Mala škola referenciranja u programskom jeziku C

Pokazivači i adrese

Memorija računara organizovana je u niz uzastopnih bajtova. Uzastopni bajtovi mogu se tretirati kao jedinstven podatak. Na primer, dva (ili četiri, u zavisnosti od sistema) uzastopna bajta mogu se tretirati kao jedinstven podatak celobrojnog tipa.

Pokazivači predstavljaju tip podataka u C-u takav da su vrednosti ovog tipa memorijske adrese. **U zavisnosti od sistema, adrese obično zauzimaju četiri (ranije dva, a novije vreme i osam) bajta.**

Pokazivačke promenljive (promenljive pokazivačkog tipa) su promenljive koje sadrže memorijske adrese. Iako su pokazivačke vrednosti (adrese) u suštini celi brojevi, pokazivački tipovi se striktno razlikuju od celobrojnih.

Takođe, jezik C razlikuje više pokazivačkih tipova i tip pokazivača se određuje na osnovu tipa podataka na koji pokazuje. **Ovo znači da pokazivači implicitno čuvaju informaciju o tipu onoga na šta ukazuju, sa izuzetkom pokazivača tipa void koji nema informaciju o tipu podataka na koji ukazuje.**

Tip pokazivača koji ukazuje na podatak tipa int zapisuje se int *. Slično važi i za druge tipove. **Prilikom deklaracije, nije bitno da li postoji razmak između zvezdice i tipa ili zvezdice i identifikatora i kako god da je napisano, zvezdica se vezuje uz identifikator, što je čest izvor grešaka.**

```
int *p1;
int* p2;
int* p3, p4;
```

Dakle, u ovom primeru, p1, p2 i p3 su pokazivači koji ukazuju na int dok je p4 običan int.

Kako bi pokazivačka promenljiva sadržala adresu nekog smislenog podatka potrebno je programeru dati mogućnost odredivanja adresa objekata.

Unarni operator & (koji zovemo „operator referenciranja“ ili „adresni operator”) vraća adresu svog operanda. On može biti primenjen samo na promenljive i elemente, a ne i na izraze ili konstante.

Na primer, ukoliko je naredni kôd deo neke funkcije

```
int a=10, *p;
p = &a;
```

onda se prilikom izvršavanja te funkcije, za promenljive a i p rezerviše prostor u njenom stek okviru. U prostor za a se upisuje vrednost 10, a u prostor za p adresa promenljive a. Za promenljivu p tada kažemo da „pokazuje“ na a.

Unarni operator * (koji zovemo „operator dereferenciranja“) se primenjuje na pokazivačku promenljivu i vraća sadržaj lokacije na koju ta promenljiva pokazuje, vodeći računa o tipu.
Simbol * koristi i za označavanje pokazivačkih tipova i za operator dereferenciranja i poželjno je jasno razlikovanje ove njegove dve različite uloge.

Derefencirani pokazivač može biti l-vrednost i u tom slučaju izmene dereferenciranog pokazivača utiču neposredno na prostor na koji se ukazuje. Ukoliko pokazivač ukazuje na neku promenljivu, posredno se menja i njen sadržaj. Na primer, nakon dodela

```
int a=10, *p;
p = &a;
*p = 5;
```

promenljiva p ukazuje na a, a u lokaciju na koju ukazuje p je upisana vrednost 5. Time je i vrednost promenljive a postala 5.

Dereferencirani pokazivač nekog tipa, može se pojaviti u bilo kom kontekstu u kojem se može pojaviti podatak tog tipa. Na primer, u datom primeru ispravna bi bila i naredba *p = *p+a+3.

Još jednom naglasimo da pokazivački i celobrojni tipovi različiti.
Tako je, na primer, ako je data deklaracija
int *pa, a;,
naredni kôd neispravan pa = a;.
Takođe, neispravno je i a = pa;, kao i pa = 1234.

Moguće je koristiti eksplicitne konverzije (na primer a = (int) pa; ili pa = (int*)a;), ali ovo treba oprezno koristiti. **Jedini izuzetak je 0 koja se može tumačiti i kao ceo broj i kao pokazivač. Tako je moguće dodeliti nulu pokazivačkoj promenljivoj i porediti pokazivač sa nulom. Simbolička konstanta NULL se često koristi umesto nule, kao jasniji indikator da je u pitanju specijalna pokazivačka vrednost. NULL je definisana u zaglavlju <stdio.h> i ubuduće ćemo uvek koristiti NULL kao vrednost za pokazivač koji ne pokazuje ni na šta smisleno. Pokazivač koji ima vrednost NULL nije moguće dereferencirati. Pokušaj dereferenciranja dovodi do greške tokom izvršavanja programa (najčešće "segmentation fault").**

Generički pokazivački tip (void*):
U nekim slučajevima, poželjno je imati mogućnost "opšteg" pokazivača, tj. pokazivača koji može da ukazuje na promenljive različitih tipova. Za to se koristi tip void*. Izraze ovog tipa je moguće eksplicitno konvertovati u bilo koji konkretni pokazivački tip (čak se u C-u, za razliku od C++-a, vrši i implicitna konverzija prilikom dodele). Međutim, na ravno, nije moguće vršiti dereferenciranje pokazivača tipa void* jer nije moguće odrediti tip takvog izraza kao ni broj bajtova u memoriji koji predstavljaju njegovu vrednost.

PRIMERI:
1. Sta je rezultat rada sledećeg programa?
Primer koji ilustruje rad sa pokazivacima

```
#include <stdio.h>
main()
{
int x = 3;

/* Adresu promenjive x zapamticemo u novoj promenljivoj.
Nova promenljiva je tipa pokazivaca na int (int*) */
int* px;
printf("Adresa promenljive x je : %p\n", &x);
printf("Vrednost promenljive x je : %d\n", x);

px = &x;
printf("Vrednost promenljive px je (tj. px) : %p\n", px);
printf("Vrednost promenljive na koju ukazuje px (tj. *px) je : %d\n", *px);

/* Menjamo vrednost promenljive na koju ukazuje px */
*px = 6;
printf("Vrednost promenljive na koju ukazuje px (tj. *px) je : %d\n", *px);

/* Posto px sadrzi adresu promenljive x, ona ukazuje na x tako da je
posredno promenjena i vrednost promenljive x */
printf("Vrednost promenljive x je : %d\n", x);
}
```

Izlaz (u konkretnom slucaju):
Adresa promenljive x je : 0012FF88
Vrednost promenljive x je : 3
Vrednost promenljive px je (tj. px) : 0012FF88
Vrednost promenljive na koju ukazuje px (tj. *px) je : 3
Vrednost promenljive na koju ukazuje px (tj. *px) je : 6
Vrednost promenljive x je : 6

2. Sta je rezultat rada sledećeg programa?
Primer koji ilustruje prenos argumenata preko pokazivaca – funkcija koja vrsi trampu vrednosti dva cela broja.
Podsetimo se:
U programskom jeziku C parametri se mogu predati funkciji po _____ i po _____.
Kada god je potrebno da funkcija menja vrednost svojih stvarnih parametara i tako izmenjene ih vrati pozivajucoj jedinici, onda se ti parametri predaju funkciji po _____.
Kada god je potrebno da funkcija svoje stvarne parametre vrati pozivajucoj jedinici nepromenjene, onda se ti parametri predaju funkciji po _____.
Kada god je potrebno da funkcija vrati pozivajucoj jedinici vise rezultata, onda se ti rezultati prenose kao parametri funkcije i predaju funkciji po _____.

```
#include <stdio.h>

/* Pogresna verzija funkcije swap. Zbog prenosa po vrednosti, funkcija
razmenjuje kopije promenljivih iz main-a, a ne samih promenljivih */
void swap_sa_greskom(int x, int y)
{
int tmp;
printf("swap_sa_greskom: ");
printf("Funkcija menja vrednosti promenljivim na adresama : \n");
printf("x : %p\n", &x);
printf("y : %p\n", &y);
tmp = x;
x = y;
y = tmp;
}

/* Resenje je prenos argumenata preko pokazivaca */
void swap(int* px, int* py)
{
int tmp;
printf("swap : Funkcija menja vrednosti promenljivim na adresama : \n");
printf("px = %p\n", px);
printf("py = %p\n", py);
tmp = *px;
*px = *py;
*py = tmp;
}

main()
{
int x = 3, y = 5;
printf("Adresa promenljive x je %p\n", &x);
printf("Vrednost promenljive x je %d\n", x);
printf("Adresa promenljive y je %p\n", &y);
printf("Vrednost promenljive y je %d\n", y);

/* Pokusavamo zamenu koristeci pogresnu verziju funkcije */
swap_sa_greskom(x, y);
printf("Posle swap_sa_greskom:\n");
printf("Vrednost promenljive x je %d\n", x);
printf("Vrednost promenljive y je %d\n", y);

/* Vrsimo ispravnu zamenu. Funkciji swap saljemo adrese promenljvih x i y, a ne njihove vrednosti */
swap(&x, &y);
```

```
printf("Posle swap:\n");
printf("Vrednost promenljive x je %d\n", x);
printf("Vrednost promenljive y je %d\n", y);
}
```

Izlaz u konkretnom slucaju:
Adresa promenljive x je 0012FF88
Vrednost promenljive x je 3
Adresa promenljive y je 0012FF84
Vrednost promenljive y je 5
swap_sa_greskom: Funkcija menja vrednosti promenljivim na adresama :
x : 0012FF78
y : 0012FF7C
Posle swap_wrong:
Vrednost promenljive x je 3
Vrednost promenljive y je 5
swap : Funkcija menja vrednosti promenljivim na adresama :
px = 0012FF88
py = 0012FF84
Posle swap:
Vrednost promenljive x je 5
Vrednost promenljive y je 3

Prenos parametara po adresi u potprogramu

Prenos pokazivača može da se iskoristi i da funkcija vrati više različitih vrednosti (preko argumenata).

PRIMER:
3. Napisati C potprogram div_and_mod koji kao rezultat vraca celobrojni deo kolicnika i ostatak pri deljenju dva cela broja x i y. NCP koji ilustruje rad sa potprogramom.

Primer koji ilustruje rad sa vise povratnih vrednosti funkcije koristeci prenos argumenata preko adrese (tj. pokazivace).

```
/* Funkcija istovremeno vraca dve vrednosti - kolicnik i ostatak dva data broja.
Ovo se postize tako sto se funkciji predaju vrednosti dva broja (x i y) koji se dele
i adrese dve promenljive na koje ce se smestiti rezultati */

void div_and_mod(int x, int y, int* div, int* mod)
{
printf("Kolicnik postavljam na adresu : %p\n", div);
printf("Ostatak postavljam na adresu : %p\n", mod);
*div = x / y;
*mod = x % y;
}

main()
{
int div, mod;
printf("Adresa promenljive div je %p\n", &div);
printf("Adresa promenljive mod je %p\n", &mod);

/* Pozivamo funkciju tako sto joj saljemo vrednosti dva broja (5 i 2)
i adrese promenljvih div i mod na koje ce se postaviti rezultati */
div_and_mod(5, 2, &div, &mod);
```

```
printf("Vrednost promenljive div je %d\n", div);
printf("Vrednost promenljive mod je %d\n", mod);
}
```

Izlaz u konkretnom slucaju:
Adresa promenljive div je 0012FF88
Adresa promenljive mod je 0012FF84
Kolicnik postavljam na adresu : 0012FF88
Ostatak postavljam na adresu : 0012FF84
Vrednost promenljive div je 2
Vrednost promenljive mod je 1

Pokazivači i nizovi

Postoji čvrsta veza pokazivača i nizova. Operacije nad nizovima mogu se iskazati i korišćenjem pokazivača i sve češće nećemo praviti razliku izmedu dva načina za pristupanje elementima niza.

Deklaracije int a[10] deklariše niz veličine 10. Početni element je a[0], a deseti element je a[9] i oni su poredani uzastopno u memoriji.
Imenu niza a pridružena je informacija o adresi početnog elementa niza, o tipu elemenata niza, kao i o broju elemenata niza. Ime niza a nije l-vrednost (jer a uvek ukazuje na isti prostor koji je rezervisan za elemente niza). Dakle, vrednost a nije pokazivačkog tipa, ali mu je vrlo bliska.

Nakon deklaracije int a[10], vrednosti a odgovara pokazivač na prvi element niza, vrednosti a+1 odgovara pokazivač na drugi element niza, itd. Dakle, umesto a[i] može se pisati *(a+i), a umesto &a[i] može se pisati a+i.
Naravno, kao i obično, nema provere granica niza, pa je dozvoljeno pisati (tj. prolazi fazu prevođenja) i a[100], *(a+100), a[-100], *(a-100), iako je veličina niza samo 10.
U fazi izvršavanja, pristupanje ovim lokacijama može da promeni vrednost drugih promenljivih koje se nalaze na tim lokacijama ili da dovede do prekida rada programa zbog pristupanja memoriji koja mu nije dodeljena.

Nakon deklaracije int a[10];,
nazivu niza a pridružena je informacija o tipu elemenata niza, ali i o njihovom tipu. Ove informacije biće iskorišćene tokom kompilacije i neće zauzimati memorijski prostor u fazi izvršavanja.
Kao što je rečeno, izraz a ima vrednost adrese početnog elementa i tip int [10] koji se, po potrebi, može konvertovati u int *.
Izraz &a ima istu vrednost kao i a, ali drugačiji tip, tip int (*)[10], a to je tip pokazivača na niz od 10 elemenata koji su tipa int. Taj tip ima u narednom primeru promenljiva b:

```
char a[10];
char (*b)[10];
char *c[10];
```

U navedenom primeru, promenljiva c je niz 10 pokazivača na char. S druge strane, promenljiva b je pokazivač na niz od 10 elemenata tipa char i zauzima onoliko prostora koliko i bilo koji drugi pokazivač. Vrednost se promenljivoj b može pridružiti, na primer, naredbom b=&a;. Tada je (*b)[0] isto što i a[0].
Ukoliko je promenljiva b deklarisana, na primer, sa char (*b)[5];, u dodeli b=&a; će biti izvršena implicitna konverzija i ova naredba će proći kompilaciju (uz upozorenje da je izvršena konverzija neusaglašenih tipova).

Niz se ne prenosi kao argument funkcije, već se kao argument funkcije se može navesti ime niza i time se prenosi samo pokazivač koji ukazuje na početak niza (tj. ne prenosi se nijedan element niza). Funkcija koja prihvata takav argument za njegov tip ima pokazivač zapisan u formi char *a ili char a[]. **Ovim se kao argument prenosi (po vrednosti) samo pokazivač na početka niza, ali ne i informacija o dužini niza.**

Na primer, kôd dat u narednom primeru (na mašini na kojoj su tip int i adresni tip reprezentovani sa 4 bajta) ispisuje, u okviru funkcije main, broj 20 (5 elemenata tipa int čija je veličina 4 bajta) i, u okviru funkcije f, broj 4 (veličina adrese koja je preneta u funkciju). Dakle, funkcija f nema informaciju o broju elemenata niza a.

```
void f(int a[]) {
printf("%d\n", sizeof(a));
}
int main() {
int a[] = {1, 2, 3, 4, 5};
printf("%d\n", sizeof(a));
f(a);
return 0;
}
```

Dakle, prilikom prenosa niza u funkciju, uz njegovo ime (tj. adresu njegovog početka), najčešće je neophodno proslediti i broj elemenata niza, jer je ovaj broj nemoguće odrediti u okviru funkcije samo na osnovu prenete adrese početka.
Izuzetak od ovog pravila predstavljaju funkcije koje obraduju prosleđene niske karaktera (stringovi iz string.h), jer je u tom slučaju na osnovu sadržaja niske moguće odrediti i njegov broj elemenata. Preciznije, i dalje nije moguće odrediti veličinu niza, već je samo moguće odrediti njegov efektivni sadržaj do pojave terminalne nule.

Kako se kao argument nikada ne prenosi čitav niz, već samo adresa početka, moguće je umesto adrese početka proslediti i pokazivač na bilo koji element niza kao i bilo koji drugi pokazivač odgovarajućeg tipa.
Na primer, ukoliko je ime niza a i ako funkcija f ima prototip f(int x[]);
(ili ekvivalentno f(int *x);),
onda se funkcija može pozivati i za početak niza (sa f(a)) ili za pokazivač na neki drugi element niza (na primer, f(&a[2]) ili ekvivalentno f(a+2)).
Ni u jednom od ovih slučajeva funkcija f, naravno, nema informaciju o tome koliko elemenata niza ima nakon prosledene adrese.

Pokazivačka aritmetika

U prethodnom poglavlju je rečeno da nakon deklaracije int a[10], vrednosti a odgovara pokazivač na prvi element niza, vrednosti a+1 odgovara pokazivač na drugi element niza, itd.

Izraz p+1 i slični uključuju izračunavanje koje koristi pokazivačku aritmetiku i koje se razlikuje od običnog izračunavanja. Naime, izrazu p+1, ne označava dodavanje vrednosti 1 na p, već dodavanje dužine jednog objekta tipa na koji ukazuje p. Na primer, ako p ukazuje na int, onda p+1 i p mogu da se razlikuju
za dva ili četiri - onoliko koliko bajtova na tom sistemu zauzima podatak tipa int. Tako, ako je p pokazivač na int koji sadrži na adresu 100, na mašini na kojoj int zauzima 4 bajta, vrednost p+3 će biti adresa 100 + 3 * 4 = 112.

PRIMER:

Zadatak 7. Popuniti sledecu tabelu:

	A	B	C	P1	P2
Init	1	2	3	&A	&C
*P1 = (*P2)++					
P1 = P2					
P1 = &B					
*P1 = *P1 - *P2					

Od pokazivača je moguće oduzimati cele brojeve (na primer, p-n), pri čemu je značenje ovih izraza analogno značenju u slučaju sabiranja.

Na pokazivače je moguće primenjivati prefiksne i postfiksne operatore ++ i --, sa sličnom semantikom.

Pored dereferenciranja, dodavanja i oduzimanja celih brojeva, nad pokaziva čima je moguće izvoditi još neke operacije.
Pokazivači se mogu porediti relacijskim operatorima (na primer, p1 < p2, p1 == p2, . . .). Ovo ima smisla ukoliko pokazivači ukazuju na elemente istog niza. Tako je, na primer, p1 < p2 tačno akko p1 ukazuje na raniji element niza od pokazivača p2.

Dva pokazivača je moguće oduzimati. I u tom slučaju se ne vrši prosto oduzimanje dve adrese, već se razmatra veličina tipa pokazivača, sa kojom se razlika deli.
Dva pokazivača nije moguće sabirati.

Unarni operatori & i * imaju viši prioritet nego binarni aritmetiči operatori.
Zato je značenje izraza *p+1 zbir sadržaja lokacije na koju ukazuje p i vrednosti 1 (a ne sadržaj na adresi p+1).
Unarni operatori &, *, i prefiksni ++ se primenjuju zdesna nalevo, pa naredba ++*p inkrementira sadržaj lokacije na koju ukazuje p. Postfiksni operator ++ kao i svi unarni operatori koji se primenjuju s leva na desno, imaju viši prioritet od unarnih operatora koji se primenjuju s desna nalevo, tako da *p++ vraća sadržaj na lokaciji p, dok se kao bočni efekat p inkrementira.

Ilustrujmo sve ovo primerom strukture podataka poznate kao stek (eng. stack).
Stek je predstavljen nizom elemenata i elementi se dodaju i uzimaju sa istog kraja.

```
#define MAX_SIZE 1024
int a[MAX_SIZE];
int *top = a;

int push(int x) {
if (top < a + SIZE - 1)  *top++ = x;
else {  printf("Greska"); exit(EXIT_FAILURE);}
}

int pop() {
if (top > a)
return *--top;
else { printf("Greska"); exit(EXIT_FAILURE);}
}

int size() { return top - a; }
```

Prilikom dodavanja elementa vrši se provera da li je stek možda pun i to izrazom top < a + SIZE - 1 koji uključuje pokazivačku aritmetku sabiranje pokazivača (preciznije adrese početka niza) i broja, i zatim oduzimanje broja od dobijenog pokazivača. Slično, prilikom uklanjanja elementa, vrši se da li je stek prazan i to izrazom top > a koji uključuje poredenje dva pokazivača (preciznije, pokazivača i niza). Napokon, funkcija koja vraća broj elemenata postavljenih na stek uključuje oduzimanje dva pokazivača (preciznije, pokazivača i niza).

Potrebno je naglasiti i suptilnu upotrebu prefiksnog operatora dekrementiranja u funkciji pop, odnosno postfiksnog operatora inkrementiranja u funkciji push, kao i njihov odnos sa operatorom dereferenciranja.

Nizovi pokazivača i višedimenzioni nizovi

Kao što je već rečeno, izuzev u slučaju ako je u pitanju argument sizeof ili & operatora, ime niza tipa T se konvertuje u pokazivač na T.
To važi i za višedimenzione nizove.

Ime niza tipa T a[d1][d2]...[dn] se konvertuje u pokazivač na n-1-dimenzioni niz, tj. u pokazivač tipa T (*)[d2]...[dn].
Razmotrimo, na primer, deklaraciju dvodimenzionog niza:

int a[10][20];

Svaki od izraza a[0], a[1], a[2], ... označava niz od 20 elemenata tipa int, te ima tip int [20] (koji se, po potrebi, implicitno konvertuje u tip int*).

Dodatno, a[0] sadrži adresu elementa a[0][0] i, opštije, a[i] sadrži adresu elementa a[i][0]. Sem u slučaju kada je argument sizeof ili & operatora, vrednost a se konvertuje u vrednost tipa int (*)[20] -ovo je tip pokazivača na niz od 20 elemenata.
Slično, izrazi &a[0], &a[1], ... su tipa pokazivača na niz od 20 elemenata, tj. int (*)[20]. Izrazi a i a[0] imaju istu vrednost (adresu početnog elementa dvodimenzionog niza), ali različite tipove: prvi ima tip int (*)[20], a drugi tip int*.

Razmotrimo razliku izmedu dvodimenzionalnog niza i niza pokazivača. Ako
su date deklaracije
int a[10][20];
int *b[10];
i a[3][4] i b[3][4] su sintaksno ispravna referisanja na pojedinačni int.
Ali a je pravi dvodimenzioni niz:
200 ´lokacija za podatak tipa int je rezervisano i uobičajena računica 20 * v + k se koristi da bi se pristupilo elementu a[v][k].
Za niz b, medutim, deklaracija alocira samo 10 pokazivača i ne inicijalizuje ih - inicijalizacija se mora izvršiti eksplicitno, bilo statički (navođenjem inicijalizatora) ili dinamički (tokom izvršavanja programa). Pod pretpostavkom da svaki element niza b zaista pokazuje na niz od 20 elemenata, u memoriji će biti 200 ´lokacija za podataka tipa int i još dodatno 10 lokacija za pokazivače. Klučna prednost niza pokazivača nad dvodimenzionalnim nizom je činjenica da vrste na koje pokazuju ovi pokazivači mogu biti različite dužine.

Tako, svaki element niza b ne mora da pokazuje na 20-to elementni niz -neki mogu da pokazuju na 2-elementni niz, neki na 50-elementi niz, a neki mogu da budu NULL i da ne pokazuju nigde.

Razmotrimo primer niza koji treba da sadrži imena meseci. Jedno rešenje je zasnovano na dvodimenzionalnom nizu (u koji se, prilikom inicijalizacije upisuju imena meseci):
```
char meseci[][9] = {
"Greska", "Januar", "Februar", "Mart", "April",
"Maj", "Jun", "Jul", "Avgust", "Septembar",
"Oktobar", "Novembar", "Decembar"};
```

Pošto meseci imaju imena različite dužine, bolje rešenje je napraviti niz pokazivača na karaktere i inicijalizovati ga da pokazuje na konstantne niske smeštene u segmentu podataka (primetimo da nije potrebno navesti broj elemenata niza pošto je izvršena inicijalizacija):
```
char *meseci[] = {
"Greska", "Januar", "Februar", "Mart", "April",
"Maj", "Jun", "Jul", "Avgust", "Septembar",
"Oktobar", "Novembar", "Decembar"};
```

Pokazivači na funkcije

Funkcije se ne mogu direktno prosleđivati kao argumenti drugim funkcijama, vraćati kao rezultat funkcija i ne mogu se dodeljivati promenljivima.

Ipak, ove operacije je moguće posredno izvršiti ukoliko se koriste pokazivači na funkcije. Razmotrimo nekoliko ilustrativnih primera.

Primer 1.

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

```
void inc1(int a[], int n, int b[]) {
int i;
for (i = 0; i < n; i++)
b[i] = a[i] + 1;
}

void mul2(int a[], int n, int b[]) {
int i;
for (i = 0; i < n; i++)
b[i] = a[i] * 2;
}

void parni0(int a[], int n, int b[]) {
int i;
for (i = 0; i < n; i++)
b[i] = a[i] % 2 == 0 ? 0 : a[i];
}

void ispisi(int a[], int n) {
int i;
for (i = 0; i < n; i++)
printf("%d ", a[i]);
putchar('\n');
}

#define N 8

int main() {
int a[N] = {1, 2, 3, 4, 5, 6, 7, 8}, b[N];
inc1(a, N, b); ispisi(b, N);
mul2(a, N, b); ispisi(b, N);
parni0(a, N, b); ispisi(b, N);
return 0;
}
```

Sve funkcije u prethodnom programu kopiraju elemente niza a u niz b, prethodno ih transformišući na neki način.

Moguće je izdvojiti ovaj zajednički postupak u zasebnu funkciju koja bi bila parametrizovana operacijom transformacije koja se primenjuje na elemente niza a:

```
#include <stdio.h>
void map(int a[], int n, int b[], int (*f) (int)) {
int i;
for (i = 0; i < n; i++)
b[i] = (*f)(a[i]);
}
int inc1(int x) { return x + 1; }
int mul2(int x) { return 2 * x; }
int parni0(int x) { return x % 2 == 0 ? 0 : x; }

void ispisi(int a[], int n) {
int i;
for (i = 0; i < n; i++)
printf("%d ", a[i]);
putchar('\n');
}
```

```
#define N 8
int main() {
int a[N] = {1, 2, 3, 4, 5, 6, 7, 8}, b[N];
map(a, N, b, &inc1); ispisi(b, N);
map(a, N, b, &mul2); ispisi(b, N);
map(a, N, b, &parni0); ispisi(b, N);
return 0;
}
```

Funkcija map ima poslednji argument tipa int (*)(int), što označava pokazivač na funkciju koja prima jedan argument tipa int i vraća argument tipa int. **Pokazivači na funkcije se razlikuju po tipu funkcije na koje ukazuju (po tipovima argumenata i tipu povratne vrednosti). Deklaracija promenljive tipa pokazivača na funkciju se vrši tako što se ime promenljive kojem prethodi karakter * navede u zagradama kojima prethodi tip povratne vrednosti funkcije, a za kojima sledi lista tipova parametara funkcije. Prisustvo zagrada je neophodno kako bi se napravila razlika izmedu pokazivača na funkcije i samih funkcija.**

U primeru
double *a(double, int);
double (*b)(double, int);
*promenljiva a označava funkciju koja vraća rezultat tipa *double, a prima argumente tipa double i int, dok promenljiva b označava pokazivač na funkciju koja vraća rezultat tipa double, a prima argumente tipa double i int.*

Najčešće korišćene operacije sa pokazivačima na funkcije su, naravno, referenciranje (&) i dereferenciranje (*). Iako kod nekih kompilatora oznake ovih operacija mogu da se izostave i da se koristi samo ime pokazivaca (na primer, u prethodnom programu je bilo moguce navesti map(a, N, b, inc1) i b[i] = f(a[i])), ovo se ne preporucuje zbog prenosivosti programa.

Moguće je kreirati i nizove pokazivača na funkcije. Ovi nizovi se mogu i incijalizovati (na uobičajeni način). U primeru
int (*fje[3]) (int) = {&inc1, &mul2, &parni0};
fje predstavlja niz od 3 pokazivača na funkcije koje vraćaju int, i primaju argument tipa int. Funkcije čije se adrese nalaze u nizu se mogu direktno i pozvati. Na primer, naredni kod ispisuje vrednost 4:
printf("%d", (*fje[0])(3));

Dinamička alokacija memorije

U većini realnih aplikacija, u trenutku pisanja programa nije moguće precizno predvideti memorijske zahteve programa. Naime, memorijski zahtevi zavise od interakcije sa korisnikom i tek u fazi izvršavanja programa korisnik svojim akcijama implicitno odreduje potrebne memorijske zahteve (na primer, koliko elemenata nekog niza će biti korišćeno). U nekim slučajevima, moguće je predvideti neko gornje ograničenje, ali ni to nije uvek zadovoljavajuće. Ukoliko je ograničenje premalo, program nije u stanju da obraduje veće ulaze, a ukoliko
je preveliko, program zauzima više memorije nego što mu je stvarno potrebno.

Rešenje ovih problema je dinamička alokacija memorije koja omogućava da program u toku svog rada, u fazi izvršavanja, zahteva (od operativnog sistema) odredenu količinu memorije. U trenutku kada mu memorija koja je dinamički alocirana više nije potrebna, program može i dužan je da je oslobodi i tako je vrati operativnom sistemu na upravljanje. Alociranje i oslobadanje vrši se funkcijama iz standardne biblioteke i pozivima runtime biblioteke. U fazi izvršavanja, vodi se evidencija i o raspoloživim i zauzetim blokovima memorije.

Funkcije standardne biblioteke za rad sa dinamičkom memorijom

Standardna biblioteka jezika C podržava dinamičko upravljanje memorijom kroz nekoliko funkcija (sve su deklarisane u zaglavlju <stdlib.h>). ***Prostor za dinamički alociranu memoriju nalazi se u segmentu memorije koji se zove hip (engl. heap).***

Funkcija malloc ima sledeći prototip:
void *malloc(size_t n);

Ona alocira blok memorije (tj. niz uzastopnih bajtova) veličine n bajtova i vraća adresu alociranog bloka u vidu generičkog pokazivača (tipa void*). U slučaju da zahtev za memorijom nije moguće ispuniti (na primer, zahteva se više memorije nego što je na raspolaganju), ova funkcija vraća NULL. Memorija na koju funkcija malloc vrati pokazivač nije inicijalizovana i njen sadržaj je, u principu, nedefinisan (tj. zavisi od podataka koji su ranije bili čuvani u tom delu memorije).
Funkcija malloc očekuje argument tipa size_t. Ovo je nenegativni celobrojni tip za čije vrednosti je rezervisano najmanje dva bajta. Ovaj tip se razlikuje od tipa unsigned int, ali su medusobne konverzije moguće i, zapravo, trivijalne. Tip size_t može da se koristi za čuvanje bilo kog indeksa niza, a on je i tip povratne vrednosti operatora sizeof.

Funkcija calloc ima sledeći prototip:
void *calloc(size_t n, size_t size)

Ona vraća pokazivač na blok memorije veličine n objekata navedene veličine size. U slučaju za zahtev nije moguće ispuniti, vraća se NULL. Za razliku od malloc, alocirana memorija je inicijalizovana na nulu.
Dinamički objekti alocirani navedenim funkcijama su neimenovani i bitno su različiti od promenljivih. Ipak, dinamički alociranim blokovima se pristupa na sličan način kao nizovima.

Navedene funkcije su generičke i koriste se za dinamičku alokaciju memorije za podatke bilo kog tipa. Kako bi se dobijenoj memoriji moglo pristupati slivcno kao u slučaju nizova, potrebno je (poželjno eksplicitno) konvertovati dobijeni pokazivač tipa void* u neki konkretni pokazivački tip.

Nakon poziva funkcije malloc() ili calloc() obavezno treba proveriti povratnu
vrednost kako bi se utvrdilo da li je alokacija uspela. Ukoliko alokacija ne uspe, pokušaj pristupa memoriji na koju ukazuje dobijeni pokazivač dovodi do dereferenciranja NULL pokazivača i greške. Ukoliko se utvrdi da je funkcija malloc() (ili calloc()) vratila vrednost NULL, može se prijaviti korisniku odgovaraju
ća poruka ili pokušati neki metod oporavka od greške. Dakle, najčešći scenario upotrebe funkcije malloc sledeći:

```
int* p = (int*) malloc(n*sizeof(int));
if (p == NULL)
/* prijaviti gresku */
```

U navedenom primeru, nakon uspešne alokacije, u nastavku programa se p
može koristiti kao (statički alociran) niz celih brojeva.

U trenutku kada dinamički alociran blok memorije više nije potreban, poželjno je osloboditi ga. To se postiže funkcijom free:
void free(void* p);

Poziv free(p) oslobada memoriju na koju ukazuje pokazivač p (a ne memorijski prostor koji sadrži s^am pokazivač p), pri čemu je neophodno da p pokazuje na blok memorije koji je alociran pozivom funkcije malloc ili calloc. Opasna je greška pokušaj oslobadanja memorije koja nije alocirana na ovaj način. Takode, ne sme se koristiti nešto što je već oslobodeno niti se sme dva puta oslobadati ista memorija. Redosled oslobadanja memorije ne mora da odgovara redosledu alokaciranja.

Ukoliko neki dinamički alociran blok nije osloboden ranije, on će biti osloboden prilikom završetka rada programa, zajedno sa svom drugom memorijom koja je dodeljena programu. Ipak, ne treba se oslanjati na to i preporučeno je eksplicitno oslobadanje sve dinamički alocirane memorije pre kraja rada programa, a poželjno čim taj prostor nije potreban.
Upotrebu ovih funkcija ilustruje naredni primer u kojem se unosi i obrnuto ispisuje niz čiji broj elemenata nije unapred poznat (niti je poznato njegovo gornje ograničenje), već se unosi sa ulaza tokom izvršavanja programa.

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

```
#include <stdlib.h>
int main() {
int n, i, *a;
/* Unos broja elemenata */
scanf("%d", &n);
/* Alocira se memorija */
if ((a = (int*)malloc(n*sizeof(int))) == NULL) {
printf("Greska prilikom alokacije memorije\n");
return 1;
}
/* Unos elemenata */
for (i = 0; i < n; i++) scanf("%d",&a[i]);
/* Ispis elemenata u obrnutom poretku */
for (i = n-1; i >= 0; i--) printf("%d ",a[i]);
/* Oslobadjanje memorije */
free(a);
return 0;
}
```

U nekim slučajevima potrebno je promeniti veličinu već alociranog bloka memorije. To se postiže korišćenjem funkcije realloc, čiji je prototip:
void *realloc(void *memblock, size_t size);
Parametar memblock je pokazivač na prethodno alocirani blok memorije, a parametar size je nova veličina u bajtovima. Funkcija realloc vraća pokazivač tipa void* na realociran blok memorije, a NULL u slučaju da zahtev ne može biti ispunjen. Zahtev za smanjivanje veličine alociranog bloka memorije uvek uspeva. U slučaju da se zahteva povećanje veličine alociranog bloka memorije, pri čemu iza postojećeg bloka postoji dovoljno slobodnog prostora, taj prostor se jednostavno koristi za proširivanje. Medutim, ukoliko iza postojećeg bloka ne postoji dovoljno slobodnog prostora, onda se u memoriji traži drugo mesto dovoljno da prihvati prošireni blok i, ako se nade, sadržaj postojećeg bloka se kopira na to novo mesto i zatim se stari blok memorije oslobada. Ova operacija može biti vremenski zahtevna.

Upotreba funkcije realloc ilustrovana je programom koji učitava cele brojeve i smešta ih u memoriju, sve dok se ne unese -1 za kraj. S obzirom na to da se broj elemenata ne zna unapred, a ne zna se ni gornje ograničenje, neophodno je postepeno povećavati skladišni prostor tokom rada programa. Kako česta realokacija može biti neefikasna, u narednom programu se izbegava realokacija prilikom unosa svakog sledećeg elementa, već se vrši nakon unošenja svakog desetog elementa. Naravno, ni ovo nije optimalna strategija _ u praksi se obično koristi pristup da se na početku realokacije vrše relativno često, a onda sve rede i rede (na primer, svaki put se veličina niza dvostruko uveća).

```
#include <stdio.h>
#include <stdlib.h>
#define KORAK 256
int main() {
int* a = NULL; /* Niz je u pocetku prazan */
int duzina = 0; /* broj popunjenih elemenata niza */
int alocirano = 0; /* broj elemenata koji mogu biti smesteni */
int i;
do {
printf("Unesi ceo broj (-1 za kraj): ");
scanf("%d", &i);
/* Ako nema vise slobodnih mesta, vrsi se prosirivanje */
if (duzina == alocirano) {
alocirano += KORAK;
a = realloc(a, alocirano*sizeof(int));
if (a == NULL) return 1;
}
a[duzina++] = i;
```

```
} while (i != -1);
/* Ispis elemenata */
printf("Uneto je %d brojeva. Alocirano je ukupno %d bajtova\n",
duzina, alocirano*sizeof(int));
printf("Brojevi su : ");
for (i = 0; i<duzina; i++)
printf("%d ", a[i]);
/* Oslobadjanje memorije */
free(a);
return 0;
}
```

Bez upotrebe funkcije realloc centralni blok navedene funkcije main bi mogao da izgleda ovako:

```
if (duzina == alocirano) {
/* Kreira se novi niz */
int* new_a;
alocirano += KORAK;
new_a = malloc(alocirano*sizeof(int));
/* Kopira se sadrzaj starog niza u novi */
for (i = 0; i < duzina; i++) new_a[i] = a[i];
/* Oslobadja se stari niz */
free(a);
/* a ukazuje na novi niz */
a = new_a;
}
```

U ovoj implementaciji, prilikom svake realokacije vrši se premeštanje memorije, tako da je ona neefikasnija od verzije sa realloc.

U gore navedenom primeru koristi se konstrukcija:

```
a = realloc(a, alocirano*sizeof(int));
```

U nekim slučajevima ova konstrukcija može da bude neadekvatna ili opasna. Naime, ukoliko zahtev za proširenje memorijskog bloka ne uspe, vraća se vrednost NULL, upisuje u promenljivu a i tako gubi jedina veza sa prethodno alociranim blokom (i on, na primer, ne može da bude osloboden).

Greške u radu sa dinamičkom memorijom

Dinamički alocirana memorija nudi mnoge mogućnosti i dobra rešenja ali često je i uzrok problema. Neki od najčešćih problema opisani su u nastavku.

Curenje memorije.
Jedna od najopasnijih grešaka u radu sa dinamički alociranom memorijom je tzv. curenje memorije (eng. memory leaking). Curenje memorije je situacija kada se u tekućem stanju programa izgubi informacija o lokaciji dinamički alociranog, a neoslobodenog bloka memorije.
U tom slučaju, program više nema mogućnost da oslobodi taj blok memorije i on biva “zauvek” (zapravo _ do kraja izvršavanja programa) izgubljen (rezervisan za korišćenje od strane programa koji više nema načina da mu pristupi).
Curenje memorije ilustruje naredni primer.

```
char* p;
p = (char*) malloc(1000);
....
p = (char*) malloc(5000);
```

Inicijalno je 1000 bajtova dinamički alocirano i adresa početka ovog bloka memorije smeštena je u pokazivačku promenljivu p. Kasnije je dinamički alocirano 5000 bajtova i adresa početka tog bloka memorije je opet smeštena u promenljivu p. Medutim, pošto originalnih 1000 bajtova nije oslobodeno korišćenjem funkcije free, a adresa početka tog bloka memorije je izgubljena promenom vrednosti pokazivačke promenljive p, tih 1000 bajtova

biva nepovratno izgubljeno za program.

Naročito su opasna curenja memorije koja se dogadaju u okviru neke petlje. U takvim situacijama može da se gubi malo po malo memorije, ali tokom dugtrajnog izvršavanja programa, pa ukupna količina izgubljene memorije može da bude veoma velika. Moguće je da u nekom trenutku program iscrpi svu raspoloživu memoriju i onda će njegovo izvršavanje biti prekinuto od strane operativnog sistema. čak i da se to ne desi, moguće je da se iscrpi raspoloživi prostor u glavnoj memoriji i da se, zbog toga, sadržaj glavne memorije prebacuje na disk i obratno (tzv. swapping), što onda ekstremno usporava rad programa.
Curenje memorije je naročito opasno zbog toga što često ne biva odmah uočeno. Obično se tokom razvoja program testira kratkotrajno i na malim ulazima. Medutim, kada se program pusti u rad i kada počne da radi duži vremenski period (možda i bez prestanka) i da obraduje veće količine ulaza, curenje memorije postaje vidljivo, čini program neupotrebljivim i može da uzrokuje velike štete.
Većina programa za otkrivanje grešaka (debagera) detektuje da u programu postoji curenje memorije, ali ne može da pomogne u lociranju odgovarajuće greške u kodu. Postoje specijalizovani programi profajleri za curenje memorije (engl. memory leaks profilers) koji olakšavaju otkrivanje uzroka curenja memorije.

Pristup oslobodjenoj memoriji.
Nakon poziva free(p), memorija na koju pokazuje pokazivač p se oslobada i ona više ne bi trebalo da se koristi. Medutim, poziv free(p) ne menja sadržaj pokazivača p. Moguće je da naredni poziv funkcije malloc vrati blok memorije upravo na toj poziciji.
Naravno, ovo ne mora da se desi i nije predvidljivo u kom će se trenutku desiti, tako da ukoliko programer nastavi da koristi memoriju na adresi p, moguće je da će greška proći neopaženo. Zbog toga, preporučuje se da se nakon poziva free(p), odmah p postavi na NULL. Tako se osigurava da će svaki pokušaj pristupa oslobodenoj memoriji biti odmah prepoznat tokom izvršavanja programa i operativni sistem će zaustaviti izvršavanje programa sa porukom o grešci (najčešće segmentation fault).

Oslobadjanje istog bloka više puta.
Nakon poziva free(p), svaki naredni poziv free(p) za istu vrednost pokazivača p prouzrokuje nedefinisano ponašanje programa i trebalo bi ga izbegavati. Takozvana višestruka oslobadanja mogu da dovedu do pada programa a poznato je da mogu da budu i izvor bezbednosnih problema.

Oslobadjanje neispravnog pokazivača.
Funkciji free(p) dopušteno je proslediti isključivo adresu vraćenu od strane funkcije malloc, calloc ili realloc. čak i prosledivanje pokazivača na lokaciju koja pripada alociranom bloku (a nije njegov početak) uzrokuje probleme. Na primer,
free(p+10); /* Oslobodi sve osim prvih 10 elemenata bloka */
neće osloboditi „sve osim prvih 10 elemenata bloka" i sasvim je moguće da će dovesti do neprijatnih posledica, pa čak i do pada programa.

Prekoračenja i potkoračenja bafera.
Nakon dinamičke alokacije, pristup memoriji je dozvoljen samo u okviru granica bloka koji je dobijen. Kao i u slučaju statički alociranih nizova, pristup elementima van granice može da stvori ozbiljne probleme. Naravno, upis je često opasniji od čitanja.
U slučaju dinamički alociranih blokova memorije, obično se nakon samog bloka smeštaju dodatne informacije potrebne alokatoru memorije kako bi uspešno vodio evidenciju koji delovi memorije su zauzeti, a koji slobodni. Prekoračenje bloka prilikom upisa menja te dodatne informacije što, naravno, uzrokuje pad sistema za dinamičko upravljanje memorijom.

Fragmentisanje memorije čest je slučaj da ispravne aplikacije u kojima ne postoji curenje memorije (a koje često vrše dinamičku alokacije i dealokacije memorije) tokom dugog rada pokazuju degradaciju u performansama i na kraju prekidaju svoj rad na nepredvideni način. Uzrok ovome je najčešće fragmentisanje memorije. U slučaju fragmentisane memorije, u memoriji se često nalazi dosta slobodnog prostora, ali on je rascepkan na male, nepovezane parčiće. Razmotrimo naredni (minijaturizovan) primer. Ukoliko 0 označava slobodni bajt, a 1 zauzet, a memorija trenutno ima sadržaj 100101011000011101010110, postoji ukupno 12 slobodnih bajtova. Medutim, pokušaj alokacije 5 bajtova ne može da uspe, jer u memoriji ne postoji prostor dovoljan za smeštanje 5 povezanih bajtova. S druge strane, memorija koja ima sadržaj 111111111111111100000000 ima samo 8 slobodnih bajtova, ali jeste u stanju da izvrši alokaciju 5 traženih bajtova.

Memoriju na hipu dodeljenu programu nije moguće automatski reorganizovati u fazi izvršavanja, jer nije moguće u toj fazi ažurirati sve pokazivače koji ukazuju na objekte na hipu.

Postoji nekoliko pristupa za izbegavanje fragmentisanja memorije. Ukoliko je moguće, poželjno je izbegavati dinamičku alokaciju memorije. Naime, alociranje svih struktura podataka statički (u slučajevima kada je to moguće) dovodi do bržeg i predvidljivijeg rada programa (po cenu većeg utroška memorije).
Alternativno, ukoliko se ipak koristi dinamička alokacija memorije, poželjno je memoriju alocirati u većim blokovima i umesto alokacije jednog objekta alocirati prostor za nekoliko njih odjednom (kao što je to, na primer, bilo radeno u primeru datom prilikom opisa funkcije realloc). Na kraju, postoje tehnike efikasnijeg rukovanja memorijom (na primer, memory pooling), u kojima se programer manje oslanja na sistemsko rešenje, a više na svoje rešenje koje je prilagodenjeno specifičnim potrebama.

Hip i dinamički životni vek
U prethodnom poglavlju, rečeno je da je memorija dodeljena programu organizovana u segment koda, segment podataka, stek segment i hip segment. Hip segment predstavlja tzv. slobodnu memoriju iz koje se crpi memorija koja se dinamički alocira. Dakle, funkcije malloc, calloc ili realloc, ukoliko uspeju, vraćaju adresu u hip segmentu. Objekti koji su alocirani u slobodnom memorijskom prostoru nisu imenovani, već im se pristupa isključivo preko adresa. Pošto ovi objekti nisu imenovani, oni nemaju definisan doseg (a doseg pokaziva ča putem kojih se pristupa dinamičkim objektima podleže standardnim pravilima). Svi objekti koji su dinamički alocirani imaju dinamički životni vek.
Ovo znači da se memorija i alocira i oslobada isključivo na eksplicitni zahtev i to tokom rada programa.
Hip segment obično počinje neposredno nakon segmenta podataka, a na suprotnom kraju memorije od stek segmenta. Obično se podaci na hipu slažu od manjih ka većim adresama (engl. upward growing), dok se na steku slažu od većih ka manjim adresama (eng;. downward growing). Ovo znači da se u trenutku kada se iscrpi memorijski prostor dodeljen programu, hip i stek potencijalno „sudaraju”, ali operativni sistemi obično to sprečavaju i tada obično dolazi do nasilnog prekida rada programa.