1] = a[998]**, itd.
Sljedeći **for** iskaz bi mogao obaviti posao:

```
for (i = 0, j = 999; i < 1000; i++, j--)
    b[j] = a[i];
```

Operator **zarez** se koristi da se inicijaliziraju dvije varijable, **i** i **j**. Također se koristi da se inkrementira dio od ove dvije varijable sa svakom petljom.
For iskaz prvo izvršava inicijalni iskaz. Onda provjerava uslov (condition). Ako je uslov **true**, iskaz se izvršava. Nakon što se iskaz završi, izraz inkrementacije se procjenjuje. **For** iskaz zatim ponovo provjerava uslov i nastavlja petlju sve dok uslov ne bude **false**.

Primjer 1

```
/* Printa vrijednosti x-a tako sto se broji od 0 do 9 */
int x;
for (x = 0; x <10; x++)
    printf( "\nThe value of x is %d", x );
```

Primjer 2

```
/* Uzima vrijednosti od korinika sve dok se ne unese broj 99 */
int nbr = 0;
for ( ; nbr != 99; )
   scanf( "%d", &nbr );
/* nakon unoshenja 99 prog. iskache iz petlje */
```

Primjer 3

```
/* Dozvoljava korisniku da unese do 10 integer vrijednosti      */
/* Vrijednosti su smjestene u niz nazvan value. Ako se unese 99 */
/* petlja se zaustavlja                                         */

int value[10];
```

```
int ctr,nbr=0;
for (ctr = 0; ctr < 10 && nbr != 99; ctr++)
{
    puts("Enter a number, 99 to quit ");
    scanf("%d", &nbr);
    value[ctr] = nbr;
}
```

→→ Gnijezđenje for iskaza

For iskaz može biti izvršen unutar drugog for iskaza. Ovo se zove **gnijezđenje** (nesting).
Gnijezđući **for** iskaze, možete raditi komplexno programiranje.
Listing 6.2 nije komplexan program, ali ilustruje gnijezđenje dva **for** iskaza:

Listing 6.2. Ugnijezđeni for iskazi.

```
1:   /* Demonstrira gnijezdjenje dva for iskaza */
2:
3:   #include <stdio.h>
4:
5:   void draw_box( int, int);
6:
7:   main()
8:   {
9:       draw_box( 8, 35 );
10:
11:      return 0;
12:  }
13:
14:  void draw_box( int row, int column )
15:  {
16:      int col;
17:      for ( ; row > 0; row--)
18:      {
19:          for (col = column; col > 0; col--)
20:              printf("X");
21:
22:          printf("\n");
23:      }
24:  }
```

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

ANALIZA: Program ima samo jednu komandu da printa **X**, ali je ugnijezđen u dvije petlje. U ovom primjeru, prototip funkcije za **draw_box()** je deklarisan u **liniji 5**. Ova funkcija uzima dvije tipa **int** varijable, red i kolonu, koje sadrže dimenzije kutije **X**-ova koje treba da se nacrtaju.
U liniji **9**, **main()** zove **draw_box()** i prosljeđuje vrijednost **8** kao red i vrijednost **35** kao kolonu.
Gledajući bliže **draw_box()** funkciju, možete vidjeti par stvari koje niste razumjeli. Prva je zašto je logička varijabla **col** deklarisana. Druga je zašto je drugi **printf()** u liniji **22** korišten. Razjasnićemo obadvije ove stvari nakon što pogledate dvije **for** petlje.
Linija **17** pokreće prvu **for** petlju. Inicijalizacija je preskočena zato što je inicijalna vrijednost reda predata funkciji. Gledajući uslov, vidite da je ova **for** petlja izvršena dok je red **0**.
Na prvom izvršenju linije **17**, red je **8**; tako da, program nastavlja do linije **19**.
Linija **19** sadrži drugi **for** iskaz. Ovjde se drugi prosljeđujući parametar, kolona, kopira lokalnoj varijabli, **col**, tipa **int**. Vrijednost **col**-a je **35** inicijalno (vrijednost prosljeđena od kolone (**column**), i kolona vraća svoju oreginalnu vrijednost. Zato što je **col** veći od **0**, **linija 20** se izvršava, printajući **X**.
col se onda dekrementira, i petlja nastavlja. Kada je **col** jednak **0**, **for** petlja se prekida, i kontrola se vraća do linije **22**.

Linija **22** uzrokuje on-screen printing da počne na novoj liniji (redu). Nakon što se prebacimo u novi red na ekranu, kontrola dođe do kraja prvog **for** iskaza petlje, što uzrokuje izvršenje inkrementirajućeg izraza, koji oduzima **1** od reda (row), čineći ga **7**. Ovo vraća kontrolu nazad na liniju **19**.
Primjetite da je vrijednost **col**-a bila **0** kada je zadnji put korištena. Ako bi se **column** koristila umjesto **col**-a, pala bi na testu uslova, zato što ne bi nikad bila veća od **0**. Bila bi printana samo prva linija.

ZAPAMTITE znak tačka-zarez (→ **;** ←) ako koristite **for** sa **null** iskazom. Stavite → **;** ← u posebnu liniju, ili stavite razmak (space) između → **;** ← i kraja for iskaza.
Jasnije je ako je stavite u posebni red (liniju)

```
for (count = 0; count < 1000; array[count] = 50) ;
    /* note space! */
```

→→→While iskaz

While iskaz, također nazvan i **while** petlja, izvršava blok iskaza sve dok je određeni uslov **true**.
While iskaz ima sljedeću formu:

```
while (condition)
    statement
```

condition (**uslov**) je bilo koji **C** izraz, i iskaz je jedinstven ili složen **C** iskaz. Kada izvršenje programa dođe do **while** petlje, događa se sljedeće:

1. Izraz condition (uslov) se procjenjuje.
2. Ako uslov procjeni **false** (nula), prekida se **while** iskaz, i izvršenje se prosljeđuje prvom iskazu nakon iskaza (*statement*).
3. Ako uslov procjeni **true** (tj., nenula), **C** iskaz(i) u iskazu (statement) se izvršavaju.
4. Izvršenje se vraća do koraka **1**.

Listing 6.3 je jednostavan program koji koristi **while** iskaz da printa brojeve od **1** do **20**. (Ovo je isti zadatak koji obavlja i **for** iskaz u Listingu **6.1**.)

Listing 6.3. Jednostavan while iskaz.

```
1:   /* Demonstrira jednostavan while iskaz */
2:
3:   #include <stdio.h>
4:
5:   int count;
6:
7:   int main()
8:   {
9:       /* Print the numbers 1 through 20 */
10:
11:      count = 1;
12:
13:      while (count <= 20)
14:      {
15:          printf("%d\n", count);
16:          count++;
17:      }
18:      return 0;
19:  }
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

```
14
15
16
17
18
19
20
```

ANALIZA: U liniji **11**, **count** se inicijalizira na **1**.
Zato što **while** iskaz ne sadrži inicijalizacijski sektor, vi morate voditi računa o inicijalizaciji bilo kojih (svih) varijabli prije početka **while**-a. **Linija 13** je stvarni **while** iskaz, i sadrži iste iskaze uslova (condition statements) kao i Listing 6.1, **count<=20**. U **while** petlji, **linija 16** se brine o inkrementiranju **count**-a. Šta mislite šta bi se desilo da ste zaboravili staviti **liniju 16** u ovaj program? Vaš program ne bi znao kad da stane, zato što bi **count** uvijek bio **1**, što je uvijek manje od **20**.
Možda ste primjetili da je **while** iskaz u biti **for** iskaz bez komponenti inkrementiranja i dekrementiranja. Tako da je:

```
for ( ; condition ; )
```

ekvivalentno sa:

```
while (condition)
```

Zbog ove ekvivalencije, sve što može biti urađeno sa **for** iskazom, takođe može biti urađeno sa **while** iskazom. Kad koristite **while** iskaz, svaka potrebna inicijalizacija mora prvo biti urađena u odvojenom iskazu, i updating mora biri urađen od strane iskaza koji je dio **while** petlje.

Primjer 1

```
int x = 0;
while (x < 10)
{
    printf("\nThe value of x is %d", x );
    x++;
}
```

Primjer 2

```
/* uzmi broj, dok ne dobijes jedan veci od 99 */
int nbr=0;
while (nbr <= 99)
    scanf("%d", &nbr );
```

Primjer 3

```
/* Dopusta korisniku da unese do 10 integer vrijednosti*/
/* Vrijednosti su smjeshtene u niizu nazvan value.     */
/* Ako se unese 99, petlja se zaustavlja               */
int value[10];
int ctr = 0;
int nbr;
while (ctr < 10 && nbr != 99)
{
    puts("Enter a number, 99 to quit ");
    scanf("%d", &nbr);
    value[ctr] = nbr;
    ctr++;
}
```

NE koristite sljedeću konvenciju ako nije neophodno:

```
while (x)
```

Umjesto, koristite ovu konvenciju:

```
while (x != 0)
```

Iako obadvije rade, druga je jasnija kada debagirate (pokušavate da nađete problem u) kood. Kada se kompajlira, oni produciraju bukvalno isti kood.
KORISTITE for iskaze umjesto **while** iskaza ako morate inicijalizirati i inkrementirati unutar vaše petlje. **For** iskaz drži inicijalizaciju, uslov i inkrement iskaze sve zajedno. **While** iskaz ne.

→→→ do...while petlja

C-ova treća petlja je **do…while** petlja, koja izvršava blokove iskaza sve dok je određeni uslov **true**.
do…while petlja testira uslov na kraju petlje, a ne na početku kao **for** i **while** petlje.
Struktura **do…while** petlje izgleda ovako:

```
do
    statement
while (condition);
```

condition (uslov) je **C** izraz, i ***statement*** (iskaz) je jedinstven ili složen **C** iskaz. Kada izvršenje programa dođe do **do…while** iskaza, dešava se sljedeće:

1. Izvršavaju se iskazi u ***iskazu*** (za do…while petlju).
2. ***condition*** (uslov) se procjenjuje. Ako je **true**, izvršenje se vraća na korak **1**. Ako je **false**, petlja se terminira (prekida).

Iskazi koji su dodijeljeni **do…while** petlji se uvijek izvršavaju bar jedanput. Ovo je zato što se testni uslov procjenjuje na kraju, a ne na početku petlje.

Do…while petlje se rjeđe koriste nego **for** i **while** petlje.

Listing **6.5** prikazuje primjer **do…while** petlje:

Listing 6.5. Jednostavna do...while petlja.

```
1:   /* Demonstrira jednostavan do...while iskaz */
2:
3:   #include <stdio.h>
4:
5:   int get_menu_choice( void );
6:
7:   main()
8:   {
9:       int choice;
10:
11:      choice = get_menu_choice();
12:
13:      printf("You chose Menu Option %d\n", choice );
14:
15:      return 0;
16:  }
17:
18:  int get_menu_choice( void )
19:  {
20:      int selection = 0;
21:
22:      do
23:      {
24:          printf("\n" );
25:          printf("\n1 - Add a Record" );
26:          printf("\n2 - Change a record");
27:          printf("\n3 - Delete a record");
28:          printf("\n4 - Quit");
29:          printf("\n" );
30:          printf("\nEnter a selection: " );
31:
32:          scanf("%d", &selection );
33:
34:       }while ( selection < 1 || selection > 4 );
35:
36:       return selection;
37:  }
       1 - Add a Record
       2 - Change a record
       3 - Delete a record
       4 - Quit
```

```
Enter a selection: 8
1 - Add a Record
2 - Change a record
3 - Delete a record
4 - Quit
Enter a selection: 4
You chose Menu Option 4
```

ANALIZA: Ovaj program obezbjeđuje **meni** sa četiri izbora. Korisnik bira jedan od četiri izbora, i onda program **printa** izabrani broj. **Main()** funkcija -> od **linije 7** do **linije 16**.

NAPOMENA: Tijelo **main()**-a može biti napisano u jednoj liniji, ovako:

```
printf( "You chose Menu Option %d", get_menu_option() );
```

Ako želite da proširite ovaj program, i djelujete po selekciji, trebate povratnu (**return**) vrijednost od strane **get_manu_choice()**-a, tako da je pametno pridružiti vrijednost varijabli (kao što je **choice**).

Linije od **18** do **37** sadrže **get_menu_choice()**. Ova funkcija prikazuje **meni** na-ekranu (on-screen) (**linije 24 do 30**) i onda slijedi izbor. Zato što morate prikazati **meni** barem jedanput da dobijete odgovor (answer), prigodno je koristiti **do…while** petlju. U ovom slučaju, **meni** je prikazan dok je unesen validan izbor. **Linija 34** sadrži **while** dio **do…while** iskaza i procjenjuje (validates) vrijednost izbora, prigodno nazvanim izborom. Ako unesena vrijednost nije između **1** i **4**, **meni** se prikazuje ponovo, i pita se korisnika na novu vrijednost. Kada se pravilan izbor unese, program nastavlja do **linije 36**, koja vraća vrijednost u odjel varijable.

```
do
{
    statement(s)
}while (condition);
```

Primjer 1

```
/* printa čak i ako uslov "padne"! */
int x = 10;
do
{
    printf("\nThe value of x is %d", x );
}while (x != 10);
```

Primjer 2

```
/* uzima brojeve dok broj nije veci od  99 */
int nbr;
do
{
    scanf("%d", &nbr );
}while (nbr <= 99);
```

Primjer 3

```
/* Omogucava korisniku da unese do 10 integer vrijednosti    */
/* Vrijednosti su smještene u niizu nazvan value.            */
/* Ako se unese 99, petlja se zaustavlja                     */
int value[10];
int ctr = 0;
int nbr;
do
{
    puts("Enter a number, 99 to quit ");
    scanf( "%d", &nbr);
    value[ctr] = nbr;
    ctr++;
}while (ctr < 10 && nbr != 99);
```

→→→ Ugnijezđene petlje

C ne postavlja uslov o broju gnijezđenja u petlji, osim što se unutrašnja petlja mora u potpunosti sadržavati u vanjskoj petlji. Ne mogu biti preklapanja. Tako da sljedeće **nije** dozvoljeno:

```
for ( count = 1; count < 100; count++)
{
    do
    {
        /* the do...while loop */
} /* kraj for petlje */
    }while (x != 0);
```

Ako se **do...while** petlja smjesti u potpunosti u **for** petlju, onda nema problema:

```
for (count = 1; count < 100; count++)
{
    do
    {
        /* the do...while loop */
    }while (x != 0);
} /* kraj for petlje */
```

Kada koristite ugnijezđene petlje, zapamtite da promjene napravljene u unutrašnjoj petlji mogu uticati na vanjsku petlju. Unutrašnja petlja mora biti nezavisna od bilo kojih varijabli u vanjskoj petlji; u ovom primjer, nisu.
U prethodnom primjeru, unutrašnja **do…while** petlja modifikuje (mijenja) vrijednost **count**-a, onoliki broj puta koliko se vanjska petlja izvrši.
Dobar stil čini kood sa ugnijezđenim petljama lakšim za čitanje. Svaki nivo petlje treba biti jedan korak dalje od zadnjeg nivoa. Ovo jasno obilježava kood koji je pridružen svakoj petlji.

NE pokušavajte preklapati petlje. Možete ih gnijezditi, ali one moraju biti u potpunosti jedni unutar drugih.
KORISTITE do…while petlju kada znate da će se petlja izršiti bar jednom.

→→ Sažetak (lekcija 6):

Iako se sve tri petlje koriste da se obavi isti zadatak, svaka je različita.
for iskaz vam dopušta da inicijalizirate, procjenite i inkrementirate sve u jednoj komandi.
while iskat radi sve dok je uslov true.
do…while petlja uvijek izvršava svoje iskaze bar jednom i nastavlja njihovo izvršenje sve dok je uslov false.

For iskaz je najbolji kada znate šta treba da inicijalizirate i inkrementirate u vašoj petlji. Ako samo imate uslov koji želite, a taj uslov nije određen broj petlji, **while** je dobar izbor. Ako znate da će se set iskaza izvršiti bar jednom, **do…while** može biti najbolji.

LEKCIJA 7: → Osnove Input-a i Output-a ←

→ Prikazivanje informacija na ekranu (on-screen)
Dva najčešća načina da prikažete informacije na ekranu, od strane vašeg programa, je sa C-ovim bibliotečnim funkcijama: **printf() i puts()**.

→→→ Printf() funkcija
Printajući tekt poruku na ekranu je jednostavno. Zovi **printf()** funkciju, prosljeđujući željenu poruku zatvorenu u znacima navoda.
Npr., da prikažete **Greska se pojavila!** na ekranu, napisali biste:

```
printf("Greska se pojavila!");
```

S dodatkom sa tektualnim porukama, često morate prikazati vrijednost programskih varijabli. Ovo je malo komplikovanije nego prikazivanje samo poruke. Npr., pretpostavimo da želite da prikažete vrijednost numeričke varijable **x** na ekranu, zajedno sa nešto indentifikacionog teksta. Čak i dalje, želite da informacija počinje na početku novog reda. Možete koristiti **printf()** funkciju kako slijedi:

```
printf("\nThe value of x is %d", x);
```

Rezultat na ekranu je, pod pretpostavkom da je vrijednost **x**-a **12**:

```
The value of x is 12
```

U ovom primjeru, dva su argumenta prosljeđena **printf()**-u. Prvi argument zatvoren pod znacima navoda i zove se format string. Drugi argument je ime varijable (**x**) koja sadrži vrijednost koja se printa.

→→ printf() Format Strings

printf() format string specificira kako se izlaz formatira. Evo tri moguće komponente format string-a:

- ***Literal text*** se prikazuje tačno onako kako je unesen u format string-u.
- ***escape sequence*** obezbjeđuje specijalnu kontrolu formatiranja.
- ***conversion specifier*** se sastoji od znaka **(%)** prije jednog karaktera.

→ Najčešće korišteni escape sequences (izlazne sekvence):

Sequence	Značenje
\a	Zvono (alert)
\b	Backspace
\n	Nova linija
\t	Horizontalni tab
****	Backslash
\?	Question mark
\'	Jedan navodnik

→ printf() Escape Sequences

Sequence	Značenje:
n	The character n
\n	Newline
\"	The double quotation character
"	The start or end of a string

Listing 7.1. Korištenje printf() escape sequences (izlazne sekvence).

```
1:  /* Demonstracija cesto koristenih escape sequences-a */
2:
3:  #include <stdio.h>
4:
5:  #define QUIT  3
6:
7:  int  get_menu_choice( void );
8:  void print_report( void );
9:
10: main()
11: {
12:     int choice = 0;
13:
14:     while (choice != QUIT)
15:     {
16:        choice = get_menu_choice();
17:
18:        if (choice == 1)
19:            printf("\nBeeping the computer\a\a\a" );
20:        else
21:        {
22:            if (choice == 2)
23:                print_report();
24:        }
25:     }
26:     printf("You chose to quit!\n");
27:
28:     return 0;
29: }
30:
31: int get_menu_choice( void )
32: {
33:     int selection = 0;
34:
35:     do
36:     {
37:         printf( "\n" );
38:         printf( "\n1 - Beep Computer" );
39:         printf( "\n2 - Display Report");
40:         printf( "\n3 - Quit");
41:         printf( "\n" );
42:         printf( "\nEnter a selection:" );
43:
44:         scanf( "%d", &selection );
45:
46:      }while ( selection < 1 || selection > 3 );
47:
48:      return selection;
49: }
50:
51: void print_report( void )
52: {
53:      printf( "\nSAMPLE REPORT" );
54:      printf( "\n\nSequence\tMeaning" );
55:      printf( "\n=========\t=======" );
56:      printf( "\n\\a\t\tbell (alert)" );
57:      printf( "\n\\b\t\tbackspace" );
58:      printf( "\n...\t\t...");
59: }
```

```
 1 - Beep Computer
 2 - Display Report
 3 - Quit
 Enter a selection:1
```

```
Beeping the computer
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:2
SAMPLE REPORT
Sequence        Meaning
=========       =======
\a              bell (alert)
\b              backspace
...             ...
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:3
You chose to quit!
```

ANALIZA: Linije **7** i **8** su prototipi funkcije.
Ovaj program ima dvije funkcije: **get_menu_choice()** i **print_report()**.
get_menu_choice() je definisan u liniji **31** do **49**. Ovo je slično sa **meni** funkcijom u Listing-u **6.5**. Linije **37** i **41** sadrže pozive **printf()** –a koje printaju novu liniju escape sequence-a. Linije **38**, **39**, **40** i **42** takođe koriste escape karakter nove linije, i oni printaju tekst. Linija **37** može biti eliminisana, mijenjajući liniju **38** ovako:

```
printf( "\n\n1 - Beep Computer" );
```

Ipak, ostavljajući liniju **37** čini program lakšim za čitanje.
Linija **16** uzima varijablin izbor, koji se zatim analizira u linijama od **18** do **25** u **if** iskazu. Ako korisnik izabere **1**, linija **19** printa karakter nove linije, poruku, a zatim tri zvuka (beeps). Ako korisnik izabere **2**, linija **23** zove funkciju **print_report()**.
print_report() funkcija je definisana na linijama od **51** do **59**. Jednostavna funkcija pokazuje lakoću korištenja **printf()**-a i escape sequence-a za printanje formatiranih informacija na ekran.

→→ printf() konverzioni specifikatori

Table 7.2. Najčešće potrebni konverzioni specifikatori.

Specifikator	Značenje	Tip Konverzije
%c	Single character	char
%d	Signed decimal integer	int, short
%ld	Signed long decimal integer	long
%f	Decimal floating-point number	float, double
%s	Character string	char arrays
%u	Unsigned decimal integer	unsigned int, unsigned short
%lu	Unsigned long decimal integer	unsigned long

Šta u vezi printanja vrijednosti više od jedne varijable?
Jedan **printf()** iskaz može printati neograničen broj varijabli, ali format string mora sadržavati jedan specifikator konverzije (conversion specifier) za svaku varijablu. Specifikatori konverzije se uparuju sa varijablama s lijeva na desno. Ako napišete:

```
printf("Rate = %f, amount = %d", rate, amount);
```

varijabla **rate** se uparuje sa **%f** specifikatorom, i varijabla **amount** se uparuje sa **%d** specifikatorom. Pozicija konverzionih specifikatora u format stringu odlučuje položaj izlaza (output-a). Ako se prosljeđuje više varijabli **printf()**-u nego što je konverzionih specifikatora, napodudarajuće varijable se ne printaju. Ako postoji više specifikatora nego varijabli, nepodudarajući specifikatori printaju smeće (garbage).
Niste ograničeni na printanje vrijednosti varijabli sa **printf()**-om. Argumenti mogu biti bilo koji validni **C** izrazi.

Npr., da printate sumu **x** i **y**-a, možete napisati:

```
z = x + y;
printf("%d", z);
```

Također ste mogli napisati:

```
printf("%d", x + y);
```

Svaki program koji koristi **printf()** treba uključivati file zaglavlja **STDIO.H**.

Listing 7.2. Upotreba printf() za prikaz numeričkih vrijednosti.

```
1:   /*Demonstracija upotrebe printf()-a za prikaz numerickih vrijednosti*/
2:
3:   #include <stdio.h>
4:
5:   int a = 2, b = 10, c = 50;
6:   float f = 1.05, g = 25.5, h = -0.1;
7:
8:   main()
9:   {
10:      printf("\nDecimal values without tabs: %d %d %d", a, b, c);
11:      printf("\nDecimal values with tabs: \t%d \t%d \t%d", a, b, c);
12:
13:      printf("\nThree floats on 1 line: \t%f\t%f\t%f", f, g, h);
14:      printf("\nThree floats on 3 lines: \n\t%f\n\t%f\n\t%f", f, g, h);
15:
16:      printf("\nThe rate is %f%%", f);
17:      printf("\nThe result of %f/%f = %f\n", g, f, g / f);
18:
19:      return 0;
20:  }
```

```
Decimal values without tabs: 2 10 50
Decimal values with tabs:       2       10      50
Three floats on 1 line:         1.050000        25.500000       -0.100000
Three floats on 3 lines:
        1.050000
        25.500000
        -0.100000
The rate is 1.050000%
The result of 25.500000/1.050000 = 24.285715
```

ANALIZA: ovaj primjer printa šest linija informacija.
Linija **17**: kada printate vrijednosti konverzionih specifikatora, ne morate koristiti varijable.
Možete koristiti izraze kao npr., **g/f**, ili čak konstante.

NE stavljajte višestruke linije teksta u **printf()** iskaz. Lakše se istakne sa višestrukim **printf()** iskazima, nego da koristite samo jednu sa nekoliko **\n**.
NE zaboravite da koristite **n** kada printate više linija u odvojenim printf() iskazima.
NE pišite pogrešno **stdio.h**. Većina **C** programera greškom napiše **studio.h**.

→→ **printf()** Funkcija

```
#include <stdio.h>
printf( format-string[,arguments,...]);
```

printf() je funkcija koja prihvata niz argumenata, svaki primjenjujući specifikator konverzije u datom format stringu.
Format string je potreban; ipak, argumenti su opcionalni. Za svaki argument, mora postojati apecifikator konverzije.

Slijede primjeri poziva **printf()**-a i njihovog izlaza (output):

Primjer 1 Input

```
#include <stdio.h>
main()
{
    printf("This is an example of something printed!");
    return 0;
}
```

Primjer 1 Output

```
This is an example of something printed!
```

Primjer 2 Input

```
printf("This prints a character, %c\na number, %d\na floating \
point, %f", `z', 123, 456.789 );
```

Primjer 2 Output

```
This prints a character, z
a number, 123
a floating point, 456.789
```

→→→ Prikazivanje poruka sa puts()

puts() funkcija takođe može biti korištena da se prikaže tekst na ekranu, ali ne može sadržavati numeričke varijable. **puts()** uzima jedan string i njegove argumente i prikazuje ih, automatski dodajući novi red na kraju. Npr., iskaz:

```
puts("Hello, world.");
```

vrši istu funkciju kao i:

```
printf("Hello, world.\n");
```

U **puts()** možete uključiti i **\n**. Svaki program koji koristi **puts()** treba sadržavati i **STDIO.H**.

KORISTITE puts() funkciju umjesto **printf()** funkcije kad god želite da printate tekst, a ne treba da printate nikakve varijable.
NE pokušavajte da koristite specifikatore konverzije sa **puts()** iskazima.

→→ puts() Funkcija

```
#include <stdio.h>
puts( string );
```

puts() je funkcija koja kopira string na standardni izlaz, obično ekran.

Slijede primjeri poziva **puts()** i njihov izlaz:
Primjer 1 Input

```
puts("This is printed with the puts() function!");
```

Primjer 1 Output

```
This is printed with the puts() function!
```

Primjer 2 Input

```
puts("This prints on the first line. \nThis prints on the second line.");
puts("This prints on the third line.");
puts("If these were printf()s, all four lines would be on two lines!");
```

Primjer 2 Output

```
This prints on the first line.
This prints on the second line.
This prints on the third line.
If these were printf()s, all four lines would be on two lines!
```

→→→ Unošenje numeričkih vrijednosti sa scanf()

Najfleksibilniji način da vaš program može čitati numeričke podatke sa vaše tastature, je korištenje **scanf()** bibliotečne funkcije.

scanf() funkcija čita podatke sa tastature prema specificiranim formatima i pridružuje unešene vrijednosti jednoj ili više programskih varijabli. Kao i **printf()**, **scanf()** koristi format string da opiše format unosa (input-a). Format string upotrebljava isti specifikator konverzije kao i **printf()** funkcija.

Npr., iskaz:

```
scanf("%d", &x);
```

čita decimalni integer sa tastature i pridružuje ga integer varijabli **x**. Tako i sljedeći iskaz čita floating-point vrijednost sa tastature i pridružuje ga varijabli **rate**:

```
scanf("%f", &rate);
```

& je simbol **"dresa-od**"u **C**-u. Zasad zapamtite da **scanf()** treba **&** simbol prije imena svake numeričke varijable u svojoj listi argumenata (osim ako je varijabla pointer, što je također objašnjeno 9. dana).

Moguće je i

```
scanf("%d %f", &x, &rate);
```

Kada se unesu višestruke varijable, **scanf()** koristi **bijela polja** (white spaces (razmak)) da odvoji unos u polju. Bijela polja mogu biti **razmak, tab-ovi ili nove linije**. Svaki specifikator konverzije u **scanf()** format stringu se podudara sa poljem unosa; kraj svakog unosa je indentifikovan sa bijelim poljem.
Ovo vam daje značajnu fleksibilnost. S obzirom na prethodno, možete napisati:

```
10 12.45
```

Također ste mogli napisati i ovo:

```
10                  12.45
```

ili ovo:

```
10
12.45
```

Sve dok je bijelih polja između varijabli, **scanf()** može pridružiti svaku vrijednost svojoj varijabli.
Scanf() -> STDIO.H.

Listing 7.3. Upotreba scanf() za sticanje numeričkih vrijednosti.

```
1:   /* Demonstracija upotrebe scanf()-a */
2:
3:   #include <stdio.h>
4:
5:   #define QUIT 4
6:
7:   int get_menu_choice( void );
8:
9:   main()
10:  {
11:      int   choice    = 0;
12:      int   int_var   = 0;
13:      float float_var = 0.0;
14:      unsigned unsigned_var = 0;
15:
16:      while (choice != QUIT)
17:      {
18:          choice = get_menu_choice();
19:
20:          if (choice == 1)
21:          {
22:              puts("\nEnter a signed decimal integer (i.e. -123)");
23:              scanf("%d", &int_var);
24:          }
25:          if (choice == 2)
26:          {
27:              puts("\nEnter a decimal floating-point number\
28                    (i.e. 1.23)");
29:              scanf("%f", &float_var);
30:          }
31:          if (choice == 3)
32:          {
33:               puts("\nEnter an unsigned decimal integer \
34                   (i.e. 123)" );
35:               scanf( "%u", &unsigned_var );
36:          }
37:      }
38:    printf("\nYour values are: int: %d  float: %f  unsigned: %u \n",
39:                                  int_var, float_var, unsigned_var );
40:
41:     return 0;
42:  }
43:
44:  int get_menu_choice( void )
45:  {
46:      int selection = 0;
47:
48:      do
49:      {
50:          puts( "\n1 - Get a signed decimal integer" );
51:          puts( "2 - Get a decimal floating-point number" );
52:          puts( "3 - Get an unsigned decimal integer" );
53:          puts( "4 - Quit" );
54:          puts( "\nEnter a selection:" );
55:
56:          scanf( "%d", &selection );
57:
58:       }while ( selection < 1 || selection > 4 );
59:
60:       return selection;
61:  }
   1 - Get a signed decimal integer
```

```
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
1
Enter a signed decimal integer (i.e. -123)
-123
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
3
Enter an unsigned decimal integer (i.e. 123)
321
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
2
Enter a decimal floating point number (i.e. 1.23)
1231.123
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
4
Your values are: int: -123  float: 1231.123047 unsigned: 321
```

ANALIZA: First, **puts()** is used instead of **printf()**. Because no variables are printed, there is no need to use **printf()**. Because **puts()** is being used, the newline escape characters have been removed from lines **51** through **53**. Line **58** was also changed to allow values from **1** to **4** because there are now four menu options. Notice that line **56** has not changed (u odnosu na Listing **7.1**); however, now it should make a little more sense.
scanf() gets a decimal value and places it in the variable selection. The function returns selection to the calling program in line **60**.
When the user selects **Quit**, the program prints the last-entered number for all three types. If the user didn't enter a value, **0** is printed, because lines **12**, **13**, and **14** initialized all three types. One final note on lines **20** through **36**: The **if** statements used here are not structured well. If you're thinking that an **if...else** structure would have been better, you're correct. Day 14, "Working with the Screen, Printer, and Keyboard," introduces a new control statement, **switch**. This statement offers an even better option.

NE zaboravite da uključite operator **adresa-od** (**&**) kada koristite **scanf()** varijable.
KORISTITE printf() ili **puts()** u konjukciji sa **scanf()**-oom. Koristite printing funkcije za prikaz poruke za podatke da **scanf()** ima (dohvati (get)).

→→ **scanf()** Funkcija

```
#include <stdio.h>
scanf( format-string[,arguments,...]);
```

Argumenti trebaju biti adrese varijabli prije nego stvarne varijable.

scanf() čita unos-na polja sa standardnog ulaza, obično tastature. Stavlja svaki od ovih unesenih polja u argument. Kada stavi informaciju, konvertuje je u format specifikatora koji korespondira (odgovara) sa format stringom. Za svaki argument, mora postojati specifikator konverzije.

Primjer 1

```
int x, y, z;
scanf( "%d %d %d", &x, &y, &z);
```

Primjer 2

```
#include <stdio.h>
main()
{
    float y;
    int x;
    puts( "Enter a float, then an int" );
    scanf( "%f %d", &y, &x);
    printf( "\nYou entered %f and %d ", y, x );
    return 0;
}
```

Lekcija 8: → Upotreba numeričkih redova ←→ Using Numeric Arrays ←

```
float expenses[12];
```

Niz je nazvan **expenses**, i sadrži **12** elemenata. Svaki od **12** elemenata je tačan ekvivalent jedne **float** varijable. Svaki od **C**-ovih tipova podataka može biti korišten za nizove. **C**-ov niz elemenata je uvijek numerisan početkom sa **0**, tako da su **12** elemenata numerisani od **0** do **11**.
Lokacija deklaracije niza u vašem izvornom koodu je važna. Zasad, stavite deklaraciju niza sa ostalim deklaracijama varijabli, upravo prije početka **main()**-a.
Niz elemenata može biti korišten u vašem programu bilo gdje i se može koristiti i neniizna varijabla istog tipa. Samim elementima u vašem nizu se pristupa koristeći ime niza i elementom indexa (subscript) u uglastim zagradama.
Npr., sljedeći iskaz storira vrijednost **89.95** u drugi element niza (zapamtite, prvi element niza je **expenses[0]**, ne **expenses[1])**:

```
expenses[1] = 89.95;
```

Isto tako, iskaz:

```
expenses[10] = expenses[11];
```

pridružuje kopiju vrijednosti storiranu (sadržanu) u elementu niza **expenses[11]** u element niza **expenses[10]**. Kada se obratite na element niza, index niza može biti literalna (brojna) konstanta, kao u ovom primjeru. Ipak, vaš program može često koristiti index koji je **C** integer varijabla ili izraz, ili čak drugi element niza. Evo par primjera:

```
float expenses[100];
int a[10];
/* additional statements go here */
expenses[i] = 100;          /* i is an integer variable */
expenses[2 + 3] = 100;      /* equivalent to expenses[5] */
expenses[a[2]] = 100;       /* a[] is an integer array */
```

Zadnji primjer možda zahtjeva objašnjenje. Ako, npr., imate integer niz nazvan **a[ ]** i vrijednost **8** je smještena u element **a[2]**, pišući:

```
expenses[a[2]]
```

ima isti efekat kao i da ste napisali:

```
expenses[8];
```

Ako je niz od **n** elemenata, dozvoljeni opseg indexa je od **0** do **n-1**.
Ako upotrebite vrijednost indexa **n**, možete imati programske greške.
Najlakši način da ovo uradite (bez opterećenja sa početkom niza od elementa **0** (zabune)) je da deklarišete niz sa jednim elementom više nego što je potrebno, i onda ignorišete element **0**.

Listing 8.1. EXPENSES.C demonstrira upotrebu niza.

```
1:   /* EXPENSES.C - Demonstrira upotrebu niza */
2:
3:   #include <stdio.h>
4:
5:   /* Deklariše niz da drži expenses, i brojačku varijablu */
6:
7:   float expenses[13];
8:   int count;
9:
10:  main()
11:  {
12:      /* Unesi podatke sa tastature u niz */
13:
14:      for (count = 1; count < 13; count++)
```

```
15:      {
16:          printf("Enter expenses for month %d: ", count);
17:          scanf("%f", &expenses[count]);
18:      }
19:
20:      /* Print array contents */
21:
22:      for (count = 1; count < 13; count++)
23:      {
24:          printf("Month %d = $%.2f\n", count, expenses[count]);
25:      }
26:      return 0;
27:  }
```

```
Enter expenses for month 1: 100
Enter expenses for month 2: 200.12
Enter expenses for month 3: 150.50
Enter expenses for month 4: 300
Enter expenses for month 5: 100.50
Enter expenses for month 6: 34.25
Enter expenses for month 7: 45.75
Enter expenses for month 8: 195.00
Enter expenses for month 9: 123.45
Enter expenses for month 10: 111.11
Enter expenses for month 11: 222.20
Enter expenses for month 12: 120.00
Month 1 = $100.00
Month 2 = $200.12
Month 3 = $150.50
Month 4 = $300.00
Month 5 = $100.50
Month 6 = $34.25
Month 7 = $45.75
Month 8 = $195.00
Month 9 = $123.45
Month 10 = $111.11
Month 11 = $222.20
Month 12 = $120.00
```

for petlja u linijama od **14** do **18** ignoriše element **0**. Ovo dozvoljava programu da koristi elemente od **1** do **12**, tako da se svaki direktno odnosi na 12 mjeseci. Vraćajući se na liniju **8**, varijabla, **count**, je deklarisana i korištena kroz program kao brojač i index niza.
U liniji **17**, **scanf()** funkcija koristi elemente niza.
Zasad, znajte da **%.2f** printa floating brojeve sa dvije decimalne cifre s desne strane.

→→→ Multidimenzionalni nizovi

Multidemenzionalni niz ima više od jednog indexa (subscript).
Npr., možete napisati program koji igra "**dame**". Tabla za "damu" sadrži 64 kvadrata poredanih u **8** redova i **8** kolona. Vaš program može predstaviti tablu kao dvo-dimenzionalni niz, kako slijedi:

```
int checker[8][8];
```

Rezultirajući niz ima 64 elementa:

```
checker[0][0], checker[0][1], checker[0][2]...checker[7][6], checker[7][7].
```

→→→ Imenovanje i deklaracija nizova

Pravila za pridruživanje imena nizovima su ista kao i za imena varijabli (Dan 3 "Storing Data: Variables and Constants."). Ime niza mora biti jedinstveno. Ne može biti korišteno za neki drugi niz ili neki drugi identifikator (varijablu, konstantu, itd.).

Kada deklarišete niz, morate specificirati broj elemenata sa literalnom konstantom (kao u ranijim primjerima) ili sa simboličkom konstantom kreiranom sa **#define** direktivom. Zato, sljedeće:

```
#define MONTHS 12
int array[MONTHS];
```

je ekvivalentno sa ovim iskazom:

```
int array[12];
```

Ipak, kod većine kompajlera, ne možete deklarisati elemente nizova sa simboličkom konstantom kreiranom sa **const** ključnom riječi:

```
const int MONTHS = 12;
int array[MONTHS];          /* Pogresno! */
```

Listing 8.2. GRADES.C storira 10 ocjena u niz.

```
1:   /*GRADES.C - Program (sample) sa nizovim */
2:   /* Uzima 10 ocjena i racuna srednju vrijednost */
3:
4:   #include <stdio.h>
5:
6:   #define MAX_GRADE 100
7:   #define STUDENTS  10
8:
9:   int grades[STUDENTS];
10:
11:  int idx;
12:  int total = 0;           /* koristi se za prosjek */
13:
14:  main()
15:  {
16:      for( idx=0;idx< STUDENTS;idx++)
17:      {
18:          printf( "Enter Person %d's grade: ", idx +1);
19:          scanf( "%d", &grades[idx] );
20:
21:          while ( grades[idx] > MAX_GRADE )
22:          {
23:              printf( "\nThe highest grade possible is %d",
24                      MAX_GRADE );
25:              printf( "\nEnter correct grade: " );
26:              scanf( "%d", &grades[idx] );
27:          }
28:
29:          total += grades[idx];
30:      }
31:
32:      printf( "\n\nThe average score is %d\n", ( total / STUDENTS) );
33:
34:      return (0);
35:  }
```

```
Enter Person 1's grade: 95
Enter Person 2's grade: 100
Enter Person 3's grade: 60
Enter Person 4's grade: 105
The highest grade possible is 100
```

```
Enter correct grade: 100
Enter Person 5's grade: 25
Enter Person 6's grade: 0
Enter Person 7's grade: 85
Enter Person 8's grade: 85
Enter Person 9's grade: 95
Enter Person 10's grade: 85
The average score is 73
```

Na liniji **9**, niz za ovaj program je nazvam **grades**. Ostale dvije varijable su deklarisane, **idx** i **total**. Opaska za index, **idx** se koristi kao brojač i index niza (subscript). Ukupan zbir svih ocjena se čuva u **total**.

Srce ovog programa je **for** petlja u linijama od **16** do **30**. **for** iskaz inicijalizira **idx** na **0**, prvi index niza. Onda se peetlja (loops) sve dok je **idx** manji od broja **students**. Svaki put kada se peetlja, inkrementira se **idx** za **1**. Za svaku petlju, program prikazuje (prompts) za ocjenu studenta (linije **18** i **19**). Primjetite da u liniji **18**, **1** se dodaje **idx**-u s ciljem da broji ljude od **1** do **10**, umjesto od **0** do **9**. Zato što niz počinje sa indexom **0**, prva ocjena je stavi ocjenu u **grade[0]**. Umjesto konfuzije (zbunjoze) korisnika s upitom za ocjenu **0** studenta (Person 0's grade), postavlja se upit za ocjenu **1** studenta (Person 1's grade).
Linije od **21** do **27** sadrže **while** petlju ugnijezđenu u **for** petlji. Ovo je editujuća provjera koja osigurava da ocjena (grade) nije veća nego maximalna ocjena, **MAX_GRADE**. Korisnici se upituju (prompted) da unesu tačnu ocjenu ako unesu ocjenu koja je prevelika. Trebate provjeriti programske podatke kad god možete.
Linija **29** dodaje unesenu ocjenu brojaču **total**. U liniji **32**, ovaj **total** se koristi da se printa prosječna ocjena (**total/STUDENTS**).

KORISTITE #define iskaz da kreirate konstante koje će biti korištene pri deklaraciji nizova. Tada možete jednostavno mijenjati broj elemenata u nizu. U GRADES.C, npr., možete promjeniti broj studenata u **#define**-u i ne biste morali praviti nikakve dodatne izmjene u programu.
IZBJEGAVAJTE multidimenzionalne nizove sa više od tri dimenzije. Zapamtite, multidimenzionalni nizovi mogu postati vrlo veliki vrlo brzo.

→→ Inicijalizacija nizova

Možete inicijalizirati sav ili dio niza kada ga prvi put deklarišete. Slijedite deklaraciju niza sa znakom jednako i listom vrijednosti zatvorenih u zagradama i razdvojenih zarezima. Lista vrijednosti se pridružuju s ciljem da elementi niza počinju od **0**.

Npr., sljedeći kood pridružuje vrijednost 100 (to)
array[0], **200** (to) **array[1]**, **300** (to) **array[2]**, i **400** (to) **array[3]**:

```
int array[4] = { 100, 200, 300, 400 };
```

Ako izostavite veličinu niza, kompajler kreira tačno onoliko prostora koliko je potrebno da drži inicijalizaciju vrijednosti. Tako da, sljedeći iskaz ima tačno isti efekat kao i prethodni iskaz deklaracije niza:

```
int array[] = { 100, 200, 300, 400 };
```

Možete, također, uključiti premalo inicijalizacijskih vrijednosti, kao npr.:

```
int array[10] = { 1, 2, 3 };
```

Ako eksplicitno ne inicijalizirate element niza, ne možete biti sigurni koju vrijednost sadrži kada se program starta. Ako uključite previše inicijalizatora (više inicijalizatora nego elemenata niza), kompajler može detektovati grešku.

→→ Inicijalizacija multidimenzionalnih nizova

Multidimenzionalni niz također može biti inicijaliziran. Lista inicijalizacionih vrijednosti se pridružuje elementima niza po redosljedu, tako da se zadnji index niza mijenja prvi. Npr.:

```
int array[4][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

rezultira sa sljedećim pridruženjima (assignments):

```
array[0][0] je jednako sa 1
array[0][1] je jednako sa 2
array[0][2] je jednako sa 3
array[1][0] je jednako sa 4
array[1][1] je jednako sa 5
array[1][2] je jednako sa 6
...
array[3][1] je jednako sa 11
array[3][2] je jednako sa 12
```

Kada inicijalizirate multidimenzionalni niz, možete učiniti vaš izvorni kood jasnijim koristeći extra zagrade da grupišete inicijalizacijske vrijednosti i još raspoređujući ih u nekoliko linija. Sljedeća inicijalizacija je ekvivalentna ja malopređašnjom:

```
int array[4][3] = { { 1, 2, 3 } , { 4, 5, 6 } ,
{ 7, 8, 9 } , { 10, 11, 12 } };
```

Zapamtite, inicijalizacijske vrijednosti moraju biti razdvojene zarezima → čak i ako su prisutne zagrade između njih. Još i koristite zagrade u parovima → zatvorena zagrada za svaku otvorenu zagradu ← ili kompajler postaje zbunjen.

Listing 8.3, **RANDOM.C**, kreira **1000** elementni niz popunjen sa nasumičnim (random) brojevima. Program onda prikazuje elemente niza na ekranu. Zamislite koliko bi linija kood bilo potrebno da se izvrši isti zadatak sa neniznim varijablama.
Vidite novu bibliotečnu funkciju, →→→ **getch()** ←←← , u ovom programu.
getch() funkcija čita jedan karakter sa tastature. U Listing-u **8.3**, **getch()** pauzira program dok korisnik ne pritisne tipku. **getch()** je objašnjen detaljno Dana 14. ☺

Listing 8.3. RANDOM.C kreira multidimenzionalni niz.

```
1:   /* RANDOM.C - Demonstrira upotrebu multidimenzionalnih nizova */
2:
3:   #include <stdio.h>
4:   #include <stdlib.h>
5:   /* Deklaracija tro-dimenzionalnog niza sa 1000 elemenata */
6:
7:   int random_array[10][10][10];
8:   int a, b, c;
9:
10:  main()
11:  {
12:      /* Popuni niz sa nasumicnim brojevima. C-ova bibliotecna */
13:      /* funkcija rand() vraca (returns) nasumican broj. Koristi jednu*/
14:      /* for petlju za svaki index niza. */
15:
16:      for (a = 0; a < 10; a++)
17:      {
18:          for (b = 0; b < 10; b++)
19:          {
20:              for (c = 0; c < 10; c++)
21:              {
22:                  random_array[a][b][c] = rand();
23:              }
24:          }
25:      }
26:
```

```
27:       /* Sad prikazi po 10 elemenata niza */
28:
29:       for (a = 0; a < 10; a++)
30:       {
31:           for (b = 0; b < 10; b++)
32:           {
33:               for (c = 0; c < 10; c++)
34:               {
35:                   printf("\nrandom_array[%d][%d][%d] = ", a, b, c);
36:                   printf("%d", random_array[a][b][c]);
37:               }
38:               printf("\nPress Enter to continue, CTRL-C to quit.");
39:
40:               getchar();
41:           }
42:       }
43:       return 0;
44:   }         /* kraj  main()-a */
```

```
random_array[0][0][0] = 346
random_array[0][0][1] = 130
random_array[0][0][2] = 10982
random_array[0][0][3] = 1090
random_array[0][0][4] = 11656
random_array[0][0][5] = 7117
random_array[0][0][6] = 17595
random_array[0][0][7] = 6415
random_array[0][0][8] = 22948
random_array[0][0][9] = 31126
Press Enter to continue, CTRL-C to quit.
random_array[0][1][0] = 9004
random_array[0][1][1] = 14558
random_array[0][1][2] = 3571
random_array[0][1][3] = 22879
random_array[0][1][4] = 18492
random_array[0][1][5] = 1360
random_array[0][1][6] = 5412
random_array[0][1][7] = 26721
random_array[0][1][8] = 22463
random_array[0][1][9] = 25047
Press Enter to continue, CTRL-C to quit
...         ...
random_array[9][8][0] = 6287
random_array[9][8][1] = 26957
random_array[9][8][2] = 1530
random_array[9][8][3] = 14171
random_array[9][8][4] = 6951
random_array[9][8][5] = 213
random_array[9][8][6] = 14003
random_array[9][8][7] = 29736
random_array[9][8][8] = 15028
random_array[9][8][9] = 18968
Press Enter to continue, CTRL-C to quit.
random_array[9][9][0] = 28559
random_array[9][9][1] = 5268
random_array[9][9][2] = 20182
random_array[9][9][3] = 3633
random_array[9][9][4] = 24779
random_array[9][9][5] = 3024
random_array[9][9][6] = 10853
random_array[9][9][7] = 28205
random_array[9][9][8] = 8930
random_array[9][9][9] = 2873
Press Enter to continue, CTRL-C to quit.
```

ANALIZA: ovaj program ima dvije ugnijezđene **for** petlje. Prije nego što pogledate **for** iskaze detaljno, primjetite da linije **7** i **8** deklarišu četiri varijable. Prva je niz nazvan **random_array**, koji se koristi da čuva nasumične vrijednosti (brojeve).
random_array je trodimenzionalni niz tipa **int** veličine **10** sa **10** sa **10**, što daje **1000** elemenata tipa **int**. Linija **8** deklariše tri varijable: **a**, **b**, i **c**, koje se koriste kao kontrola **for** petljee.

Ovak program još uključuje i file zaglavlja **STDLIB.H** (za standardnu biblioteku) u liniji **4**. Uključen je s ciljem da obezbjedi prototip za **rand()** funkciju u liniji **22**.

Srce programa su dva ugnijezđena **for** iskaza. Prvi je u liniji **16** do **25**, a drugi od linije **29** do **42**. Obadva **for** gnijezda imaju istu strukturu. Rade kao i petlje u Listing-u **6.2**, ali idu jedan nivo dublje.
U prvom setu **for** iskaza, linija **22** se izvršava ponavljano (repeatedly). Linija **22** pridružuje povratnu (return) vrijednost funkcije, **rand()**, elementu **random_array** nizu, gdje je **rand()** bibliotečna funkcija koja vraća nasumičan broj.
Vraćajući se kroz listing, možete primjetiti da linija **20** mijenja varijablu **c** od **0** u **9**. Ovo peetlja kroz najdalji desni index **random_array** niz. Linija **18** peetlja kroz **b**, srednji index nasumičnog niza. Svaki put kada se **b** promjeni, peetlja se kroz sve **c** elemente. Linija **16** inkrementira varijablu **a**, koja peetlja kroz najdalji lijevi index. Svaki put kada se ovaj index promjeni, peetlja se kroz svih **10** vrijednosti indexa **b**, koji u povratku peetlja kroz svih **10** vrijednosti od **c**. Ova petlja inicijalizira svaku vrijednost nasumičnog niza u nasumičan broj.
Linije **29** do **42** sadrže drugo gnijezdo **for** iskaza. One rade kao i prethodni **for** iskaz, ali ova petlja printa svaku od vrijednosti koje su prije pridružene. Nakon što ih se prikaže **10**, linija **38** printa poruku i čeka da se pritisne **Enter**. Linija **40** vodi računa o pritisku tipke koristeći **getchar()**. Da se **Enter** ne pritisne, **getchar()** čeka dok se ne pritisne.

→ Maximalna veličina niza

Zbog načina na koji model memorije radi, ne biste trebali kreirati više od **64KB** podatkovnih varijabli zasad.

NAPOMENA: Neki operativni sistemi nemaju 64KB limit. DOS ima.

Da izračunate smještajni prostor potreban za niz, pomnožite broj elemenata u nizu sa veličinom elemenata. Npr., **500-elementni niz tipa float** zahtjeva smještajni prostor od **500*4=2000 byte-ova**.
Možete odrediti smještajni prostor –u- programu koristeći **C**-ov **sizeof()** operator. **sizeof()** operator je unarni operator, a ne funkcija. Uzima kao i njegovi argumenti ime varijable ili ime tipa podataka i vraća veličinu, u byte-ovima, njihovih argumenata. Upotreba **sizeof()**-a je ilustrovano u **Listing-u 8.4**.

Listing 8.4. Korištenje sizeof() operatora da se odluči potrebni smještajni prostor za niz.

```
1:   /* Demonstrira sizeof() operator */
2:
3:   #include <stdio.h>
4:
5:   /* Declaracija nekoliko 100-elementnih nizova */
6:
7:   int intarray[100];
8:   float floatarray[100];
9:   double doublearray[100];
10:
11:  main()
12:  {
13:      /* Prikazuje veličinu numeričkih tipova podataka */
14:
15:      printf("\n\nSize of int = %d bytes", sizeof(int));
16:      printf("\nSize of short = %d bytes", sizeof(short));
17:      printf("\nSize of long = %d bytes", sizeof(long));
18:      printf("\nSize of float = %d bytes", sizeof(float));
19:      printf("\nSize of double = %d bytes", sizeof(double));
20:
21:      /* Prikazuje veličine od tri niza */
22:
23:      printf("\nSize of intarray = %d bytes", sizeof(intarray));
24:      printf("\nSize of floatarray = %d bytes",
25               sizeof(floatarray));
26:      printf("\nSize of doublearray = %d bytes\n",
27               sizeof(doublearray));
28:
29:      return 0;
30:  }
```

Ovo je izlaz sa 16-bitne Windows 3.1 mašine:

```
Size of int = 2 bytes
Size of short = 2 bytes
Size of long = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of intarray = 200 bytes
Size of floatarray = 400 bytes
Size of doublearray = 800 bytes
```

Ovo je izlaz sa 32-bitnog Windows NT mašine, kao i 32-bitne UNIX mašine:

```
Size of int = 4 bytes
Size of short = 2 bytes
Size of long = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of intarray = 400 bytes
Size of floatarray = 400 bytes
Size of doublearray = 800 bytes
```

Q Mogu li sabrati dva niza zajedno (ili množiti, dijeliti ili ih oduzeti)?
A Ako deklarišete dva niza, **NE** možete ih sabirati. Svaki element mora biti sabran pojedinačno. Vježba 10 ovo ilustruje.

Vježbe:

1. Napišite **C** programsku liniju koja bi deklarisala tri jednodimenzionalna **int**eger niza, s imenima jedan, dva i tri, sa **1000** elemenata svaki.
2. Napištite iskaz koji bi deklarisao **10**-elementni **int**eger niz i inicijalizirao sve njegove elemente na **1**.
3. Dat je sljedeći niz, napišite kood da inicijalizirate sve elemente niza do **88**:
int osamdesetosam[88];
4. Dat je sljedeći niz, napišite kood da inicijalizirate sve elemente niza do **0**:
int stuff[12][10];

LEKCIJA 9: → Razumijevanje pointera ←

→→ Memorija vašeg kompjutera

PC-ev RAM se sastoji od više hiljada sekvencijalnih smještajnih lokacija, i svaka lokacija je indentifikovana sa jedinstvenom adresom. Memorijske adrese u datom kompjuteru su u opsegu od **0** do maximalne vrijednosti koja zavisi od veličine instalirane memorije.
Kada koristite vaš kompjuter, operativni sistem koristi nešto sistemske memorije. Kada pokrenete program, programski kood (instrukcije mašinskog jezika za različite programske zadatke) i podaci (informacije koje program koristi) takođe koriste nešto sistemske memorije. Ovaj dio (teorije) ispituje memorijsko smještanje za programske podatke.

→→→ Kreiranje pointer-a

Primjetite da je adresa varijable **rate** (ili bilo koje druge varijable) broj i može se smatrati kao bilo koji drugi broj u **C**-u. Ako znate adresu varijable, možete kreirati drugu varijablu, u koju ćete smjestiti adresu one prve varijable. Prvi korak je da deklarišete varijablu koja će držati adresu **rate**-a. Dajte joj ime **p_rate**, npr. Na početku, **p_rate** je neinicijalizirana. Prostor je alociran za **p_rate**, ali je njena vrijednost neodređena.
Sljedeći korak je da smjestite adresu varijable **rate** u varijablu **p_rate**. Zato što sad **p_rate** sadrži adresu **rate**-a, ona indicira (pokazuje) lokaciju gdje je rate smještena u memoriji. Tako reći, **p_rate** pokazuje (points) na **rate**, ili je pointer na **rate**.

→→→ Pointeri i jednostavne varijable

→→ Deklaracija pointera

Pointer je numerička vrijednost i, kao i sve varijable, mora biti deklarisan prije upotrebe. Za imena pointer varijabli slijede ista pravila kao i za sve varijable; i ono mora biti jedinstveno.
Ovo poglavlje koristi konvenciju da pointer na varijablu **name** se zove **p_name**. Ovo nije neophodno, ipak, možete nazvati pointere kako god želite (sa **C**-ovim pravilima imenovanja).

Deklaracija pointera ima sljedeću formu:

```
typename *ptrname;
```

typename je **C**-ov tip varijable i pokazuje (indicira) tip varijable na koju pokazuje pointer. Asterisk (*****) je indirektni operator, i on indicira da ***ptrname*** je pointer na tip **typename**, a ne varijabla tipa **typename**. Pointeri mogu biti deklarisani zajedno sa nepointer varijabalama. Evo par primjera:

```
char *ch1, *ch2;          /* ch1 i ch2 su oba pointeri na tip char */
float *value, percent;    /* value je pointer na tip float, i      */
                          /* percent je obicna float varijabla     */
```

NAPOMENA: simbol → ***** ← se koristi i kao indirektni operator i kao operator množenja. Ne brinite da li će kompajler doći u zabunu. Kontext u kojem se → ***** ← koristi obezbjeđuje dovoljno informacija tako da kompajler može shvatiti da li mislite na indirekciju ili množenje.

→→→ Inicijalizacija pointera

Sad kad ste deklarisali pointer, šta možete raditi s njim? Ne možete ništa raditi s njim dok ga ne napravite da pokazuje na nešto. Kao i regularne varijable, neinicijalizirani pointer može biti korišten, ali su rezultati nepredvidivi i potencijalno destruktivni. Dok pointer ne drži adresu varijable, on je beskoristan. Adrese se ne smještaju u pointer putem magije; vaš program je mora tamo smjestiti koristeći **address-of** (adresa-od) operator, (**&**) (ampersand). Kada se stavi (**&**) ispred imena varijable, **address-of** operator vraća adresu varijable. Tako da, inicijalizirate pointer sa iskazom u formi:

```
pointer = &variable;
```

Programski iskaz da inicijalizirate varijablu **p_rate** da pokazuje na varijablu **rate** bi bio:

```
p_rate = &rate;     /* pridruži adresu rate-a, p_rate-u */
```

Prije inicijalizacije, **p_rate** nije pokazivao ni na šta naročito. Nakon inicijalizacije, **p_rate** je pointer na **rate** (pokazuje na **rate**).

→→ Koristeći pointere

Sad, kad znate kako da deklarišete i inicijalizirate pointere, vjerovatno se pitate kako se oni koriste. Indirektni operator (*) dolazi u igru ponovo. Kada → * ← prethodi (je ispred) imenu pointera, on se odnosi na varijablu na koju pokazuje.

Nastavimo sa prethodnim primjerom, u kojem je pointer **p_rate** bio inicijaliziran da pokazuje na varijablu **rate**. Ako napišete ***p_rate**, on se odnosi na varijablu **rate**. Ako hpćete da printate vrijednost **rate**-a, (koja je u ovom primjeru **100**), možete napisati:

```
printf("%d", rate);
```

ili ovo:

```
printf("%d", *p_rate);
```

U **C**-u, ova dva iskaza su ekvivalentna.
Pristup sadržaju varijable, koristeći ime varijable, se zove **direktni pristup**.
Pristup sadržaju varijable, koristeći pointer na varijablu, se zove **inirektan pristup** ili indirekcija.

Ako imate pointer nazvan **ptr**, koji je bio inicijaliziran da pokazuje na varijablu **var**, sljedeće je istinito:

- ***ptr** i **var** -> obadva se odnose na sadržaj od **var** (tj., na bilo koju vrijednost da se nalazi tu).
- **ptr** i **&var** -> odnose se na adresu od **var**

Kao što možete vidjeti, ime pointera bez operatora indirekcije pristupa samoj vrijednosti pointera, koja je, naravno, adresa varijable na koju pokazuje.

Listing 9.1 demonstrira osnovnu upotrebu pointera. Trebali biste unijeti, kompajlirati i pokrenuti ovaj program.

Listing 9.1. Osnovna upotreba pointera.

```
1:  /* Demonstrira osnovnu upotrebu pointera. */
2:
3:  #include <stdio.h>
4:
5:  /* Deklaracija i inicijalizacija int varijable */
6:
7:  int var = 1;
8:
9:  /* Deklaracija pointera na int */
10:
11: int *ptr;
12:
13: main()
14: {
15:     /* Inicijalizacija ptr da pokazuje na var */
16:
17:     ptr = &var;
18:
19:     /* Pristupi var direktni i indirektno */
20:
21:     printf("\nDirect access, var = %d", var);
22:     printf("\nIndirect access, var = %d", *ptr);
23:
24:     /* Prikazi adresu var-a na dva nacina */
25:
```

```
26:     printf("\n\nThe address of var = %d", &var);
27:     printf("\nThe address of var = %d\n", ptr);
28:
29:     return 0;
30: }
      Direct access, var = 1
      Indirect access, var = 1
      The address of var = 4264228
      The address of var = 4264228
```

NAPOMENA: Adresa prijavljena za var možda nije **4264228** na vašem sistemu.

Ovaj listing pokazuje odnos između varijable, njene adrese, pointera i dereferencije pointera ☺.

RAZUMITE šta su pointeri i kako rade. Mastering **C**-a zahtjeva mastering pointera.
NE koristite neinicijalizirane pointere. Rezultati mogu biti razarajući ako ih ne inicijalizirate

→→→ Pointer tipovi varijabli

Svaki byte memorije ima svoju adresu, tako da multibyte-na varijabla u stvari zauzme nekoliko adresa. **Kako, onda, pointeri rade s adresama multibyte-nih varijabli?** Evo kako: Adresa varijable je u stvari adresa prvog (najnižeg) byte-a koji zauzima. Ovo može biti ilustrirano sa primjerom koji deklariše i inicijalizira tri varijable:

```
int vint = 12252;
char vchar = 90;
float vfloat = 1200.156004;
```

U ovom primjeru, **int** varijabla zauzima dva byte-a, **char** varijabla zauzima jedan byte, i **float** varijabla zauzima četiri byte-a:

```
int *p_vint;
char *p_vchar;
float *p_vfloat;
/* additional code goes here */
p_vint = &vint;
p_vchar = &vchar;
p_vfloat = &vfloat;
```

Svaki pointer je jednak sa adresom prvog byte-a od one-na-koju-se-pokazuje varijable.
Svaki pointer je deklarisan da pokazuje na određeni tip varijable. Kompajler zna da pointer na tip **int** pokazuje na prvi od dva byte-a, itd.

→→→→ Pointeri i nizovi

Pointeri mogu biti korisni kada radite sa jednostavnim varijablama, ali su od veće pomoći sa nizovima. Postoji poseban odnos (relacija) između pointera i nizova u **C**-u. U stvari, kada koristite indexnu notaciju niza, koju ste naučili u lekciji 8, vi u stvari koristite pointere. Slijedi objašnjenje.

→→ Ime niza kao pointer

Ime niza bez zagrada je pointer na prvi element niza. Tako da, ako ste deklarisali niz **data[]**, data je adresa prvog elementa niza.

“Stani malo ba”, možda ćete reći, “Zar nam ne treba address-of operator da dobijemo adresu?”. DA.

Takođe možete koristiti izraz **&data[0]** da dobijete adresu prvog elementa niza. U **C**-u, odnos **(data==&data[0])** je true.

Možete deklarisati pointer varijablu i inicijalizirati je da pokazuje na niz.

Npr., sljedeći kood inicijalizira pointer varijablu **p_array** sa adresom prvog elementa od **array[]**:

```
int array[100], *p_array;
/* additional code goes here */
p_array = array;
```

Zato što je **p_array** pointer varijabla, ona se može modifikovati da pokazuje negdje drugo. Za razliku od nizova, **p_array** nije ograničen (locked) da pokazuje na prvi element niza **array[]**. Npr., on može pokazivati na druge elemente niza **array[]**. Kako biste ovo uradili? Prvo, trebate pogledati kako su elementi nizova smješteni u memoriju.

→→ Smještanje elemenata niza

Elementi niza su smješteni u sekvencijalne memorijske lokacije sa prvim elementom na najmanjoj adresi. Koliko visoko zavisi od tipa podataka niza (**char**, **int**, **float**, itd.).
Možemo reći i: da ostvarimo pristup elementima niza nekog naročitog tipa, pointer mora biti uvećan za **sizeof(datatype)**. -> **sizeof()** operator vraća veličinu u byte-ovima **C** tipova podataka.

Listing 9.2 ilustruje odnos između adresa i elemenata različitog tipa nizova, tako što deklarišući nizova tipova **int**, **float** i **double** i prikazujući adrese sukcesivnih (zaredom) elemenata:

Listing 9.2. Prikazuje adrese elemenada niza zaredom (successive array elements).

```
1:  /* Demonstrira odnos izmedju adresa i */
2:  /* elemenata niza razlicitih tipova podataka. */
3:
4:  #include <stdio.h>
5:
6:  /* Deklaracija tri niza i brojacke varijable. */
7:
8:  int i[10], x;
9:  float f[10];
10: double d[10];
11:
12: main()
13: {
14:     /* Printa tebelu zaglavlja */
15:
16:     printf("\t\tInteger\t\tFloat\t\tDouble");
17:
18:     printf("\n================================");
19:     printf("======================");
20:
21:     /* Printa adresu svakog elementa niza. */
22:
23:     for (x = 0; x < 10; x++)
24:         printf("\nElement %d:\t%ld\t\t%ld\t\t%ld", x, &i[x],
25:             &f[x], &d[x]);
26:
27:     printf("\n================================");
28:     printf("======================\n");
29:
30:     return 0;
31: }
```

```
                Integer     Float     Double
 ============================================
 Element 0:     1392        1414      1454
 Element 1:     1394        1418      1462
 Element 2:     1396        1422      1470
 Element 3:     1398        1426      1478
 Element 4:     1400        1430      1486
 Element 5:     1402        1434      1494
 Element 6:     1404        1438      1502
```

```
Element 7:     1406        1442      1510
Element 8:     1408        1446      1518
Element 9:     1410        1450      1526
================================================
```

ANALIZA: Tačne adrese koje prikazuje vaš sistem mogu se razlikovati od ovih, ali je odnos između njih (razmak (gaap)) isti.
printf() pozivi u linijama **16** i **24** koriste tab escape character (**\t**) da se organizuje poravnanje u kolonama. Tri niza su kreirana u linijama **8**, **9** i **10**. Linija **8** deklariše niz **i** tipa **int**, linija **9** deklariše niz **f** tipa **float**, i linija **10** deklariše niz **d** tipa **double**. Linija **16** printa zaglavlje za tabelu koja će da se prikaže. Linije **23**, **24** i **25** su **for** petlja koja printa svaki od tabelinih redova. Prvo je printa broj elemenata **x**. Nakon ovoga slijedi adresa elementa u svakom od tri niza.

→ Pointer Arithmetic

Za pristup elementima niza koristi se pointer aritmetika. Imamo pointer na prvi element niza, pointer mora inkrementirati za veličinu koja je jednaka veličini tipa podataka koji su smješteni u niz.
Trebaju nam samo dvije pointer operacije: **inkrementiranje** i **dekrementiranje**.

→ Inkrementiranje pointera

Kada inkrementirate pointer, povećate mu vrijednost da pokazuje na "sljedeću" adresu (element).
C zna tip podataka na koje pokazuje pointer (iz deklaracije pointera) i poveća adresu smještenu u pointeru za veličinu tipa podatka.
Pretpostavimo da je **ptr_to_int** pointer varijabla na neki element int niza. Ako izvršite iskaz:

```
ptr_to_int++;
```

vrijednost od **ptr_to_int** se poveća za veličinu tipa **int** (obično 2 byte-a), i **ptr_to_int** sad pokazuje na sljedeći element niza.

Istom logikom i **ptr_to_float**:

```
ptr_to_float++;
```

Ovo je true za inkrement veći od **1**. Ako dodate broj **n** pointeru, **C** inkrementira pointer za **n** elemenata niza pridruženog tipa podataka. Tako da:

```
ptr_to_int += 4;
```

poveća vrijednost smještenu u **ptr_to_int** za **8** (pretpostavljajući da je integer 2 byte-a), tako da sad pokazuje na 4 elementa unaprijed.

Tako i :

```
ptr_to_float += 10;
```

poveća vrijednost smještenu u **ptr_to_float** za **40** (pretpostavljajući da je float 4 byte-a), tako da pokazuje 10 elemenata niza unaprijed.

→ Dekrementiranje pointera

Primjenjuje se isti koncept kao i maloprije. Dekrementiranje pointera je u stvari specijalni slučaj inkrementiranja, dodavajući negativnu vrijednost. Ako dekrementirate pointer sa **--** ili **-=** operatorom, pointer aritmetika prilagodi veličinu elemenata niza.

Listing 9.3. Upotreba pointer aritmetike i pointer notacije za pristup elementima niza:

```
1:  /* Demonstrira upotrebu pointer aritmetike za pristup */
2:  /* elementima niza sa pointer notacijom. */
```

```
3:
4:  #include <stdio.h>
5:  #define MAX 10
6:
7:  /* Deklaracija i inicijalizacija integer niza. */
8:
9:  int i_array[MAX] = { 0,1,2,3,4,5,6,7,8,9 };
10:
11: /* Deklaracija pointera na int i int varijable. */
12:
13: int *i_ptr, count;
14:
15: /* Deklaracija i inicijalizacija niza float. */
16:
17: float f_array[MAX] = { .0, .1, .2, .3, .4, .5, .6, .7, .8, .9 };
18:
19: /* Deklaracija pointera na float. */
20:
21: float *f_ptr;
22:
23: main()
24: {
25:     /* Inicijalizacija pointera. */
26:
27:     i_ptr = i_array;
28:     f_ptr = f_array;
29:
30:     /* Printaj elemente niza. */
31:
32:     for (count = 0; count < MAX; count++)
33:         printf("%d\t%f\n", *i_ptr++, *f_ptr++);
34:
35:     return 0;
36: }
            0       0.000000
            1       0.100000
            2       0.200000
            3       0.300000
            4       0.400000
            5       0.500000
            6       0.600000
            7       0.700000
            8       0.800000
            9       0.900000
```

ANALIZA: U liniji **9**, **MAX** se koristi da se odredi broj elemenata u nizu **int**-ova nazvan **i_array**. Elementi u ovom nizu su inicijalizirani u isto vrijeme kad je niz i deklarisan. Linija **13** deklariše dvije dodatne **int** varijable. Prva je pointer nazvan **i_ptr**. Znamo da je ovo pointer jer se koristi indirektni operator (*****). Druga varijabla je jednostavna tipa **int** varijabla nazvana **count**. U liniji **17**, drugi niz je definisan i inicijaliziran. Ovaj niz je tipa **float**, sadrži **MAX** vrijednosti, i inicijaliziran je sa **float** vrijednostima. Linija **21** deklariše pointer na **float** nazvan **f_ptr**.
main() funkcija je od linije **23** do **36**. Program pridružuje adrese od dva niza pointerima sa njihovim tipovima respektivno u linijiama **27** i **28**. Zapamtite, ime niza bez indexa je isto kao i adresa na početku niza. **for** iskaz u linijama **32** i **33** koristi **int** varijablu **count** da broji od **0** do vrijednosti **MAX**. Za svako brojanje, linija **33** dereferencira dva pointera i printa njihove vrijednosti u pozivu **printf()** funkcije. Tada inkrement operator inkrementira svaki od pointera tako da svaki pokazuje na sljedeći element u niizu prije nastavka sljedeće iteracije **for** petlje.

Kad počnete pisati komplexnije programe, upotreba pointera će imati svojih prednosti.

Zapamtite da ne možete raditi operacije inkrementiranja i dekrementiranja na pointer konstantama. (Ime niza bez uglastih zagrada je pointer konstanta). Također zapamtite da kad manipulišete s pointerima na elemente niza, **C** kompajler ne vodi računa (keep track) o početku i kraju niza. Ako niste pažljivi, možete inkrementirati ili dekrementirati pointer tako da pokazuje negdje u memoriju prije ili poslije niza. Nešto je smješteno tamo, ali nisu elementi niza. Trebate voditi računa o pointerima i gdje oni pokazuju.

→→ Ostale manipulacije pointerima

Jedina druga pointer aritmetika se zove differencing, koje se odnosi na oduzimanje dva pointera.

```
       ptr1 - ptr2
/*ovo oduzimanje dva pointera na istom nizu nam govori o međusobnoj
udaljenosti između ova dva elementa na koja pokazuju ova dva pointera */
```

Pointer komparacija je validna samo između pointera koji pokazuju na isti niz. /* **==**, **!=**, **>**, **<**, **>=**, i **<=** */

Niži elementi niza (oni sa manjim indexom) uvijek imaju nižu adresu od većih elemenata.
Tako da je:

```
ptr1 < ptr2
```

istina ako **ptr1** pokazuje na raniji član niza nego **ptr2**.

Table 9.1. Operacije s pointerima.

Operacija	Opis
Pridruživanje	Možete pridružiti vrijednost pointeru. Vrijednost treba biti adresa, pribavljena pomoću **address-of** operatora (**&**) ili od pointer konstante (ime niza).
Indirekcija	**Operator inderekcije** (*****) daje vrijednost smještenu u pointed-to lokaciji.
Address of	Možete koristiti **address-of** operator da pronađete adresu pointera, tako da možete imati pointere na pointere. Ovo je glavna tema dana 15. "Pointers: Beyond the Basics."
Inkrementacija	Možete **dodati int**eger pointeru s ciljem da pokazuje na drugu memorijsku lokaciju.
Dekrementacija	Možete **oduzimati int**eger (cijeli broj) od pointera s ciljem da pokazuje da neku drugu memorijsku lokaciju.
Differencing	Možete oduzeti **int**eger od pointera s ciljem da pokazuje na neku drugu memorijsku lokaciju.
Komparacija	Vrijedi samo za dva pointera koji pokazuju u istom nizu.

→ Oprez s pointerima (Pointer Cautions)

Kada pišete program koji koristi pointere, morate izbjeći jednu ozbiljnu grešku: koristeći neinicijalizirani pointer na lijevoj strani iskaza pridruživanja.
Npr., sljedeći iskaz deklariše pointer na tip **int**:

```
int *ptr;
```

Ovaj pointer još nije inicijaliziran, tako da ne pokazuje ninašta. Da budemo precizniji, ne pokazuje na ništa **znano**. Neinicijalizirani pointer ima neku vrijednost, samo što ne znate koja je. U većini slučajeva je **nula**.

Ako koristite neinicijalizirani pointer u iskazu pridruživanja, evo šta se dešava:

```
*ptr = 12;
```

Vrijednost **12** se pridružuje “bilo kojoj da je” adresi na koju **ptr** pokazuje. Ta adresa može biti skoro bilo gdje u memoriji --- gdje je smješten operativni sistem ili negdje u programskom koodu. **12** koje je smješteno tu može overwrite neke važne informacije, i rezultat može biti od prijavljivanja greške do pada sistema.
Vi morate saami inicijalizirati pointer; kompajler to neće uraditi umjesto vas.

NE pokušavajte da radite matematičke operacije kao što su djeljenje, množenje i modul na pointerima. Sabiranje (inkrementiranje) i oduzimanje (diferencija) pointera je prihvatljiva.
NE zaboravite da oduzimanje ili sabiranje sa pointerom mijenja pointer koji se nalazi na veličini tipa podatka na koji pokazuje. Ne mjenja se za **1** ili za broj koji mu se dodaje (osim ukoliko je pointer na jedan-byte-ni karakter).
RAZUMITE veličine tipova varijabli na vašem kompjuteru.
NE pokušavajte da inkrementirate ili dekrementirate varijablu niza. Pridružite pointer na početnu adresu niza i inkrementirajte se (Listing 9.3).

→→ Notacija indexa niza i pointera

Ime niza bez uglastih zagrada je pointer na prvi element niza. Tako da, možete pristupiti prvom elementu niza koristeći indirektni operator. Ako je **array[]** deklarisan niz, izraz ***array** je **array**-ev prvi element, ***(array+1)** drugi, itd. Ako generalizujete ovo za cijeli niz, sljedeći odnosi su true:

```
*(array) == array[0]
*(array + 1) == array[1]
*(array + 2) == array[2]
...
*(array + n) == array[n]
```

Ovo ilustruje ekvivalenciju index notaciju niza sa pointer notacijom. Možete koristiti bilo koju u vašem programu; **C** kompajler ih vidi kao dva različita načina za pristup podacima niza koristeći pointere.

→→→ Prosljeđivanje niza funkcijama

Ovaj odnos dolazi do izražaja kada želite da prosljedite niz kao argument funkciji. Jedini način da prosljedite niz funkciji je pomoću pointera.

Argument je vrijednost koju pozivajući program (onaj koji poziva) prosljeđuje funkciji. Može biti **int**, **float**, ili bilo koji drugi jednostavni tip podataka, ali mora biti jedna numerička vrijednost. Može biti jedan element niza, ali ne može biti cijeli niz.

Šta ako želite da proslijedite cijeli niz funkciji? Pa, možete imati pointer na niz, i taj pointer je jedna numerička vrijednost (adresa od prvog elementa niza). Ako proslijedite tu vrijednost funkciji, funkcija zna adresu niza i može pristupiti elementima niza koristeći pointer notaciju.

Razmislite drugi problem; ako napišete funkciju koja uzima niz kao argument, vi želite funkciju koja može raditi sa nizovima različitih veličina. Npr., možete napisati funkciju koja nalazi najveći element u integer nizu. Funkcija ne bi bila od velike koristi ako bi bila ograničena na upotrebu sa nizovima jedne fiksne veličine.

Kako funkcija zna veličinu niza, čija je adresa prosljeđena? Zapamtite, vrijednost prosljeđena funkciji je pointer na prvi element niza. On može biti prvi od 10 ili prvi od 10000 elemenata. Postoje dvije metode da kažete funkciji veličinu niza:

Prvi metod: možete identifikovati zadnji element niza smještajući specijalnu vrijednost tamo. Kad funkcija procesira niz, gleda tu vrijednost u svakom elementu. Kada nađe tu vrijednost, doseže se kraj niza. Nedostatak ove metode je da forsira da rezervišete vrijednost kao kraj-od-niza indikator, smanjujući flexibilnost koju imate smještajući prave podatke u niz.

Drugi metod je više flexibilan i izravniji, i koristi se u ovoj knjizi; Prosljedi funkciji veličinu niza kao argument. Ovo može biti jednostavni argument tipa **int**. Tako, funkciji se prosljeđuju dva argumenta: pointer na prvi element niza i **integer** koji specificira broj elemenata u nizu.

Listing 9.4 prihvata vrijednost od korisnika i smješta ih u niz. Onda zove funkciju s imenom **largest()**, prosljeđuje niz (sa pointerom i veličinom). Funkcija nalazi najveću vrijednost u nizu i vraća je pozivajućem programu.

Listing 9.4. Prosljeđivanje niza funkciji.

```
1:  /* Prosljeđivanje niza funkciji. */
2:
3:  #include <stdio.h>
4:
5:  #define MAX 10
6:
7:  int array[MAX], count;
8:
9:  int largest(int x[], int y);
10:
11: main()
12: {
13:     /* Unesi MAX vrijednosti sa tastature. */
14:
15:     for (count = 0; count < MAX; count++)
16:     {
17:         printf("Enter an integer value: ");
18:         scanf("%d", &array[count]);
19:     }
20:
21:     /* Pozovi funkciju i prikaži povratnu vrijednost. */
22:     printf("\n\nLargest value = %d\n", largest(array, MAX));
23:
24:     return 0;
25: }
26: /* Funkcija largest() vraća najveću vrijednost */
27: /* integer nizu */
28:
29: int largest(int x[], int y)
30: {
31:     int count, biggest = -12000;
32:
33:     for ( count = 0; count < y; count++)
34:     {
35:         if (x[count] > biggest)
36:             biggest = x[count];
37:     }
38:
39:     return biggest;
40: }
```

```
Enter an integer value: 1
Enter an integer value: 2
Enter an integer value: 3
Enter an integer value: 4
Enter an integer value: 5
Enter an integer value: 10
Enter an integer value: 9
Enter an integer value: 8
Enter an integer value: 7
Enter an integer value: 6
Largest value = 10
```

ANALIZA: largest() je funkcija koja vraća **int** pozivajućem programu; njen drugi argument je **int** predstavljen kao parametar **y**. **int x[]** indicira da je prvi argument pointer na tip **int**, predstavljen kao parametar **x**. **int x[]** i **int *x** su ekvivalentni -> misle na "**pointer na int**". Prva forma je poželjnija (vidi se da parametar predtavlja pointer na niz). Naravno, pointer ne zna da pokazuje na niz, ali funkcija ga koristi na taj način.

Kada se pozove funkcija **largest()**, parametar **x** drži vrijednost prvog argumenta, tako da je pointer na prvi element niza. Možete koristiti **x** bilo gdje se može koristiti pointer niza. U **largest()**, elementima niza se pristupa koristeći index notaciju u linijama **35** do **36**. Možete, također, koristiti pointer notaciju, prepišite **for** petlju ovako:

```
for (count = 0; count < y; count++)
{
   if (*(x+count) > biggest)
      biggest = *(x+count);
}
```

Listing 9.5 pokazuje drugi način prosljeđivanja niza funkcijama.

Listing 9.5. Alternativni način prosljeđivanja niza funkciji.

```
1:  /* Prosljeđivanje niza funkciji. Alternativni način. */
2:
3:  #include <stdio.h>
4:
5:  #define MAX 10
6:
7:  int array[MAX+1], count;
8:
9:  int largest(int x[]);
10:
11: main()
12: {
13:     /* Unesi MAX vrijednosti sa tastature. */
14:
15:     for (count = 0; count < MAX; count++)
16:     {
17:         printf("Enter an integer value: ");
18:         scanf("%d", &array[count]);
19:
20:         if ( array[count] == 0 )
21:             count = MAX;              /* will exit for loop */
22:     }
23:     array[MAX] = 0;
24:
25:     /* Pozovi funkciju i prikaži povratnu vrijednost. */
26:     printf("\n\nLargest value = %d\n", largest(array));
27:
28:     return 0;
29: }
30: /* Funkcija largest() vraća (returns) najveću vrijednost */
31: /* u integer niz */
32:
33: int largest(int x[])
34: {
35:     int count, biggest = -12000;
36:
37:     for ( count = 0; x[count] != 0; count++)
38:     {
39:         if (x[count] > biggest)
40:             biggest = x[count];
41:     }
42:
43:     return biggest;
44: }
```

```
Enter an integer value: 1
Enter an integer value: 2
Enter an integer value: 3
Enter an integer value: 4
Enter an integer value: 5
Enter an integer value: 10
Enter an integer value: 9
Enter an integer value: 8
Enter an integer value: 7
Enter an integer value: 6
Largest value = 10
```

Evo output-a nakon drugog startanja programa:

```
Enter an integer value: 10
Enter an integer value: 20
Enter an integer value: 55
Enter an integer value: 3
Enter an integer value: 12
Enter an integer value: 0
Largest value = 55
```

ANALIZA: Ovaj program koristi **largest()** koja ima istu funkcionalnost kao i u **Listing-u 9.4**. Razlika je da je potreban samo petlja niza (array tag). **for** petlja u liniji **37** nastavlja na traži najveću vrijednost dok ne neiđe na **0**, tada zna da je gotova.
Razlika između **Listing-a 9.4** i **9.5**:
u liniji **7** morate dodati extra element nizu da smjesti vrijednost koja indicira kraj. U linijama **20** i **21**, **if** iskaz se dodaje da se vidi da li je korisnik unio **0**, što signalizira da je završio sa unosom podataka (vrijednosti). Ako se unese **0**, **count** se smješta na maximalnu vrijednost tako da se iz **for** petlje može čisto izaći. Linija **23** osigurava da je zadnji element **0** u čijem je slučaju korisnik unio maximalan broj vrijednosti (**MAX**).
Šta se desi kada zaboravite staviti **0** na kraju niza? **largest()** nastavlja nakon kraja niza, upoređujući vrijednosti u memoriji dok ne nađe **0**.
Kao što možete vidjeti, prosljeđivanje niza funkciji nije naročito teško. Jednostavno trebate prosljediti pointer na prvi element niza. U većini situacija, trebate prosljediti i broj elemenata niza. U funkciji, vrijednost poitera može biti korištena za pristup elementima niza pomoći indexa ili pointer notacijom.

UPOZORENJE: Sjetite se iz Dana 5 da kada se jednostavna varijabla prosljeđuje funkciji, samo se kopija vrijednosti varijable prosljeđuje. Funkcija može koristiti vrijednost, ali ne može promijeniti oreginalnu varijablu jer nema pristup saamoj varijabli. Kada prosljedite niz funkciji, stvari su drugačije. Funkciji se proslijedi adresa niza, ne samo kopija vrijednosti u nizu. Kood u funkciji radi sa stvarnim elementima niza koje može izmjeniti (promjeniti (modifikovati)) vrijednosti koje su smještene u nizu.

SAŽETAK:

Ovo vas je poglavlje upoznalo sa pointerima, centralnom dijelu **C** programiranja. **Pointer** je varijabla koja drži adresu neke druge varijable; za pointer je rečeno da “pokazuje na” varijablu čiju adresu drži.

Dva operatora koja su potrebna sa pointerima su **address-of operator(&)** i **indirektni operator (*)**.

Kada se stavi ipred imena pointera, indirektni operator vraća sadržaj varijable na-koju-pokazuje.

Pointeri i nizovi imaju poseban odnos. Ime niza bez uglastih zagrada je pointer na prvi element niza. Specijalna karakteristika pointer aritmetike ga čini laganim da pristupi elementima niza (pointere). Notacija indexa niza je u stvari specijalan oblik pointer notacije.
Takođe ste naučili da prosljeđujete niz funkcijama, prosljeđujući pointer na niz. Jednom kada funkcija zna adresu niza i njegovu veličinu, ona može pristupiti elementima niza koristeći pointer notaciju ili indexnu notaciju.

P&O (pitanja&odgovori)

P Kako kompajler zna razliku između * za množenje, za dereferenciranje i za deklaraciju pointera?
O Kompajler interpretira različite upotrebe asteriska * zavisno od kontexta u kojem se koristi. Ako iskaz koji se procjenjuje počinje sa tipom varijable, može pretpostaviti da se asterisk koristi za deklaraciju pointera. Ako se asterisk koristi sa varijablom koja je bila deklarisana kao pointer, ali ne u deklaraciji varijable, asterisk se podrazumjeva za dereferenciranje. Ako se koristi u matematičkim izrazima, ali ne sa poiter varijablom, za asterisk se pretpostavlja da se koristi kao operator množenja.
P Šta se dešava ako koristimo address-of operator na pointeru?
O Dobijete adresu pointer varijable. Zapamtite, pointer je samo još jedna varijabla koja drži adresu varijable na koju pokazuje.
P Jesu li varijable uvijek smještene na istoj lokaciji?
O Ne. Svaki put kada se program starta, njegove varijable mogu biti smještene na druge adrese unutar kompjutera. Nikad nemojte pridruživati konstantne adresne vrijednosti pointeru.

LEKCIJA 10: → Characters and Strings ←

Karakter (character) je jedno slovo, broj, tačka, zarez, ili neki takav specijalan znak.
String je svaka sekvenca tih karaktera.
Stringovi se koriste da čuvaju tekstualne podatke, koji se sastoje od slova, brojeva, tačke, zareza, i ostalih simbola.

→→→ char tip podataka

C koristi **char** tip podataka da drži karaktere.
char je jedna od **C**-ovih numeričko integerskih tipova podataka. Ako je **char** numerički tip, kako se on može koristiti da drži (čuva) karaktere?

Odgovor leži u kako **C** smješta karaktere. Memorija vašeg kompjutera smješta sve podatke u numeričkom formatu (formi). Nema direktnog načina da se smjeste karakteri.
Ipak, postoji numerički kood za svaki karakter. Ovo se zove **ASCII** kood ili **ASCII** karakter skup (American Standard Code for Information Interchange).
Kood pridružuje vrijednosti između **0** i **255** za velika i mala slova, brojeve, znakova punktacije, i druge simbole.
Vjerovatno ste malo munjeni, jelda da jeste, jelda jelda?. Ako **C** smješta karaktere kao brojeve, kako vaš program zna da li je data **char** varijabla karakter ili broj? Pa, kao što ste naučili, deklarisati varijablu kao tip **char** nije dovoljno; morate uraditi nešto drugo sa varijablom:

- Ako se **char** varijabla koristi negdje u **C** programu gdje se očekuje karakter, ona se interpretira kao karakter.
- Ako se **char** varijabla koristi negdje u **C** programu gdje se očekuje broj, ona se interpretira kao broj.

→→ Upotreba Character varijabli

Kao i ostale varijable, morate deklarisati karaktere prije nego što ih koristite, i možete ih inicijalizirati za vrijeme deklaracije. Evo par primjera:

```
char a, b, c;         /* Declare three uninitialized char variables */
char code = `x';      /* Declare the char variable named code */
                      /* and store the character x there */
code = `!';           /* Store ! in the variable named code */
```

Da kreirate literalnu karakter konstantu, zatvorite jedan karakter u znake apostrofa. Kompajler automatski prevodi literalne karakterne konstante u korespondirajući (odgovarajući) **ASCII** kood, i pridružuje numeričku vrijednost varijabli.

Možete kreirati simbolično karakternu konstantu koristeći ili **#define** direktivu ili **const** ključnu riječ:

```
#define EX `x'
char code = EX;      /* Sets code equal to `x' */
const char A = `Z';
```

Funkcija **printf()** se može koristiti da printa i karaktere i brojeve. Format string **%c** govori (instructs) **printf()** da printa karakter, dok **%d** govori **printf()** da printa decimalni integer.

Listing 10.1 inicijalizira dva tipa **char** varijabli i printa svaku od njih, prvo kao karakter, a onda kao broj:
Listing 10.1. Numerička priroda varijabli tipa char**.**

```
1:  /* Demonstrira numeričku prirodu char varijabli */
2:
3:  #include <stdio.h>
4:
5:  /* Deklaracija i inicijalizacija dvije char varijable */
6:
7:  char c1 = `a';
8:  char c2 = 90;
9:
```

```
10: main()
11: {
12:     /* Print varijablu c1 kao karakter, onda kao broj */
13:
14:     printf("\nAs a character, variable c1 is %c", c1);
15:     printf("\nAs a number, variable c1 is %d", c1);
16:
17:     /* Uradi isto za varjiablu c2 */
18:
19:     printf("\nAs a character, variable c2 is %c", c2);
20:     printf("\nAs a number, variable c2 is %d\n", c2);
21:
22:     return 0;
23: }
      As a character, variable c1 is a
      As a number, variable c1 is 97
      As a character, variable c2 is Z
      As a number, variable c2 is 90
```

ANALIZA: Kao što ste naučili Dana 3 da dozvoljeni opseg za varijablu tipa **char** ide samo do **127**, gdje **ASCII** kood ide do **255**. **ASCII** kood je u stvari podjeljen u dva dijela. Standardni **ASCII** kood ide samo do **127**, ovaj opseg uključuje sva slova, brojeve, znakova interpunkcije i ostale simbole na tastaturi. Kood od **128** do **255** je proširenі **ASCII** kood i predstavlja specijalne karaktere kao što su stranjska slova i grafičke simbole. Tako da, za standardne tekstualne podatke, možete koristiti varijable tipa char, ako želite da printate proširene **ASCII** karaktere, morate koristiti **unsigned** char.

Listing 10.2 printa neke od proširenih **ASCI**I karaktera.

Listing 10.2. Printanje proširenih ASCII karaktera.

```
1:  /* Demonstrira printanje proširenih ASCII karaktera */
2:
3:  #include <stdio.h>
4:
5:  unsigned char x;    /* Mora biti unsigned za proširenі ASCII */
6:
7:  main()
8:  {
9:      /* Printa proširene ASCII karaktere od 180 do 203 */
10:
11:     for (x = 180; x < 204; x++)
12:     {
13:         printf("ASCII code %d is character %c\n", x, x);
14:     }
15:
16:         return 0;
17: }
      ASCII code 180 is character ¥
      ASCII code 181 is character µ
      ASCII code 182 is character [partialdiff]
      ASCII code 183 is character [Sigma]
      ASCII code 184 is character [Pi]
      ASCII code 185 is character [pi]
      ASCII code 186 is character [integral]
      ASCII code 187 is character ª
      ASCII code 188 is character º
      ASCII code 189 is character [Omega]
      ASCII code 190 is character æ
      ASCII code 191 is character ø
      ASCII code 192 is character ¿
      ASCII code 193 is character ¡
      ASCII code 194 is character ¬
      ASCII code 195 is character [radical]
      ASCII code 196 is character [florin]
      ASCII code 197 is character ~
```

```
ASCII code 198 is character [Delta]
ASCII code 199 is character «
ASCII code 200 is character »
ASCII code 201 is character ...
ASCII code 202 is character g
ASCII code 203 is character À
```

ANALIZA: Gledajući ovaj program, možete vidjeti da linija **5** deklariše **unsigned** karakter varijablu, **x**. Ovo daje opseg od **0** do **255**. Kao i sa numeričkim tipovima podataka, NE smijete inicijalizirati **char** varijablu na vrijednost izvan dozvoljenog opsega, u kojem slučaju možete dobiti neočekivane rezultate.
U liniji **11**, **x** nije inicijaliziran van opsega; nego je inicijaliziran na **180**. U **for** iskazu, **x** se inkrementira za **1** dok ne nostigne **204**. Svaki put kada se **x** inkrementira, linija **13** printa vrijednost **x**-a i karakternu vrijednost **x**-a.
Zapamtite da **%c** printa karaktere, ili **ASCII**, vrijednost **x**-a.

KORISTITE %c da printate karakternu vrijednost broja
NE koristite znake navoda (nego **apostrofa**) kada inicijalizirate karakternu varijablu.
NE pokušavajte da stavite proširenu **ASCII** karakternu vrijednost u **signed char** varijablu.

NAPOMENA: Neki komp. sistemi mogu koristiti drugačiji karakterni set (skup), ipak, većina ih koristi iste vrijednosti za **0** do **127**.

→→→ Korištenje (upotreba) stringova

Varijable tipa **char** mogu držati samo jedan karakter, tako da one imaju ograničenu upotrebu.
Također, trebate način kako da smještate stringove (strings), koji su sekvence karaktera. Ime osobe i adresa su primjeri stringova. Iako ne postoji poseban tip podataka za stringove, **C** rješava ovaj tip informacija sa nizovima karaktera.

→→ Nizovi karaktera

Da bi držali string od šest karaktera, npr., trebate deklarisati niz tipa **char** sa sedam elemenata.
Nizovi tipa **char** se deklarišu kao i nizovi drugih tipova podataka. Npr., iskaz:

```
char string[10];
```

deklariše 10-elementni niz tipa **char**. Ovaj se niz može koristiti da drži string od devet ili manje karaktera.

"Čekaj zeru", možda ćete reći, "to je 10-elementni niz, pa kako onda može držati samo devet karaktera?". U **C**-u, string je definisan kao sekvenca karaktera koja završava sa **null** karakterom, specijalnim karakterom koji se predstavlja sa **\0**.
Iako se predstavlja sa dva karaktera (backslash i nula), **null** karakter se interpretira kao jedan karakter i ima **ASCII** vrijednost **0**. To je jedan od **C**-ovih escape sequences (izlaznih sekvenci), koje su obrađene Dana 7.
Kada **C** program smjesti string **Guru**, npr., on smjesti sedam karaktera: **G**, **u**, **r**, **u** i **null** karakter **\0**, što čini total od pet karaktera. Tako da, niz karaktera može sadržavati string karaktera koji broje jedan manje element nego ukupni broj elemenata u niizu.
Varijabla tipa **char** je veličine jedan byte, tako da broj byte-ova u niizu varijable tipa **char** je isti kao i broj elemenata u niizu.

→→ Inicijalizacija karakternog niiza

Kao i ostali **C**-ovi tipovi podataka, karakterni niz može biti inicijaliziran kada se deklariše.
Karakternom niizu mogu biti pridružene vrijednosti element po element, kako slijedi:

```
char string[10] = { `G', `u', `r', `u', `\0' };
```

Više je konvencionalno, koristiti **literalni string**, koji se predstavlja kao sekvenca karaktera između znakova navoda:

```
char string[10] = "Guru";
```

Kada koristite literalni string u vašem programu, kompajler automatski dodaje terminirajući **null** karakter na kraju stringa. Ako ne specifišete broj indexa kada deklarišete niz, kompajler izračuna veličinu niza umjesto vas. Tako da, sljedeća linija kreira i inicijalizira peto-elementni niz:

```
char string[] = "Guru";
```

Zapamtite da stringovi zahtjevaju terminirajući **null** karakter.
C-ove funkcija koje manipulišu stringovima (pokrivene Dana 17, "Manipulating String") određuju dužinu stringa gledajući na **null** karakter. Ove funkcije nemaju drugog načina kako da prepoznaju kraj stringa. Ako **null** karakter nedostaje, vaš program misli da se string produžava do sljedećeg **null** karaktera u memoriji. Zeznuti programski bugovi mogu rezultirati od ovakve vrste greške.

→→→ Stringovi i pointeri

Vidjeli ste da se stringovi smještaju u nizove tipa **char**, sa krajem stringa (koji ne mora okupirati cijeli niz) markiranim sa **null** karakterom. Zato što je kraj niza markiran, sve što treba da uradite s ciljem da definišete dati string, je nešto što će da pokazuje na njegov početak. (da li je pokazuje prava riječ? Naravno da jeste!!!). S gore rečenim, vjerovatno skačete unaprijed sa zaključcima. Iz Dana 9, "Understanding Pointers" znate da je ime niza pointer na prvi element tog niza. Tako da, za string koji je smješten u taj niz, trebate samo ime tog niza da biste mu pristupili. U stvari, upotreba imena niza je **C**-ov standardni metod za pristupanje stringovima. Da budemo precizniji, korištenje imena niza za pristup stringovima je metod koji **C** bibliotečna funkcija očekuje. **C** standardna biblioteka uključuje broj funkcija koje manipulišu stringovima. (Dan 17).

Da prosljedite string jednoj od ovih funkcija, vi prosljedite ime niza. Isto je true za funkcije prikazivanja stringa: **printf()** i **puts()**, o kojima smo pričali ranije u ovom poglavlju.
Možda ste primjetili da sam spomenuo "stringovi smješteni u niz" maloprije.
Da li ovo implicira da neki stringovi nisu smješteni u nizove??? Naravno da da, I nastavak objačnjava zašto.

→→ Stringovi bez nizova

Iz prethodnog dijela, znate da je string definisan sa imenom niza karaktera i **null** karakterom. Ime niiza je pointer tipa **char** na početak stringa. **null** označava kraj stringa. Stvarni prostor koji zauzima string u nizu je slučajan (incidential). U stvari, jedina svrha kojoj služi niz, je da obezbjedi alociranje prostora za string.
Šta ako biste mogli naći nešto memorijskog smještajnog prostora bez alociranja niza? Tada biste mogli smjestiti string sa svojim terminirajućim **null** karakterom . Pointer na prvi karakter može služiti za specificiranje (određivanje) početka stringa, baš kao i da je string u alociranom nizu. Kako naći smještajni prostor memorije? Postoje dvije metode:

Jedna alocira prostor za literalni string kada se program kompajlira, a druga koristi **malloc()** funkciju da alocira prostor dok se program izvršava, proces znan kao **dinamička alokacija**.

→→→ Alokacija string prostora pri kompilaciji

Početak stringa, kako se pomenuto ranije, je indiciran sa pointerom na varijablu tipa **char**. Možete se sjetiti kako da deklarišete takav pointer.

```
char *message;
```

Ovaj iskaz deklariše pointer na varijablu tipa **char** s imenom **message**. On ne pokazuje na ništa sad, ali šta ako promjenite pointer deklaraciju da čita:

```
char *message = "Great Caesar\'s Ghost!";
```

Kada se iskaz izvrši, string **Great Caesar's Ghost!** (sa terminirajućim **null** karakterom) je smješten negdje u memoriju, i pointer **message** je inicijaliziran da pokazuje na prvi karakter stringa. Nije bitno gdje je u memoriji string smješten; kompajler to radi automatski.
Jednom definisan, **message** je pointer na string i može se koristiti kao takav.

Prethodna deklaracija / inicijalizacija je ekvivalentna sa sljedećim, i dvije notacije ***message** i **message[]** su takođe ekvivalentne, obadvije znače “pointer na”:

```
char message[] = "Great Caesar\'s Ghost!";
```

Ovaj metod alociranja prostora za smještanje stringa je ok kada znate šta trebate dok pišete program. Šta ako program ima različite potrebe za smještanje stringa, zavisno od unosa korisnika ili nekog drugog faktora koji je nepoznat dok pišete program?

Koristiite **malloc()** funkciju, koja vam dozvoljava da alocirate smještajni prostor u-leetu (“on-the-fly”).

→→→ malloc() Funkcija

malloc() funkcija je jedna od **C**-ovih alocirajućih funkcija. Kada pozovete **malloc()**, vi prosljedite broj byte-ova koji su potrebni memoriji. **malloc()** pronalazi i rezerviše blok memorije potrebne veličine i vraća adresu prvog byte-a u bloku. Ne trebate se brinuti gdje je nađena memorija; to se rješava automatski.
malloc() funkcija vraća adresu, i njen povratni tip je pointer na tip void. **Zašto void???** Pointer na tip void je kompatibilan sa svim tipovima podataka. Zato što, memorija koja je alocirana od strane **malloc()**-a, može biti korištena za smještanje bilo kojeg **C**-ovog tipa podataka, tip void-ovog povratnog tipa je prigodan.

→→ malloc() Funkcija

```
#include <stdlib.h>
void *malloc(size_t size);
```

malloc() alocira blok memorije koji je broj byte-ova iskazan u **size**. Alocirajući memoriju koja je potrebna sa **malloc()**, umjesto sve odjednom kada program starta, možete koristiti kompjutersku memoriju efikasnije.
Kada koristite **malloc()**, trebate uključiti **STDLIB.H** file zaglavlja.
Neki kompajleri imaju druge file-ove zaglavlja koje treba uključiti, ipak, radi portabilnosti, najbolje je uključiti (include) **STDLIB.H**.

malloc() vraća pointer na alocirani blok memorije. **Ako malloc() nije u mogućnosti da alocira potrebnu količinu memorije, on vraća null.** Kad god pokušate da alocirate memoriju, trebate uvijek provjeriti povratnu vrijednost, čak i ako je količina memorije koja se alocira mala.

Primjer 1

```
#include <stdlib.h>
#include <stdio.h>
main()
{
   /* allocate memory for a 100-character string */
   char *str;
   if (( str = (char *) malloc(100)) == NULL)
   {
      printf( "Not enough memory to allocate buffer\n");
      exit(1);
   }
   printf( "String was allocated!"\n );
   return 0;
}
```

Primjer 2

```
/* allocate memory for an array of 50 integers */
int *numbers;
numbers = (int *) malloc(50 * sizeof(int));
```

Primjer 3

```
/* allocate memory for an array of 10 float values */
```

```
float *numbers;
numbers = (float *) malloc(10 * sizeof(float));
```

Korištenje malloc() Funkcije

Možete koristiti **malloc()** da alocirate memoriju da smjesti jedan tip **char**. Prvi, deklarišete pointer na tip **char**.

```
char *ptr;
```

Onda, pozovite **malloc()** i prosljedite željenu veličinu memorijskog bloka. Zato što obično tip **char** zauzima jedan byte, trebate blok od jednog byte-a. Povratna vrijednost od strane **malloc()**-a je pridružena pointeru.

```
ptr = malloc(1);
```

Ovaj iskaz alocira memorijski blok od jednog byte-a i pridružuje njegovu adresu **ptr**-u. Za razliku od varijabli koje su deklarisane u programu, ovaj byte memorije nema ime. Samo pointer može da referencira varijablu. Npr., da smjestite karakter **'x'** tu, napisali biste:

```
*ptr = `x';
```

Alociranje prostora za string sa **malloc()**-om je skoro identično sa korištenjem **malloc()**-a za alociranje prostora za jednu varijablu tipa **char**. Glavna razlika je da treba da znate veličinu prostora za alocranje → maximalni broj karaktera u stringu. Ovaj maximum zavisi od potreba vašeg programa. Za ovaj primjer, recimo da želite da alocirate prostor za string od **99** karaktera, plus jedan za terminirajući **null** karakter, što je ukupno **100** elemenata.
Prvo deklarišete pointer na tip **char**, a onda zovnete **malloc()**:

```
char *ptr;
ptr = malloc(100);
```

Sad **ptr** pokazuje na rezervisani blok od **100** byte-ova koji može biti korišten za smještanje stringa i manipulaciju. Možete koristiti **ptr** kao da je vaš program eksplicitno alocirao taj prostor sa sljedećom deklaracijom niza:

```
char ptr[100];
```

Upotreba **malloc()**-a dozvoljava vašem programu da alocira smještajni prostor koji je potreban kao odgovor na vaše potrebe. Naravno, dostupni prostor nije neograničen; on zavisi od količine memorije instalirane na vešem kopjuteru i o drugim programskim smještajnim zahtjevima.
Ako nema dovoljno raspoložive memorije, **malloc()** vraća **0** (**null**). Vaš program bi trebao testirati povratnu vrijednost **malloc()**-a tako da znate da je zahtjevana memorija alocirana uspješno. Uvijek trebate testirati **malloc()**-ovu povratnu vrijednost u odnosu na simboličnu konstatnu **NULL**, koja je definisana u **STDLIB.H**.

Listing 10.3 ilustrira upotrebu **malloc()**-a. Svaki program koji koristi **malloc()** mora **#include** file zaglavlja **STDLIB.H**.

Listing 10.3. Upotreba malloc() funkcije za alociranje smještajnog prostora za string podatke.

```
1:  /* Demonstrira upotrebu malloc() za alociranje smjestajnog */
2:  /* prostora za string podatke. */
3:
4:  #include <stdio.h>
5:  #include <stdlib.h>
6:
7:  char count, *ptr, *p;
8:
9:  main()
10: {
11:    /* Allocira blok od 35 byte-a. Testira za uspjeh (alociranja). */
12:    /* exit() bibliotecna funkcija terminates (obustavlja) program. */
13:
14:    ptr = malloc(35 * sizeof(char));
15:
```

```
16:     if (ptr == NULL)
17:     {
18:         puts("Memory allocation error.");
19:         exit(1);
20:     }
21:
22:     /* Popuni string sa vrijednostima od 65 do 90, */
23:     /* koje su ASCII koodovi za A-Z. */
24:
25:     /* p je pointer koristen da se prodje kroz string. */
26:     /* Zelite da ptr ostane pointer na pocetak */
27:     /* stringa. */
28:
29:     p = ptr;
30:
31:     for (count = 65; count < 91 ; count++)
32:         *p++ = count;
33:
34:     /* Dodaj terminirajuci null karakter. */
35:
36:     *p = `\0';
37:
38:     /* Prikazi string na ekranu. */
39:
40:     puts(ptr);
41:
42:     return 0;
43: }
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

ANALIZA: Ovaj program koristi **malloc()** na jednostavan način.
Linija **7** deklariše dva pointera i karakternu varijablu koji se koriste kasnije u listing-u. Nijedna od ovih varijabli nije inicijalizirana, tako da ne bi trebale biti korištene – još.
malloc() funkcija se poziva u liniji **14** sa parametrima **35** pomnoženo sa **size of char**. Jeste li mogli koristiti samo **35**??? DA, ali pretpostavljate da će svi koji pokrenu ovaj program koristiti kompjuter koji smješta varijable tipa **char** kao jedan byte veličine. Zapamtite od Dana 3, da različiti kompajleri mogu koristiti različite varijable različitih veličina. Koristeći **size-of** operator je lak put ka kreiranju portabilnog kooda.

Nikad nemojte pretpostavljati da **malloc()** uzima onoliko memorije koliko mu vi kažete. U stvari, ne govorite mu da uzme memoriju → vi ga pitate. Linija **16** pokazuje najlakši put da provjerite da li je **malloc()** obezbjedio memoriju. Ako je memorija alocirana, **ptr** pokazuje na nju, a ako nije, **ptr** je **null**. Ako program ne uspije da dobije memoriju, linije **18** i **19** prikazuju error poruku (greška) i nadajmo se, izlaze iz programa.
Linija **29** inicijalizira drugi pointer deklarisan u liniji **7**, **p**. On je pridružen istoj adresnoj vrijednosti kao i **ptr**. **for** petlja koristi ovaj novi pointer da smjesti vrijednosti u alociranu memoriju. Gledajući u liniju **31**, vidite da je **count** inicijaliziran na **65** i inkrementiran za **1** sve dok ne dostigne **91**. Za svaku petlju **for** iskaza, vrijednost **count**-a se pridružuje adresi na koju pokazuje **p**. Primjetite da svaki put kada se **count** inkrementira, adresa na koju pokazuje **p** se također inkrementira. Ovo znači da svaka vrijednost se smješta jedna iza druge u memoriji.
Trebali ste primjetiti da se **count**-u pridružuju brojevi, koji su varijable tipa **char**. Sjećate li se diskusije o **ASCII** karakterima i njihovim numeričkim ekvivalentima??? → broj **65** je ekvivalentan sa **A**, **66** ekvivalentan sa **B**, itd. **for** petlja završava nakon što se alfabet pridruži memorijskoj lokaciji na koju pokazuje.
Linija **36** odsjeca vrijednosti karaktera na koje se pokazuje (pointed to) stavljajući **null** na zadnjoj adresi na koju pokazuje **p**. Primjenjujući **null**, sad možete koristiti ove vrijednosti kao string.
Zapamtite da **ptr** još uvijek pokazuje na prvu vrijednost **A**, tako da ako je koristite kao string, on printa svaki karakter dok ne dosegne **null**. Linija **40** korist **puts()** da dokaže ovo i da pokaže rezultate šta je urađeno.

NE alocirajte više memorije nego što je potrebno. Nemaju svi puno memorije, tako da ne treba da se razbacujete s njom.
NE pokušavajte da pridružite novi string karakternom niizu koji je prethodno alociran sa dovoljno memorije da drži mali string. Npr., u ovoj deklaraciji:

```
char a_string[] = "NO";
```

string pokazuje na **"NO"**. Ako pokužate da pridružite **"YES"** ovom niizu, možete imati ozbiljnih problema. Niiz inicijalno može držati samo tri karaktera → **'N'**, **'O'**, i **null**. **"YES"** je četvoro-karakterni → **'Y'**, **'E'**, **'S'**, i **null**. Nemate predstavu šta četvrti karakter, **null**, prepisuje (overwrites).

→→→→ Prikazivanje stringova i karaktera

Ako vaš program koristi string podatke, on vjerovatno mora i da prikaže podatke na ekranu. Prikazivanje stringa se obično radi sa ili **puts()** funkcijom ili sa **printf()** funkcijom.

→→→ puts() ←←← Funkcija

puts() funkcija stavlja string na-ekran, potom njegovo ime. Pointer na string koji će biti prikazan, je jedini argument koji **puts()** uzima. Zato što se literalni string procjenjuje kao pointer na tring, **puts()** se može koristiti da prikaže kako literalne stringove tako i string varijable. **puts()** funkcija automatski ubacuje karakter nova-linija na kraju svakog prikaza stringa, tako da je svaki naredni string, koji prikazuje **puts()** u novoj liniji.

Listing 10.4 ilustrira upotrebu (korištenje) **puts()**-a.

Listing 10.4. Upotreba puts() funkcije za prikaz text-a na-ekranu.

```
1:  /* Demonstrira prikazivanje stringa sa puts(). */
2:
3:  #include <stdio.h>
4:
5:  char *message1 = "C";
6:  char *message2 = "is the";
7:  char *message3 = "best";
8:  char *message4 = "programming";
9:  char *message5 = "language!!";
10:
11: main()
12: {
13:    puts(message1);
14:    puts(message2);
15:    puts(message3);
16:    puts(message4);
17:    puts(message5);
18:
19:    return 0;
20: }
```

```
C
is the
best
programming
language!!
```

→→→ printf() Funkcija

Stringove također možete prikazivati koristeći **printf()** bibliotečnu funkciju. Sjetite se iz Dana 7 da **printf()** koristi format string i konverzioni specifikator da oblikuje svoj izlaz (output). Da prikažete string, koristite konverzioni specifikator **%s**.
Kada **printf()** susretne **%s** u svom format string-u, funkcija upoređuje **%s** sa korespondirajućim (odgovarajućim) argumentom u svojoj listi argumenata. Za string, ovaj argument mora biti pointer na string koji želite da prikažete. **printf()** funkcija prikazuje string na-ekranu, zaustavlja se kada dođe do stringovog terminirajućeg **null** karaktera. Npr.:

```
char *str = "A message to display";
printf("%s", str);
```

Takođe možete prikazati višestruke stringove i miješati (mixati) ih sa literalnim textom i/ili numeričkim varijablama:

```
char *bank = "First Federal";
char *name = "John Doe";
int balance = 1000;
printf("The balance at %s for %s is %d.", bank, name, balance);
```

Rezultirajući izlaz je:

```
The balance at First Federal for John Doe is 1000.
```

Zasada, ove informacije bi vam trebale biti dovoljne da prikazujete string podatke u vašim programima. Kompletni detalji za korištenje **printf()**-a su dati Dana 14, "Working with the Screen, Printer, and Keyboard."

→→→→ Čitanje string-ova sa tastature

S dodatkom prikazivanja string-ova, program često treba da prihvata unesene string podatke od strane korisnika via tastature. **C** biblioteka ima dvije funkcije koje mogu biti korištene za ovu svrhu:
gets() i **scanf()**.
Prije nego što možete čitati string sa tastature, ipak, morate imati nešto da stavite. Možete kreirati prostor za smještanje string-a koristeći bilo koji od metoda o kojima smo govorili ranije → deklaracija niza ili → **malloc()** funkcija.

→→→ Unošenje string-ova koristeći →→→ gets() ←←← funkciju

gets() funkcija uzima string sa tastature. Kada se pozove **gets()**, on čita sve utipkane karaktere sa tastature sve do prvog karaktera novi-red (newline) (koji VI generišete pritiskom na <enter>). Ova funkcija odbacuje novi red, dodaje **null** karakter, i daje string pozivajućem programu. String je smješten na lokaciji na koju pokazuje pointer na tip **char** koji je prosljeđen do **gets()**-a. Program koji koristi **gets()** mora uključiti (**#include**) file **STDIO.H**.

Listing 10.5 prezentuje primjer:

Listing 10.5. Korištenje gets()-a za unos string podataka sa tastature.

```
1:  /* Demonstrira koristenje gets() bibliotecne funkcije. */
2:
3:  #include <stdio.h>
4:
5:  /* Alocira karakterni niz da drzi unos (input). */
6:
7:  char input[81];
8:
9:  main()
10: {
11:     puts("Enter some text, then press Enter: ");
12:     gets(input);
13:     printf("You entered: %s\n", input);
14:
15:     return 0;
16: }
        Enter some text, then press Enter:
        This is a test
        You entered: This is a test
```

ANALIZA: U ovom primjeru, argument za **gets()** je unosni izraz (expression input), koji je ime niiza tipa **char** i samim tim pointer na prvi element niiza.

Niz je deklarisan sa **81** elemenata u liniji **7**. Zato što maximalna moguća dužina linije na većini kompjutera iznosi **80** karaktera (op.a.), ova veličina niiza obezbjeđuje prostor za najdužu moguću unosnu liniju (plus **null** karakter koji **gets()** koristi na kraju).
gets() funkcija ima povratnu vrijednost, koja je ignorisana u prethodnom primjeru. **gets()** vraća pointer na tip **char** sa adresom gdje je smješten uneseni string. DA, ovo je ista vrijednost koja je prosljeđena **gets()**-u, ali vrijednost koja se vraća programu na ovaj način omogućava vašem programu da testira za praznu liniju (blank line).

Listing 10.6 pokazuje kako da uradite ovo:
Listing 10.6. Koristeći **gets()**-ovu povratnu vrijednost za test prazne linije (blank line).

```
1:  /* Demonstrira koristenje gets() povratne vrijednosti. */
2:
3:  #include <stdio.h>
4:
5:  /* Deklaracija karakternog niza da drzi unos, i pointer. */
6:
7:  char input[81], *ptr;
8:
9:  main()
10: {
11:     /* Prikazi instrukcije. */
12:
13:     puts("Enter text a line at a time, then press Enter.");
14:     puts("Enter a blank line when done.");
15:
16:    /* Peetlja sve dok unos ne bude prazna linija (blank line). */
17:
18:     while ( *(ptr = gets(input)) != NULL)
19:         printf("You entered %s\n", input);
20:
21:     puts("Thank you and good-bye\n");
22:
23:     return 0;
24: }
```

```
Enter text a line at a time, then press Enter.
Enter a blank line when done.
First string
You entered First string
Two
You entered Two
Bradley L. Jones
You entered Bradley L. Jones
Thank you and good-bye
```

ANALIZA: Sad možete vidjeti kako program radi. Ako unesete praznu liniju (tj., ako jednostavno pritisnete **<enter>**) u odgovoru na liniju **18**, string (koji sadrži **0** karaktera) je još uvijek smješten sa **null** karakterom na kraju. Zato što string ima dužinu **0**, **null** karakter je smješten na prvu poziciju. Ovo je pozicija na koju pokazuje povratna vrijednost **gets()**-a, tako da ako testirate ovu poziciju i naiđete na **null** karakter, znate da je unešena prazna linija.

Listing 10.6 obavlja ovaj test sa **while** iskazom u liniji **18**. Ovaj iskaz je malo koplikovan, tako da pažljivo pogledajte detalje po redu:

1. **gets()** funkcija prihvata unos sa tastature dok ne dostigne karakter nova-linija.
2. Unešeni string, minus nova-linija i sa **null** karakterom, je smješteno u memorijskoj lokaciji na koju pokazuje unos.
3. Adresa stringa (ista vrijednost kao i unos) je vraćena pointeru **ptr**.
4. Iskaz pridruživanja je izraz koji procjenjuje na vrijednost varijable na lijevoj strani operatora pridruživanja. Tako da, cijeli izraz **ptr = gets(input)** se procjenjuje na prijednost **ptr**-a. Zatvarajući izraz u zagrade i pišući indirektni operator (*****) ispred njih, vi dobavljate vrijednost smještenu na pokazujuću adresi (poited-to address). Ovo je, naravno, prvi karakter unešenog string-a.

5. null je simbolna konstanta koja je definisana u file-u zaglavlja **STDIO.H**. Ona ima vrijednost **null** karaktera (**0**).
6. Ako prvi karakter unošećeg string-a nije **null** karakter (ako nije unešena prazna linija), upoređujući operator vraća **true**, i izvršava se **while** petlja. Ako je prvi karakter **null** karakter (ako je unešena prazna linija), upoređujući operator vraća **false**, i **while** petlja se prekida (terminira).

Kada koristite **gets()** ili bilo koju drugu funkciju koja smješta podatke koristeći pointer, uvjerite se da pointer pokazuje na alociran prostor. Lako je napraviti grešku kako slijedi:

```
char *ptr;
gets(ptr);
```

Pointer **ptr** je deklarisan ali ne i inicijaliziran. On pokazuje negdje, ali ne znate gdje. **gets()** funkcija ovo ne zna, tako da jednostavno ide naprijed i smješta uneseni string na adresu dobavljenu od strane **ptr**-a. String može prebrisati (overwrite) nešto bitno, kao što je programski kood ili operativni sistem. Kompajler na hvata ovakve greške, tako da vi, kao programer, morate biti vigilat-ni (op.a.).

→→ gets() Funkcija

```
#include <stdio.h>
char *gets(char *str);
```

gets() funkcija uzima string, **str**, sa standardnog ulaza, obično tastature. String se sastoji od bilo kojih unošenih karaktera dok se ne pročita karakter novi-red (newline). Tada, kada se to desi, **null** se dedaje na kraj string-a.
Onda **gets()** funkcija vraća pointer na string koji je upravo pročitan. Ako postoji problem za dobavljanje string-a, **gets()** vraća null.

Primjer

```
/* gets() example */
#include <stdio.h>
char line[256];
void main()
{
   printf( "Enter a string:\n");
   gets( line );
   printf( "\nYou entered the following string:\n" );
   printf( "%s\n", line );
}
```

→→→ Unošenje string-a korištenjem scanf() funkcije

Vidjeli ste Dana 7 da **scanf()** bibliotečne funkcija prihvataju numeričke unošene podatke sa tastature. Ove funkcije mogu takođe unositi stringove.
Zapamtite da **scanf()** koristi format string koji nam govori kako da se čita unos.
Da se čita string, uključite specifikator **%s** u **scanf()**-ov format string. Kao i **gets()**, **scanf()** prosljeđuje pointer na string-ovu smještajnu lokaciju.

Kako scanf() odlučuje gdje string počinje a gdje završava???
Početak je prvi nebijeloprostorni (nonwhitespace) karakter na koji se naleti (encountered). Kraj može biti specificiran na jedan od dva načina:
Ako koristite **%s** u format string-u, string ide do (ali ne uključuje) sljedećeg bijelog karaktera (space, tab, ili novalinija). Ako koristite **%ns** (gdje je **n** integer konstanta koja specificira (označava) širinu polja), **scanf()** unosi sljedećih **n** karaktera ili do sljedećeg bijelog karaktera, šta god da naiđe prvo.

Možete čitati višestruke string-ove sa **scanf()**-om tako što uključite više od jednog **%s**, u format stringu. Za svaki **%s** u format string-u, **scanf()** koristi sljedeća pravila da pronađe traženi broj stringova pri unosu.

Na primjer:

```
scanf("%s%s%s", s1, s2, s3);
```

Ako u odgovoru na ovaj iskaz vi unesete **Januar Februar Mart**; **Januar** je pridružen string-u **s1**, **Februar** string-u **s2**, i **Mart s3**.

Šta je sa korištenjem operatora širine polja? Ako izvršite iskaz:

```
scanf("%3s%3s%3s", s1, s2, s3);
```

I kao odgovor vi unesete **Septembar**; **Sep** je pridružen **s1**, **tem** je pridružen **s2**, i **bar** je pridružen **s3**.

Šta ako unesete manje ili više string-ova nego što scanf() funkcija očekuje???
Ako unesete manje stringova, **scanf()** nastavlja da traži nedostajuće string-ove, i program ne nastavlja sve dok se oni ne unesu. Na primjer, u odgovoru na ovaj iskaz:

```
scanf("%s%s%s", s1, s2, s3);
```

vi unesete **Januar Februar**, program sjedne i čeka na treći string koji je specificiran (naveden) u **scanf()**-ovom format string-u.
Ako unesete više string-ova nego što je zahtjevano, nepodudarajući string-ovi ostaju u toku (pending) (čekaju u tastaturinom buffer-u) i čitani su od strane bilo kojeg kasnijeg **scanf()**-ovog ili nekog drugog unošenog iskaza. Na primjer, u odgovoru na iskaze:

```
scanf("%s%s", s1, s2);
scanf("%s", s3);
```

vi unesete **Januar Ferbuar Mart**, rezultat je da je **Januar** pridružen string-u **s1**, **Februar** je pridružen **s2**, i **Mart** je pridružen **s3**.

scanf() funkcija ima povratnu vrijednost, **integer** vrijednost koja je jednaka broju predmeta koji su unešeni uspješno. Vraćena vrijednost je često ignorisana. Kada samo čitate text, **gets()** funkcija je često poželjnija od **scanf()**-a. Najbolje je da koristite **scanf()** funkciju kada čitate u kombinaciji text i numeričke vrijednosti (podatke).
Ovo je ilustrovano u Listing 10.7.

& Zapamtite iz Dana 7 da morate koristiti **adresa-od** (address-of) **operator** (**&**) kada unosite numeričke vrijednosti sa **scanf()**-om.

Listing 10.7. Unošenje numeričkih i textualnih podataka sa **scanf()**-om.

```
1:  /* Demonstrira korištenje scanf()za unos numerickih i text podataka. */
2:
3:  #include <stdio.h>
4:
5:  char lname[81], fname[81];
6:  int count, id_num;
7:
8:  main()
9:  {
10:     /* Prompt the user. */
11:
12:     puts("Enter last name, first name, ID number separated");
13:     puts("by spaces, then press Enter.");
14:
15:     /* Input the three data items. */
16:
17:     count = scanf("%s%s%d", lname, fname, &id_num);
18:
19:     /* Display the data. */
20:
21:     printf("%d items entered: %s %s %d \n", count, fname, lname, id_num);
22:
```

```
23:     return 0;
24: }
            Enter last name, first name, ID number separated
            by spaces, then press Enter.
            Jones Bradley 12345
            3 items entered: Bradley Jones 12345
```

ANALIZA: Zapamtite da **scanf()** zahtjeva adresu od varijable za parametre. U Listing-u 10.7, **lname** i **fname** su pointeri (tj. adrese), tako da oni ne trebaju **adresa-od** (address-of) operator (**&**). Suprotno, **id_num** je regularno ime varijable, tako da joj treba **&** kada prosljeđuje **scanf()**-u u liniji **17**.

Neki programeri osjećaju da je unos podataka sa **scanf()**-om sklon greškama. Oni preferiraju da unose sve podatke, numeričke i string-ove, koristeći **gets()**, i onda program odvaja brojeve i konvertuje ih u numeričke varijable. Takve tehnike su van dometa ove knjige, ali bi bile dobre programerske vježbe. Za taj zadatak, trebate funkcije za manipulaciju string-ovima koje su pokrivene Dana 17.

SAŽETAK

Ovo poglavlje je pokrilo **C**-ove **char** tipove podataka. Jedna upotreba varijabli tipa **char** je da smješta individualne karaktere. Vidjeli ste da se karakteri smještaju kao numeričke vrijednosti: **ASCII** kood pridružuje numerički kood svakom karakteru. Tako da, možete koristiti tip **char** da smještate male **integer** vrijednosti takođe. Obadva **signed** i **unsigned** tipa **char** su dozvoljena.

String je sekvenca karaktera koja završava sa **null** karakterom. String-ovi mogu biti korišteni za text-ualne podatke. **C** smješta string-ove u nizove tipa char. Da smjestite sting dužine **n**, trebate niz tipa **char** sa **n+1** elemenata.

Možete koristiti funkcije alokacije memorije kao što su **malloc()**, da dodate dinamiku vašim programima. Koristeći **malloc()**, možete alocirati tačnu količinu memorije za vaš program. Bez takvih funkcija, trebali biste nagađati količinu smještajne memorije koju vaš program zahtjeva (treba). Vaša pretpostavka će vjerovatno biti velika, tako da alocirate više memorije nego što vam je potrebno.

P&O

P Koja je razlika između string i niiza karaktera???
O String je definisan kao sekvenca karaktera koja završava sa **null** karakterom. Niz je sekvenca karaktera. Tako da je string, null-terminirajući niz karaktera.
Ako definišete niz tipa **char**, stvarni alocirani smještajni prostor za niz je specificirana veličina, ne veličina minus **1**. Ograničeni ste na tu veličinu, ne možete smještati veće stringove. Evo ga i primjer:

```
char state[10]="Minneapolis";  /* Wrong! String longer than array. */
char state2[10]="MN";          /* OK, but wastes space because */
                               /* string is shorter than array. */
```

Ako, s druge strane, definišete pointer na tip **char**, ova ograničenja se ne odnose. Varijabla je smještajni prostor samo za pointer. Stvarni string-ovi su smješteni negdje drugo u memoriji (ali ne treba da brinete gdje). Nema ograničenja dužine ili bačenog prostora. Stvarni string je smješten negdje drugo. Pointer može pokazivati na string bilo koje dužine.

P Šta se desi ako stavimo string u karakterni niz koji je veći od niiza???
O Ovo može prouzrokovati teška-za-naći grešku. Ovo možete uraditi u **C**-u, ali sve što je smješteno u memoriji direktno nakon karakternog niza se prepisuje (overwritten). Ovo može biti memorijska oblast koja se ne koristi, neki drugi podaci, ili neke vitalne sistemske informacije. Vaši rezultati zavise od toga šta prepisujete. Često, ništa se ne desi neko vrijeme. NE želite ovo da radite.

Vježbe:

1. Napišite liniju kooda koja deklariše varijablu tipa **char** s imenom letter, i inicijalizirajte je sa karakterom **$**.
2. Napišite liniju kooda koja deklariše niz tipa **char**, i inicijalizirajte ga na string **"Pointeri su zabavni!"**. Napravite niz taman toliki koliko mu je potrebno da drži string.
3. Napišite liniju kooda koja alocira smještanje za string **"Pointeri su zabavni!"**, kao u vježbi 2, ali bez korištenja nizova.
4. Napišite kood koji alocira prostor za **80**-karakterni string i onda unosi string sa tastature i smješta ga u alocirani prostor.
5. Napišite funkciju koja kopira jedan niz karaktera u drugi. (Hint: Uradite ovo kao programi koje ste pisali Dana 9).
6. Napišite funkciju koja prihvata dva string-a. Izbrojite broj karaktera u svakom, i vratite pointer na duži string.
7. SAMI: Napišite funkciju koja prihvata dva stringa. Koristite **malloc()** funkciju da alocirate dovoljno memorije da drži dva stringa nakon što se spoje (linkuju). Vratite pointer na ovaj novi string.
Na primjer, ako prosljedim **"Hello" i "World!"**, funkcija vraća pointer na **"Hello World!"**. Imajući da je spojena vrijednost treći string je najlakše. (možda možete iskoristiti odgovore iz vježbi 5 i 6).
8. BUG BUSTER: Da li nešto nije u redu sa sljedećim?
```
char a_string[10] = "This is a string";
```
9. BUG BUSTER: Da li nešto nije u redu sa sljedećim?
```
char *quote[100] = { "Smile, Friday is almost here!" };
```
10. BUG BUSTER: Da li nešto nije u redu sa sljedećim?
```
char *string1;
char *string2 = "Second";
string1 = string2;
```
11. BUG BUSTER: Da li nešto nije u redu sa sljedećim?
```
char string1[];
char string2[] = "Second";
string1 = string2;
```
12. SAMI: Koristeći **ASCII** chart, napišite program koji printa kutiju (box) na-ekranu (on-screen) koristeći double-line (duple-linijske) karaktere.

LEKCIJA 11: → Structures ← → Jednostavne strukture ←

Struktura je kolekcija (skup) jedne ili više varijabli, grupisanih pod jednim imenom radi lakše manipulacije. Varijable u strukturi, za razliku od onih u niizu, mogu biti varijable različitih tipova.
Struktura može sadržavati bilo koji od **C**-ovih tipova podataka, uključujući nizove i druge strukture. Svaka varijabla unutar strukture se naziva **član** (member) strukture. Sljedi jednostavan primjer:
Trebali biste početi sa jednostavnim strukturama. Primjetite da **C** jezik ne pravi razliku između jednostavne i komplexne strukture, ali je lakše objasniti strukture na ovaj način.

→→→ Definisanje i deklaracija struktura

Ako pišete grafički program, vaš kood treba da radi sa koordinatama tačaka na ekranu. Koordinate ekrana su pisane kao: **x** vrijednost, data za horizontalan položaj, i **y** vrijednost, data ze vertikalni položaj. Možete definisati strukturu s imenom (nazvanu) **coord** koja sadrži i **x** i **y** vrijednosti ekranskih lokacija kako slijedi:

```
struct coord {
    int x;
    int y;
};
```

Ključna riječ **struct**, koja identifikuje početak definicije strukture, mora biti slijeđena odmah sa imenom strukture, ili tag-om (koji prati ista pravila kao i imena drugih **C** varijabli).
Unutar zagrada, koje slijede nakon imena strukture, je lista strukturinih članskih varijabli. Varijabli morate dati tip i ime za svaki član.

Prethodni iskaz definiše tip strukturu nazvanu **coord** koja sadrži dvije integer varijable; **x** i **y**. One, ipak, ne stvaraju stvarno instance strukture **coord**. U drugim riječima, one NE deklarišu (smještaju prostor sa strane za) nikakve strukture.

→→ Postoje dva načina da se deklariše struktura:
Jedan je da nakon definicije strukture slijedi lista sa jednom ili više imena varijabli, kako je urađeno ovdje:

```
struct coord {
    int x;
    int y;
} first, second;
```

Ovi iskazi definišu tip strukture **coord** i deklarišu dvije strukture; **first** i **second**, tipa **coord**.
first i **second** su instance od tipa **coord**;
first sadrži dva integer člana nazvani **x** i **y**, a također i **second**.

Ovaj metod deklaracije strukture kombinuje deklaraciju sa definicijom.

Drugi metod je da se deklarišu strukturine varijable na drugačijoj lokaciji, u vašem izvornom koodu, u odnosu na definiciju.
Sljedeći iskazi također deklarišu dvije instance tipa **coord**:

```
struct coord {
    int x;
    int y;
};
/* Additional code may go here */
struct coord first, second;
```

→→ Pristupanje članovima struktura

Individualni članovi strukture mogu biti korišteni kao i ostale varijable istog tipa.
Članovima strukture se pristupa koristeći strukturno članski operator (**.**), takođe nazvan i **tačka-operator**, između imena strukture i imena člana. Tako da se struktura s imenom **first** pojavi na lokaciji na ekranu koji ima koordinate **x=50**, **y=100**, napisali biste:

```
first.x = 50;
first.y = 100;
```

Da prikažete lokacije ekrana koje su smještene u strukturi **second**, napisali biste:

```
printf("%d,%d", second.x, second.y);
```

Do ove tačke, možda se pitate koje su prednosti koristeći strukture u odnosu na individualne varijable? Jedna bitna prednost je da možete kopirati informacije između struktura istog tipa sa jednostavnim jednačinskim iskazima. Nadovezujući se na prethodni primjer, iskaz:

```
first = second;
```

je ekvivalentan sa ovim iskazom:

```
first.x = second.x;
first.y = second.y;
```

Kada vaš program koristi komplexne strukture sa puno članova, ova notacija vam može uštediti dosta vremena. Druge prednosti struktura će postati očite kako budete učili naprednije tehnike.

Uopšte, naći ćete da su strukture korisne kad god informacija različitih tipova varijabli treba da se tretira kao grupa. Na primjer, u bazi podataka mailing list-e, svaki unos može biti struktura, i svaki komad informacije (ime, adresa, grad, itd.) može biti član strukture.

→→→ struct ←←← Ključna riječ

```
struct tag {
    structure_member(s);
    /* dodatni iskazi mogu ici ovdje */
} instance;
```

→ struct ključna riječ se koristi da deklariše strukturu.←

Struktura je skup jedne ili više varijabli (structure_members) koje su grupisane pod jednim imenom radi lakše manipulacije.
Varijable ne moraju biti istog tipa varijable, niti moraju biti jednostavne varijable.
Strukture takođe mogu čuvati nizove, pointere i druge strukture.

Ključna riječ **struct** označava početak definicije (definisanja) strukture. Nakon **struct** ključne riječi slijedi tag → dato ime strukture. Nakon tag-a slijede članovi strukture, zatvoreni u zagradama.
Instanca → stvarna deklaracija strukture, takođe može biti definisana. Ako definišete strukturu bez instance, onda je ona samo tamplate koji može biti kasnije korišten u programu da deklariše strukture. Evo primjera template-a:

```
struct tag {
    structure_member(s);
    /* additional statements may go here */
};
```

Da koristite template, koristite sljedeći format:

```
struct tag instance;
```

Da koristite ovaj format, morali ste prethodno deklarisati strukturu sa datim tag-om (imenom).

Primjer 1

```
/* Deklarisi strukturin template nazvan SSN */
struct SSN {
    int first_three;
    char dash1;
    int second_two;
    char dash2;
    int last_four;
}
/* Use the structure template */
struct SSN customer_ssn;
```

Primjer 2

```
/* Deklarisi strukturu i instancu zajedno */
struct date {
    char month[2];
    char day[2];
    char year[4];
} current_date;
```

Primjer 3

```
/* Deklarisi i inicijaliziraj strukturu  */
struct time {
    int hours;
    int minutes;
    int seconds;
} time_of_birth = { 8, 45, 0 };
```

→→ Više – Komplexne strukture

Sad kad ste upoznati sa jednostavnim strukturama, možete preći na interesantnije i komplexnije tipove struktura. Ovo su strukture koje sadrže druge strukture kao članove i strukture koje sadrže niizove kao članove.

→ Strukture koje sadrže strukture

Kao što je spomenuto ranije, **C**-ove strukture mogu sadržavati bilo koje **C**-ove tipove podataka. Npr., struktura može sadržavati drugu strukturu. Prethodni primjer može biti proširen da ilustruje ovo.

Pretpostavimo da vaš grafički program treba da radi sa kvadratima (geometrijskim). Kvadrat se može definisati sa koordinatama dva dijagonalno suprotna ugla. Već ste vidjeli kako da definišete strukturu koja može držati dvije koordinate koje su potrebne za jednu tačku. Trebate dvije takve strukture da definišete kvadrat. Možete definisati strukturu kako slijedi (pretpostavljajući, naravno, da ste prethodno definisali strukturu tipa **coord**):

```
struct kvadrat {
    struct coord topleft;
    struct coord bottomrt;
};
```

Ovaj iskaz definiše strukturu tipa **kvadrat** koja sadrži dvije strukture tipa **coord**. Ove dvije strukture tipa **coord** su nazvane **topleft** i **bottomrt**.
Sljedeći iskaz definiše samo strukturu tipa **kvadrat**. Da deklarišete strukturu, morate onda uključiti iskaz kao:

```
struct kvadrat mybox;
```

Mogli ste kombinirati definiciju i deklaraciju, kao što ste maloprije za tip **coord**:

```
struct kvadrat {
    struct coord topleft;
    struct coord bottomrt;
} mybox;
```

Da pristupite stvarnim podatkovnim lokacijama (članovima tipa **int**), morate primjeniti članski operator (**.**) dva puta. Tako da, izraz:

```
mybox.topleft.x
```

se odnosi na član **x**, **topleft** člana strukture tipa **kvadrat** nazvane **mybox**. Da definišete kvadrat sa koordinatama (**0,10**),(**100,200**), napisali biste:

```
mybox.topleft.x = 0;
mybox.topleft.y = 10;
mybox.bottomrt.x = 100;
mybox.bottomrt.y = 200;
```

Listing 11.1. Demonstracija struktura koje sadrže druge strukture.

```
1:  /* Demonstrira strukture koje sadrze strukture. */
2:
3:  /* Prima unos za tchoshkovne koorsinate kvadrata i
4:     racuna oblast. Pretpostavljano da je y koordinata
5:     gornjeg-lijevog ugla veca od y koordinate
6:     donjeg-desnog ugla, da je x koordinata
7:     donjeg-desnog ugla veca od x koordinate
8:     gornjeg-lijevog ugla i da su sve koordinate pozitivne. */
9:
10: #include <stdio.h>
11:
12: int length, width;
13: long area;
14:
15: struct coord{
16:     int x;
17:     int y;
18: };
19:
20: struct kvadrat{
21:     struct coord topleft;
22:     struct coord bottomrt;
23: } mybox;
24:
25: main()
26: {
27:     /* Unesi koordinate */
28:
29:     printf("\nEnter the top left x coordinate: ");
30:     scanf("%d", &mybox.topleft.x);
31:
32:     printf("\nEnter the top left y coordinate: ");
33:     scanf("%d", &mybox.topleft.y);
34:
35:     printf("\nEnter the bottom right x coordinate: ");
36:     scanf("%d", &mybox.bottomrt.x);
37:
38:     printf("\nEnter the bottom right y coordinate: ");
39:     scanf("%d", &mybox.bottomrt.y);
40:
41:     /* Izracunaj duzinu i shirinu */
42:
43:     width = mybox.bottomrt.x - mybox.topleft.x;
44:     length = mybox.bottomrt.y - mybox.topleft.y;
45:
46:     /* Izracunaj i prikazi oblast */
47:
48:     area = width * length;
49:     printf("\nThe area is %ld units.\n", area);
50:
```

```
51:     return 0;
52: }
```

```
Enter the top left x coordinate: 1
Enter the top left y coordinate: 1
Enter the bottom right x coordinate: 10
Enter the bottom right y coordinate: 10
The area is 81 units.
```

ANALIZA: struktura **coord** je definisana u linijama **15** do **18** sa svoja dva člana, **x** i **y**. Linije **20** do **23** deklarišu i definišu instancu, nazvanu **mybox**, strukture **kvadrat**. Dva člana strukture **kvadrat** su **topleft** i **bottomrt**, obadvije strukture tipa **coord**.

mybox ima dva člana, a svaki od njih ima po još dva člana → ukupno četiri člana da se popune.

Kada koristite vrijednosti **x**-a i **y**-a, morate uključiti ime instance strukture. Zato što su **x** i **y** u strukturi unutar strukture, morate koristiti imena instaci za obadvije strukture → **mybox.bottomrt.x**, **mybox.bottort.y**, **mybox.topleft.x** i **mybox.topleeft.y** → u proračunima.

Postoji limit iza kojeg gnijezđenje postaje neproduktivno. Rijetko postoje više od **tri** nivoa gnijezđenja u bilo kojem **C** programu.

→→ Strukture koje sadrže niizove

Možete definisati strukturu koja sadrži jedan ili više nizova kao članove. Niiz može biti bilo kojeg **C**-ovog tipa podataka (**int**, **char**, itd.). Npr., iskazi:

```
struct data{
    int  x[4];
    char y[10];
};
```

definišu strukturu tipa **data**, koja sadrži četvoro-elementni **integer** niz kao član, nazvan **x** i **10-elementni** karakterni niz kao član nazvan **y**. Onda možete deklarisati strukturu nazvanu **record** tipa **data** kako slijedi:

```
struct data record;
```

Elementi niza **x** zauzimaju dva puta više prostora od elemenata niza **y**.

Pristupate elementima niza koji su članovi strukture, kombinujući članski operator i index niiza:

```
record.x[2] = 100;
record.y[1] = `x';
```

izraz → **`record.y`** ← je pointer na prvi element niza **y[]** u strukturi **record**. Tako da, možete printati sadržaj **y[]**-a na ekranu koristeći iskaz:

```
puts(record.y);
```

Sad pogledajte drugi primjer. Listing **11.2** koristi strukturu koja sadrži varijablu tipa **float** i dva niiza tipa **char**.

Listing 11.2. Struktura koja sadrži niz-ne članove.

```
1:  /* Demonstrira strukturu koja ima niz-ne clanove. */
2:
3:  #include <stdio.h>
4:
5:  /* Definisite i deklarisite strukturu da drzi podatke. */
6:  /* Ona sadrzi jednu float varijablu i dva char niza. */
7:
8:  struct data{
9:      float amount;
```

```
10:     char fname[30];
11:     char lname[30];
12: } rec;
13:
14: main()
15: {
16:     /* Unesi podatke sa tastature. */
17:
18:     printf("Enter the donor's first and last names,\n");
19:     printf("separated by a space: ");
20:     scanf("%s %s", rec.fname, rec.lname);
21:
22:     printf("\nEnter the donation amount: ");
23:     scanf("%f", &rec.amount);
24:
25:     /* Prikazi informacije. */
26:     /* Napomena: %.2f specificira floating-point vrijednost */
27:     /* koja ce se prikazati sa dvije cifre desno, nakon */
28:     /* decimalne tachke. */
29:
30:     /* prikazi podatke na ekran. */
31:
32:     printf("\nDonor %s %s gave $%.2f.\n", rec.fname, rec.lname,
33:             rec.amount);
34:
35:     return 0;
36: }
    Enter the donor's first and last names,
    separated by a space: Bradley Jones
    Enter the donation amount: 1000.00
    Donor Bradley Jones gave $1000.00.
```

ANALIZA: Ovaj program uključuje strukture koje sadrže članove nizove nazvane **fname[30]** i **lname[30]**. Obadva su niizovi karaktera koje drže osobino prvo ime i prezime, respektivno. Struktura, deklarisana u linijama od **8** do **12** se zove **data**. Ona sadrži **fname** i **lname** karakterne nizove sa varijablom tipa **float** nazvanu **amount**. Stuktura je idealna za čuvanje imena osobe (u dva dijela, prvo ime i dugo prezime) i vrijednosti, kao što je količina koju je osoba donirala dobrotvornoj organizaciji (charitable organization).

Instanca niza, nazvana **rec**, takođe je deklarisana u liniji **12**. Ostatak programa koristi rec da dobije vrijednosti od korisnika (linije **18** do **23**) i onda i printa (linije **32** do **33**).

→→→ Niizovi struktura

Ako možete imati nizove koji sadrže strukture, možete li takođe imati i nizove struktura???
Naravno da možete…
U stvari, nizovi struktura su veoma snažni (jaki) programerski alat. Evo kako rade:
Vidjeli ste kako se definicija strukture može prekrojiti da odgovara podacima s kojim vaš program treba da radi. Obično, program treba da radi sa više od jednom instancom podataka. Npr., u programu da se održava lista telefonskih brojeva, možete definisati strukturu da drži ime svake osobe i broj.

```
struct entry{
    char fname[10];
    char lname[12];
    char phone[8];
};
```

Telefonska lista moža držati mnogo unosa, ipak, tako da jedna instanca u cijeloj strukturi i nije od velike pomoći. Ono što trebate je niz struktura od tipa **entry**. Nakon što je struktura definisana, možete deklarisati niz ovako:

```
struct entry list[1000];
```

Ovaj iskaz deklariše niz nazvan **list** koji sadrži **1000** elemenata. Svaki element je struktura od tipa **entry** i identifikovana je pomoću indexa kao i odrugi tipovi elemenata niza. Svaki od ovih struktura ima tri elementa, od kojih je svaki niz tipa **char**.

Kada ste deklarisali niz struktura, možete manipulisati s podacima na puno načina. Npr., da pridružite podatke u jednom elementu niza, drugom elementu niza, napisali biste:

```
list[1] = list[5];
```

Ovaj iskaz pridružuje svakom članu strukture **list[1]** vrijednosti koje su sadržane u korespondirajućim članovima of **list[]**. Takođe možete premiještati podatke između pojedinačnih članova strukture.
Iskaz:

```
strcpy(list[1].phone, list[5].phone);
```

kopira string **list[5].phone** u **list[1].phone**.

→ **strcpy()** bibliotečna funkcija kopira jedan string u drugi string. ←

Ako želite, takođe možete premiještati podatke pojedinačnih elemenata nizova članova strkture:

```
list[5].phone[1] = list[2].phone[3];
```

Ovaj iskaz premiješta drugi karakter od **list[5]**-ovog telefonskog broja na četvrtu poziciju u **list[2]**-ovog telefonskog broja. (NE zaboravite da indexi počinju sa offset-om **0**).

Listing **11.3** demonstrira upotrebu nizova struktura. Čak i više, demonstrira nizove struktura koje sadrže nizove kao članove.

Listing 11.3. Nizovi struktura (Arrays of structures).

```
1:  /* Demonstrira koristenje nizova struktura. */
2:
3:  #include <stdio.h>
4:
5:  /* Definisi strukturu da drzi unose. */
6:
7:  struct entry {
8:      char fname[20];
9:      char lname[20];
10:     char phone[10];
11: };
12:
13: /* Deklarisi niz struktura. */
14:
15: struct entry list[4];
16:
17: int i;
18:
19: main()
20: {
21:
22:     /* Peetljaj do unosa podataka za četiri osobe. */
23:
24:     for (i = 0; i < 4; i++)
25:     {
26:         printf("\nEnter first name: ");
27:         scanf("%s", list[i].fname);
28:         printf("Enter last name: ");
29:         scanf("%s", list[i].lname);
30:         printf("Enter phone in 123-4567 format: ");
31:         scanf("%s", list[i].phone);
32:     }
33:
34:     /* Printaj dvije prazne linije. */
35:
```

```
36:     printf("\n\n");
37:
38:     /* Peetljaj da prikazes podatke. */
39:
40:     for (i = 0; i < 4; i++)
41:     {
42:         printf("Name: %s %s", list[i].fname, list[i].lname);
43:         printf("\t\tPhone: %s\n", list[i].phone);
44:     }
45:
46:     return 0;
47: }
    Enter first name: Bradley
    Enter last name: Jones
    Enter phone in 123-4567 format: 555-1212
    Enter first name: Peter
    Enter last name: Aitken
    Enter phone in 123-4567 format: 555-3434
    Enter first name: Melissa
    Enter last name: Jones
    Enter phone in 123-4567 format: 555-1212
    Enter first name: Deanna
    Enter last name: Townsend
    Enter phone in 123-4567 format: 555-1234
    Name: Bradley Jones             Phone: 555-1212
    Name: Peter Aitken              Phone: 555-3434
    Name: Melissa Jones             Phone: 555-1212
    Name: Deanna Townsend           Phone: 555-1234
```

ANALIZA: Linije **7** do **11** definišu template strukturu koja se zove **entry**, koja sadrži tri karakterna niiza: **fname**, **lname** i **phone**.
Linija **15** koristi template da definiše niz od četiri **entry** strukturine varijable nazvane **list**.
Linija **17** definiše varijablu tipa **int** koja će biti korištena kao **brojač** kroz program.
main() počinje u liniji **19**. Prva funkcija **main()**-a je da obavi petlju četiri puta sa **for** iskazom. Ova petlja je korištena da dobije informacije za niz struktura. Ovo može biti viđeno u linijama **24** do **32**.
Primjetite da je **list** korišten sa indexom na isti način kako su indexovane i varijable niza Dana 8.

Linija **36** obezbjeđuje prekid unosa prije početka izlaza. Ona printa dvije nove linije.
Linije **40** do **44** prikazuju podatke koje je korisnik unio u prethodnim koracima. Vrijednosti nizova struktura se printaju sa indexovanim imenima nizova, nakon kojih slijedi članski operator (**.**) i ime člana strukture.

NE zaboravite ime strukturine instance i članski operator (**.**) kada koristite strukturine članove.
NE miješajte strukturin tag (oznaku (ime)) sa njenim instancama!!! Tag se koristi da se deklariše strukturin template, ili format. Instanca je varijabla koja je deklarisana koristeći tag.
NE zaboravite **struct** ključnu riječ kada deklarišete instance od prethodno definisane strukture.
DEKLARIŠITE strukturine instance sa istim pravilima vidljivosti (scope) kao i ostale varijable.

→→→ Inicijaliziranje struktura

Kao i ostali tipovi **C**-ovih varijabli, strukture mogu biti inicijalizirane kada se deklarišu. Ova procedura je slična kao i inicijalizacija nizova.
Deklaraciju strukture slijedi znak jednako i lista inicijalizacijskih vrijednosti odvojenih zarezom i zatvoreni u zagradama. Na primjer, pogledajte sljedeće iskaze:

```
1: struct sale {
2:     char customer[20];
3:     char item[20];
4:     float amount;
5: } mysale = { "Acme Industries",
6:              "Left-handed widget",
7:               1000.00
8:           };
```

Kada se ovi iskazi izvrše, oni obavljaju sljedeće akcije:

1. Definiše tip strukture s imenom **sale** (linije **1** do **5**).
2. Declariše instancu tipa strukture **sale** s imenom **mysale** (linija **5**).
3. Inicijalizira član strukture **mysale.customer** na string **"Acme Industries"** (linija **5**).
4. Inicijalizira član strukture **mysale.item** na string **"Left-handed widget"** (linija **6**).
5. Inicijalizira član strukture **mysale.amount** na vrijednost **1000.00** (linija **7**).

Za strukturu koja sadrži strukture kao članove, listaj inicijalizacijske vrijednosti po redu. One smještaju u članove strukture po redu u kojem su izlistani članovi u definiciji strukture.
Evo primjera koji se proširuje na prethodni:

```
1:  struct customer {
2:      char firm[20];
3:      char contact[25];
4:  }
5:
6:  struct sale {
7:     struct customer buyer;
8:     char item[20];
9:     float amount;
10: } mysale = { { "Acme Industries", "George Adams"},
11:                "Left-handed widget",
12:                 1000.00
13:            };
```

Ovi iskazi obavljaju sljedeće inicijalizacije:

1. Član strukture **mysale.buyer.firm** je inicijaliziran stringu **"Acme Industries"** (linija **10**).
2. Član strukture **mysale.buyer.contact** je inicijaliziran stringu **"George Adams"** (linija **10**).
3. Član strukture **mysale.item** je inicijaliziran stringu **"Left-handed widget"** (linija **11**).
4. Član strukture **mysale.amount** je inicijaliziran na iznos **1000.00** (linija **12**).

Također, možete inicijalizirati nizove struktura. Inicijalizacioni podaci koje vi obezbjedite se primjenjuju, po redu, strkturama u niizu.

Na primjer, da deklarišete niz struktura tipa **sale** i inicijalizirate prva dva elementa niza (tj., prve dvije strukture), možete napisati:

```
1:  struct customer {
2:      char firm[20];
3:      char contact[25];
4:  };
5:
6:  struct sale {
7:      struct customer buyer;
8:      char item[20];
9:      float amount;
10: };
11:
12:
13: struct sale y1990[100] = {
14:      { { "Acme Industries", "George Adams"},
15:          "Left-handed widget",
16:           1000.00
17:      }
18:      { { "Wilson & Co.", "Ed Wilson"},
19:          "Type 12 gizmo",
20:           290.00
21:      }
22: };
```

Evo šta se dešava u ovom koodu:

1. Član strkture **y1990[0].buyer.firm** je inicijaliziran na string **"Acme Industries"** (line **14**).
2. Član strkture **y1990[0].buyer.contact** je inicijaliziran na string **"George Adams"** (line **14**).
3. Član strkture **y1990[0].item** je inicijaliziran na string **"Left-handed widget"** (line **15**).

4. Član strkture **y1990[0].amount** je inicijaliziran na vrijednost **1000.00** (line **16**).
5. Član strkture **y1990[1].buyer.firm** je inicijaliziran na string **"Wilson & Co."** (line **18**).
6. Član strkture **y1990[1].buyer.contact** je inicijaliziran na string **"Ed Wilson"** (line **18**).
7. Član strkture **y1990[1].item** je inicijaliziran na string **"Type 12 gizmo"** (line **19**).
8. Član strkture **y1990[1].amount** je inicijaliziran na vrijednost **290.00** (line **20**).

→→→ Strukture i Pointeri

Možete korstiti pointere kao članove strukture, i također možete deklarisati pointere na strukture.

→→ Pointeri kao članovi strukture

Imate potpunu flexibilnost u korištenju pointera kao članove strukture. Pointer članovi su deklarisani na isti način kao i pointeri koji nisu članoci strukture → tj., koristeći indirektni operator (*).
Evo ga i primjerćić:

```
struct data {
    int *value;
    int *rate;
} first;
```

Ovi iskazi definišu i deklarišu strukturu čija su obadva člana pointeri na tip **int**. Kao i kod svih pointera, deklarisati ih nije dovoljno; morate, pridružujući im adresu varijable, inicijalizirati ih da pokazuju na nešto. Da su **cost** i **interest** bili deklarisani da budu varijable tipa **int**, mogli ste napisati:

```
first.value = &cost;
first.rate = &interest;
```

Sad kad su pointeri inicijalizirani, možete koristiti indirektni operator (*).

Izraz ***first.value** se procjenjuje na vrijednosti od **cost**, i izraz ***first.rate** se procjenjuje na vrijednost od **interest**.

Možda je najčešće korišteni tip pointera, kao član strukture, pointer na tip **char**.

Da se podsjetimo, možete deklarisati pointer na tip **char** i inicijalizirati ga da pokazuje na string ovako:

```
char *p_message;
p_message = "Teach Yourself C In 21 Days";
```

Istu stvar možete uraditi sa pointerima na tip **char** koji su članovi strukture:

```
struct msg {
    char *p1;
    char *p2;
} myptrs;
myptrs.p1 = "Teach Yourself C In 21 Days";
myptrs.p2 = "By SAMS Publishing";
```

Svaki pointer član strukture pokazuje na prvi byte stringa, smještenog negdje u memoriji.
Pointere, članove strukture, možete koristiti svugdje gdje možete koristiti i pointere.

Na primjer, da printate pokazuje-na string, napisali biste:

```
printf("%s %s", myptrs.p1, myptrs.p2);
```

→ Koja je razlika u korištenju nizova tipa char kao člana strukture i korištenja pointera na tip char???
Ovo su obadva metoda za "smještanje" stringa u strukturu, kako je pokazano ovdje u strukturi **msg**, koja koristi obadva metoda:

```
struct msg {
    char p1[30];
    char *p2;
} myptrs;
```

Sjetite se da je ime niza bez uglastih zagrada pointer na prvi element niza. Tako da, možete koristiti ova dva člana strukture na sličan način:

```
strcpy(myptrs.p1, "Teach Yourself C In 21 Days");
strcpy(myptrs.p2, "By SAMS Publishing");
/* additional code goes here */
puts(myptrs.p1);
puts(myptrs.p2);
```

Koja je razlika između ovih metoda???
U ovome: → ako definišete strukturu koja sadrži niz tipa **char**, svaka instanca tog tipa strukture sadrži smještajni prostor za niz specificirane (naznačene) veličine. Još i više, ograničeni ste na naznačenu veličinu; NE možete smještati veći string u strukturu. Evo ga i primjerćićić:

```
struct msg {
    char p1[10];
    char p2[10];
} myptrs;
...
strcpy(p1, "Minneapolis"); /* Wrong! String longer than array.*/
strcpy(p2, "MN");          /* OK, but wastes space because    */
                           /* string shorter than array.      */
```

Ako, u drugu ruku, definišete strukturu koja sadrži pointer na tip **char**, ova ograničenja se ne primjenjuju. Svaka instanca strukture sadrži smještajni prostor samo za pointer. Stvarni stringovi su smješteni negdje drugo u memoriji (ali vas briga gdje). Ne postoji ograničenje dužine ili zalud potrošenog prostora. Stvarni string nisu smješteni kao dio strukture. Svaki pointer u strukturi može pokazivati na string bilo koje dužine. Taj string postaje dio dtrukture, iako nije smješten u strukturi.

→→→ Pointeri na Strukture

C program može deklarisati i koristiti pointere na strukture, baš kao što može deklarisati pointere na bilo koji drugi smještajni tip podataka. Kao što ćete vidjeti kasnije u ovom poglavlju, pointeri na strukture se često koriste kada se prosljeđuje struktura kao argument funkciji.
Pointeri na strukture se takođe koriste na veoma snažan podatkovno smještajni metod (način), u narodu poznatiji kao **uvezane – liste** (ulančane liste (povezane – liste (**linked lists**))) (Dan 15).

Za sada, pogledajte kako vaš program može kreirati i koristiti pointere na strukture. Prvo, definišete strukturu:

```
struct part {
    int number;
    char name[10];
};
```

Sada deklarišete pointer na tip **part**:

```
struct part *p_part;
```

Zapamtite da , indirektni operator (*) u deklariaciji kaže da je **p_part** pointer na tip **part**, a ne instanca tipa **part**.

Može li se sad inicijalizirati pointer???
NE, jer iako je struktura **part** definisana, nikakve njene instance nisu deklarisane.
Zapamtite da, je njena deklaracija, ne definicija, ta koja ostavlja sa strane smještajni prostor u memoriji za podatkovni objekat (data object). Zato što pointer zahtjeva memorijsku adresu da pokazuje na nju, vi morate deklarisati instancu od tipa **part** prije nego što nešto može pokazivati na nju. Evo deklaracije:

```
struct part gizmo;
```

Sad možete obaviti inicijalizaciju pointera:

```
p_part = &gizmo;
```

Ovaj iskaz pridružuje adresu od **gizmo** na **p_part**.

Sad kad imate pointer na strukturu gizmo, kako možete da ga koristite???
Jedna metoda koristi indirektni operator (*****). Sjetite se iz Dana 9, da ako je **ptr** pointer na podatkovni objekat, onda se izraz ***prt** odnosi na objekat na koji pokazuje.
Primjenjujući ovo na naš primjer, znate da je **p_part** pointer na strukturu **gizmo**, tako da se ***p_part** odnosi na **gizmo**. Tada primjenjujute operator člana strukture (**.**) da pristupite pojedinačnim članovima **gizmo** –a. Da pridružite vrijednost **100** na **gizmo.number**, možete napisati:

```
(*p_part).number = 100;
```

***p_part** mora biti zatvore u zagradama, jer (**.**) operator ima veće značenje (precedence) nego (******) operator.

Drugi način da pristupite članovima strukture koristeći pointe na strukturu je da koristite **indirektni članski operator**, koji se sastoji od karaktera **->** (povlaka i znak veće). (Kada se koriste ovako, **C** ih tretira kao jedan operator, ne dva) dakle; **->**

Simbol se postavlja između imena pointera i imena člana.
Na primjer, da pristupite članu **member** od **gizmo**-a sa **p_part** pointerom, možete napisati:

```
p_part->number
```

Još jedan primjer, ako je **str** struktura, **p_str** pointer na **str**, i **memb** je član od **str**, možete pristupiti **str.memb** pišući:

```
p_str->memb
```

Tako da, postoje tri načina kako da pristupite člane strukture:

- Koristeći ime strukture
- Koristeći pointer na strukturu sa indirektnim operatorom (*****)
- Koristeći pointer na strukturu sa indirektnim članskim operatorom (**->**)

Ako je p_str pointer na strukturu str, sljedeći izrazi su ekvivalentni:

```
str.memb
(*p_str).memb
p_str->memb
```

→→→ Pointeri i nizovi struktura

Vidjeli ste da nizovi struktura mogu biti veoma snažan programerski alat, kao što mogu i pointeri na strukture. Možete kombinovati njih dvoje, koristeći pointere za pristup strukturama koje su elementi niiza.
Da ovo ilustrujemo, evo definicije strukture iz ranijeg primjera:

```
struct part {
    int number;
    char name[10];
};
```

Nakon što se **part** struktura definiše, možete deklarisati niiz od tipa **part** (array of type part):

```
struct part data[100];
```

Zatim možete deklarisati pointer na tip **part** i inicijalizirati ga da pokazuje na prvu strukturu u nizu podataka:

```
struct part *p_part;
p_part = &data[0];
```

Sjetite se da je ime niza bez uglastih zagrada, pointer na prvu element niza, tako da druga linija može da se napiše i ovako:

```
p_part = data;
```

Sad imete niiz struktura tipa **part** i pointer na prvi element niza (tj., prvu strukturu u niizu). Na primjer, možete printati sadržaj prvog elementa koristeći iskaz:

```
printf("%d %s", p_part->number, p_part->name);
```

A šta ako želite da printate sve elemente niza???
Vjerovatno bi morali koristiti **for** petlju, printajući jedan element niza sa svakom iteracijom petlje. Za pristup članovima koristeći pointer notaciju, morate promjeniti pointer **p_part** tako da sa svakom iteracijom petlje, on pokazuje na sljedeći element niza (tj., sljedeću strukturu u niizu).

Kako ovo izvesti???
Ovdje vam priskače u pomoć pointerova aritmetika. Unarni inkremetni operator (**++**) ima specijalno značenje kada se primjenjuje na pointer. On znači **"inkrementiraj pointer za veličinu objekta na koji pokazuje."**. Tj., ako imate pointer **ptr** koji pokazuje na podatkovni objekat tipa **obj**, iskaz

```
ptr++;
```

ima isti efekat kao i:

```
ptr += sizeof(obj);
```

Listing 11.4. Pristup sukcesivnim (zaredom) elementima niza inkrementirajući pointer.

```
1:  /* Demonstrira korake kroz niz od struktura */
2:  /* koristeci pointer notaciju. */
3:
4:  #include <stdio.h>
5:
6:  #define MAX 4
7:
8:  /* Definisi strukturu, onda deklarisite i inicijalizirajte */
9:  /* niz od cetiri strukture. */
10:
11: struct part {
12:     int number;
13:     char name[10];
14: } data[MAX] = {1, "Smith",
15:                2, "Jones",
16:                3, "Adams",
17:                4, "Wilson"
18:                };
19:
20: /* Deklarisite pointer na tip part, i brojacku varijablu. */
21:
22: struct part *p_part;
23: int count;
24:
25: main()
26: {
```

```
27:     /* Inicijalizirajte pointer na prvi element niza. */
28:
29:     p_part = data;
30:
31:     /* Peetljajte kroz niz, inkrementirajuci pointer */
32:     /* sa svakom iteracijom. */
33:
34:     for (count = 0; count < MAX; count++)
35:     {
36:         printf("At address %d: %d %s\n", p_part, p_part->number,
37:                 p_part->name);
38:         p_part++;
39:     }
40:
41:     return 0;
42: }
            At address 96: 1 Smith
            At address 108: 2 Jones
            At address 120: 3 Adams
            At address 132: 4 Wilson
```

ANALIZA: Prvo, u linijama **11** do **18**, ovaj program deklariše i inicijalizira niz od struktura, nazvan **data**. Pointer nazvan **p_part** je onda definisan u liniji **22** da se koristi da pokazuje na strukturu **data**. **main()** funkcija ima prvi zadatak da postavi pointer, **p_part**, da pokazuje na strukturu **part** koja je deklarisana. Onda se printaju svi elementi koristeći **for** petlju u linijama **34** do **39** koja inkrementira pointer na niz sa svakom iteracijom. Program takođe prikazuje adrese od svakog elementa.

Pogledajte adrese koje su prikazane. Precizne vrijednosti mogu biti različite na vašem sistemu, ali su u jednakim inkrementnim veličinama → u tačnoj veličini strukture **part** (većina sistema će imati inkrement od **12**). Ovo jasno ilustruje da inkrementirajući pointer, povećava se za veličinu (količinu) jednaku veličini podatkovnog objekta na koji pokazuje.

→ Prosljeđujući strukture kao argumente funkciji ←

Kao i ostali tipovi podataka, struktura može biti prosljeđena kao argument funkciji.
Listing **11.3** koristi funkciju da prikaže podatke na ekran, dok su u Listingu **11.2** korišćeni iskazi koji su dio **main()**-a.

Listing 11.5. Prosljeđivanje strukture kao funkcijskog argumenta.

```
1:  /* Demonstrira prosljedjenje strukture funkciji. */
2:
3:  #include <stdio.h>
4:
5:  /* Deklarisi i definisi strukturu da drzi podatke. */
6:
7:  struct data{
8:      float amount;
9:      char fname[30];
10:     char lname[30];
11: } rec;
12:
13: /* Funkcijski prototip. Funkcija nema povratnu vrijednost, */
14: /* i ona uzima strukturu tipa data kao svoj argument. */
15:
16: void print_rec(struct data x);
17:
18: main()
19: {
20:     /* Unesi podatke sa tastature. */
21:
22:     printf("Enter the donor's first and last names,\n");
23:     printf("separated by a space: ");
24:     scanf("%s %s", rec.fname, rec.lname);
25:
```

```
26:     printf("\nEnter the donation amount: ");
27:     scanf("%f", &rec.amount);
28:
29:     /* Pozovi prikaznu funkciju. */
30:     print_rec( rec );
31:
32:     return 0;
33: }
34: void print_rec(struct data x)
35: {
36:     printf("\nDonor %s %s gave $%.2f.\n", x.fname, x.lname,
37:             x.amount);
38: }
        Enter the donor's first and last names,
        separated by a space: Bradley Jones
        Enter the donation amount: 1000.00
        Donor Bradley Jones gave $1000.00.
```

ANALIZA: U liniji **16** je prototip funkcije za funkciju koja treba da prihvati strukturu. Kao što biste za svaki drugi tip podataka koji treba da bude prosljeđen, trebate da uključite (include) pravilne argumente. U ovom slučaju, to je struktura tipa **data**. Ovo je ponovljeno u file-u zaglavlja za funkciju u liniji **34**. Kada se pozove funkcija, vi samo trebate strukturi prosljediti ime instance → u ovom slučaju, **rec** (linija **30**).
Prosljeđivanje strukture funkciji se ne razlikuje puno od prosljeđivanja jednostavne varijable.

Možete takođe prosljediti strukturu funkciji, prosljeđujući adresu strukture (tj., pointer na strukturu). U stvari, u starijim verzijama **C**-a, to je bio jedini način da se prosljedi struktura kao argument. Sad to nije neophodno, ali možete vidjeti starije programe koji još koriste ovaj metod. Ako prosljedite pointer na strukturu kao argument, zapamtite da morate koristiti indirektni operator (**->**) da pristupite članu strukture u funkciji.

NE mješajte nizove sa strukturama!!!
ISKORISTITE deklaraciju pointera na strukturu → naročito kada koristite nizove struktura
NE zaboravite da kada inkrementirate pointer, on pomjera ekvivalent razmaka za veličinu podataka na koji pokazuje. U slučaju pointera na strukturu, ovo je veličina strukture.
KORISTITE indirektni članski operator (**->**) kad radite sa pointerima na strukturu.

→→→ Unije ←←←

Unije su slične strukturama.
Unija je deklarisana i korištena na isti način kao i struktura.
Unija se razlikuje od strukture samo da se jedan po jedan njen član može koristiti. Razlog za ovo je jednostavan. Svi članovi unije zauzimaju isto područje u memoriji. Ono se smještaju jedan na drugi.

→→→ Definisanje, deklarisanje i inicijalizacija Unija

Unija se definiše i deklariše na isti način kao i strukture.
Jedina razlika u deklaraciji je ključna riječ **union** koja se koristi umjesto **struct**. Da definišete jednostavne uniju od **char** varijable i **integer** varijable, možete napisati sljedeće:

```
union shared {
    char c;
    int i;
};
```

Ova unija, **shared**, može se koristiti da se kreiraju instance unije koja može držati karakterne vrijednosti **c** ili **integer** vrijednosti **i**. Ovo je **ili** (OR) uslov.
Za razliku od struktura koja bi držala obadvije vrijednosti, unija može držati samo jednu vrijednost (at a time).

Unija može biti inicijalizirana pri svojoj deklaraciji. Zato što se samo jedan član može koristiti (at a time), samo se jedan može i inicijalizirati.

Da bi izbjegli zbunjozu, samo se prvi član unije može inicijalizirati. Slijedi kood koji pokazuje instancu od **shared** unije, koja se deklariše i definiše:

```
union shared generic_variable = {`@'};
```

Primjetite da je **generic_variable** unija inicijalizirana, baš kako bi i prvi član strukture bio inicijaliziran.

→→ Pristupanje članovima unije

Pojedinačni članovi unije mogu biti korišteni na isti način kao što mogu biti korišteni i članovi strukture → koristeći članski operator (→ **.** ←). Ipak, postoji bitna razlika u pristupanju članovima unije. Samo jednom članu unije treba da se pristupa (at a time). Zato što unija smješta svoje članove na vrh jedan drugog, važno je da se pristupa samo jednom članu (at a time).

Listing 11.6 prezentuje primjerćić:

Listing 11.6. Primjer POGRŠNE upotrebe unije

```
1:   /* Primjer koristenja vise od jedne unije u isto vrijeme. */
2:   #include <stdio.h>
3:
4:   main()
5:   {
6:       union shared_tag {
7:           char   c;
8:           int    i;
9:           long   l;
10:          float  f;
11:          double d;
12:      } shared;
13:
14:      shared.c = `$';
15:
16:      printf("\nchar c   = %c",  shared.c);
17:      printf("\nint i    = %d",  shared.i);
18:      printf("\nlong l   = %ld", shared.l);
19:      printf("\nfloat f  = %f",  shared.f);
20:      printf("\ndouble d = %f",  shared.d);
21:
22:      shared.d = 123456789.8765;
23:
24:      printf("\n\nchar c   = %c",  shared.c);
25:      printf("\nint i    = %d",  shared.i);
26:      printf("\nlong l   = %ld", shared.l);
27:      printf("\nfloat f  = %f",  shared.f);
28:      printf("\ndouble d = %f\n",  shared.d);
29:
30:      return 0;
31:  }
           char c   = $
           int i    = 4900
           long l   = 437785380
           float f  = 0.000000
           double d = 0.000000
           char c   = 7
           int i    = -30409
           long l   = 1468107063
           float f  = 284852666499072.000000
           double d = 123456789.876500
```

ANALIZA: U linijama od **6** do **12** se definiše i deklariše unija nazvana **shared**.
shared sadrži pet članova, svaki različitog tipa.
Linije **14** do **22** inicijaliziraju pojedinačne članove od **shared**.
Linije **16** do **20**, i od **24** do **28** onda predstavljaju vrijednosti svakih članova koristeći **printf()** iskaze.

Zato što je karakterna varijabla, **c**, inicijalizirana u liniji **14**, to je jedina vrijednost koja bi trebala biti korištena dok se ne inicijalizira drugi član. Rezultat printanja članova varijabli druge unije (**i**, **l**, **f**, i **d**) mogu biti nepredvidivi (linije **16** do **20**).
Linija **22** stavlja (puts) vrijednost u **double** varijablu, **d**. Primjetite da je printanje varijabli opet nepredvidivo za sve osim za **d**. Vrijednost unesena u **c**, u liniji **14** je izgubljena zato što je prepisana (overwriten) kada je vrijednost **d**-a u liniji **22** unešena. Ovo je dokaz da svi članovi zauzimaju isti prostor.

→→→ union ←←← Ključna riječ

```
union tag {
    union_member(s);
    /* additional statements may go here */
}instance;
```

union ključna riječ se koristi da se deklariše unija.
Unija je skup jedne ili više varijabli (**union_members** (clanovi_unije)) koje su grupisane pod jednim imenom. S dodatkom, svaki od ovih članova unije zauzima istu oblast memorije.

Ključna riječ **union** indentifikuje početak definicije unije. Nakon ključne riječi unije slijedi tag (obilježje (ime)) unije. Nakon imena (tag-a) su članovi unije zatvoreni u zagrade.

Instanca, stvarna deklaracija unije, takođe može biti definisana. Ako definišete strukturu bez instace, ona je onda samo template koji može biti korišten kasnije u programu da deklariše strukture.
Slijedi template-ov format:

```
union tag {
    union_member(s);
    /* additional statements may go here */
};
```

Da koristite ovaj template, koristili biste sljedeći format:

```
union tag instance;
```

Da bi koristili ovaj format, morate prethodno deklarisati uniju sa datim tag-om(imenom).

Primjer 1

```
/* Declare a union template called tag */
union tag {
    int nbr;
    char character;
}
/* Use the union template */
union tag mixed_variable;
```

Primjer 2

```
/* Declare a union and instance together */
union generic_type_tag {
    char c;
    int i;
    float f;
    double d;
} generic;
```

Primjer 3

```
/* Initialize a union. */
union date_tag {
    char full_date[9];
    struct part_date_tag {
        char month[2];
        char break_value1;
        char day[2];
        char break_value2;
        char year[2];
    } part_date;
}date = {"01/01/97"};
```

Listing **11.7** pokazuje više praktičnu primjenu unije. Iako je ova upotreba pojednostavljena, ovo je jedna od najčešćih upotreba unije.

Listing 11.7. Praktična upotreba unije

```
1:   /* Primjer tipicne upotrebe unije */
2:
3:   #include <stdio.h>
4:
5:   #define CHARACTER   `C'
6:   #define INTEGER     `I'
7:   #define FLOAT       `F'
8:
9:   struct generic_tag{
10:      char type;
11:      union shared_tag {
12:          char   c;
13:          int    i;
14:          float  f;
15:      } shared;
16:  };
17:
18:  void print_function( struct generic_tag generic );
19:
20:  main()
21:  {
22:      struct generic_tag var;
23:
24:      var.type = CHARACTER;
25:      var.shared.c = `$';
26:      print_function( var );
27:
28:      var.type = FLOAT;
29:      var.shared.f = (float) 12345.67890;
30:      print_function( var );
31:
32:      var.type = `x';
33:      var.shared.i = 111;
34:      print_function( var );
35:      return 0;
36:  }
37:  void print_function( struct generic_tag generic )
38:  {
39:      printf("\n\nThe generic value is...");
40:      switch( generic.type )
41:      {
42:          case CHARACTER: printf("%c",  generic.shared.c);
43:                          break;
44:          case INTEGER:   printf("%d",  generic.shared.i);
45:                          break;
46:          case FLOAT:     printf("%f",  generic.shared.f);
```

```
47:                             break;
48:          default:           printf("an unknown type: %c\n",
49:                                     generic.type);
50:                             break;
51:      }
52:  }
      The generic value is...$
      The generic value is...12345.678711
      The generic value is...an unknown type: x
```

ANALIZA: Ovaj program obezbjeđuje način za smještanje tipova podataka u jedan smještajni prostor. **generic_tag** struktura vam dozvoljava da smještate ili karakter, ili integer, ili floating-point broj u istoj oblasi. Ova oblast je unija nazvana **shared** koja radi kao i u primjeru Listing **11.6**.
Primjetite da **generic_tag** struktura takođe dodaje dodatno polje nazvano **type**. Ovo se polje koristi da smjesti informacije na tip od varijable koja je sadržana u **shared**.
type pomaže u sprječavanju da se **shared** koristi na pogrešan način, tako što pomaže da se izbjegnu veliki podaci kao što su prezentovani u primjeru Listing **11.6**.

Linije **5**,**6** i **7** definišu konstante **CHARACTER**, **INTEGER** i **FLOAT**, koje se koriste kasnije u programu radi lakše čitljivosti.
Linija **9** do **16** definiše **generic_tag** strukturu koja će se koristiti kasnije.
Linija **18** predstavlja prototip za funkciju **print_function()**.
Struktura **var** je deklarisana u liniji **22** i inicijalizirana je da prvo drži vrijednost karaktera u linijama **24** i **25**.
Poziv da **print_function()** u liniji **26** omogućava da se printaju vrijednosti.
Linije **28** do **30** i od **32** do **34** ponavljaju ovaj proces sa ostalim varijablama.

print_function() je srce ovog listinga.
Iako je ova funkcija korištena da printa vrijednosti od **generic_tag** varijable, slična funkcija može biti korištena da je inicijalizira.
print_function() će procjeniti tip varijable s ciljem da printa iskaz sa odgovarajućim tipom varijable. Ovo sprječava dobijenje velikih podataka kao u listingu **11.6**.

NE pokušavajte da inicijalizirate više od jednog člana unije.
ZAPAMTITE koji član unije se koristi. Ako popunite član jednog tipa i onda pokušate da koristite drugi tip, možete dobiti nepredvidive rezultate.
NE zaboravite da je veličina unije jednaka sa njenim najvećim članom.
PRIMJETITE da su unije napredna tema **C**-a.

→→→ typedef i Strukture

Možete koristiti **typedef** ključnu riječ da kreirate sinonim za tip strukture ili unije.
Na primjer, sljedeći iskaz definiše **coord** kao sinonim za naznačenu strukturu:

```
typedef struct {
    int x;
    int y;
} coord;
```

Onda možete deklarisati instance od ove strukture koristeći **coord** identifikator:

```
coord topleft, bottomright;
```

Primjetite da je **typedef** različit od imena strukture (structure tag). Ako napišete:

```
struct coord {
    int x;
    int y;
};
```

identifikator **coord** je tag (oznaka (ime)) za strukturu.

Možete koristiti tag da deklarišete instance za strukturu, ali za razliku od **typedef**-a, morate uključiti **struct** ključnu riječ:

```
struct coord topleft, bottomright;
```

Da li koristite **typedef** ili strukturin tag da deklarišete strukturu, ne pravi veliku razliku.
Koristeći **typedef** rezultira sa malo konciznijim koodom, zato što **struct** ključna riječ ne mora biti upotrebljena.
S druge strane, koristeći tag i imajući **struct** ključnu riječ, eksplicitno čini jasnim da je struktura deklarisana.

Sažetak

Ovo poglavlje pokazuje kako se koriste strukture, tip podataka koji vi dizajnirate da odgovara potrebama vašeg programa.
Struktura može sadržavati bilo koji **C**-ov tip podataka, uključujući druge strukture, pointere, i nizove. Svaki podatak unutar strukture se naziva **član**, i njemu se pristupa koristeći strukturin članski operator (**.**) između imena strukture i imena člana. Struktura se može koristiti pojedinačno, i također mogu biti korištene u nizovima.
Unije su predstavljene kao slične strukturama. Glavna razlika između unije i strukture je da unija smješta sve svoje članove u istu oblast. Ovo znači da se samo jedan član unije može koristiti (at a time (u isto vrijeme)).

P&O

P Da li postoji razlog da se deklariše struktura bez instance???
O Pokazao sam ti dva načina kako da deklarišeš strukturu. **Prvi je** da deklarišeš, tijelo strukture, tag, i instancu, sve odjednom. **Drugi je** da deklarišeš tijelo strukture i tag bez instance. Instanca tada može biti deklarisana kasnije koristeći ključnu riječ **struct**, tag i ime za instancu. Česta je programerska praxa da se koristi drugi metod. Većina programera deklariše tijelo strukture i tag bez instance. Instance se zatim deklarišu kasnije u programu. Sljedeće poglavlje objašnjava pogled (domet) varijable. Domet (scope) će se primjeniti na instancu, ali ne i na tag i tijelo strukture.
P Da li je češća upotreba typedef-a ili strukturinog tag-a (obilježja (imena))?
O Puno programera koristi **typedef** da učine svoj kood lakšim za čitanje, ali to ne pravi veliku praktičnu razliku. Većina add-in biblioteka koje sadrže funkcije su dostupne za kupovinu. Ovi add-in –I često imaju puno **typedef**-ova da učine svoj proizvod unikatnim. Ovo je naročito tačno za database add-in proizvode.
P Mogu li jednostavno pridružiti jednu strukturu drugoj sa operatorom peridruživanja???
O Da i Ne. Noivije verzije **C** kompajlera (prevodioca) vam dopuštaju da pridružite jednu strukturu drugoj, ali starije verzije ne. U starijim verzijama **C**-a, možda trebate pridružiti svaki član strukture pojedinačno! Ovo je true i za unije.
P Koiko je velika unija???
O Zato što je svaki član unije smješten na istu memorijsku lokaciju, iznos potrebnog prostora da se smjesti unija je jednaka sa njenim najvećim članom.

Vježbe:

1. Napišite kood koji definiše strukturu nazvanu **time**, koja sadrži tri **int** člana.
2. Napišite kood koji obavlja dva zadatka: definiše strukturu nazvanu **data**, koja sadrži jedan član tipa **int** i dva člana tipa **float**, i deklariše instancu tipa data nazvanu **info**.
3. Nastavljajući sa vježbom **2**, kako biste pridružili vrijednost **100 integer** članu strukture **info**?
4. Napišite kood koji deklariše i inicijalizira pointer na **info**.
5. Nastavljajući sa vježbom **4**, pokažite dva načina koristeći pointer notaciju, da pridružite vrijednost **5.5** na prvi **float** član od **info**.
6. Napišite definiciju za tip strukture nazvanu **data** koja može držati jedan string do **20** karaktera.
7. Kreirajte strukturu koja sadrži pet stringova: **address1**, **address2**, **city**, **state**, i **zip**. Kreirajte **typedef** nazvanu **RECORD** koja se može koristiti za kreiranje instance za ovu strukturu.
8. Koristeći **typedef** iz vježbe **7**, alocirajte i inicijalizirajte element nazvan **myaddress**.
9. **BUG BUSTER: Šta nije u redu sa sljedećim koodom???**

```
struct {
    char zodiac_sign[21];
    int month;
} sign = "Leo", 8;
```

10. **BUG BUSTER: Šta nije u redu sa sljedećim koodom???**

```
/* setting up a union */
union data{
    char a_word[4];
    long a_number;
}generic_variable = { "WOW", 1000 };
```

LEKCIJA 12: → Understanding Variable Scope ←

→→ Šta je domet → (Scope)?

Domet varijable se odnosi na produžetak na koji različiti dijelovi programa imaju pristup varijabli → tj., gdje je varijabla vidljiva (visible).

Kada govorimo o vidljivosti, pojam varijable se odnosi na sve **C**-ove tipove podataka: jednostavne varijable, nizove, strukture, pointere, itd. Takođe se odnosi na simboličke konstante definisane sa ključnom riječi **const**.

Domet se takođe odražava na životni vijek (**lifetime**) varijable: koliko dugo varijabla opstaje u memoriji, ili kada se varijablin smještaj alocira i dealocira. Prvo, ovo poglavlje ispituje vidljivost (visibility).

→ Demonstracija dometa (Scope)

Listing **12.1** definiše varijablu **x** u liniji **5**, koristi **printf()** da prikaže vrijednost **x**-a u liniji **11**, i onda zove funkciju **print_value()** da prikaže vrijednost **x**-a ponovo.
Primjetite da se funkciji **print_value()** ne prosljeđuje vrijednost od **x** kao argument; ona jednostavno koristi **x** kao argument za **printf()** u liniji **19**.

Listing 12.1. Varijabla x je pristupna unutar funkcije print_value().

```
1:  /* Ilustruje domet varijable (scope). */
2:
3:  #include <stdio.h>
4:
5:  int x = 999;
6:
7:  void print_value(void);
8:
9:  main()
10: {
11:     printf("%d\n", x);
12:     print_value();
13:
14:     return 0;
15: }
16:
17: void print_value(void)
18: {
19:     printf("%d\n", x);
20: }
        999
        999
```

Ovaj se program kompajlira i pokreće bez problema.
Sad napravimo malu modifikaciju u programu, pomjerajući definiciju varijable **x** na lokaciju unutar **main()** funkcije. Novi izvorni kood je prikazan u Listingu **12.2**.

Listing 12.2. Varijabla x nije pristupna unutar funkcije print_value().

```
1:  /* Ilustruje domet varijable (scope). */
2:
3:  #include <stdio.h>
4:
5:  void print_value(void);
6:
7:  main()
8:  {
9:     int x = 999;
10:
```

```
11:     printf("%d\n", x);
12:     print_value();
13:
14:     return 0;
15: }
16:
17: void print_value(void)
18: {
19:     printf("%d\n", x);
20: }
```

ANALIZA: Ako pokušate da kompajlirate Listing **12.2**, kompajler generiše poruku greške sličnu ovoj:

```
list1202.c(19) : Error: undefined identifier `x'.
```

Zapamtite da u poruci greške, **broj u zagradi** se odnosi na programsku liniju gdje je pronađena greška. Linija **19** je poziv na **printf()** unutar **print_value()** funkcije.

Ova greška vam govori da je unutar funkcije **print_value()**, varijabla **x** nedefinisana, ili u drugim riječima, nije vidljiva (not visible). Primjetite, ipak, da poziv za **printf()**-om u liniji **11** ne generiše poruku greške; u ovom dijelu programa, varijabla **x** je vidljiva.

Jedina razlika između Listinga **12.1** i Listinga **12.2** je gdje je varijabla **x** definisana.
Pomjerajući definiciju varijable **x**, vi ste joj promjenili domet. U Listingu **12.1**, **x** je externa varijabla (external variable), i njen domet je cijeli program. Pristupna je (akcesibilna) i u **main()** funkciji i u **print_value()** funkciji.

U Listingu **12.2**, **x** je lokalna varijabla, i njen domet je ograničen na **main()** funkciju. Što se tiče **print_value()** funkcije, **x** ne postoji.

→→→ Externe Varijable

Externa **varijabla** je varijabla koja je definisana izvan bilo kojih funkcija. To znači i izvan **main()**-a, jer je i **main()** takođe funkcija. Do sada, većina varijabli u ovoj knjizi je bila **extern**a, smještena u izvornom koodu prije početka **main()**-a. **Extern**a varijable se nekad nazivaju i globalne varijable.

NAPOMENA: Ako explicitno ne inicijalizirate externu varijablu kada je definišete, kompajler je inicijalizira na **0**.

→ Domet externe varijable

Domet **extern**e varijable je cijeli program. To znači da je **extern**a varijabla vidljiva kroz **main()** i bilo koju drugu funkciju u programu. Na primjer, varijabla **x** u Listingu **12.1** je **extern**a varijabla.
Precizno govoreći, nije tačno kad se kaže da je domet **extern**e varijable cijeli program. Umjesto toga, domet je cijeli file izvornog kooda koji sadrži definiciju varijable.

Moguće je, da izvorni kood programa bude sadržan u dva ili više odvojenih file-ova. Ovo ćete naučiti Dana 21.

→→ Kada koristiti externe varijable

U praxi trebate koristiti **extern**e varijable što je rjeđe moguće. **Zašto?**
Zato što kad koristite **extern**e varijable, vi kršite princip modularne nezavisnosti koji je centralni u strukturnom programiranju. Modularna nezavisnost je ideja da svaka funkcija, ili modul, u programu sadrži sav kood i podatke koji su potrebni da se uradi taj posao. Sa relativno malim programima, koje sad pišete, ovo možda izgleda nebitno, ali kako napredujete na veće i komplexnije programe, “overreliance” na **extern**im varijablama može početi da stvara probleme.

Kad treba koristiti externe varijable???
Napravite **extern**u varijablu samo onda kada svi ili većina programskih funkcija treba pristup toj varijabli. Simbolične konstante definisane sa ključnom riječi **const** su često dobri kandidati za **extern**i status. Ako samo neke od funkcija trebaju pristup varijabli, prosljedite varijablu funkciji kao argument radije nego praveći je **extern**om.

→ extern ← Ključna riječ

Kada funkcija koristi **extern**u varijablu, dobra je programerska vještina da deklarišete varijablu unutar funkcije, koristeći ključnu riječ **extern**. Deklaracija ima formu:

```
extern type name;
```

u kojem je **type** tip varijable i **name** je ime varijable. Na primjer, dodali biste deklaraciju **x**-a funkcijama **main()** i **print_value()** u Listingu **12.1**. Rezultirajući program je prikazan u Listingu **12.3**.

Listing 12.3. externa varijabla x je deklarisana kao extern unutar funkcija main() i print_value().

```
1:  /* Ilustruje deklaraciju externih varijabli. */
2:
3:  #include <stdio.h>
4:
5:  int x = 999;
6:
7:  void print_value(void);
8:
9:  main()
10: {
11:     extern int x;
12:
13:     printf("%d\n", x);
14:     print_value();
15:
16:     return 0;
17: }
18:
19: void print_value(void)
20: {
21:     extern int x;
22:     printf("%d\n", x);
23: }
        999
        999
```

ANALIZA: Ovaj program printa vrijednost **x**-a dva puta, prvo u liniji **13**, kao dio **main()**-a, a onda u liniji **21** kao dio **print_value()**-a.
Linija **5** definiše **x** kao varijablu tipa **int**, jednaku sa **999**.
Linije **11** i **21** deklarišu **x** kao **extern int**. Primjetite razliku između definicija varijable, koja ostavlja sa strane smještajni prostor za varijablu, i **extern**u deklaraciju.

Kaže se: “Ova funkcija koristi **extern**u varijablu sa tim-i-tim imenom i tipom koji je definisan negdje drugo”. U ovom slučaju, **extern** deklaracija nije potrebna, precizno govoreći → program će raditi jednako isto kao i bez linija **11** i **21**. Ipak, da je funkcija **print_value()** bila u različitom (drugom) kood modulu nego globalna deklaracija varijable **x** (u liniji **5**), tada bi **extern**a deklaracija bila potrebna.

→→→ Lokalne Varijable

Lokalna varijabla je ona koja je definisana unutar funkcije.
Domet lokalne varijable je ograničen na funkciju u kojoj je definisana.
Dan 5 opisuje lokalne varijable unutar funkcija, kako da ih definišete, i koje su prednosti njene upotrebe.

Lokalne varijable nisu automatski inicijalizirane na **0** od strane kopajlera. Ako ne inicijalizirate lokalnu varijablu kada je definišete, ona ima nedefinisanu "smeće" vrijednost.
Vi **MORATE** explicitno pridružiti vrijednost lokalnim varijablama prije nego što ih koristite prvi put.

Varijabla može biti lokalna i **main()** funkciji takođe. Ovo je slučaj za **x** u Listingu **12.2**.
Ona je definisana unutar **main()**-a, i dok se kompajlira i izvršava taj program (**12.2**), ona je takođe vidljiva samo unutar **main()**-a.

KORISTITE lokalne varijable za predmete kao što su brojači petlji.
NE koristite **extern**a varijable ako nisu potrebne većini programskih funkcija.
KORSITITE lokalne varijable da izolirate vrijednosti varijabli koje su sadržane u ostatku programa.

→ Static ← nasupros → Auto**m**atic ← varijablama

Lokalne varijable su **auto**matic po defaultu. To znači da se lokalne varijable kreiraju ponovo svaki put kada se zove funkcija, i one se uništavaju kada izvršenje napusti funkciju. Ovo u praxi znači da, **auto**matic varijabla ne zadržava vrijednost između poziva funkcije u kojoj je definisana.

Pretpostavimo da vaš program ima funkciju koja koristi lokalnu varijablu **x**. Takođe pretpostavimo da prvi put kada se zove, funkcija pridružuje vrijednost **100** **x**-u. Izvršenje se vraća pozivajućem programu, i funkcija se zove kasnije ponovo. Da li varijabla **x** još uvijek drži vrijednost **100**? **NE!!!** Prva instanca varijable **x** je uništena kada je izvršenje napustilo funkciju nakon prvog poziva. Kada je funkcija bila pozvana ponovo, nova instanca od **x** se mora kreirati. Stari **x** je otišao.

Šta ako funkcija treba da zadrži vrijednost lokalne varijable između poziva???
Na primjer, printajuća funkcija možda treba da pamti broj linija koje su poslane printeru da odluči kada je potrebna nova stranica. S ciljem da lokalna varijabla zadrži vrijednost između poziva, ona mora biti definisana kao statična sa ključnom riječi →→→ **static**. ←←←

Na primjer:

```
void func1(int x)
{
    static int a;
    /* Additional code goes here */
}
```

Listing 12.4 ilustruje razliku između **auto**matic i **static** lokalne varijable.
Listing 12.4. Razlika između automatske i static lokalne varijable.

```
1:  /* Demonstrira  automatic i static lokalne varijable. */
2:  #include <stdio.h>
3:  void func1(void);
4:  main()
5:  {
6:     int count;
7:
8:     for (count = 0; count < 20; count++)
9:     {
10:         printf("At iteration %d: ", count);
11:         func1();
12:     }
```

```
13:
14:     return 0;
15: }
16:
17: void func1(void)
18: {
19:     static int x = 0;
20:     int y = 0;
21:
22:     printf("x = %d, y = %d\n", x++, y++);
23: }
```

```
At iteration 0: x = 0, y = 0
At iteration 1: x = 1, y = 0
At iteration 2: x = 2, y = 0
At iteration 3: x = 3, y = 0
At iteration 4: x = 4, y = 0
At iteration 5: x = 5, y = 0
At iteration 6: x = 6, y = 0
At iteration 7: x = 7, y = 0
At iteration 8: x = 8, y = 0
At iteration 9: x = 9, y = 0
At iteration 10: x = 10, y = 0
At iteration 11: x = 11, y = 0
At iteration 12: x = 12, y = 0
At iteration 13: x = 13, y = 0
At iteration 14: x = 14, y = 0
At iteration 15: x = 15, y = 0
At iteration 16: x = 16, y = 0
At iteration 17: x = 17, y = 0
At iteration 18: x = 18, y = 0
At iteration 19: x = 19, y = 0
```

ANALIZA: Ovaj program ima funkciju koja definiše i inicijalizira jednu varijablu od svakog tipa. Ova funkcija je funkcija **func1()** u linijama **17** do **23**. Svaki put kada se pozove funkcija, obadvije varijable se prikazuju na ekranu i inkrementiraju (linija **22**).
main() funkcija, u linijama **4** do **15** sadrži **for** petlju (linije **8** do **12**) koja printa poruku (linija **10**) i onda zove **func1()** (linija **11**).
for petlja iterira **20** puta.
U izlazu, primjetite da se **x**, **static** varijabla, povećava sa svakom iteracijom zato što ona zadržava vrijednost između poziva. Varijabla **auto**matic, **y**, s druge strne, se reinicijalizira na **0** sa svakim pozivom.
Ovaj program takođe ilustruje razliku na koji način se explicitno vrši inicijalizacija varijable (tj., kada se varijabla inicijalizira za vrijeme definicije). Varijabla **static** je incijalizirana samo prvi put, kada je pozvana funkcija. U sljedećim pozivima, program pamti da je varijabla već inicijalizirana i tako ne zahtjeva reinicijalizaciju.
Varijabla vraća vrijednost koju je imala kada je izašlo zadnje izvršenje funkcije.
Nasuprot tome, varijabla **auto**matic je inicijalizirana na određenu vrijednost, svaki put kada je funkcija pozvana.

Ako experimentišete sa **auto**matskim varijablama, možete dobiti rezultate koji se ne slažu sa onim što ste pročitali ovdje. Na primjer, ako modifikujete Listing **12.4**, tako da se dvije lokalne varijable ne inicijaliziraju kada su definisane, funkcija **func1()** u linijama **17** do **23** čita:

```
17: void func1(void)
18: {
19:     static int x;
20:     int y;
21:
22:     printf("x = %d, y = %d\n", x++, y++);
23: }
```

Kada pokrenete modifikovan program, možete naći da je vrijednost od **y**-a povećana za **1** sa svakom iteracijom. Ovo znači da **y** čuva vrijednost između poziva funkcije.

Da li je ovo što ste pročitali o varijablama automatic bunch of mumbo jumbo???
NE, nije (have faith!). Ako dobijete rezultate, kao što su opisani maloprije, u kojiem je varijabla **auto**matic držala svoju vrijednost tokom repetivnih poziva funkcije, oni su samo slučajni.

Evo šta se dešava: Svaki put kada se pozove funkcija, novi **y** se kreira. Kompajler može koristiti istu memorijsku lokaciju za novi **y** koji je korišten za **y** u prethodnom slučaju kad je funkcija pozvana. Ako **y** nije explicitno inicijaliziran od strane funkcije, smještajna lokacija možda sadrži vrijednost koju je **y** imao tokom prethodnog poziva. Varijabla izgleda kao da je sačuvala svoju staru vrijednost, ali je to igra slučaja; definitivno ne možete računati da će se to desiti svaki put.

→ **auto**matic je default za lokalne varijable, pa zato ne mora biti posebno naglašeno u definiciji varijable. Ako želite, možete uključiti ključnu riječ → **auto** ← u definiciji prije tipa ključne riječi, kako je pokazano:

```
void func1(int y)
{
    auto int count;
    /* Additional code goes here */
}
```

→→→ Domet (Scope) funkcijskih parametara (of Function Parameters)

Varijabla koja je sadržana u listi parametara zaglavlja funkcije, ima lokalni domet (local scope). Na primjer, pogledajte sljedeću funkciju:

```
void func1(int x)
{
    int y;
    /* Additional code goes here */
}
```

I **x** i **y** su lokalne varijable sa dometom koji je cijela funkcija **func1()**. Naravno, **x** inicijalno sadrži onu (bilokoju) vrijednost koja je prosljeđena funkciji od strane pozivajućeg programa. Jednom kada ste koristili ovu vrijednost, možete koristiti **x** kao bilo koju drugu lokalnu varijablu.

Zato što parametarske varijable uvijek počinju sa vrijednošću prosljeđenom kao korespondirajući (odgovarajući) argument, besmisleno je da mislite o njima kao o da su **static** ili **auto**matic.

→→→ Externe static varijable

Možete napraviti **extern**u (globalnu) varijablu **static**, uključujući ključnu riječ **static** u njenoj definiciji:

```
static float rate;
main()
{
    /* Additional code goes here */
}
```

Razlika između obične externe varijable i static externe varijable je u dometu. Obična **extern**a varijabla je vidljiva za sve funkcije u file-u i može se koristiti od strane funkcija u drugim file-ovima. **static extern**a varijabla je vidljiva samo funkcijama u svom file-u i ispod, nakon definicije.

Ove razlike se očigledno primjenjuju većinom na programe sa izvornim koodom koji je sadržan u dva ili više file-ova. (Dan 21).

→→→ Register Varijable

Ključna reiječ → → → **register** ← ← ← se koristi da sugeriše kompajleru da se **auto**matic lokalna varijabla radije smješta u procesorski registar, nego u regularnu memoriju.
Šta je procesorski registar, i koje su prednosti u njegovom korištenju???

Central processing unit (CPU) sadrži nekoliko podatkovno smještajnih lokacija nazvani registri. U njima se odvijaju stvarne podatkovne operacije, kao što su sabiranje i dijeljenje. Da manipuliše podacima, CPU mora premjestiti podatke iz memorije u svoje registre, obaviti manipulacije, i onda premjestiti podatke nazad u memoriju. Premještajući podatke u i iz memorije uzima konačnu količinu vremena. Ako se neka varijabla može čuvati u registru sa kojom se počinje, manipulacija varijable će početi puno brže.

Koristeći ključnu riječ **register** u definiciji varijable **auto**matic, vi pitate kompajler da smjesti tu varijablu u registar. Pogledajte sljedeći primjer:

```
void func1(void)
{
    register int x;
    /* Additional code goes here */
}
```

Primjetite da sam rekao pita, a ne kaže. Zavisno od potreba programa, registar možda neće biti dostupan za varijablu. U tom slučaju, kompajler je tretira kao običnu varijablu **auto**matic.
Ključna riječ **register** je sugestija, ne naredba.
Korist od smještajne klase **register** je veća za varijable koje funkcija koristi često, kao što je brojačka varijabla za petlju.

Ključna riječ **register** može biti korištena samo sa jednostavnim numeričkim varijablama, ne sa nizovima ili strukturama. Također, ne može biti korištena sa ili statičnom (**static**) ili externom (**extern**) smještajnom klasom. Ne možete definisati pointer na varijablu **register**.

INICIJALIZIRAJTE lokalne varijable, ili nećete znati koje vrijednosti ona sadrži.
INICIJALIZIRAJTE gloabalne varijable, iako su inicijalizirane na **0** po defaultu. Ako uvijek inicijalizirate vaše varijable, izbjeći ćete probleme kao što je zaboravljanje inicijalizacije lokalnih varijabli.
PROSLJEĐUJTE podatkovne predmete (data items) kao parametre funkcije umjesto da ih deklarišete kao global, ako su potrebni u samo nekoliko funkcija.
NE koristite varijable **register** za nenumeričke vrijednosti, strukture ili nizove.

→→ Lokalne Varijable i main() Funkcija

Sve što je dosad rečeno za lokalne varijable se primjenjuje (odnosi) i na **main()** kao i na sve ostale funkcije. Precizno govoreći, **main()** je funkcija kao i bilo koja drugo. **main()** funkcija se poziva kada se pokrene program od strane vašeg operativnog sistema, i kontrola se vraća operativnom sistemu iz **main()**-a kada se program izvrši (terminira).

Ovo znači da lokalne varijable, definisane u **main()**-u se kreiraju kada program počne, i njihov životni vijek je gotov kada program završi. Ideja da static lokalna varijabla zadržava vrijednost između poziva **main()**-a nema smisla: Varijabla ne može opstati između izvršenja prograama. Unutar **main()**-a, tako, nema razlike između lokalnih varijabli **auto**matic i **static**. Možete definisati lokalnu varijablu i **main()**-u kao **static**, ali to nema efekta.

ZAPAMTITE da je **main()** funkcija slična u puno stvari bilo kojoj drugoj funkciji
NE deklarišite varijable **static** u **main()**-u, jer radeći to, ne dobijate ništa.

Koje smještajne klase trebam koristiti???
Kada odlučujete koju smještajnu klasu da koristite za neku naročitu varijablu u vašem programu, može biti od koristiti koristiti tabelu **12.1**, koja skraćuje (summarizes) pet smještajnih klasa dostupnih u **C**-u.

Tabela 12.1. C-ovih pet varijablinih smještajnih klasa.

Klasa (Class)	Ključna riječ (Keyword)	Vrijeme života (Storage Lifetime)	Gdje je definisana :	Domet (Scope)
Automatic	None[1]	Temporary	In a function	Local
Static	**static**	Temporary	In a function	Local
Register	**register**	Temporary	In a function	Local
External	None[2]	Permanent	Outside a function	Global (all files)
External	**static**	Permanent	Outside a function	Global (one file)

[1] **auto** ključna riječ je opcionalna.

[2] **extern** ključna riječ se koristi u funkcijama da deklariše static externu varijablu koja je definisana negdje drugo.

Kada odlučujete o smještajnoj klasi, trebate koristiti **auto**matic amještajnu klasu kad god je to moguće i koristite ostale klase samo kad je to potrebno.

Evo par stvari na koje trebate obratiti pažnju:
- Dajte svakoj varijabli **auto**matic smještajnu klasu za početak.
- Ako će se s varijablom manipulisati često (frekventno), dodajte ključnu riječ **register** u njenu definiciju.
- U funkciji koja nije **main()**, napravite **static** varijablu ako se njena vrijednost treba sačuvati između poziva funkcije.
- Ako se varijabla koristi u većini ili svim programskim funkcijama, definišite je sa **extern**om smještajnom klasom.

→→ Lokalne varijable i Blokovi

Do sada, ovo poglavlje je diskutovalo samo varijable koje su lokalne funkciji. Ovo je primarni način na koji se koriste lokalne varijable, ali možete definisati varijable koje su lokalne bilo kojem programskom bloku (bilo koji dio zatvoren u zagradama).

Kada deklarišete varijable unutar bloka, morate zapamtiti da deklaracija mora biti prva.
Listing 12.5 pokazuje primjer:

Listing 12.5. Definisanje lokalnih varijabli unutar programskog bloka.
```
1:  /* Demonstrira lokalne varijable unutar bloka. */
2:
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:      /* Definicija lokalne varijable za main(). */
8:
9:      int count = 0;
10:
11:     printf("\nOutside the block, count = %d", count);
12:
13:     /* Pocetak bloka. */
14:     {
15:       /* Definicija lokalne varijable za blok. */
16:
17:       int count = 999;
18:       printf("\nWithin the block, count = %d", count);
19:     }
20:
```

```
21:     printf("\nOutside the block again, count = %d\n", count);
22:     return 0;
23: }
       Outside the block, count = 0
       Within the block, count = 999
       Outside the block again, count = 0
```

ANALIZA: Iz ovog programa, možete vidjeti da je **count** definisan unutar bloka nezavisan od **count**-a definisanog izvan bloka.
Linija **9** definiše **count** kao varijablu tipa **int** koja je jednaka **0**. Zato što je deklarisana na početku **main()**-a, može se koristiti kroz cijelu **main()** funkciju.
Kood u liniji **11** pokazuje da je varijabla **count** inicijalizirana na **0** printajući njenu vrijednost.
Blok je deklarisan u linijama **14** do **19**, i unutar bloka, druga **count** varijabla je definisana kao varijabla tipa **int**. Ova **count** varijabla je inicijalizirana na **999** u liniji **17**.
Linija **18** printa vrijednost **999**, od blokove varijable **count**. Zato što blok završava u liniji **19**, print iskaz u liniji **21** koristi oreginalni **count** inicijalno deklarisan u liniji **9** od **main()**-a.

Upotreba ovog tipa lokalne varijable nije uobičajena u **C** programiranju, i možda nikad ne nađete je za potrebnu.
Češće se koristi kada programer pokušava da izolira problem unutar programa.
Možete privremeno izolirati dio kooda u zagradam i ostvariti lokalne varijable da vam asistiraju u pronalaženju problema.
Druga prednost je da varijablina deklaracija/inicijalizacija može biti smještena bliže tački gdje se koristi, što može pomoći u razumijevanju programa.

NE pokušavajte da stavljate definicije varijabli nigdje drugo nego na početak funkcije ili na početak bloka.
NE koristite varijable na početku bloka ukoliko ne čini program jasnijim.
KORISTITE varijable na početku bloka (privremeno) da pomognu pronalaženju problema.

Sažetak

Ovo poglavlje je pokrilo **C**-ove smještajne klase varijabli. Svaka **C**-ova varijabla, bilo jednostavna, niz, struktura, ili bilo koja, ima specifičnu smještajnu klasu koja odlučuje dvije stvari:
njen domet, ili gdje je vidljiva u memoriji;
i njen životni vijek, ili koliko dugo varijabla opstaje (ostaje) u memoriji.

Pravilna upotreba smještajnih klasa (storage classes) je važan aspekt strukturalnog programiranja. Držeći većinu varijabli lokalnim funkciji koja ih koristi, vi ste povećali funkcijinu nezavisnost od jedna druge.
Varijabli treba dati smještajnu klasu **auto**matic, osim ako postoji neki poseban razlog da se ona koristi kao externa ili statična (**extern** i **static**).

P&O

P Dan 11, "Structures," je rečeno da domet utiče na strukturinu instancu, ali ne i na strukturin tag (ime) ili tijelo. Kako domet ne utiče na strukturin tag ili tijelo???
O Kada deklarišete strukturu bez instance, vi kreirate template. Vi u stvari ne deklarišete nikakve varijable. Ne deklarišete varijablu sve dok ne kreirate instancu strukture. Zbog ovoga, možete ostaviti tijelo strukture externom, bilo kojoj funkciji koja nema stvarnog efekta na externu memoriju. Većina programera stavlja često korištena tijela strukture sa tagovima u file-u zaglavlja i onda uključi ove file-ove zaglavlja kada su potrebni za kreiranje instance od strukture. (File-ovi zaglavlja se obrađuju Dana 21).
P Kako kompjuter zna razliku između globalne i lokalne varijable koje imaju isto ime?
O Važna stvar koju trebate znati je da kad se deklariše lokalna varijabla sa istim imenom kao i globalna varijabla, program privremeno ignoriše globalnu varijablu. On nastavlja da je ignoriše dok lokalna varijabla ne izađe van dometa.

Vježbice:

3. Napišite program koji deklariše globalnu varijablu tipa int nazvanu var. Inicijalizirajte var na neku vrijednost. Program treba da printa vrijednost od var u funkciji (ne main()). Da li treba da prosljedite var kao parametar funkciji?
4. Promjenite program u vježbi 3. Umjesto deklaracije var kao globalne varijable, promjenite je u lokalnu varijablu u main()-u. Program još treba da printa var u posebnoj (odvojenoj) funkciji. Da li treba da prosljedite var kao parametar funkciji?
6. BUG BUSTER: Možete li pronaći problem u ovom koodu? Hint: Ima veze sa gdje je varijabla deklarisana.

```
void a_sample_function( void )
{
   int ctr1;
   for ( ctr1 = 0; ctr1 < 25; ctr1++ )
      printf( "*" );
   puts( "\nThis is a sample function" );
   {
      char star = `*';
      puts( "\nIt has a problem\n" );
      for ( int ctr2 = 0; ctr2 < 25; ctr2++ )
      {
         printf( "%c", star);
      }
   }
}
```

7. BUG BUSTER: Šta nije u redu sa sljedećim koodom?

```
/*Count the number of even numbers between 0 and 100. */
#include <stdio.h>
main()
{
   int x = 1;
   static int tally = 0;
   for (x = 0; x < 101; x++)
   {
      if (x % 2 == 0)  /*if x is even...*/
      tally++;..  /*add 1 to tally.*/
   }
   printf("There are %d even numbers.\n", tally);
   return 0;
}
```

8. BUG BUSTER: Da li nešto nije u redu sa sljedećim programom?

```
#include <stdio.h>
void print_function( char star );
int ctr;
main()
{
   char star;
   print_function( star );
   return 0;
}
void print_function( char star )
{
   char dash;
   for ( ctr = 0; ctr < 25; ctr++ )
   {
      printf( "%c%c", star, dash );
   }
}
```

9. Šta sljedeći program printa? Ne pokrećite program → pokušajte sami provaliti čitajući kood.

```
#include <stdio.h>
void print_letter2(void);           /* function prototype */
int ctr;
char letter1 = `X';
char letter2 = `=';
main()
{
   for( ctr = 0; ctr < 10; ctr++ )
   {
      printf( "%c", letter1 );
      print_letter2();
   }
   return 0;
}
void print_letter2(void)
{
   for( ctr = 0; ctr < 2; ctr++ )
      printf( "%c", letter2 );
}
```

10. BUG BUSTER: Da li će se prethodni program pokrenuti? Ako ne, u čemu je problem? Prepišite (rewrite) ga tako da bude tačan.

LEKCIJA 13: → Advanced Program Control ←

→→→ Prekinuti petlje ranija (Ending Loops Early)

Dana 6. ste naučili kako petlje: **for**, **while** i **do…while** mogu kontrolisati izvršenje programa. Ovakve kontrukcije petlje izvršavaju blok **C** iskaza nikad, jednom ili više od jednom, zavisno od uslova u programu. U sva tri slučaja, prekidanje (terminacija) ili izlaz (exit) iz petlje se dešava samo kad se desi neki određeni uslov.

Nekad, ipak, želite da imate više kontrole nad izvršenjem petlje.
break i **continue** iskazi vam ovo omogućuju.

→→→ break iskaz (Statement)

break iskaz može biti smješten samo u tijelu **for**, **while** ili **do…while**, petlje. (ovo vrijedi i za **switch** iskaz, ali je on obrađen kasnije u ovom poglavlju.)
Kada se naiđe na **break** iskaz, izvršenje izlazi iz petlje.
Slijedi primjer:

```
for ( count = 0; count < 10; count++ )
{
   if ( count == 5 )
      break;
}
```

Prepuštena sebi, for petlja bi se izvršila **10** puta. U šestoj iteraciji, ipak, **count** je jednak sa **5**, i **break** iskaz se izvršava, što uzrokuje da se **for** petlja terminira. Izvršenje se prosljeđuje iskazu koji slijedi odmah iza zatvorene zagrade **for** petlje. Kada se naiđe na **break** iskaz unutar ugnijezđene petlje, on uzrokuje da program izađe (exit) samo iz unutrašnje petlje (innermost loop only).

Listing 13.1 demonstrira upotrebu **break**-a.

Listing 13.1. Korištenje break iskaza.

```
1:  /* Demonstrira break iskaz. */
2:
3:  #include <stdio.h>
4:
5:  char s[] = "This is a test string. It contains two sentences.";
6:
7:  main()
8:  {
9:     int count;
10:
11:     printf("\nOriginal string: %s", s);
12:
13:     for (count = 0; s[count]!='\0'; count++)
14:     {
15:         if (s[count] == `.')
16:         {
17:             s[count+1] = `\0';
18:             break;
19:         }
20:     }
21:     printf("\nModified string: %s\n", s);
22:
23:     return 0;
24: }
  Original string: This is a test string. It contains two sentences.
  Modified string: This is a test string.
```

ANALIZA: Ovaj program vadi (extraktuje (extracts)) prvu rečenicu iz stringa. On traži string, karakter po karakter, za prvi period (koji bi trebao označavati kraj rečenice).
Ovo se radi u **for** petlji u linijama **13** do **20**.
Linija **13** počinje **for** iskaz, inkrementirajući **count** da ide od karaktera do karaktera u stringu, **s**.
Linija **15** provjerava da vidi da li je trenutni karakter u stringu jednak periodu. Ako jeste, **null** karakter se ubacuje odmah nakon perioda (linija **17**). Ovo, u stvari, uređuje string (trims string). Jednom kada uredite string, više nemate potrebu da nastavljate petlju, tako da **break** iskaz (linija **18**) brzo terminira petlju i šalje kontrolu prvoj liniji nakon petlje (linija **21**). Ako se ne nađe nijedan period, string se ne mijenja (promijeni (altered)).
Petlja može sadržavati više **break** iskaza, ali samo prvi **break** koji se izvršava (ako se) ima efekta. Ako se ne izvrši ni jedan **break**, petlja se terminira (prekida) normalno (prema svom testnom uslovu).

→ break iskaz (The break Statement)

```
break;
```

break se koristi unutar petlje ili **switch** iskaza. On prouzrokuje da se kontrola programa proslijedi na kraju petlje koja se izvršava (**for**, **while**, **do…while**) ili **switch** iskaza. Ne izvršavaju se nikakve buduće iteracije petlje; prva komanda koja slijedi nakon petlje ili **switch** iskaza se izvršava.

Primjer

```
int x;
printf ( "Counting from 1 to 10\n" );
/* having no condition in the for loop will cause it to loop forever */
for( x = 1; ; x++ )
{
   if( x == 10 )   /* This checks for the value of 10 */
      break;      /* This ends the loop */
   printf( "\n%d", x );
}
```

→→→ continue iskaz

Kao i **break** iskaz, **continue** iskaz može biti smješten samo unutar tijela petlji: **for**, **while** ili **do…while**.
Kada se izvrši **continue** iskaz, sljedeća iteracija od zatvorene (enclosing) petlje se izvršava odmah. Iskaz između **continue** iskaza i kraja petlje se ne izvršavaju.

Listing **13.2** koristi **continue** iskaz. Ovaj program prihvata linije ulaza (input) sa tastature i onda ga prikazuje sa svim odstranjenim samoglanicima malih slova.

Listing 13.2. Korištenje continue iskaza.

```
1:  /* Demonstrira continue iskaz. */
2:
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:     /* Deklaracija buffer-a za unos i brojacku varijablu. */
8:
9:     char buffer[81];
10:     int ctr;
11:
12:     /* Unesi liniju texta. */
13:
14:     puts("Enter a line of text:");
15:     gets(buffer);
16:
17:     /* Idi kroz string, prikazi samo one karaktere  */
```

```
18:     /* koji nisu samoglasnici malih slova (lowercase vowels). */
19:
20:     for (ctr = 0; buffer[ctr] !='\0'; ctr++)
21:     {
22:
23:         /* Ako je karakter samoglasnik malih slova, peetljaj */
24:         /* nazad bez prikazivanja (without displaying it). */
25:
26:         if (buffer[ctr] == `a' || buffer[ctr] == `e'
27:            || buffer[ctr] == `i' || buffer[ctr] == `o'
28:            || buffer[ctr] == `u')
29:               continue;
30:
31:         /* Ako nije samoglasnik, prikazi ga. */
32:
33:         putchar(buffer[ctr]);
34:     }
35:     return 0;
36: }
```

```
Enter a line of text:
This is a line of text
Ths s  ln f txt
```

ANALIZA: Iako ovo nije najpraktičniji program, on koristi **continue** iskaz efikasno.
Linije **9** i **10** deklarišu programske varijable, **buffer[]** drži string koji korisnik unese u liniji **15**.
Druga varijabla, **ctr**, se inkrementira kroz elementa niza **buffer[]**, dok **for** petlja, u linijama od **20** do **34** traži samoglasnike.
Za svako slovo u petlji, **if** iskaz u linijama **26** do **28** provjerava slovo nasuprot samoglasnika malih slova. Ako postoji odgovaranje (match), **continue** iskaz se izvršava, koji šalje kontrolu programa nazad na liniju **20**, **for** iskaz. Ako se ne nađe slovo samoglasnik, kontrola se prosljeđuje na **if** iskaz, i linija **33** se izvršava.
→ Linija **33** sadrži novu bibliotečnu funkciju, → → → **putchar()** ← ← ←, koji prikazuje jedan karakter na ekranu.

→→ continue iskaz

```
continue;
```

continue se koristi unutar petlje. On prouzrokuje da kontrola programa preskoči ostatak trenutnih (current) iteracija petlje i počinje sljedeći iteraciju.

Primjer

```
int x;
printf("Printing only the even numbers from 1 to 10\n");
for( x = 1; x <= 10; x++ )
{
   if( x % 2 != 0 )     /* See if the number is NOT even */
      continue;       /* Get next instance x */
   printf( "\n%d", x );
}
```

→→→ goto iskaz (The goto Statement)

goto iskaz je jedan od **C**-ovih bezuslovnih skokova, ili granajućih (branching) iskaza.
Kada izvršenje programa naiđe na **goto** iskaz, izvršenje odmah skače, ili granaa (branches), na lokaciju koja je specificirana (navedena) od strane **goto** iskaza.
Ovaj iskaz je bezuslovni jer izvršenje uvijek skače kad god se naiđe na **goto** iskaz; grananje (branch) ne zavisi od programskog uslova (za razliku od **if** iskaza, npr.).

goto iskaz i njegova meta moraju biti u istoj funkciji, ali oni mogu biti u različitim blokovima.

Listing **13.3** je jednostavan program koji koristi **goto** iskaz.

Listing 13.3. Upotreba goto iskaza.

```
1: /* Demonstrira goto iskaz */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     int n;
8:
9: start: ;
10:
11:     puts("Enter a number between 0 and 10: ");
12:     scanf("%d", &n);
13:
14:     if (n < 0 ||n > 10 )
15:         goto start;
16:     else if (n == 0)
17:         goto location0;
18:     else if (n == 1)
19:         goto location1;
20:     else
21:         goto location2;
22:
23: location0: ;
24:     puts("You entered 0.\n");
25:     goto end;
26:
27: location1: ;
28:     puts("You entered 1.\n");
29:     goto end;
30:
31: location2: ;
32:     puts("You entered something between 2 and 10.\n");
33:
34: end: ;
35:     return 0;
36: }
```

```
Enter a number between 0 and 10:
1
You entered 1.
Enter a number between 0 and 10:
9
You entered something between 2 and 10.
```

ANALIZA: Ovo je jednostavan program koji prihvata brojeve između **0** i **10**. Ako broj nije između **0** i **10**, program koristi **goto** iskaz u liniji **15** da ide na početak, koji je u liniji **9**. U drugom slučaju, program provjerava u liniji **16** da vidi da li je broj jednak sa **0**. Ako jeste, **goto** iskaz u liniji **17** šalje kontrolu na lokaciju**0** (linija **23**), koja printa iskaz u liniji **24** i izvršava drugi **goto**.
goto u liniji **25** šalje kontrolu na kraj na kraju programa. Program izvršava istu logiku za vrijednost (broj) **1** i sve vrijednosti između **2** i **10** u cjelini.

Meta (target) **goto** iskaza može biti ili prije ili poslije tog iskaza u koodu. Jedino ograničenje, kao što je rečeno, je da i **goto** i meta moraju biti u istoj funkciji. Ipak, oni mogu biti u različitim blokovima. Možete koristiti **goto** da prebacujete izvršenje i u i iz petlji, kao što je **for** iskaz, ali ovo ikad ne biste rebali raditi. U stvari, čvrsto se preporučuje da nikad ne koristite **goto** iskaz nigdje u vašem programu.
Evo dva razloga:

- Ne treba vam. Nijedan programski zadatak ne zahtjeva **goto** iskaz. Uvijek možete napisati potreban kood koristeći ostale **C**-ove granajuće (braching) iskaze.
- Opasno je. **goto** iskaz može da izgleda kao idealno rješenje za neke programerske probleme, ali je lako pogriješiti. Kada se izvršenje programa grana sa **goto** iskazom, nikakav zapis (record) se ne drži o odakle je došlo izvršenje programa, tako da izvršenje može talasati kroz program tamo-amo. Ovaj tip programiranja je poznat kao **spaghetti code**.

IZBJEGAVAJTE upotrebu **goto** iskaza ako je moguće.
NE miješajte (brkajte) **break** i **continue**.
break prekida petlju, dok **continue** nastavlja sljedeći iteraciju.

→→ goto iskaz

```
goto location;
```

lokacija je oznaka koja ideftifikuje (označava) lokaciju u prrogramu, gdje će izvršenje da se granaa. Označavajući iskaz se sastoji iz identifikatora, nakon koga slijedi dvotačka "**:**" (colon) i **C** iskaz:

```
location: a C statement;
```

Ako želite da je oznaka (label) sama od sebe u lednoj liniji, možete nakon nje staviti **null** iskaz (tačka-zarez samo "**;**"):

```
location: ;
```

→→→ Beskonačne petlje

Šta su beskonačna petlje, i zašto bi želio imati jednu u programu???
Beskonačna petlja je ona koja, ako se prepusti svojim spravama, izvršava beskonačno mnogo puta. Ona može biti **for** petlja, **while** petlja, ili **do…while** petlja.

Na primjer, ako napišete:

```
while (1)
{
    /* additional code goes here */
}
```

vi kreirate beskonačnu petlju. Uslov koji **while** testira je konstanta **1**, koja je uvijek **true** i ne može biti promijenjena od strane programa. Zato što **1** ne može biti promjenjena sama od sebe, petlja se nikad ne terminira.

U prethodnom dijelu, vidjeli ste da se **break** iskaz može koristiti za izlaz (exit) iz petlje. Bez **break** iskaza, beskonačna petlja bi bila beskorisna. Sa **break**, možete iskoristiti prednosti beskonačnih petlji.

Takođe, možete kreirati beskonačnu **for** petlju, ili beskonačnu **do…while** petlji, kako slijedi:

```
for (;;)
{
    /* additional code goes here */
}
do
{
    /* additional code goes here */
} while (1);
```

Princip ostaje isti za sva tri tipa petlji.
Ovaj dio objašnjava upotrebu **while** petlje.

Beskonačna petlja može biti korištena da se testiraju mnogu uslovi i odluči da li se petlja treba terminirati (prekinuti). Možda će biti teško uključiti sve testne uslove u zagrade nakon while iskaza.
Možda izgleda lakše da se uslovi testiraju pojedinačno u tijelu petlje, i onda izaći, izvršavajući **break**, kada je potrebno.

Beskonačne petlje mogu kreirati meni (manu) sistem koji upravlja operacijama vašeg programa. Možda se sjećate iz Dana 5, “Functions: The Basics,” da programska **main()** funkcija često služi kao neka vrsta “saobraćajnog drota” (“traffic cop”), usmjeravajući izvršenje među različitim funkcijama koje obavljaju stvarni posao programa.
Ovo se često radi pomoću neke vrste meni-a: Korisniku je predstavljena lista izbora i on čini unos neke, birajući neki od njih. Jedan od mogućih izbora bi trebao biti da terminira program. Jednom kada je izbor napravljen, jedna od **C**-ovih iskaza odlučivanja se koristi da usmjeri izvršenje programa prema tom izboru.

Listing **13.4** demonstrira meni (menu) system.

Listing 13.4. Upotreba beskonačne petlje za implementiranje sistema meni-ja.

```
1:  /* Demonstrira beskonacnu petlju ua implementiranje */
2:  /* meni sistema (manu system). */
3:  #include <stdio.h>
4:  #define DELAY  1500000       /* Koristi se u petlji delay. */
5:
6:  int menu(void);
7:  void delay(void);
8:
9:  main()
10: {
11:    int choice;
12:
13:    while (1)
14:    {
15:
16:     /* Uzmi korisnikov izbor. */
17;
18:     choice = menu();
19:
20:     /* Granaj na osnovu unosa (input-a). */
21:
22;     if (choice == 1)
23:     {
24:         puts("\nExecuting choice 1.");
25:         delay();
26:     }
27:     else if (choice == 2)
28:          {
29:              puts("\nExecuting choice 2.");
30:              delay();
31:          }
32;     else if (choice == 3)
33:          {
34:              puts("\nExecuting choice 3.");
35:              delay();
36:          }
37:     else if (choice == 4)
38:          {
39:              puts("\nExecuting choice 4.");
40:              delay();
41:          }
42:     else if (choice == 5)       /* Exit program. */
43:          {
44:              puts("\nExiting program now...\n");
45:              delay();
46:              break;
47:          }
48:          else
49:          {
50:              puts("\nInvalid choice, try again.");
51:              delay();
52:          }
53:    }
```

```
54:    return 0;
55: }
56:
57: /* Prikazi meni i unose korisnikovog izbora. */
58: int menu(void)
59: {
60:     int reply;
61:
62:     puts("\nEnter 1 for task A.");
63:     puts("Enter 2 for task B.");
64:     puts("Enter 3 for task C.");
65:     puts("Enter 4 for task D.");
66:     puts("Enter 5 to exit program.");
67:
68:     scanf("%d", &reply);
69:
70:     return reply;
71: }
72:
73: void delay( void )
74: {
75:     long x;
76:     for ( x = 0; x < DELAY; x++ )
77:         ;
78: }
```

```
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
1
Executing choice 1.
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
6
Invalid choice, try again.
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
5
Exiting program now...
```

ANALIZA: U Listingu **13.4**, funkcija nazvana **menu()** se poziva u liniji **18** i definiše u linijama **58** do **71**. **menu()** prikazuje meni na-ekranu, prihvata korisnikov unos, i vraća unos do **main** programa (glavnom). U **main()**-u, serija ugnijezđenih **if** iskaza testira vraćenu vrijednost i usmjerava izvršenje na osnovu nje. Jedina stvar koju ovaj program radi je prikazivanje meni-a na-ekran. U stvarnom programu, kood bi pozivao razne funkcije da obave određene zadatke.

Ovaj program takođe koristi drugu funkciju nazvanu **delay()**.
delay() je definisana u linijama **73** do **78** i ne radi puno toga. Jednostavno rečeno, **for** iskaz u liniji **76** peetlja, čineći ništa (linija **77**).
Iskaz peelja **DELAY** puta.
Ovo je efektivan metod da se program pauzira momentalno. Ako je pauza (delay) prekratka ili preduga, definisana vrijednost od **DELAY**-a se može podesiti (accordingly).

I Borland i Symantec obezbjeđuju funkciju sličnu **delay()**-u, nazvanu **sleep()**.
Ova funkcija pauzira izvršenje programa za onaj broj sekundi koje su prošle kao njen argument. Da koristite **sleep()**, program mora uključivati file zaglavlja **TIME.H** ako koristite Symantec-ov kompajler.
Morate uključiti **DOS.H** ako koristite Borland-ov kompajler.

Ako koristite bilo koji od ovih kompajlera ili kompajler koji podržava **sleep()**, trebali bi ga koristiti umjesto **delay()**.

UPOZORENJE: Postoje bolji načini da se pauzira kompjuter, nego od onih pokazanih u Listing-u **13.4**. Ako izaberete da koristite funkciju kao što je **sleep()**, kao što je pomenuto, budite oprezni. **sleep()** funkcija nije **ANSI**-kompatibilna. To znači da možda neće raditi sa drugim kompajlerima ili na svim platformama.

→→→→ switch ←←←← iskaz

C-ovi najflexibilniji iskaz kontrole programa je **switch** iskaz, koji dopušta vašem programu izvršenje različitih iskaza na osnovu izraza koji može imati više od dvije vrijednosti.
Raniji kontrolni iskazi, kao što je **if**, je bio ograničen na procjenu izraza koji je mogao imati samo dvije vrijednosti: **true** ili **false**.
Da kontrolišete protok programa, baziran na više od dvije vrijednosti, vi ste morali koristiti višestruke ugnijezđene **if** iskaze, kao što je pokazano u Listing-u **13.4**.
Switch iskaz čini takvo gnijezđenje nepotrebnim.

Generalna (opšta) forma **switch** iskaza je kako slijedi:

```
switch (expression)
{
    case  template_1: statement(s);
    case  template_2: statement(s);
    ...
    case  template_n: statement(s);
    default: statement(s);
}
```

U ovom iskazu, **expression** je bilo koji izraz koji je procijenjen na **integer** vrijednost; tipa **long**, **int** ili **char**. **switch** iskazi procjenjuju **expression** i upoređuju tu vrijednost sa template-ovima koji slijede nakon svake **case** oznake, i onda se jedno od navedenog dešava:

- Ako je par (match) nađen između **expression**-a i jednog od **template**-a, izvršenje se prebacuje na iskaz koji slijedi nakon **case** oznake.
- Ako par (match) nije nađen, izvršenje se prosljeđuje (prebacuje (transferred)) na iskaz koji slijedi nakon **default**-ne opcionalne (izborne) **oznake** (labele).
- Ako par (match) nije nađen, i ako nema **default**-ne **oznake** (labele), izvršenje se prosljeđuje na prvi iskaz koji na nalazi nakon zatvorene zagrade **switch** iskaza.

switch iskaz je demonsriran u Listing-u **13.5**, koji prikazuje poruku na osnovu korisnikovog unosa.
Listing 13.5. Upotreba switch iskaza.

```
1:  /* Demonstracija switch iskaza. */
2:
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:      int reply;
8:
9:      puts("Enter a number between 1 and 5:");
10:     scanf("%d", &reply);
11:
12:     switch (reply)
13:     {
14:         case 1:
15:             puts("You entered 1.");
16:         case 2:
17:             puts("You entered 2.");
18:         case 3:
19:             puts("You entered 3.");
```

```
20:          case 4:
21:              puts("You entered 4.");
22:          case 5:
23:              puts("You entered 5.");
24:          default:
25:              puts("Out of range, try again.");
26:      }
27:
28:      return 0;
29: }
```

```
Enter a number between 1 and 5:
2
You entered 2.
You entered 3.
You entered 4.
You entered 5.
Out of range, try again.
```

ANALIZA: Paa, to sigurno nije tačno, zar ne?
Izgleda kao da **switch** iskaz nalazi prvi odgovarajući template (matching template) i onda izvršava sve što slijedi (ne samo iskaze pridružene template-u).
To je upravo ono šta se dešava. To je kako **switch** iskaz treba da radi.
U stvari, on obavlja **goto** na odgovarajući template (maiching template).
Da biste bili sigurni da se izvršavaju samo iskazi koji su pridruženi odgovarajućem template-u (matching template), uključite (include) **break** iskaz tamo gdje je potreban.
Listing **13.6** pokazuje program koji je ponovo napisan uz upotrebu **break** iskaza.
Sada on radi kako treba

Listing 13.6. Pravilna upotreba switch-a, uključujući break iskaze kao potrebne.

```
1: /* Demonstrira switch iskaz pravilno. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     int reply;
8:
9:     puts("\nEnter a number between 1 and 5:");
10:    scanf("%d", &reply);
11:
12:    switch (reply)
13:    {
14:      case 0:
15:          break;
16       case 1:
17:        {
18:          puts("You entered 1.\n");
19:          break;
20:        }
21:      case 2:
22:        {
23:          puts("You entered 2.\n");
24:          break;
25:        }
26:      case 3:
27:        {
28:          puts("You entered 3.\n");
29:          break;
30:        }
31:      case 4:
32:        {
33:          puts("You entered 4.\n");
34:          break;
```

```
35:          }
36:       case 5:
37:          {
38:            puts("You entered 5.\n");
39:            break;
40:          }
41:       default:
42:          {
43:            puts("Out of range, try again.\n");
44:          }
45:    }              /* Kraj od switch-a */
46:
47: }
```

```
Enter a number between 1 and 5:
1
You entered 1.
Enter a number between 1 and 5:
6
Out of range, try again.
```

Kompajlirajte i pokrenite ovu verziju; radi pravilno.

Jedna od čestih upotreba **switch** iskaza je da se implementira takva vrsta meni-a kao što je pokazano u Listing-u **13.6**.
Listing **13.7** koristi **switch** iskaz umjesto da implementira meni. Upotreba **switch** iskaza je puno bolja od ugnijezđenih **if** iskaza, koji su korišteni u ranijim verzijama meni programa, pokazan u Listing-u **13.4**.

Listing 13.7. Korištenje switch iskaza za izvršenje meni (menu) sistema.

```
1:  /* Demonstrira koristenje beskonacne petlje i switch */
2:  /* iskaza za implementaciju meni sistema (menu system). */
3:  #include <stdio.h>
4:  #include <stdlib.h>
5:
6:  #define DELAY 150000
7:
8:  int menu(void);
9:  void delay(void);
10:
11: main()
12: {
13:
14:    while (1)
15:    {
16:          /* Uzmi korisnikovu selekciju i granaj se na osnovu unosa. */
17:
18:          switch(menu())
19:          {
20:             case 1:
21:                 {
22:                   puts("\nExecuting choice 1.");
23:                   delay();
24:                   break;
25:                 }
26:             case 2:
27:                 {
28:                   puts("\nExecuting choice 2.");
29:                   delay();
30:                   break;
31:                 }
32:             case 3:
33:                 {
34:                   puts("\nExecuting choice 3.");
35:                   delay();
36:                   break;
```

```
37:                 }
38:             case 4:
39:                 {
40:                   puts("\nExecuting choice 4.");
41:                   delay();
42:                   break;
43:                 }
44:             case 5:     /* Exit program. */
45:                 {
46:                   puts("\nExiting program now...\n");
47:                   delay();
48:                   exit(0);
49:                 }
50:             default:
51:                 {
52:                   puts("\nInvalid choice, try again.");
53:                   delay();
54:                 }
55:          }  /* End of switch */
56:     }       /* End of while  */
57:
58: }
59:
60: /* Prikazuje meni (menu) i unose korisnikove selekcije (izbora). */
61: int menu(void)
62: {
63:     int reply;
64:
65:     puts("\nEnter 1 for task A.");
66:     puts("Enter 2 for task B.");
67:     puts("Enter 3 for task C.");
68:     puts("Enter 4 for task D.");
69:     puts("Enter 5 to exit program.");
70:
71:     scanf("%d", &reply);
72:
73:     return reply;
74: }
75:
76: void delay( void )
77: {
78:     long x;
79:     for( x = 0; x < DELAY; x++ )
80:         ;
81: }
```

```
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
1
Executing choice 1.
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
6
Invalid choice, try again.
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
5
```

```
        Exiting program now...
```

Jedan novi iskaz u programu je: → → → **exit()** ← ← ←bibliotečna funkcija u iskazima koji su prifruženi sa **case 5**: u liniji **48**.
Ovdje ne možete koristiti **break**, kao što ste u Listing-u **13.4**. Izvršenje **break**-a bi samo izašlo (**break** out of) iz **switch** iskaza; ne bi izašlo (break out of) beskonačne **while** petlje.
Kao što ćete naučiti u sljedećem dijelu, **exit()** funkcija prekida (terminira (terminates)) program.

Ipak, imajući izvršenje "propad" ("fall through") kao dio **switch** konstrukcije može biti korisno nekad. Recimo, na primjer, da želite isti blok izvršenih iskaza ako se jedna od nekoliko vrijednosti susretne. Jednostavno izostavite **break** iskaze i izlistajte sve **case** template-ove prije iskaza. Ako testni izraz odgovara (matches) bilo kojem od **case** iskaza, izvršenje će "propasti" ("fall through") kroz sljedeće **case** iskaze sve dok ne naiđe blok kooda koji želite da se izvrši.
Ovo je ilustrovano u Listing-u **13.8**.

Listing 13.8. Još jedan način da se korsti switch iskaz.

```
1:  /* Jos jedna upotreba switch iskaza. */
2:
3:  #include <stdio.h>
4:  #include <stdlib.h>
5:
6:  main()
7:  {
8:     int reply;
9:
10:     while (1)
11:     {
12:         puts("\nEnter a value between 1 and 10, 0 to exit: ");
13:         scanf("%d", &reply);
14:
15:         switch (reply)
16:         {
17:             case 0:
18:                 exit(0);
19:             case 1:
20:             case 2:
21:             case 3:
22:             case 4:
23:             case 5:
24:                 {
25:                   puts("You entered 5 or below.\n");
26:                   break;
27:                 }
28:             case 6:
29:             case 7:
30:             case 8:
31:             case 9:
32:             case 10:
33:                 {
34:                   puts("You entered 6 or higher.\n");
35:                   break;
36:                 }
37:             default:
38:                   puts("Between 1 and 10, please!\n");
39:         } /* end of switch */
40:     }     /*end of while */
41:
43: }
```

```
Enter a value between 1 and 10, 0 to exit:
11
Between 1 and 10, please!
Enter a value between 1 and 10, 0 to exit:
1
```

```
You entered 5 or below.
Enter a value between 1 and 10, 0 to exit:
6
You entered 6 or higher.
Enter a value between 1 and 10, 0 to exit:
0
```

ANALIZA: Ovaj program prihvata vrijednosti sa tastature i onda konstatuje da li je vrijednost **5** ili niža, **6** ili viša, ili ako nije između **1** i **10**. Ako je vrijednost **0**, linija **18** izvršava poziv na **exit()** funkciju, što prouzrokuje izlaz iz programa.

→→→ switch iskaz (The switch Statement)

```
switch  (expression)
{
    case  template_1: statement(s);
    case  template_2: statement(s);
    ...
    case  template_n: statement(s);
    default: statement(s);
}
```

switch iskaz dopušta višestruka grananja iz jedinstvenog izraza. Efikasnije je i lakše da pratite nego da višestrukonivoišete (multileveled) **if** iskaze.
switch iskaz procjenjuje izraz i onda granaa na **case** iskaz koji sadrži template, koji odgovara (matching) rezultatu izraza (expression's result). Ako nikakav template ne odgovara (matching) sa rezultatom izraza, kontrola ide na default-ni iskaz. Ako nema default-nog iskaza, kontrola ide na kraj **switch** iskaza.

Took programa nastavlja od **case** iskaza prema dole, osim ako se ne naiđe na **break** iskaz. U tom slučaju, kontrola ide na kraj od **switch** iskaza.

Primjer 1

```
switch( letter )
{
   case `A':
   case `a':
       printf( "You entered A" );
       break;
   case `B':
   case `b':
       printf( "You entered B");
       break;
   ...
   ...
   default:
       printf( "I don't have a case for %c", letter );
}
```

Primjer 2

```
switch( number )
{
   case 0:    puts( "Your number is 0 or less.");
   case 1:    puts( "Your number is 1 or less.");
   case 2:    puts( "Your number is 2 or less.");
   case 3:    puts( "Your number is 3 or less.");
   ...
   ...
   case 99:   puts( "Your number is 99 or less.");
              break;
   default:   puts( "Your number is greater than 99.");
}
```

Zato što nema **break** iskaza za prve **case** iskaze, ovaj primjer pronalazi **case** koji odgovara broju i printa svaki **case** od te tačke prema dole, do **break**-a u **case**-u **99**.
Ako je broj bio **3**, bilo bi vam rečeno da je taj broj jednak sa **3** ili manje, **4** ili manje, sve do **99** ili manje.
Program nastavlja printanje sve dok ne naiđe na **break** iskaz u **case**-u **99**.

NE zaboravite da koristite **break** iskaze, ako ih vaši **switch** iskazi trebaju.
KORSITITE default **case** u **switch** iskazu, čak iako mislite da ste pokrili sve moguće slučajeve (possible cases).
KORSTITE switch iskaz, umjesto **if** iskaza ako se procjenjuju više od dva uslova za istu vrijednost.
POREDAJTE vaše **case** iskaze tako da budu lakši za čitanje.

→→ Izlaženje iz programa (Exiting the Program)

C program se normalno terminira kada izvršenje dođe do zatvorene zagrade od **main()** funkcije. Ipak, vi možete terminirati program u bilo koje vrijeme pozivajući bibliotečnu funkciju **exit()**. Također, možete navesti (specificirati) jednu ili više funkcija da budu automatski izvršene nakon terminacije.

→→→ exit() funkcija (The exit() Function)

exit() funkcija terminira izvršenje programa i vraća kontrolu operativnom sistemu.
Ova funkcija uzima jedan argument tipa **int**, koji se prosljeđuje nazad operativnom sistemu da indicira uspješnost programa ili kvar (program's success or failure).

Syntax-a za **exit()** funkciju je:

```
exit(status);
```

Ako **status** ima vrijednost **0**, ona indicira da se program terminirao (završio) **normalno**.
Vrijednost **1** indicira da je program završio sa nekom vrstom **greške**.
Povratna vrijednost se obično ignoriše. U DOS sistemu, možete testirati povratnu vrijednost sa DOS-ovim batch file-om i sa **if errorlevel** iskazom. Ovo nije knjiga o DOS-u, tako da vam ne treba DOS-ova dokumentacija ako želite da koristite programsku povratnu vrijednost.
Ako koristite operativni sistem koji nije DOS, trebali biste provjeriti njegovu dokumentaciju da odlučite kako da koristite povratnu vrijednost (return value) iz programa.

Da koristite **exit()** funkciju, program mora uključivati file zaglavlja **STDLIB.H**.
Ovaj file zaglavlja također definiše **dvije simbolične konstante** koje se koriste kao argumenti za **exit()** funkciju:

```
#define EXIT_SUCCESS   0
#define EXIT_FAILURE   1
```

Tako da, za izlaz sa povratnom vrijednošću **0**, pozovite izlaz (**EXIT_SUCCESS**); za povratnu vrijednost **1**, pozovite izlaz (**EXIT_FAILURE**).

KORISTITE exit() komandu da izađete iz programa ako postoji problem.
PROSLJEDITE značajnu (meaningful) vrijednosti od **exit()** funkcije.

→→→ Izvršenje komandi operativnog sistema u programu

C-ova standardna biblioteka uključuje funkciju, → → → **system()** ← ← ←.
Ona vam omogućava da izvršavate komande operativnog sistema u izvršavajućem **C**-ovom programu (running **C** program).
Ovo može biti od koristi, dozvoljavajući vam da čitate listinge direktorija s diska, ili formatirate disk bez izlaženja iz programa.
Da koristite **system()** bibliotečnu funkciju, morate uključiti file zaglavlja **STDLIB.H**.

→→→ Format **system()**-a je:

```
system(command);
```

Argument **command** može biti ili string ili pointer na string. Na primjer, da dobijete listing direktorija u DOS-u, možete napisati ili:

```
system("dir");
```

ili

```
char *command = "dir";
system(command);
```

Nakon što se izvrši sistemska komanda, izvršenje se vraća programu na lokaciji koja neposredno slijedi nakon poziva **system()**-a. Ako, komanda koju prosljeđujete, **system()**, nije validna sistemska komanda, dobićete Bad command ili file name error poruku prije pokretanja programa.

Upotreba **system()**-a je ilustrovano u Listing-u **13.9**.

Listing 13.9. Upotreba system() funkcije za izvršenje sistemskih komandi.

```
1:  /* Demonstrira system() funkciju. */
2:  #include <stdio.h>
3:  #include <stdlib.h>
4:
5:  main()
6:  {
7:      /* Deklarisi buffer da drzi unos (input). */
8:
9:      char input[40];
10:
11:     while (1)
12:     {
13:         /* Uzmi (dohvati (Get)) korisnikovu komandu. */
14:
15:         puts("\nInput the desired system command, blank to exit");
16:         gets(input);
17:
18:         /* Izadji, ako je unesena prazna linija. */
19:
20:         if (input[0] == `\0')
21:             exit(0);
22:
23:         /* Izvrsi komandu. */
24:
25:         system(input);
26:     }
27:
28: }
      Input the desired system command, blank to exit
      dir *.bak
      Volume in drive E is BRAD_VOL_B
      Directory of E:\BOOK\LISTINGS
      LIST1414 BAK      1416 05-22-97   5:18p
      1 file(s)       1416 bytes
      240068096 bytes free
      Input the desired DOS command, blank to exit
```

NAPOMENA: dir *.bak je DOS-ova komanda koja izlista file-ove u tekućem direktoriju koji imaju **.BAK** extenziju. Ova komanda takođe radi nad Microsoft Windows-om. Za UNIX mašine, napisali biste **ls *.bak** i dobili sličan rezultat. Ako koristite System 7 ili neki drugi operativni sistem, morali biste napisati odgovarajuće komande operativnog sistema.

ANALIZA: Listing **13.9** ilustruje upotrebu **system()**-a.
Koristeći **while** petlju u linijama **11** do **26**, ovaj program omogućava komande operativnog sistema.
Linije **15** i **16** izbacuju (prompts) da korisnik unese komandu operativnog sistema. Ako korisnik pritisne <**enter**>, bez unošenja komande, linije **20** i **21** zovu **exit()** da prekine program.
Linija **25** poziva **system()** sa unesenom komandom od strane korisnika. Ako pokrenete ovaj program na vašem sistemu, dobićete različit izlaz, naravno.

Komande koje prosljeđujete **system()**-u nisu ograničene na jednostavne operativne komande, kao što je izlistavanje direktorija ili formatiranje diska. Takođe možete prosljediti ime ili neki exekutabilni file ili batch file → i taj se program izvršava normalno.
Na primjer, ako prosljedite argument LIST1308, izvršili biste program koji se zove LIST1308. Kada izađete iz programa, izvršenje se prosljeđuje nazad na gdje je napravljen poziv za **system()**.

Jedino ograničenje na korištenje **system()**-a ima veze sa memorijom.
Kada se izvrši **system()**, oreginalni program ostaje učitan u RAM-u vašeg kompjutera, i nova kopija procesora komande operativnog sistema i bilo koji program koji pokrenete, su takođe učitani (loaded). Ovo radi ako kompjuter ima dovoljno memorije. Ako ne, dobićete error poruku.

Sažetak

Ovo poglavlje je pokrilo razne teme kontrole programa.
Naučili ste o **goto** iskazima i zašto ih treba izbjegavati u vašim programima.
Vidjeli ste da **break** i **continue** iskazi daju dodatnu kontrolu nad izvršenjem petlji i da se ovi iskazi mogu koristiti u konjukciji sa beskonačnim petljama, da obavljaju koriste programerske zadatke.
Ovo poglavlje, takođe, objašnjava kako se koristi **exit()** funkcija za kontrolu programske terminacije.
I, vidjeli ste kako se koristi **system()** funkcija da se izvrše komande operativnog sistema unutar vašeg programa.

P&O

P: Da li je bolje koristiti switch iskaz ili ugnijezđenu petlju?
O: Ako provjeravate varijable koje mogu uzeti (imati) više od dvije vrijednosti, **switch** iskaz je skoro uvijek bolji. Rezultirajući kood je lakši za čitanje, takođe. Ako provjeravate samo **true/false** uslov, koristite **if** iskaz.

P: Zašto treba da izbjegavam goto iskaz???
O: Kada vidite **goto** prvi put, lakše je vjerovati da može biti korista. Ipak, **goto** može prouzrokovati više problema nego što ih može riješiti. **goto** iskaz je nestruktuirana komanda koja vas vodi na drugu tačku programa. Većina debugger-a (software koji vam pomaže da pronađete neki problem) ne može ispitati (interrogate) gogo kako treba. **goto** iskazi takođe dovode ka **spaghetti kooda** → kooda koji se svuda rasteže.

Vježbe: (lekcija 13)

1. Napišite iskaz koji prozrokuje kontrolu programa da ide na sljedeću iteraciju u petlji.
2. Napišite iskaz(e) koji šalju kontrolu programa na kraj petlje.
3. Napišite liniju kooda koji prokazuje listing svih file-ova u tekućem direktoriju (za DOS sistem).
4. BUG BUSTER: Da li nešto nije u redu sa sljedećim koodom?

```
switch( answer )
{
    case `Y': printf("You answered yes");
              break;
    case `N': printf( "You answered no");
}
```

5. BUG BUSTER: Da li nešto nije u redu sa sljedećim koodom?

```
switch( choice )
{
    default:
        printf("You did not choose 1 or 2");
    case 1:
        printf("You answered 1");
        break;
    case 2:
        printf( "You answered 2");
        break;
}
```

6. Prepišite (rewrite) vježbu 5 koristeći **if** iskaze.
7. Napišite beskonačnu **do…while** petlju.
Zbog obilnosti mogućih rješenja za sljedeće vježbe, odgovori nisu dati. Ovo su vježbe koje treba da uradite sami.
8. ON YOUR OWN: Napišite program koji radi kao **calculator**. Program bi također trebao dozvoljavati sabiranje, oduzimanje, množenje i djeljenje.
9. ON YOUR OWN: Napišite program koji obezbjeđuje meni sa pet različitih izbora. Peta opcija bi trebala biti izlaz iz programa. Svaka od ostalih opcija bi trebala izvršavati sistemske komande koristeći **system()** funkciju.

→→→→→ Rezervisane ključne riječi u C-u.

Ključna riječ	OPIS
asm	Ključna riječ koja navodi u-linijijsku (inline) assembly language kood.
auto	Default-na smještajna klasa (storage class).
break	Komanda koja postoji za, **while**, **switch**, i **do...while** iskaze bezuslovno.
case	Komanda korištena unutar switch iskaza.
char	Najjednostavniji C-ov tip podataka.
const	Modifikator podataka koji sprječava da se varijabla promjeni. Vidi volatile.
continue	Komanda koja resetuje **for**, **while**, ili **do...while** iskaz do sljedeće iteracije.
default	Komanda korištena unutar **switch** iskaza da uhvati bilo koje instance koje nisu navedene sa **case** iskazom.
Do	Peetljajuća komanda korištena u konjukciji sa **while** iskazom. Petlja će se uvijek izvršiti bar jednom.
double	Tip podataka koji može držati vrijednosti duple-preciznosti floating-point-a.
else	Iskaz koji signalizira da se izvrše alternativni iskazi kada se **if** iskaz procjeni na **FALSE**.
enum	Tip podataka koji dozvoljava varijablama da budu deklarisane da prihvataju samo određene vrijednosti.
extern	Modifikator podataka koji pokazuje da će varijabla biti deklarisana u nekom drugom dijelu programa.
float	Tip podataka korišten za **float**ing-point brojeve.
for	Peetljajuća komanda koja sadrži inicijalizaciju, inkrementaciju, I uslovne dijelove.
goto	Komanda koja uzrokuje skok na predefinisanu labelu (oznaku).
if	Komanda korištena da se promjeni took programa na osnovu TRUE/FALSE odluke.
int	Tip podataka korišten da drži **int**eger vrijednosti.
long	Tip podataka korišten da drži veće integer vrijednosti od int-a..
register	Smještajni modifikator koji navodi da bi varijabla trebala biti smještena u registar, ako je moguće.
return	Komanda koja uzrokuje da programski took izađe iz tekuće funkcije i vrati se pozivajućem programu. Također se može koristiti da vrati jednu (single) vrijednost.
short	Tip podataka korišten da drži integere. Ne koristi se uobičajeno, i iste je veličine kao i int na većini kompjutera.
signed	Modifikator korišten da označi da varijabla može imati obje, i pozitivne i negativne vrijednosti. Pogledaj unsigned.
sizeof	Operator koji vraća veličinu predmeta u byte-ovima.
static	Modifikator korišten da naznači da bi kompajler trebao rezervisati (sačuvati (zapamtiti (retain))) vrijednost varijable.
struct	Ključna riječ korištena da kombinuje **C**-ove varijable bilo kojeg tipa u grupu.
switch	Komanda korištena da promjeni took programa u mnoštvu smjerova. Korišten u konjukciji sa **case** iskazom.
typedef	Modifikator korišten da kreira nova imena za postojeću varijablu i tipove funkcija.
union	Ključna riječ korištena da dozvoli da više varijabli dijeli isti memorijski prostor.
unsigned	Modifikator korišten da naznači da će varijabla sadržavati samo pozitivne vrijednosti. Vidi signed.
void	Ključna riječ korištena na naznači ili da funkcija ne vraća ništa ili da je pointer koji se koristi smatra generički ili da je u mogućnosti da pokazuje na bilo koji tip podataka.
volatile	Modifikator koji naznačava da varijabla može biti promjenjena. Vidi const.
while	Peetljajući iskaz koji izvršava sekciju (dio) kooda sve dok uslov ostaje TRUE.

S dodatkom na prethodne ključne riječi, sljedeće su **C++-ove rezervisane riječi**:

catch	**inline**	**template**
class	**new**	**this**
delete	**operator**	**throw**
except	**private**	**try**
finally	**protected**	**virtual**
friend	**public**	

Funkcija	Opis
isalnum()	Provjerava da vidi da li je karakter alphanumeric.
isalpha()	Provjerava da vidi da li je karakter alphabetic.
iscntrl()	Provjerava da vidi da li je karakter kontrolni karakter.
isdigit()	Provjerava da vidi da li je karakter decimalna cifra.
isgraph()	Provjerava da vidi da li je karakter printabilan (space je izuzetak).
islower()	Provjerava da vidi da li je karakter malo slovo (lowercase).
isprint()	Provjerava da vidi da li je karakter printabilan.
ispunct()	Provjerava da vidi da li je karakter punctuation character.
isspace()	Provjerava da vidi da li je karakter whitespace character.
isupper()	Provjerava da vidi da li je karakter uppercase (veliko slovo).
isxdigit()	Provjerava da vidi da li je karakter hexadecimalna cifra (digit).

Veličina Tipova Varijabli:
==========
char 1
short 2
int 2
float 4
double 8
unsigned char 1
unsigned short 2
unsigned int 2

NE koristie numeričke vrijednosti kada određujete maximume za varijable.
KORISTITE definisane konstante ako pišete portabilan program.
KORISTITE karakternu klasifikaciju funkcija kad god je to moguće.
ZAPAMTITE da se **"!="** smatra kao provjera jednakosti.
UPOZORENJE: Morate biti oprezni kada koristite karakterne numerike. Karakterni numerici možda nisu portabilni.
NAPOENA: Većina kompajlera sa integrisanim **IDE**-ovima (Development Environments obezbjeđuju **ANSI** opciju. Odabirajući **ANSI** opciju, vi ste zagarantovali **ANSI** kompatibilnost.
KORISTITE više nego samo **"Case"** da razdvojite imena varijabli.
NE pretpostavljajte numeričke vrijednosti za karaktere.

ANSI standardne preprocessor-ske direktive.

#define	**#if**
#elif	**#ifdef**
#else	**#ifndef**
#endif	**#include**

#error	#pragma

```
1:  /*=================================================*
2:   * Program: listD07.c                              *
3:   * Purpose: This program demonstrates using defined     *
4:   *          constants for creating a portable program.  *
5:   * Note:    This program gets different results with    *
6:   *          different compilers.                        *
7:   *=================================================*/
8:  #include <stdio.h>
9:  #ifdef _WINDOWS
10:
11: #define STRING "DOING A WINDOWS PROGRAM!\n"
12:
13: #else
14:
15: #define STRING "NOT DOING A WINDOWS PROGRAM\n"
16:
17: #endif
18:
19: int main(void)
20: {
21:    printf( "\n\n") ;
22:    printf( STRING );
23:
24: #ifdef _MSC_VER
25:
26:    printf( "\n\nUsing a Microsoft compiler!" );
27:    printf( "\n   Your Compiler version is %s\n", _MSC_VER );
28:
29: #endif
30:
31: #ifdef __TURBOC__
32:
33:    printf( "\n\nUsing the Turbo C compiler!" );
34:    printf( "\n   Your compiler version is %x\n", __TURBOC__ );
35:
36: #endif
37:
38: #ifdef __BORLANDC__
39:
40:    printf( "\n\nUsing a Borland compiler!\n" );
41:
42: #endif
43:
44:    return(0);
45: }
```

Evo izlaza kojeg ćete vidjeti kada pokrenete ovaj program koristeći **Turbo C for DOS 3.0 kompajler**:
NOT DOING A WINDOWS PROGRAM
Using the Turbo C compiler!
  Your compiler version is 300

Evo izlaza kojeg ćete vidjeti kada pokrenete ovaj program koristeći **Borland C++ compiler under DOS**:
NOT DOING A WINDOWS PROGRAM
Using a Borland compiler!

Evo izlaza kojeg ćete vidjeti kada pokrenete ovaj program koristeći **Microsoft compiler under DOS**:
NOT DOING A WINDOWS PROGRAM
Using a Microsoft compiler!
  Your compiler version is >>

→→→→ Najčešće C Funkcije ←←←←

Ovaj dodatak iznosi prototipe funkcija koje su sadržane u svakom od file-u zaglavlja (**header files**) obezbjeđeni sa većinom **C** kompajlera.
Funkcije koje imaju zvjezdicu poslije imena su pokrivene u ovom priručniku.
Funkcije su sortirane alfabetskim redosljedom. Poslije svakog imena i "header file"-a su kompletni prototipi.

Primjetite da prototipi file-ova zaglavlja koriste notaciju koja je drugačija od onih korištenih u ovom priručniku. Za svaki parametar koji funkcija uzima, samo je tip dat u prototipu, nijedno ime parametra nije uključeno. Evo dva primjera:

```
int func1(int, int *);
int func1(int x, int *y);
```

obadvije deklaracije specificiraju dva parametra → **prvi tipa int**, i **drugi** → **pointer na tip int**. što se tiče kompajlera, ove dvije deklaracije su ekvivalentne.

Najčešće C funkcije izlistane u alfabetskom redosljedu:

Funkcija	File zaglavlja	Prototip funkcije
abort*	STDLIB.H	void abort(void);
Abs	STDLIB.H	int abs(int);
acos*	MATH.H	double acos(double);
asctime*	TIME.H	char *asctime(const struct tm *);
asin*	MATH.H	double asin(double);
assert*	ASSERT.H	void assert(int);
atan*	MATH.H	double atan(double);
atan2*	MATH.H	double atan2(double, double);
atexit*	STDLIB.H	int atexit(void (*)(void));
atof*	STDLIB.H	double atof(const char *);
atof*	MATH.H	double atof(const char *);
atoi*	STDLIB.H	int atoi(const char *);
atol*	STDLIB.H	long atol(const char *);
bsearch*	STDLIB.H	void *bsearch(const void *, const void *, size_t, size_t, int(*) (const void *, const void *));
calloc*	STDLIB.H	void *calloc(size_t, size_t);
ceil*	MATH.H	double ceil(double);
clearerr	STDIO.H	void clearerr(FILE *);
clock*	TIME.H	clock_t clock(void);
cos*	MATH.H	double cos(double);
cosh*	MATH.H	double cosh(double);
ctime*	TIME.H	char *ctime(const time_t *);
difftime	TIME.H	double difftime(time_t, time_t);
div	STDLIB.H	div_t div(int, int);
exit*	STDLIB.H	void exit(int);
exp*	MATH.H	double exp(double);
fabs*	MATH.H	double fabs(double);
fclose*	STDIO.H	int fclose(FILE *);
fcloseall*	STDIO.H	int fcloseall(void);
feof*	STDIO.H	int feof(FILE *);

fflush*	STDIO.H	int fflush(FILE *);
fgetc*	STDIO.H	int fgetc(FILE *);
fgetpos	STDIO.H	int fgetpos(FILE *, fpos_t *);
fgets*	STDIO.H	char *fgets(char *, int, FILE *);
floor*	MATH.H	double floor(double);
flushall*	STDIO.H	int flushall(void);
fmod*	MATH.H	double fmod(double, double);
fopen*	STDIO.H	FILE *fopen(const char *, const char *);
fprintf*	STDIO.H	int fprintf(FILE *, const char *, ...);
fputc*	STDIO.H	int fputc(int, FILE *);
fputs*	STDIO.H	int fputs(const char *, FILE *);
fread*	STDIO.H	size_t fread(void *, size_t, size_t, FILE *);
free*	STDLIB.H	void free(void *);
freopen	STDIO.H	FILE *freopen(const char *, const char *, FILE *);
frexp*	MATH.H	double frexp(double, int *);
fscanf*	STDIO.H	int fscanf(FILE *, const char *, ...);
fseek*	STDIO.H	int fseek(FILE *, long, int);
fsetpos	STDIO.H	int fsetpos(FILE *, const fpos_t *);
ftell*	STDIO.H	long ftell(FILE *);
fwrite*	STDIO.H	size_t fwrite(const void *, size_t, size_t, FILE *);
getc*	STDIO.H	int getc(FILE *);
getch*	STDIO.H	int getch(void);
getchar*	STDIO.H	int getchar(void);
getche*	STDIO.H	int getche(void);
getenv	STDLIB.H	char *getenv(const char *);
gets*	STDIO.H	char *gets(char *);
gmtime	TIME.H	struct tm *gmtime(const time_t *);
isalnum*	CTYPE.H	int isalnum(int);
isalpha*	CTYPE.H	int isalpha(int);
isascii*	CTYPE.H	int isascii(int);
iscntrl*	CTYPE.H	int iscntrl(int);
isdigit*	CTYPE.H	int isdigit(int);
isgraph*	CTYPE.H	int isgraph(int);
islower*	CTYPE.H	int islower(int);
isprint*	CTYPE.H	int isprint(int);
ispunct*	CTYPE.H	int ispunct(int);
isspace*	CTYPE.H	int isspace(int);
isupper*	CTYPE.H	int isupper(int);
isxdigit*	CTYPE.H	int isxdigit(int);
labs	STDLIB.H	long int labs(long int);
ldexp	MATH.H	double ldexp(double, int);
ldiv	STDLIB.H	ldiv_t div(long int, long int);
localtime*	TIME.H	struct tm *localtime(const time_t *);
log*	MATH.H	double log(double);
log10*	MATH.H	double log10(double);
malloc*	STDLIB.H	void *malloc(size_t);

mblen	STDLIB.H	int mblen(const char *, size_t);
mbstowcs	STDLIB.H	size_t mbstowcs(wchar_t *, const char *, size_t);
mbtowc	STDLIB.H	int mbtowc(wchar_t *, const char *, size_t);
memchr	STRING.H	void *memchr(const void *, int, size_t);
memcmp	STRING.H	int memcmp(const void *, const void *, size_t);
memcpy	STRING.H	void *memcpy(void *, const void *, size_t);
memmove	STRING.H	void *memmove(void *, const void*, size_t);
memset	STRING.H	void *memset(void *, int, size_t);
mktime*	TIME.H	time_t mktime(struct tm *);
modf	MATH.H	double modf(double, double *);
perror*	STDIO.H	void perror(const char *);
pow*	MATH.H	double pow(double, double);
printf*	STDIO.H	int printf(const char *, ...);
putc*	STDIO.H	int putc(int, FILE *);
putchar*	STDIO.H	int putchar(int);
puts*	STDIO.H	int puts(const char *);
qsort*	STDLIB.H	void qsort(void*, size_t, size_t, int (*)(const void*, const void *));
rand	STDLIB.H	int rand(void);
realloc*	STDLIB.H	void *realloc(void *, size_t);
remove*	STDIO.H	int remove(const char *);
rename*	STDIO.H	int rename(const char *, const char *);
rewind*	STDIO.H	void rewind(FILE *);
scanf*	STDIO.H	int scanf(const char *, ...);
setbuf	STDIO.H	void setbuf(FILE *, char *);
setvbuf	STDIO.H	int setvbuf(FILE *, char *, int, size_t);
sin*	MATH.H	double sin(double);
sinh*	MATH.H	double sinh(double);
sleep*	TIME.H	void sleep(time_t);
sprintf	STDIO.H	int sprintf(char *, const char *, ...);
sqrt*	MATH.H	double sqrt(double);
srand	STDLIB.H	void srand(unsigned);
sscanf	STDIO.H	int sscanf(const char *, const char *, ...);
strcat*	STRING.H	char *strcat(char *,const char *);
strchr*	STRING.H	char *strchr(const char *, int);
strcmp*	STRING.H	int strcmp(const char *, const char *);
strcmpl*	STRING.H	int strcmpl(const char *, const char *);
strcpy*	STRING.H	char *strcpy(char *, const char *);
strcspn*	STRING.H	size_t strcspn(const char *, const char *);
strdup*	STRING.H	char *strdup(const char *);
strerror	STRING.H	char *strerror(int);
strftime*	TIME.H	size_t strftime(char *, size_t, const char *, const struct tm *);
strlen*	STRING.H	size_t strlen(const char *);
strlwr*	STRING.H	char *strlwr(char *);
strncat*	STRING.H	char *strncat(char *, const char *, size_t);
strncmp*	STRING.H	int strncmp(const char *, const char *, size_t);
strncpy*	STRING.H	char *strncpy(char *, const char *, size_t);

strnset*	STRING.H	char *strnset(char *, int, size_t);
strpbrk*	STRING.H	char *strpbrk(const char *, const char *);
strrchr*	STRING.H	char *strrchr(const char *, int);
strspn*	STRING.H	size_t strspn(const char *, const char *);
strstr*	STRING.H	char *strstr(const char *, const char *);
strtod	STDLIB.H	double strtod(const char *, char **);
strtok	STRING.H	char *strtok(char *, const char*);
strtol	STDLIB.H	long strtol(const char *, char **, int);
strtoul	STDLIB.H	unsigned long strtoul(const char*, char **, int);
strupr*	STRING.H	char *strupr(char *);
system*	STDLIB.H	int system(const char *);
tan*	MATH.H	double tan(double);
tanh*	MATH.H	double tanh(double);
time*	TIME.H	time_t time(time_t *);
tmpfile	STDIO.H	FILE *tmpfile(void);
tmpnam*	STDIO.H	char *tmpnam(char *);
tolower	CTYPE.H	int tolower(int);
toupper	CTYPE.H	int toupper(int);
ungetc*	STDIO.H	int ungetc(int, FILE *);
va_arg*	STDARG.H	(type) va_arg(va_list, (type));
va_end*	STDARG.H	void va_end(va_list);
va_start*	STDARG.H	void va_start(va_list, lastfix);
vfprintf	STDIO.H	int vfprintf(FILE *, constchar *, ...);
vprintf	STDIO.H	int vprintf(FILE*, constchar *, ...);
vsprintf	STDIO.H	int vsprintf(char *, constchar *, ...);
wcstombs	STDLIB.H	size_t wcstombs(char *, const wchar_t *, size_t);
wctomb	STDLIB.H	int wctomb(char *, wchar_t);

(IDE) -> integrated development environment

Tabela C.1. Hexadecimalni brojevi I njihovi decimalni I binarni ekvivalenti.

Hex. Digit	Dec. Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111
10	16	10000
F0	240	11110000
FF	255	11111111

Tabela D.1. ANSI C ključne riječi.

asm	**auto**	**break**
case	**char**	**const**
continue	**default**	**do**
double	**else**	**enum**
extern	**float**	**for**
goto	**if**	**int**
long	**register**	**return**
short	**signed**	**sizeof**
static	**struct**	**switch**
typedef	**union**	**unsigned**
void	**volatile**	**while**

Tabela D.2. Moguće vrijednosti na osnovu veličine byte-a.

Broj byte-ova	Unsigned	Signed	Signed
Integral Types	Maximum	Minimum	Maximum
1	255	-128	127
2	65,535	-32,768	32,767
4	4,294,967,295	-2,147,483,648	2,147,438,647
8		1.844674 * E19	

Table D.3. ANSI-definisane konstante unutar LIMITS.H.

Konstanta	Vrijednost
CHAR_BIT	Character variable's number of bits.
CHAR_MIN	Character variable's minimum value (signed).
CHAR_MAX	Character variable's maximum value (signed).
SCHAR_MIN	Signed character variable's minimum value.
SCHAR_MAX	Signed character variable's maximum value.
UCHAR_MAX	Unsigned character's maximum value.
INT_MIN	Integer variable's minimum value.
INT_MAX	Integer variable's maximum value.
UINT_MAX	Unsigned integer variable's maximum value.
SHRT_MIN	Short variable's minimum value.
SHRT_MAX	Short variable's maximum value.
USHRT_MAX	Unsigned short variable's maximum value.
LONG_MIN	Long variable's minimum value.
LONG_MAX	Long variable's maximum value.
ULONG_MAX	Unsigned long variable's maximum value.

Table D.4. ANSI-definisane konstante unutar FLOAT.H.

Konstanta	Vrijednost
FLT_DIG	Precision digits in a variable of type float.
DBL_DIG	Precision digits in a variable of type double.
LDBL_DIG	Precision digits in a variable of type long double.
FLT_MAX	Float variable's maximum value.
FLT_MAX_10_EXP	Float variable's exponent maximum value (base 10).
FLT_MAX_EXP	Float variable's exponent maximum value (base 2).
FLT_MIN	Float variable's minimum value.
FLT_MIN_10_EXP	Float variable's exponent minimum value (base 10).
FLT_MIN_EXP	Float variable's exponent minimum value (base 2).
DBL_MAX	Double variable's maximum value.
DBL_MAX_10_EXP	Double variable's exponent maximum value (base 10).
DBL_MAX_EXP	Double variable's exponent maximum value (base 2).
DBL_MIN	Double variable's minimum value.
DBL_MIN_10_EXP	Double variable's exponent minimum value (base 10).
DBL_MIN_EXP	Double variable's exponent minimum value (base 2).
LDBL_MAX	Long double variable's maximum value.
LDBL_MAX_10_DBL	Long double variable's exponent maximum value (base 10).
LDBL_MAX_EXP	Long double variable's exponent maximum value (base 2).
LDBL_MIN	Long double variable's minimum value.
LDBL_MIN_10_EXP	Long double variable's exponent minimum value (base 10).
LDBL_MIN_EXP	Long double variable's exponent minimum value (base 2).