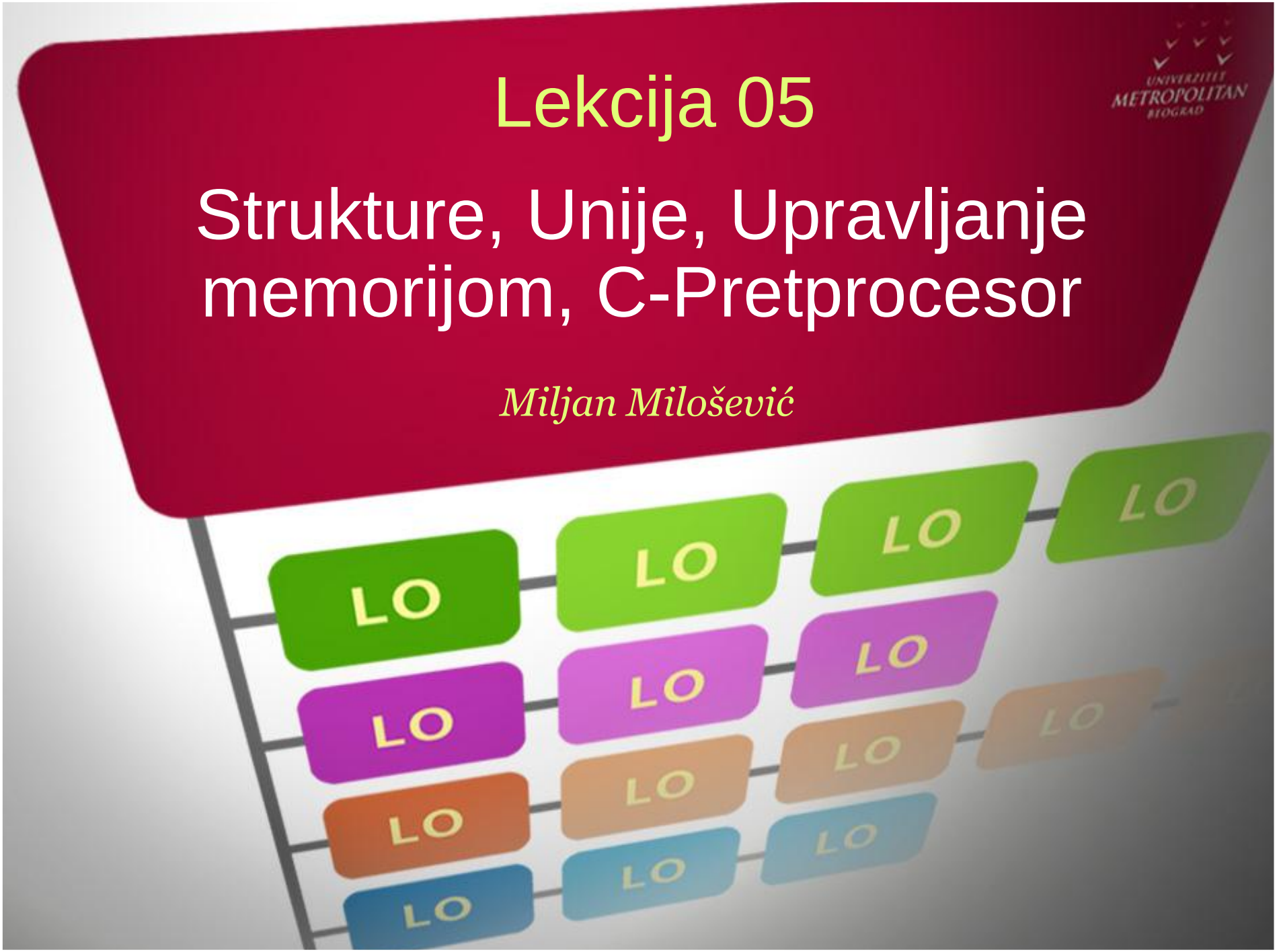
Lekcija 05

Strukture, Unije, Upravljanje memorijom, C-Pretprocesor

Miljan Milošević



STRUKTURE, UNIJE, UPRAVLJANJE MEMORIJOM, C-PRETPROCESOR

	01	02	03	04
Uvod	**Strukture - Osnovi**	**Strukture – Upotreba nizova, funkcija i pokazivača**	**Unije**	**Polja bitova**
	➢ *Uvod u strukture* ➢ *Definicija strukture* ➢ *Osnovna pravila kod definisanja struktura* ➢ *Deklaracija strukturnih promenljivih i typedef* ➢ *Pristup članu strukture* ➢ *Korišćenje typedef deklaracije kod struktura*	➢ *Strukture i pokazivači*	➢ *Deklaracija unije* ➢ *Memorijski prostor kod upotrebe unija* ➢ *Pristup članu unije* ➢ *Unije kao članovi strukture*	➢ *Osnovi o polju bitova*

STRUKTURE, UNIJE, UPRAVLJANJE MEMORIJOM, C-PRETPROCESOR

05

Dinamičko upravljanje memorijom

- *Alociranje memorije*
- *Realociranje i oslobađanje memorije*
- *Upravljanje memorijom u teoriji i praksi*
- *Upravljanje memorijom - rad sa strukturama i funkcijama*

06

Pretprocesorske naredbe

- *Uvod u pretprocesorske direktive*
- *Uključivanje datoteka*
- *Makro zamena – osnovna primena*
- *Makro zamena – dodatne mogućnosti*
- *Parametrizovani makroi*
- *Uslovno prevođenje*
- *Primeri korišćenja pretprocesora - Rezime*

07

Vežba - Strukture, Unije, Polja bitova

- *Primer. Strukture i typedef*
- *Primer. Korišćenje struktura kod geometrijskih oblika*
- *Primer. Pokazivači na strukture*
- *Primer. Pokazivači i strukture*
- *Primer. Pokazivači na strukture unutar struktura*
- *Primer. Pokazivači na strukture unutar struktura*
- *Primer. Korišćenje*

08

Vežba – Dinamičko upravljanje memorijom

- *Primer. Generisanje nizova proizvoljne dužine*
- *Primer. Upotreba funkcije realloc()*
- *Primer. Upotreba funkcija calloc() i realloc() kod nizova*
- *Primer. Vraćanja niza iz funkcije korišćenjem pokazivača*
- *Funkcija vraća pokazivač na dinamički alociranu memoriju – Primer1*
- *Funkcija vraća*

09

Vežba – Pretprocesorske naredbe

- *Primer. Upotreba pretprocesorskih direktiva #define i #include*
- *Primer. Upotreba pretprocesorskih direktiva za uslovno prevođenje:*
- *Primer. Upotreba direktiva za uslovno prevođenje i #undef*
- *Primer. Upotreba pretprocesorske direktive #pragma*
- *Primer. Korišćenje makroa unutar drugog makroa*

10

Zadaci za samostalan rad

- *Zadaci iz struktura, upravljanja memorijom i pretprocesora*

UVOD

Ova lekcija treba da ostvari sledeće ciljeve:

U okviru ove lekcije studenti se upoznaju sa raznim bitnim pojmovima u programskom jeziku C:

- ❑ Strukture, unije i polja bitova
- ❑ Upravljanje memorijom
- ❑ Pretpocesorske naredbe

Objekti koje susrećemo u svakodnevnom životu su veoma kompleksni da bi se opisali pomoću jedne promenljive. Nizovi, sa kojima smo se susreli do sada, mogu da čuvaju veliki broj podataka ali je problem u tome što ti podaci moraju biti istog tipa, i takvo ograničenje nije pogodno za objekte. Svaka osoba, na primer, deli iste osobine kao što su ime i visina. Vrednost svake od osobina mora biti smešten u posebnu promenljivu, koje moraju biti različitog tipa podatka.

Pojedinačne informacije koje opisuju karakteristike objekata, kao što su informacije o preduzećima ili zaposlenima se često grupišu u takozvane slogove odnosno strukture (redords*, struct*). **Slogovi su korisnički definisani tipovi podataka** koji su pogodni za bolju organizaciju, predstavljanje, i naravno čuvanje informacija o sličnim objektima. Slogovi se sastoje iz polja koja sadrže jedan individualni detalj, kao što je na primer naziv, adresa, ili registarski broj nekog preduzeća.

Unije se definišu na sličan način kao i strukture. Za razliku od članova struktura svi članovi unije dele istu adresu pa je stoga moguće koristiti unije kada se želi uštedeti memorijski prostor. Kao dodatak na primitivne i izvedene tipove, članovi struktura i unija mogu biti i polja bitova. Polje bita je celobrojna promenljiva koja se sastoji od određenog broja bitova. Definisanjem polja bitova moguće je deo memorije iscepkati u individualne grupe kojima se posle može pristupiti preko imena.

U toku pisanja programa često nije poznato unapred koliko je podataka potrebno da bi se rešio neki problem. U ovakvim slučajevima je pogodno alocirati, korišćenjem principa upravljanja memorijom, neophodan deo prostora koji će biti oslobođen čim je to moguće.

Već smo opisali faze generisanja programa. Prvi korak je poziv pretprocesora koji priprema izvorni kod za kompajler. Rezultat pripreme je modifikovani kod u kome su obrisani komentari a direktive pretprocesora zamenjene rezultatima izvršenja tih direktiva.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Strukture - Osnovi

Strukture,deklaracija,definicija,pristup, typedef

- *Uvod u strukture*
- *Definicija strukture*
- *Osnovna pravila kod definisanja struktura*
- *Deklaracija strukturnih promenljivih i typedef*
- *Pristup članu strukture*

01

UVOD U STRUKTURE

Struktura je skup jedne ili više promenljih, koje mogu biti različitih tipova, grupisanih zajedno radi lakše manipulacije

Nizovi u C-u omogućavaju da se definiše promenljiva koja može da čuva više podataka istog tipa, dok je struktura još jedan korisnički definisan tip podatka u C-u koji dozvoljava čuvanje podataka različitog tipa. Struktura je skup jedne ili više promenljivih, koje mogu biti različitih tipova, grupisanih zajedno radi lakše manipulacije. Strukture pomažu pri organizaciji kompleksnih podataka, posebno u velikim programima, jer one omogućavaju obradu grupe međusobno povezanih promenljivih kao jedne celine. Tradicionalan primer strukture je stvaranje platnog spiska: zaposleni je opisan skupom atributa tipa imena, adrese, broja socijalnog osiguranja, plate i sl. Neki od ovih atributa mogu se, takođe, predstaviti preko struktura jer se mogu sastojati od više komponenti. Drugi primer: tačka u prostoru je definisana uređenim parom koordinata, pravougaonik se (barem deo njih) može definisati parom tačaka, itd.

Pretpostavimo da želimo da pratimo knjige u biblioteci tako da će neki od sledećih atributa biti uzeti u razmatranje:

- Naziv knjige
- Naziv autora
- Tema knjige
- ID knjige

U nastavku će biti opisano korišćenje struktura za ovaj primer.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

DEFINICIJA STRUKTURE

U cilju definisanja strukture koristi se ključna reč struct. Nakon toga se navodi ime strukture (opciono), a zatim, između vitičastih zagrada, opis njenih članova (ili polja)

U cilju definisanja strukture koristi se ključna reč struct. Korišćenjem ključne reči struct definiše se novi tip podatka, koji može da ima više od jednog atributa tj. člana. Osnovni format definicije strukture je dat u nastavku:

```
struct [naziv strukture]
{
   member definition;
   member definition;
   ...
   member definition;
} [jedna ili više strukturna promenljiva];
```

Nakon ključne reči struct navodi se ime strukture (opciono, može i da se ne navede), a nakon toga, između vitičastih zagrada, opis njenih članova (ili polja). Imena članova strukture se ne mogu koristiti kao samostalne promenljive, one postoje samo kao deo složenijeg objekta. Članovi strukture mogu biti proizvoljnog tipa podatka, int, char, itd, a mogu se čak definisati i kao nizovi. Na kraju definicije strukture, pre poslednjeg znaka " ; " može se navesti lista strukturnih promenljivih, i ovo je takođe opciono. U nastavku je dat primer deklarisanja strukture Books, koja se sastoji iz podataka kao što su na primer: naziv knjige, ime autora, oblast knjige i identifikacioni broj:

```
struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

OSNOVNA PRAVILA KOD DEFINISANJA STRUKTURA

Struktura mora da se sastoji iz bar jednog člana. Struktura ne može da sadrži element koji je istog tipa kao i struktura osim ako se radi o pokazivaču na taj tip

Struktura mora da se sastoji iz bar jednog člana. U nastavku je dat primer gde se definiše struktura *struct Date*, koja ima tri člana tipa short:

```
struct Date { short year, month, day; };
```

U narednom primeru je definisana struktura struct Song, i ima 5 članova u kojima se smešta 5 različitih informacija o muzičkom zapisu. Član strukture published je tipa struct Date, a ovaj tip je definisan u prethodnom primeru:

```
struct Song {
  char title[64];
  char artist[32];
  char composer[32];
  short duration;          // Playing time in seconds.
  struct Date published;   // Date of publication.
};
```

Strukturni tip ne može da sadrži element koji je istog tipa kao i struktura jer njena definicija nije kompletna dok se ne zatvori vitičasta zagrada (}). To znači da u prethodnom primeru struktura Song ne može da sadrži element koji je tipa Song. Međutim, strukturni tip može i često sadrži pokazivač na taj isti tip podatka. Takve "samo-referencirajuće strukture" se često koriste kod dinamičkih struktura podataka kao što su povezane liste, binarna stabla, itd. U nastavku je dat primer definisanja novog tipa podatka koji će da posluži kao čvor povezane liste:

```
struct Cell
{
          struct Song song;      // This record's data.
          struct Cell *pNext;    // A pointer to the next record.
};
```

Ukoliko se novodefinisani strukturni tip koristi u više izvornih fajlova, onda je neophodno definiciju strukture smestiti u neki fajl zaglavlja, koji će zatim biti uključen u odgovarajući izvorni fajl.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

DEKLARACIJA STRUKTURNIH PROMENLJIVIH I TYPEDEF

Struktura se može definisati izostavljanjem naziva. U slučaju da vam naziv strukture treba u nastavku moguće je koristiti typedef u cilju defisanja novog tipa radi lakšeg i kraćeg pisanja

Kada smo deklarisali strukturu onda je moguće definisati promenljive koje će biti tog strukturnog tipa podataka. U nastavku su dati primeri deklaracije:

```
typedef struct Song Song_t;   // Song_t is now a synonym for
                              // struct Song.
Song_t song1, song2, *pSong = &song1; // Two struct Song objects and a
                              // struct Song pointer.
```

U prethodnom primeru smo definisali novi naziv za struct Song, a to je Song_t, tako da je u nastavku moguće koristiti Song_t pri deklaraciji objekata strukturnog tipa. Objekti strukturnog tipa, kao što su u našem primeru song1 i song2, se nazivaju strukturni objekti (ili strukturne promenljive).
Kao što smo već napomenuli, struktura se može definisati izostavljanjem naziva, odnosno taga. Ovo ima smisla i praktično je ako se prilikom deklaracije strukture deklarišu i strukturne promenljive koje će biti korišćenje dalje u programu. U slučaju da vam tip strukture treba u nastavku za deklaraciju još nekih strukturnih promenljivih onda je moguće koristiti typedef da bi ste imali bar neko ime tipa podatka:

```
typedef struct
{
          struct Cell *pFirst, *pLast;
} SongList_t;
```

Ova typedef deklaracija definiše SongList_t kao ime strukturnog tipa čiji su članovi pokazivači na struct Cell, označeni imenima pFirst i pLast.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRISTUP ČLANU STRUKTURE

Članu strukture se pristupa preko imena promenljive (čiji je tip struktura) iza koga se navodi tačka (.) a onda ime člana.

Članu strukture se pristupa preko imena promenljive (čiji je tip struktura) iza koga se navodi tačka (.) a onda ime člana. Tačka se u ovom izrazu drugačije naziva operator pristupa. U nastavku su dati neki primeri korišćenjem strukture struct Song i pristupanje njenim članovima:

```
#include <string.h>              // Prototypes of
string functions.
Song_t song1, song2;             // Two objects of
type Song_t

// Copy a string to the title of song1:
strcpy( song1.title, "Havana Club" );

// Likewise for the composer member:
strcpy( song1.composer, "Ottmar Liebert" );

song1.duration = 251;                // Playing time.

// The member published is itself a structure:
song1.published.year = 1998;         // Year of
publication
```

Kod struktura se može koristiti operator dodele da se ceo sadržaj jedne strukturne promenljive iskopira u drugu strukturnu promenljivu koja je istog tipa:

```
song2 = song1;
```

Nakon prethodnog iskaza dodele, svi članovi objekta song2 će imatu istu vrednost kao i odgovarajući članovi strukturne promenljve song1.

U nastavku je dat još jedan primer pristupa elementima strukture

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

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( )
{
   struct Books Book1;        /* Declare Book1 of
type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Metropolitan");
   strcpy( Book1.author, "Marko Markovic");
   strcpy( Book1.subject, "C Metropolitan
Tutorial");
   Book1.book_id = 6495407;

   /* print Book1 info */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   return 0;
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

KORIŠĆENJE TYPEDEF DEKLARACIJE KOD STRUKTURA

Korišćenjem typedef deklaracije možemo definisati novi tip podatka i kada je u pitanju rad sa strukturama. Tako pri deklaraciji novih promenljivih izostavljamo pisanje ključne reči struct

U nastavku je dat primer gde smo korišćenjem typedef deklaracije definisali novi tip podatka koji smo zatim koristili u programu:

```
typedef struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} Book;
```

Stoga je moguće deklarisati strukturnu promenljivu korišćenjem sledećeg iskaza:

```
Book book;
```

i zatim je kao takvu koristiti u programu:

```
int main( )
{
   Book book;

   strcpy( book.title, "C Programming");
   strcpy( book.author, "Marko Markovic");
   strcpy( book.subject, "C Tutorial");
   book.book_id = 6495407;

   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);

   return 0;
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Strukture – Upotreba nizova, funkcija i pokazivača

Strukture, nizovi, funkcije, pokayivači

- *Strukture i pokazivači*

02

NIZOVI STRUKTURA

Umesto da se skup složenih podataka čuva u nezavisnim promenljvama, nekada je bolje čuvati taj skup podataka u nizovama struktura

Veoma često u programerskim problemima postoji povezan skup složenih podataka. Umesto da se oni čuvaju u nezavisnim promenljivama (što bi vodilo programima teškim za održavanje) bolje je koristiti nizove struktura. Na primer, ako je potrebno imati podatke o imenima i broju dana i meseci u godini, moguće je te podatke čuvati u nizu sa brojevima dana i u (nezavisnom) nizu imena meseci. Bolje je, međutim, opisati strukturu mesec koja sadrži brojDana i ime:

```
struct mesec {
          char ime[10];
          int brojDana;
};
```

i koristiti niz ovakvih struktura:

```
struct opis_meseca meseci[13];
```

U prethodnom iskazu deklarisan je niz dužine 13 da bi se meseci mogli referisati po svojim rednim brojevima, pri čemu se početni element niza ne koristi. Moguća je i deklaracija sa inicijalizacijom (u kojoj nije neophodno navođenje broja elemenata niza):

```
struct opis_meseca meseci[] = {
          { "",0 },
          { "januar",31 },
          { "februar",28 },
          { "mart",31 },
          ...
          { "decembar",31 }
}
```

U prethodno navednoj inicijalizaciji unutrašnje vitičaste zagrade se mogu izostaviti:

```
struct opis_meseca meseci[] = {
          "",0,
          "januar",31,
          "februar",28,
          "mart",31,
          ...
          "decembar",31
}
```

Nakon navedene deklaracije, ime prvog meseca u godini se može dobiti sa meseci[1].ime, njegov broj dana sa meseci[1].brojDana itd. Kao i obično, broj elemenata ovako inicijalizovanog niza može se izračunati na sledeći način:

```
sizeof(meseci)/sizeof(struct opis_meseca)
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

STRUKTURA KAO ARGUMENT FUNKCIJE

Strukture se prosleđuju funkciji na isti način kao što je moguće proslediti bilo koju drugu promenljivu ili pokazivač. Pozivanje po vrednosti nije efikasno za ogromne strukture

Strukture se prosleđuju funkciji na isti način kao što je moguće proslediti bilo koju drugu promenljivu ili pokazivač. Pristup članovima strukture koja je prosleđena funkciji se vrši na identičan način kako je to prikazano u prethodnim primerima. Neka je za prethodni primer koji smo imali definisana funkcija na sledeći način:

```
void printBook( struct Books book );
```

Štampanje podataka o knizi je moguće izvršiti korišćenjem:

```
printBook( Book1 );
```

gde bi definicija funkcije printBook imala sledeći oblik

```
void printBook( struct Books book )
{
   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);
}
```

Ukoliko je parameter funkcije strukturnog tipa, onda se sadržaj stvarnih argumenata kopira u fiktivne parametre funkcije koja se poziva. Ovaj pristup nije efikasan osim u slučaju kada strukture sadrže mali broj članova, kako je to prikazano u sledećem primeru.

```
// The function dateAsString( ) converts a date from
a structure of type
// struct Date into a string of the form mm/dd/yyyy.
// Argument:     A date value of type struct Date.
// Return value: A pointer to a static buffer
containing the date string.

const char *dateAsString( struct Date d )
{
  static char strDate[12];
  sprintf( strDate, "%02d/%02d/%04d", d.month,
d.day, d.year );
  return strDate;
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

UGNJEŽDENE STRUKTURE

U nekom od prethodnih lekcija imali smo ugnježdenu if instrukciju unutar druge if instrukcije, kao i ugnježdenu petlju unutar druge petlje. Na sličan način koristimo ugnježdene strukture

Neka je potrebno napisati strukturu za osobu koja će osim podataka o imenu i visini, imati i podatak o datumu rođenja. Datum rođenja može biti definisan kao složeni tip podatka odnosno struktura koja će imati tri člana za tri podatka: dan, mesec i godina rođenja. Stoga, strukturu za datum rođenja možemo napisati na sledeći način:

```
struct Date
{
   int month;
   int day;
   int year;
};
```

Sada možemo deklarisati strukturu za osobu koja osim članova za ime i visinu, ima i član za datum koji je tipa Date

```
struct Person
{
   string name;
   int height;
   Date bDay;
};
```

U programu koji sledi su definisane dve funkcije za učitavanje odnosno štampanje podataka o elementima objekta strukturnog tipa. Funkcije kao argumente koriste pokazivače jer se na taj način vrši ušteda memorijskog prostora.

Funkcija koja vrši štampanje podataka na ekran kao argument preuzima pokazivač koji je setovan kao const da ne bi došlo do slučajnih izmena podataka u okviru funkcije:

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

struct Date
{
   int month;
   int day;
   int year;
};

struct Person {
   char name[20];
   int height;
   struct Date bDay;
};

void setValues(struct Person *pers);
void getValues(const struct Person *pers);

int main ()
{
   struct Person p1;
   setValues(&p1);
   printf("Outputting person data\n");
   printf("======================\n");
   getValues(&p1);
   return 0;
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

UGNJEŽDENE STRUKTURE

U nekom od prethodnih lekcija imali smo ugnježdenu if instrukciju unutar druge if instrukcije, kao i ugnježdenu petlju unutar druge petlje. Na sličan način koristimo ugnježdene strukture

Neka je potrebno napisati strukturu za osobu koja će osim podataka o imenu i visini, imati i podatak o datumu rođenja. Datum rođenja može biti definisan kao složeni tip podatka odnosno struktura koja će imati tri člana za tri podatka: dan, mesec i godina rođenja. Stoga, strukturu za datum rođenja možemo napisati na sledeći način:

```
struct Date
{
   int month;
   int day;
   int year;
};
```

Sada možemo deklarisati strukturu za osobu koja osim članova za ime i visinu, ima i član za datum koji je tipa Date

```
struct Person
{
   string name;
   int height;
   Date bDay;
};
```

U programu koji sledi su definisane dve funkcije za učitavanje odnosno štampanje podataka o elementima objekta strukturnog tipa. Funkcije kao argumente koriste pokazivače jer se na taj način vrši ušteda memorijskog prostora.

Funkcija koja vrši štampanje podataka na ekran kao argument preuzima pokazivač koji je setovan kao const da ne bi došlo do slučajnih izmena podataka u okviru funkcije:

```
void setValues(struct Person* pers)
{
   printf("Enter person's name: ");
   scanf("%s", pers->name);
   printf("Enter height in inches: ");
   scanf("%d",&pers->height);

   printf("Enter month, day and year of
birthday separated by spaces: ");
   scanf("%d%d%d", &pers->bDay.month, &pers-
>bDay.day, &pers->bDay.year);
}

void getValues(const struct Person* pers)
{
   printf("Person's name: %s\n", pers->name);
   printf("Person's height in inches is: %d\n",
pers->height);
   printf("Person's birthday in mm/dd/yyyy
format is: %d/%d/%d",
      pers->bDay.month, pers->bDay.day, pers-
>bDay.year);
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Strukture i pokazivači

Strukture i pokazivači, pokazivač na strukturu

- *Upotreba pokazivača kod struktura*
- *Pokazivači na strukture*
- *Pokazivač na strukturu kao argument funkcije*

02

UPOTREBA POKAZIVAČA KOD STRUKTURA

Pokazivači i strukture se javljaju zajedno u sledećim slučajevima: pokazivač kao element strukture, pokazivači na strukture, i pokazivač na strukturu je element strukture

Strukture i pokazivači se mogu u C programima pojaviti kroz više različitih funkcionalnih veza. Osnovni oblici korišćenja pokazivača sa strukturama su:

- pokazivači kao elementi strukture
- pokazivači na strukture
- pokazivač na strukturu je element strukture

U prvom slučaju pokazivač na neki tip podatka je element strukture. Pristup ovim članovima predstavlja kombinaciju upotrebe operatora posrednog pristupa "*" i operatora za pristup elementima strukture ".". Opšti oblik sintakse za upotrebu pokazivača u strukturama je:

```
*ime_strukture.ime_pointera
```

Analogno pokazivačima na osnovne tipove, mogu se deklarisati pokazivači na strukture. Potpuno je isti način inicijalizovanja pokazivača, u ovom slučaju dodeljuje mu se adresa strukture. Pristup elementima strukture u ovom slučaju se vrši korišćenjem operatora "->" jer je, iako ispravano, korišćenje tačke ". " povezano sa višestrukom upotrebom operatora "*" i zagrada, što program čini nepreglednijim. Sintaksa ima sledeći izgled:

```
ime_pointera_strukture->ime_elementa_strukture
```

ili manje uobičajen način:

```
(*ime_pointera_strukture).ime_elementa_strukture
```

Ako se u strukturi kojoj se pristupa preko pokazivača nalazi pokazivač, onda se takvom elementu pristupa na sledeći način:

```
*ime_pointera_strukture->ime_pointera_u_strukturi
```

ili (ispravno i neuobičajeno):

```
*(*ime_pointera_strukture).ime_pointera_u_strukture
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

POKAZIVAČI NA STRUKTURE

Pokazivač na strukturu se može definisati na sličan način kao i pokazivač na bilo koju drugu promenljivu. Kada koristimo pokazivače, elementu strukture se pristupa korišćenjem „->“

Pokazivač na strukturu se može definisati na sličan način kao i pokazivač na bilo koju drugu promenljivu, kao u sledećem primeru:

```
struct Books *struct_pointer;
```

Sada je moguće čuvati adresu strukturne promenljive u prethodno definisanoj pokazivačkoj promenljivoj. Da bi se adresa strukturne promenljive dodelila pokazivaču neophodno je koristiti adresni operator & ispred imena strukturne promenljive:

```
struct_pointer = &Book1;
```

Da bi ste pristupili članu strukture korišćenjem pokazivača na strukturu, koristi se operator -> , umesto operatora pristupa (.), na sledeći način:

```
struct_pointer->title;
```

Pogledajmo sada ponovo primer gde smo izvršili deklaraciju promenljivih za rad sa mužičkim numerama korišćenjem pokazivača (kod je u nastavku).
Pošto pokazivač pSong pokazuje na objekat song1, izraz *pSong ustvari predstavlja vrednost onoga na šta pSong pokazuje a to je song1. Stoga se iskaz (*pSong).duration odnosi na član duration promenljive song1. Korišćenje zagrada je neophodno jer operator tačka “.” ima veću prednost od indirektnog operatora (*).

```
#include <string.h>        // Prototypes of string
functions.
Song_t song1,*pSong;       // One object of type Song_t,
                                        // and a pointer
to Song_t.

// Copy a string to the title of song1:
strcpy( song1.title, "Havana Club" );
// Likewise for the composer member:
strcpy( song1.composer, "Ottmar Liebert" );
song1.duration = 251;                // Playing time.
// The member published is itself a structure:
song1.published.year = 1998;         // Year of publication.

*pSong = &song1;
if ( (*pSong).duration > 180 )
  printf( "The song %s is more than 3 minutes long.\n",
(*pSong).title );
```

Kao što smo već napomenuli, ukoliko imamo pokazivač na strukturu, onda je moguće koristiti operator strelice, odnosno ->, da bi se pristupilo članovima strukturne promenljive na koji pokazivač pokazuje, umesto indirektnog i “tačka” operatora (* i .). Drugim rečima, izraz oblika p->m je ekvivalentan izrazu(*p).m. Stoga je moguće if iskaz u prethodnom primeru napisati na sledeći način:

```
if ( pSong->duration > 180 )
  printf( "The song %s is more than 3 minutes long.\n", pSong-
>title );
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

POKAZIVAČ NA STRUKTURU KAO ARGUMENT FUNKCIJE

Ogromne strukture se uglavnom prosleđuju po adresi jer se na taj način ne vrši kopiranje vrednosti u fiktivne parametre i ne zauzima se nepotrebno mesto u stek memoriji

Sada možemo da izmenimo prethodni program gde smo koristili strukturnu promenljivu kao argument funkcije, i da napišemo nove funkcije koje će kao argumente imati pokazivače na strukture. Deklaracija funkcije koja štampa podatke o knjizi korišćenjem pokazivača kao argumenata bi imala sledeći oblik

```
/* function declaration */
void printBook( const struct Books *book );
```

Funkcija se iz glavnog programa poziva na sledeći način:

```
/* print Book1 info by passing address of Book1 */
printBook( &Book1 );
```

dok bi definicija funkcije korišćenjem pokazivača izgledala ovako:

```
void printBook( const struct Books *book )
{
   printf( "Book title : %s\n", book->title);
   printf( "Book author : %s\n", book->author);
   printf( "Book subject : %s\n", book->subject);
   printf( "Book book_id : %d\n", book->book_id);
}
```

Ogromne strukture se uglavnom prosleđuju po adresi, odnosno po pokazivaču. U narednom primeru se kopira samo adresa promenljive tipa Song, ne ceo sadržaj strukture. Osim toga, ukoliko se unapred zna da neće biti vršene izmene nad objektom onda se parametar funkcije može definisati kao “read-only” odnosno konstantan objekat korišćenjem rezervisane reči const.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Unije

Unije,deklaracija i primena, strukture i unije

- *Deklaracija unije*
- *Memorijski prostor kod upotrebe unija*
- *Pristup članu unije*
- *Unije kao članovi strukture*

03

DEKLARACIJA UNIJE

Osnovna svrha unija je ušteda memorije jer se dozvoljava da se različiti tipovi podataka smeštaju u istoj memorijskoj lokaciji

Unija (union**)** je specijalni tip podatka u C-u koja dozvoljava da se različiti tipovi podataka smeštaju u istoj memorijskoj lokaciji. Unija može da bude definisana tako da ima više članova, ali samo jedan član unije u datom vremenskom trenutku može da sadrži vrednost. Unije obezbeđuju efikasan način korišćenja jedne iste memorijske lokacije za višestruku upotrebu. Osnovna svrha unija je ušteda memorije.

```
union [naziv unije]
{
   member definition;
   member definition;
   ...
   member definition;
} [jedna ili više promenljivih];
```

Pri deklaraciji unije važe potpuno identična pravila kao kod struktura. U nastavku je dat primer definisanja unije kao tipa podatka čiji je naziv Data, i koja ima tri članice i, f, i str:

```
union Data
{
   int i;
   float f;
   char  str[20];
} data;
```

Sada, promenljiva koja je tipa **Data** može u jednom vremenskom trenutku da čuva ili ceo broj, ili realan broj float, ili niz karatera. Ovo znači da jedna promenljiva, tj. jedna memorijska lokacija može da se koristi za čuvanje različitih podataka. Bilo koji standardni ili korisnički definisani tip podatka može biti korišćen za definisanje tipa promenljive koja je članica unije.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

MEMORIJSKI PROSTOR KOD UPOTREBE UNIJA

Memorijska lokacija koja je zauzeta pri deklaraciji unije će biti dovoljno velika da se u njoj smesti najveći član unije

Memorijska lokacija koja je zauzeta pri deklaraciji unije će biti dovoljno velika da se u njoj smesti najveći član unije. Tako, u prethodnom primeru, tip Data će zauzeti 20 bajtova memorijskog prostora. U nastavku je dat primer koji štampa ukupan memorijski prostor koji je zauzela promenljiva tipa union Data:

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

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;

   printf( "Memory size occupied by data : %d\n", sizeof(data));

   return 0;
}
```

Rezultat programa biće:

```
Memory size occupied by data : 20
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRISTUP ČLANU UNIJE

Pristup članu unije se ostvaruje na isti način kao kod struktura korišćenjem tačke koja se navodi između naziva promenljive i naziva člana unije kome se pristupa

U nastavku je dat primer koji detaljnije opisuje upotrebu unije:

```
#include <stdio.h>
#include <string.h>
union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

Rezultat prethodnog programa biće:

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

U prethodnom primeru možemo videti da su vrednosti članica i i f nedostupni, jer je poslednja operacija dodele pomoću strcpy zauzela memorijsku lokaciju za uniju pa je stoga vidljiv samo član str koji je i oštampan kako treba. Sada pogledajmo izmenjeni primer, gde se svaka promenljiva koristi u različitim vremenskim trenucima što je i svrha korišćenja unija u programu:

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

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

UNIJE KAO ČLANOVI STRUKTURE

Unije se često koriste i kao članovi struktura. Pravila su gotovo slična kao i kod korišćenja ugnježdenih struktura

Unije se često koriste i kao članovi struktura. Neka se, na primer, u programu čuvaju i obrađuju informacije o studentima i zaposlenima na nekom fakultetu. Za svakoga se čuva ime, prezime i matični broj, za zaposlene se još čuva i koeficijent za platu, dok se za studente čuva broj indeksa:

```
struct akademac
{
            char ime_i_prezime[50];
            char jmbg[14];
            char vrsta;
            union {
                        double plata;
                        char indeks[7];
            } dodatno;
};
```

U ovom slučaju poslednji član strukture (dodatno) je unijskog tipa (sam tip unije nije imenovan). Član strukture *vrsta* tipa char sadrži informaciju o tome da li se radi o zaposlenom (na primer, vrednost z) ili o studentu (na primer, vrednost s). Promenljive plata i indeks dele zajednički memorijski prostor i podrazumeva se da se u jednom trenutku koristi samo jedan podatak u uniji.

Na primer:

```
int main()
{
            struct akademac pera = {
                        "Pera Peric",
"0101970810001", 'z'};
            struct akademac ana = {
                        "Ana Anic", "1212992750501",
's'};

            pera.dodatno.plata = 56789.5;
            printf("%f\n", pera.dodatno.plata);
            strcpy(ana.dodatno.indeks, "12/123");
            printf("%s\n", ana.dodatno.indeks);
}
```

Pokušaj promene polja pera.dodatno.indeks bi narušio podatak o koeficijentu plate, dok bi pokušaj promene polja ana.dodatno.koeficijent narušio podatak o broju indeksa.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Polja bitova

Bolja bitova, stukture

- *Osnovi o polju bitova*

04

OSNOVI O POLJU BITOVA

Definisanjem polja bitova moguće je deo memorije iscepkati u individualne grupe kojima se posle može pristupiti preko imena

Još jedan od načina uštede memorije u C programima su polja bitova (engl. bit fields). Naime, najmanji celobrojni tip podataka je char koji zauzima jedan bajt, a za predstavljanje neke vrste podataka dovoljan je manji broj bitova. Na primer, zamislimo da želimo da predstavimo grafičke karakteristike pravougaonika u nekom programu za crtanje. Ako je dopušteno samo osnovnih 8 boja (crvena, plava, zelena, cijan, magenta, žuta, bela i crna) za predstavljanje boje dovoljno je 3 bita. Vrsta okvira (pun, isprekidan, tačkast) može da se predstavi sa 2 bita. Na kraju, da li pravougaonik treba ili ne treba popunjavati možemo kodirati sa jednim bitom. Ovo možemo iskoristiti da definišemo sledeće polje bitova.

```
struct osobine_pravougaonika {
          unsigned char popunjen : 1;
          unsigned char boja : 3;
          unsigned char vrsta_okvira : 2;
};
```

Iza svake promenljive naveden je broj bitova koji je potrebno odvojiti za tu promenljivu. Veličina ovako definisanog polja bitova (vrednost izraza sizeof (struct osobine_pravougaonika)) je samo 1 bajt (iako je potrebno samo 7 bitova, svaki podatak mora da zauzima ceo broj bajtova, tako da je odvojen 1 bit više nego sto je potrebno). Da je u pitanju bila obična struktura, zauzimala bi 3 bajta. Polje bitova se nadalje koristi kao obična struktura. Na primer:

```
...
#define ZELENA 02
...
#define PUN 00
struct osobine_pravougaonika op;
op.popunjen = 0;
op.boja = ZELENA;
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Dinamičko upravljanje memorijom

Upravljanje memorijom, organizacija memorije

1. *Alociranje memorije*
2. *Realociranje i oslobađanje memorije*
3. *Upravljanje memorijom u teoriji i praksi*
4. *Upravljanje memorijom - rad sa strukturama i funkcijama*

05

UVOD U DINAMIČKO UPRAVLJANJE MEMORIJOM

U velikom broju problema broj podataka sa kojima se operiše nije unapred poznat. Programski jezik C obezbeđuje nekoliko funkcija za alociranje i upravljanje memorijom

Pomoću tipova podataka koje smo do sada upoznali može se obaviti širok spektar zadataka i problema. Međutim, postoji ogroman deo zadataka gde broj objekata sa kojima se radi nije unapred poznat, pa je u tom slučaju neophodno procenjivati koliko će ih biti. U ovom delu će biti opisan postupak dinamičkog upravljanja memorijom u C-u. Programski jezik C obezbeđuje nekoliko funkcija za alociranje i upravljanje memorijom. Ove funkcije su deo **<stdlib.h>** fajla zaglavlja. Prostor za dinamički alociranu memoriju nalazi se u segmentu memorije koji se zove hip (engl. heap). U nastavku je dat spisak osnovnih funkcija za upravljanje memorijom u C-u

	Naziv funkcije i opis
1	**void *calloc(int num, int size);** Funkcija koja alocira niz od **num** elemenata gde je memorija neophodna za svaki element niza veličine **size** bajtova.
2	**void free(void *address);** Funkcija koja oslobađa blok memorije predstavljen pomoću adrese
3	**void *malloc(int num);** Funkcija koja alocira niz od **num** bajtova.
4	**void *realloc(void *address, int newsize);** Funkcija realocira memoriju tako da zauzima novu veličinu prostora, specificiranu veličinom **newsize** u bajtovima

Slika-1 Osnovne funkcije za upravljanje memorijom u C-u

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Alociranje memorije

Alociranje memorije,malloc, calloc

- *Funkcije za dinamičko rezervisanje memorije*
- *Upotreba funkcija malloc() i calloc()*
- *Konverzija pri alociranju memorije*

05

FUNKCIJE ZA DINAMIČKO REZERVISANJE MEMORIJE

Za odvajanje (alociranje) memorije u C-u se koriste dve funkcije i to: malloc() i calloc()

Tokom programiranja, ukoliko unapred znate veličinu niza koga ćete koristiti da bi ste rešili neki problem, onda su stvari jasne i onda možete koristiti statički niz. Na primer, ukoliko želite da čuvate ime neke osobe i ukoliko znate da ime neće imati više od 100 karaktera, onda je moguće deklarisati niz na sledeći način:

```
char name[100];
```

Ali, uzmimo sada u obzir situaciju gde nemamo predstavu o tome kolika će biti dužina teksta koju želimo da smestimo u memoriji, kao što je npr. detaljan opis neke teme. U ovom slučaju, nama je potrebno da definišemo pokazivač na karakter bez specificiranja niza, odnosno koliko će maksimalno elemenata da bude, da bi kasnije mogli po potrebi da alociramo neophodnu veličinu memorije. Dve funkcije koje se koriste za alociranje memorije malloc( ) i calloc( ) imaju za nijansu različite parametre:

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

Prethodna funkcija rezerviše kontinualni blok memorije čija je veličina size bajtova. Kada program zauzme ovaj deo memorije, sadržaj ostaje nedefinisan. Funkciju calloc možemo napisati na sledeći način:

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

Ova funkcija rezerviše blok memorije čija je veličina označena sa count x size bajtova. Drugim rečima, ovaj blok memorije je dovoljno veliki da se u njemu čuva niz od count elemenata, gde svaki element ima veličinu od size bajtova. Osim toga, funkcija calloc( ) inicijalizuje svaki bajt odvojene memorije tako da ima vrednost 0.

Dinamički objekti alocirani navedenim funkcijama su neimenovani (tipa void) i bitno su različiti od promenljivih. Ipak, dinamički alociranim blokovima se pristupa na sličan način kao nizovima. Vrednost pokazivača je adresa prvog bajta memorije u alociranom memorijskom bloku, ili nul pokazivač ukoliko memorija nije uspešno alocirana.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

UPOTREBA FUNKCIJA MALLOC() I CALLOC()

Da bi ste koristili funkcije malloc i calloc neophodno je pretprocesorskom direktivom uključiti fajl zaglavlja u kome se nalaze deklaracije ovih funkcija (#include <stdlib.h>)

U sledećem primeru je opisano korišćenje funkcije malloc u cilju dinamičkog alociranja memorije:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
   char name[100];
   char *description;

   strcpy(name, "Marko Markovici");
   /* allocate memory dynamically */
   description = malloc( 200 * sizeof(char) );
   if( description == NULL )
   {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   }
   else
   {
      strcpy( description, "Marko Markovic, student Metropolitana");
   }
   printf("Name = %s\n", name );
   printf("Description: %s\n", description );
}
```

Prethodni program može veoma slično da se napiše korišćenjem funkcije calloc(), umesto funkcije malloc(), na sledeći način:

```
calloc(200, sizeof(char));
```

Na ovaj način imamo veću fleksibilnost u programu jer imamo mogućnost da menjamo vrednost zauzete memorije, što ne bi smo mogli u slučaju da smo unapred deklarisali niz fiksne dužine.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

KONVERZIJA PRI ALOCIRANJU MEMORIJE

Konverzija pri dinamičkom alociranju se vrši automatski, ali je praksa da se vrši eksplicitna konverzija korišćenjem cast-operatora

Pogledajmo sledeći primer:

```
#include <stdlib.h>  // Provides function prototypes.
typedef struct {
            long key;
    /* ... more members ... */
} Record;            // A structure type.

float *myFunc( size_t n )
{
  // Reserve storage for an object of type double.
  double *dPtr = malloc( sizeof(double) );
  if  ( dPtr == NULL )// Insufficient memory.
  {
    /* ... Handle the error ... */
    return NULL;
  }
  else                // Got the memory: use it.
  {
    *dPtr = 0.07;
    /* ... */
  }

  // Get storage for two objects of type Record.
  Record *rPtr;
  if  ( ( rPtr = malloc( 2 * sizeof(Record) ) == NULL )
  {
    /* ... Handle the insufficient-memory error ... */
    return NULL;
  }
  // Get storage for an array of n elements of type float.
  float *fPtr = malloc( n * sizeof(float) );
  if ( fPtr == NULL )
  {
    /* ... Handle the error ... */
    return NULL;
  }
  /* ... */
  return fPtr;
}
```

Kada program dodeli neimenovani pokazivač pokazivačkoj promenljivoj nekog drugog tipa, kompajler automatski vrši odgovarajuću i prikladnu konverziju (cast). Neki programeri međutim imaju naviku da eksplicitno vrše konverziju tipova. Kada pristupate lokaciji u alociranom memorijskom bloku, tip pokazivača je taj koji određuje na koji način će sadržaj memorijskog bloka biti interpretiran, kao što je pokazano u prethodnom primeru.

Nekada je veoma korisno inicijalizovati nulom svaki bajt alocirane memorije, i u tim slučajevima je pogodnije koristiti funkciju calloc( ) umesto funkcije malloc( ). Veličina bloka memorije koji će biti alociran se drugačije izražava korišćenjem funkcije calloc( ). Moguće je u prethodnom primeru zameniti linije teksta gde se vrši alociranje memorije, na sledeći način:

```
// Get storage for an object of type double.
double *dPtr = calloc( 1, sizeof(double) );

// Get storage for two objects of type Record.
Record *rPtr;
if  ( ( rPtr = calloc( 2, sizeof(Record) ) == NULL )
{  /* ... Handle the insufficient-memory error ...
*/  }

// Get storage for an array of n elements of type
float.
float *fPtr = calloc( n, sizeof(float));
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Realociranje i oslobađanje memorije

Realociranje, oslobađanje memorije, realoc, free

- *Funkcije za realociranje i oslobađanje memorije*
- *Upotreba funkcija realoc() i free()*

05

FUNKCIJE ZA REALOCIRANJE I OSLOBAĐANJE MEMORIJE

Realociranje već odvojene memorije se vrši korišćenjem funkcije realloc(), dok se oslobađanje memorije vrši korišćenjem funkcije free()

Kada više ne postoji potreba za dinamički alociranom memorijom, onda tu memoriju treba osloboditi i vratiti je na korišćenje operativnom sistemu. Oslobađanje memorije se ostvaruje pozivom funkcije free( ). Osim toga, ukoliko je potrebno smanjiti ili povećati veličinu već alociranog memorijskog bloka onda je to moguće uraditi korišćenjem funkcije realloc( ). Prototipovi ovih funkcija su dati u nastavku:

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

Ova funkcija oslobađa dinamički alociran blok memorije na adresi na koju pokazuje ptr. Dozvoljeno je da argument funkcije bude nul pokazivač i tada poziv funkcije neće imati nikakvog efekta. Poziv free(*ptr*) oslobađa memoriju na koju ukazuje pokazivač *ptr* (a ne memorijski prostor koji sadrži sam pokazivač ptr), pri čemu je neophodno da ptr pokazuje na blok memorije koji je alociran pozivom funkcije malloc ili calloc. Opasna je greška pokušaj oslobađanja memorije koja nije alocirana na ovaj način. Takođe, ne sme se koristiti nešto što je već oslobođeno niti se sme dva puta oslobađati ista memorija. Redosled oslobađanja memorije ne mora da odgovara redosledu alociranja.

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

Ova funkcija oslobađa memorijski blok na koji pokazuje ptr i alocira novi memorijski blok veličine size bajtova. Novi memorijski blok može da počinje od iste adrese kao i stari, oslobođeni blok.

Funckija realloc( ) kao rezultat vraća nul pokazivač ukoliko nije mogla da odvoji memorijski prostor dužine size bajtova. U tom slučaju funkcija neće da oslobodi stari memorijski blok niti da ošteti njegov sadržaj.

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. Međutim, ukoliko iza postojećeg bloka ne postoji dovoljno slobodnog prostora, onda se u memoriji traži drugo mesto koje je dovoljno da prihvati prošireni blok i, ako se nade, sadržaj postojećeg bloka se kopira na to novo mesto i zatim se stari blok memorije oslobađa. Ova operacija može biti vremenski zahtevna.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

UPOTREBA FUNKCIJA REALOC() I FREE()

Ukoliko neki dinamički alocirani blok nije oslobođen ranije, on će biti oslobođen prilikom završetka rada programa, zajedno sa svom drugom memorijom koja je dodeljena programu

Ukoliko neki dinamički alociran blok nije oslobođen ranije, on će biti oslobođen prilikom završetka rada programa, zajedno sa svom drugom memorijom koja je dodeljena programu. Ipak, ne treba se oslanjati na to i preporučeno je eksplicitno oslobađanje sve dinamički alocirane memorije pre kraja rada programa, a poželjno čim taj prostor više nije potreban. U nastavku je dat primer korišćenja funkcija realloc() i free().

Može se u narednom programu probati bez realociranja memorije, ali će se pojaviti greška prilikom poziva funkcije strcat() jer neće imati dovoljno memorije za izvršavanje spajanja stringova.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
   char name[100];
   char *description;
   strcpy(name, "Marko Markovic");
   description = malloc( 30 * sizeof(char) );
   if( description == NULL )
   {
      fprintf(stderr, "Error - unable to allocate
required memory\n");
   }
   else
   {
      strcpy( description, "Marko je student
Metropolitana.");
   }
   description = realloc( description, 100 *
sizeof(char) );
   if( description == NULL )
   {
      fprintf(stderr, "Error - unable to allocate
required memory\n");
   }
   else
   {
      strcat( description, "She is in class 10th");
   }
   printf("Name = %s\n", name );
   printf("Description: %s\n", description );
   free(description);
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Upravljanje memorijom u teoriji i praksi

Stek, hip, karakteristika memorije, curenje memorije

- *Stek i hip memorija*
- *Karateristike zauzete memorije*

05

STEK I HIP MEMORIJA

O stek memoriji je već bilo reči u lekciji o funkcijama. Hip je deo memorije sa kojim se operiše pri dinamičkom alociranju, realociranju i oslobađanju memorije

Način organizovanja i korišćenja memorije u fazi izvršavanja programa može se razlikovati od jednog do drugog operativnog sistema. Tekst u nastavku odnosi se, ako to nije drugačije naglašeno, na širok spektar platformi, pa su, zbog toga, načinjena i neka pojednostavljenja. Kada se izvršni program učita u radnu memoriju računara, biva mu dodeljena određena memorija i započinje njegovo izvršavanje. Dodeljena memorija organizovana je u nekoliko delova koje zovemo segmenti ili zone:

- segment koda (engl. code segment ili text segment);
- segment podataka (engl. data segment);
- stek segment (engl. stack segment);
- hip segment (engl. heap segment).

Stek memorija

U stek segmentu (koji se naziva i programski stek poziva (engl. call stack)) čuvaju se svi podaci koji karakterišu izvršavanje funkcija. Podaci koji odgovaraju jednoj funkciji (ili, preciznije, jednoj instanci jedne funkcije — jer, na primer, rekurzivna funkcija može da poziva samu sebe i da tako u jednom trenutku bude aktivno više njenih instanci) organizovani su u takozvani stek okvir (engl. stack frame).

Stek okvir jedne instance funkcije obično, između ostalog, sadrži:

- argumente funkcije;
- lokalne promenljive (promenljive deklarisane unutar funkcije);
- međurezultate izračunavanja;
- adresu povratka (koja ukazuje na to odakle treba nastaviti izvršavanje programa nakon povratka iz funkcije);
- adresu stek okvira funkcije pozivaoca.

Hip memorija

Hip memorija je neiskorišćena memorija programa koja može biti upotrebljena da se u toku izvršavanja programa dinamički zauzme za određene potrebe. Mnogo puta, programeri nisu unapred svesni koliko im je potrebno memorije za čuvanje određenog tipa informacija ili podataka već će taj podatak o veličini memorije biti određen tek u toku izvršavanja programa. Kao što smo već napomenuli, u toku izvršavanja programa moguće je dinamički alocirati memoriju u hipu korišćenjem malloc ili calloc naredbi. Korišćenjem naredbe free izvršiće se oslobađanje memorijskog prostora koji je deo hip memorije.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

KARATERISTIKE ZAUZETE MEMORIJE

Neophodno je znati karakteristike dinamički alocirane memorije, jer nepoznavanje pravila može dovesti do nepredviđenih situacija tokom izvršavanja vašeg programa

Uspešno odvojena memorija pozivom neke od funkcija za alociranje rezultira u formiranju pokazivača na početak memorijskog bloka. "Početak" memorijskog bloka znači da je vrednost pokazivača jednaka adresi najmanjeg bajta u bloku memorije. Memorijski blok je takav da se u njega može smestiti bilo koji tip podatka.

Alocirani memorijski blok ostaje rezervisan ili do kraja programa ili do ekplicitnog oslobađanja ili realociranja memorije korišćenjem funkcija free( ) ili realloc( ). Drukčije rečeno, vreme zauzeća memorije traje od trenutka odvajanja do trenutka oslobađanja, odnosno do trenutka prekida izvršenja programa.

Ono što treba znati je da organizacija memorije nakon uzastopnih poziva funkcija malloc( ), calloc( ), i/ili realloc( ) nije specificirana, tj ne mora da znači da će se zauzeti susedni memorijski blokovi.

Takođe nije specificirano da li zahtev za alociranjem bloka memorije veličine 0 rezultira u formiranju nul pokazivača ili pokazivač dobija neku uobičajenu vrednost. U svakom slučaju, međutim, ne postoji način da se koristi pokazivač na blok memorije veličine 0 bajtova, osim možda u slučaju kao argument funkcija realloc( ) ili free( ).

Najznačajnija karakteristika dinamičkog alociranja memorije nije opseg važenja već životni vek. Kao i kod bilo koje globalne ili statičke promenljive, dinamički alociran prostor živi sve dok se program izvršava. Međutim, ukoliko je pre završetka programa pokazivač koji pokazuje na dinamički alociranu promenljivu izašao iz opsega važenja, onda je izgubljen pristup dinamički kreiranoj memoriji. U tom slučaju dinamički kreirana promenljiva i dalje zauzima memoriju, ali se tom delu memorije ne može pristupiti. Ovakav problem se naziva curenje memorije (memory leak).

S obzirom na prethodnu konstantaciju, pošto je pokazivač jedini način da se pristupi dinamički alociranoj memoriji, treba voditi računa i o tome da se vrednost pokazivača ne promeni da pokazuje na neku adresu, osim ako je prethodno ta dinamička adresa referencirana korišćenjem drugog pokazivača. Ukoliko se to ne odradi kako treba opet će se pojaviti curenje memorije.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Upravljanje memorijom - rad sa strukturama i funkcijama

Dinamičko upravljanje memorijom, funkcije, strukture

- *Dinamičko alociranje i strukture*
- *Rezultat funkcije je pokazivač na dinamički alociranu memoriju*

05

DINAMIČKO ALOCIRANJE I STRUKTURE

Struktura takodje može biti korišćena u kombinaciji sa alociranjem memorije pomoću funkcije malloc()

U nastavku je dat primer:

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

typedef struct rec
{
                    int i;
                    float PI;
                    char A;
}RECORD;

int main()
{
                    RECORD *ptr_one;

                    ptr_one = (RECORD *) malloc (sizeof(RECORD));

                    (*ptr_one).i = 10;
                    (*ptr_one).PI = 3.14;
                    (*ptr_one).A = 'a';

                    printf("First value: %d\n",(*ptr_one).i);
                    printf("Second value: %f\n", (*ptr_one).PI);
                    printf("Third value: %c\n", (*ptr_one).A);

                    free(ptr_one);

                    return 0;
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

REZULTAT FUNKCIJE JE POKAZIVAČ NA DINAMIČKI ALOCIRANU MEMORIJU

Dinamički alocirana promenljiva može da bude rezultat funkcije. Rezultat se u tom slučaju preko pokazivača vraća pozivaocu funkcije

Već smo u nekoj od prethodnih lekcija pomenuli da rezultat funkcije može biti pokazivač, i da su česti primeri da taj pokazivač pokazuje na memoriju koja je dinamički alocirana u okviru funkcije. Pogledajmo sledeći primer:

```
#include <stdio.h>
#include <stdlib.h>
char * setName();
int main (void)
{
          char* str;
          str = setName();
          printf("%s",str);
          free(str);
          return 0;
}
char* setName (void)
{
          char* name;
          name = malloc(80*sizeof(char));
          printf("Enter your name: ");
          scanf("%s", name);
          return name;
}
```

Prethodni deo koda radi jer pokazivač koji je vraćen pozivaocu iz funkcije setName pokazuje na niz karaktera čiji se životni vek, s obzirom da je definisana korišćenjem dinamičkog alociranja, ne završava izlaskom iz funkcije i povratkom u glavni program.

Ono što ovde treba napomenuti je to da ako se memorija dinamički alocira unutar funkcije korišćenjem nekog lokalnog pokazivača, taj pokazivač će biti uništen po završetku funkcije ali ne i memorija koja je dinamički alocirana jer ne postoji način da joj se pristupi i da se onda funkcijom free() oslobodi. Ovakav problem je izbegnut u prethodnom primeru pošto je adresa lokalnog pokazivača kao rezultat vraćena pozivaocu funkcije setName i dodeljena drugoj promenljivoj str koja živi u okviru main funkcije. Pokazivačka promenljiva str se zatim koristi do kraja main funkcije gde se poziva funkcija free() koja oslobađa memorijski prostor dinamički alociran u okviru funkcije setName.

Prethodni primer je ustvari primer koji ilustruje korišćenje različitih pokazivača koji pokazuju na istu memorijsku lokaciju. Međutim, u slučaju dinamički kreirane memorije sasvim je svejedno nad kojim pokazivačem ćete izvršiti oslobađanje memorije funkcijom free(). Jedino je bitno da funkciju free() koristite samo za onaj prostor koji je dinamički alociran. Ukoliko je dinamički alocirana memorija već oslobođena korišćenjem free(), ponovno korišćenje funkcije free() nad istom lokacijom može dovesti do nepredvidljivog rezultata.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Pretprocesorske naredbe

Pretprocesor, pretprocesorske naredbe

- *Uvod u pretprocesorske direktive*
- *Uključivanje datoteka*
- *Makro zamena – osnovna primena*
- *Makro zamena – dodatne mogućnosti*
- *Parametrizovani makroi*
- *Uslovno prevođenje*

06

UVOD U PRETPROCESORSKE DIREKTIVE

Pretprocesiranje predstavlja pripremnu fazu pre kompajliranja u kojoj se vrše određene transformacije ulaznog teksta, brisanje komentara itd. Sve direktive započinju znakom #

Programski jezik C omogućuje određene olakšice ako se koristi preprocesor kao prvi korak u prevođenju. Pretprocesiranje, dakle, predstavlja samo pripremnu fazu, pre kompajliranja. Suštinski, pretprocesor vrši samo jednostavne operacije nad tekstualnim sadržajem programa i ne koristi nikakvo znanje o jeziku C. Pretprocesor ne analizira značenje naredbi napisanih u jeziku C već samo pretprocesorske direktive (engl. preprocessing directive) (one se ne smatraju delom jezika C) na osnovu kojih vrši određene transformacije teksta izvornog programa. Dve konstrukcije su najčešće u upotrebi. Jedna je #include, za uključivanje datoteka za prevođenje, a druga je #define, koja zamenjuje simbol proizvoljnim nizom znakova. Ostale konstrukcije opisane u ovom delu odnose se na uslovno prevođenje i makro zamenu sa argumentima.

Sve pretrocesorske naredbe počinju znakom taraba (#). U nastavku je data lista svih najbitnijih pretprocesorskih direktiva:

Direktiva	Opis
#define	Pretprocesorska marko zamena
#include	Uključivanje zaglavlja iz nekog drugog fajla
#undef	Poništavanje definicije pretprocesorskog makroa
#ifdef	Vraća tačno (true) ako je makro definisan
#ifndef	Vraća tačno (true) ako makro nije definisan
#if	Ispituje da li su ispunjeni odredjeni uslovi
#else	Alternativa za #if
#elif	Skraćeni zapis za #else #if
#endif	Oznaka za kraj pretprocesorskog uslova
#error	Štampa poruku o grešci na standardnom izlazu stderr
#pragma	Prouzrukuje sprovođenje specijanih komandi kompajlera, korišćenjem standardizovanih metoda

Slika-1 Lista najbitnijih direktiva C-pretprocesora

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

UKLJUČIVANJE DATOTEKA

Uključivanje datoteke olakšava manipulaciju funkcijama i deklaracijama. Razlikujemo uključivanje datoteka standardnih biblioteka i uključivanje korisnički definisanih datoteka

Veliki programi obično su podeljeni u više datoteka, radi preglednosti i lakšeg održavanja. Uključivanje datoteke olakšava manipulaciju funkcijama, deklaracijama i #define skupovima. Svaka linija oblika

```
#include "ime datoteke"
```

ili

```
#include <ime datoteke>
```

se menja sadržajem imena datoteke. Ako je ime datoteke pod navodnicima, onda je pretpostavka da je ona korisnički definisana, i počinje se tražiti u direktoriju u kojem je i izvorni program. Ako se ona tamo ne nalazi, potraga se nastavlja od mesta koje je definisano implementacijom prevodioca (moguće je definisati mesta na kojima će se tražiti datoteke iz #include konstrukcije).
Često se može pojaviti nekoliko #include linija na početku izvorne datoteke, koje uključuju zajedničke #define naredbe i extern deklaracije ili eventualno pristupaju deklaracijama prototipa funkcije iz standardne biblioteke (kako bi bili do kraja precizni, to ne moraju biti datoteke jer način pristupanja datotekama zavisi od implementacije).

Naredba #include je najbolja za međusobno povezivanje deklaracija velikog programa. Ona garantuje da će sve izvorne datoteke videti definicije i deklaracije promenljivih, što će eliminisati razne poteškoće. Naravno, kad se posmatrana datoteka izmeni, sve datoteke koje je spominju u izvornom kodu, moraju se ponovo prevoditi.

❑ Ugnježdene #include direktive

Direktive #include mogu biti ugnježdene tj. može da se desi da fajl koji je uključen u drugi fajl direktivom #include ustvari već sadrži #include direktivu. Pretprocesor dozvoljava do 15 nivoa ugnježdenih direktiva. S obzirom da fajlovi zaglavlja međusobno uključuju jedan drugog, može lako da se desi da je jedan isti fajl uključen više puta direktivom #include. Na primer, pretpostavimo da fajl *myProject.h* sadrži sledeću liniju:

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

Tako će izvorni fajl koji sadrži sledeće #include direktive uključiti fajl *stdio.h* dva puta, prvi direktno a drugi indirektno:

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

Jedan od načina da se spreči ova pojava je korišćenje uslovnog prevođenja što će biti opisano u nastavku lekcije.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

MAKRO ZAMENA – OSNOVNA PRIMENA

Pretprocesorska direktiva #define omogućava zamenjivanje niza karaktera u datoteci, tj. makroa (engl. macro) drugim nizom karaktera pre samog prevođenja

Definicija makro zamene ima oblik:

```
#define ime tekst zamene
```

Prethodna naredba vrši makrozamenu u najprostijem obliku, tj. učestala pojavljivanja imena simbola zameniće se tekstom zamene. Ime u #define naredbi istog je oblika kao i ime promenljive, dok je tekst zamene proizvoljan. Jasno, tekst zamene je ostali deo linije, mada dugačka definicija može prelaziti preko više linija ako se na kraj svake linije koja se nastavlja umetne \. Opseg imena definisanog pomoću #define konstrukcije proteže se od mesta na kojem je definisan pa do kraja izvorne datoteke koja se prevodi. Definicija može koristiti prethodne definicije. Zamene se obavljaju samo sa simbolima, a ne obavljaju se u nizovima unutar navodnika. Primera radi, ako je YES definisano ime, zamena neće biti obavljena kod

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

ili:

```
YESMAN=123;
```

Svako ime može biti definisano pomoću bilo kojeg teksta zamene. Tako npr.,

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

definiše novu reč, forever, za beskonačnu petlju.

Takođe je moguće definisati makronaredbe s argumentima, tako da tekst zamene može biti različit za različita pojavljivanja makronaredbe. Možemo definisati makronaredbu čije je ime max:

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

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

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

Biti zamenjena linijom:

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

Sve dok se dosledno postupa s argumentima, ova makronaredba poslužiće za bilo koji tip podatka. Dakle, nema potrebe za postojanjem različitih oblika makronaredbe za različite tipove podataka što je slučaj kod funkcija.
Ako dobro razmotrimo proširivanje makronaredbe max, možemo uočiti neke nejasnoće. Izrazi se računaju dva puta. Tako, nije baš preporučljivo u argumente umetati operatore inkrementiranja

```
max(i++, j++) /* pogrešno */
```

jer će se veći broj u gornjem izrazu uvećati dvaput.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

MAKRO ZAMENA – DODATNE MOGUĆNOSTI

Makro zamena može da se koristi za definisanje makro funkcija, zamenu teksta, povezivanje argumenata, itd.

Važno je voditi računa i o zagradama u tekstu zamene, kako bi bio očuvan poredak primene operacija. Na primer, ukoliko se definicija

```
#define kvadrat(x) x*x
```

primeni na kvadrat(a+2), tekst zamene će biti a+2*a+2, a ne (a+2)*(a+2), kao što smo verovatno želeli i u tom slučaju bi trebalo koristiti:

```
#define kvadrat(x) (x)*(x)
```

Tekst zamene može da sadrži čitave blokove sa deklaracijama, kao u sledećem primeru:

```
#define swap(t, x, y) { t z; z=x; x=y; y=z; }
```

koji definiše zamenjivanje vrednosti dve promenljive tipa t. Celobrojnim promenljivama a i b se, zahvaljujući ovom makrou, mogu razmeniti vrednosti sa

```
swap(int, a, b);
```

Bez obzira na sve mane i poteškoće u radu, makronaredbe su ipak korisne. Konkretan primer nalazimo i u zaglavlju <stdio.h>, u kojem se funkcije getchar i putchar često definišu kao makronaredbe da bi se skratilo vreme pozivanja funkcije po učitanom znaku. Isto tako, funkcije iz zaglavlja <ctype.h> obično se uvode kao makronaredbe. Imena funkcija mogu se poništiti pomoću #undef naredbe, što se čini kad se želi naglasiti da je potprogram ustvari funkcija, a ne makronaredba:

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

Ukoliko se u direktivi #define, u novom tekstu, ispred imena parametra navede simbol #, kombinacija će biti zamenjena vrednošću parametra navedenog izmedu dvostrukih navodnika. Ovo može da se kombinuje sa nadovezivanjem niski. Na primer, naredni makro može da posluži za otkrivanje grešaka:

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

Tako će tekst:

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

unutar nekog programa biti zamenjen tekstom:

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

Pošto se niske automatski nadovezuju, efekat ove naredbe je isti kao efekat naredbe:

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

Preprocesorski operator ## osigurava način povezivanja argumenata za vreme obavljanja makronaredbe. Ako se parametar u tekstu nalazi u blizini ##, on se zamenjuje argumentom, operator i eventualni okolni razmaci se premeštaju, a rezultat ponovo proverava.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PARAMETRIZOVANI MAKROI

Jedna od moćnih funkcija pretprocesora je mogućnost da simulira funkcije korišćenjem parametrizovanih makroa odnosno makrofunkcija

Jedna od moćnih funkcija pretprocesora je mogućnost da simulira funkcije korišćenjem parametrizovanih makroa. Pretpostavimo da u kodu imamo funkciju koja računa kvadrat broja:

```
int square(int x) {
   return x * x;
}
```

Korišćenjem makroa moguće je prethodnu funkciju definisati na sledeći način:

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

Makroi sa argumentima moraju biti definisani direktivom #define pre nego što se mogu koristiti. Lista argumenata se nalazi u zagradama koje slede za imenom makroa. Prazna polja nisu dozvoljena između imena makroa i zagrada. Na primer:

```
#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void)
{
   printf("Max between 20 and 10 is %d\n", MAX(10, 20));
   return 0;
}
```

Rezultat koda je:

```
Max between 20 and 10 is 20
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

USLOVNO PREVOĐENJE

Pretprocesorskim direktivama je moguće isključiti delove koda iz procesa prevođenja, u zavisnosti od vrednosti uslova koji se računa u fazi pretprocesiranja.

Moguća je kontrola samog pretprocesiranja uslovnim izrazima koji se računaju tokom pretprocesiranja. To određuje selektivno uključivanje koda, zavisno od rezultata izračunatih tokom prevođenja.

Naredba #if računa izraz konstantnog celog broja. Ako izraz nije nula, linije koje slede do naredbi #endif ili #elif ili #else biće obrađene (preprocesorska naredba #elif liči na #else if). Izraz defined(ime); u #if naredbi ima vrednost 1 ako je ime već definisano, a 0 ako nije. Na primer, ako želimo da uključivanje datoteke hdr.h obavimo samo jednom, možemo napisati

```
#if !defined(HDR)
#define HDR
/* sadržaj hdr.h se uključuje */
#endif
```

Prvo uključivanje hdr.h definiše ime HDR. Eventualni kasniji pokušaji uključivanja pronaći će definisano ime, te će preskočiti preko #endif. Takav stil valja koristiti kako bi se izbegao veliki broj uključivanja datoteka. Ako se ovaj stil dosledno poštuje, tada svako pojedino zaglavlje može uključiti sva zaglavlja od kojih zavisi, a da korisnik ne zna ništa o njihovoj međusobnoj zavisnosti.

Ovaj niz testira ime SYSTEM kako bi doneo odluku o uključivanju zaglavlja:

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

Linije #ifdef i #ifndef su specijalizirane forme za ispitivanje definisanosti imena. Prvi primer #if konstrukcije može se napisati i na sledeći način:

```
#ifndef HDR
#define HDR
/* sadržaj HDR se uključuje */
#endif
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMERI KORIŠĆENJA PRETPROCESORA - REZIME

U nastavku su dati razni primeri u cilju boljeg razumevanja svih najbitnijih pretprocesorskih direktiva.

```
#define MAX_ARRAY_LENGTH 20
```

Ova direktiva govori pretprocesoru da identifikatoru MAX_ARRAY_LENGTH dodeli vrednost 20. Direktiva *#define* se uglavnom koristi za konstante zbog bolje čitljivosti.

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

Prva direktiva ukazuje pretprocesoru da uzme fajl stdio.h iz standardne C biblioteke i da tekst iskopira u tekući izvorni fajl. Druga linije ukazuje pretprocesoru da uzme fajl myheader.h iz lokalnog direktorijuma i da njegov sadržaj doda u tekući fajl sa kodom.

```
#undef  FILE_SIZE
#define FILE_SIZE 42
```

Prva linija govori pretprocesoru da poništi definiciju identifikatora FILE_SIZE a druga da mu dodeli vrednost 42.

```
#ifndef MESSAGE
   #define MESSAGE "You wish!"
#endif
```

Prethodni primer ukazuje kompajleru da definiše identifikator MESSAGE samo u slučaju da identifikator MESSAGE nije već definisan.

```
#ifdef DEBUG
   /* Your debugging statements here */
#endif
```

Prethodne naredbe ukazuju pretprocesoru da izvrši procesiranje naredbi nakon #if ukoliko je definisan makro DEBUG. Ovo je korisna opcija kada u toku faze debagiranja (debug – ispravljanja grešaka) programa želite da se izvrši deo programa koji smatrate da ima grešku a koji neće biti izvršen u toku konačne (release) verzije programa.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

OSTALE DIREKTIVE I PREDEFINISANI MAKROI

Od ostalih direktiva koje možete susresti u C programima bitno je pomenuti: #error, #line i #pragma

Naredne direktive spadaju u "ostale" direktive pretprocesora:

- #error
- #line
- #pragma

❑ **Direktiva #error**

Kada se naiđe na direktivu #error program prekida sa radom i odgovarajuća poruka (koja je navedena posle iskaza #error) će biti prikazana od strane kompajlera.
Pogledajmo sledeći primer:

```
#ifndef __cplusplus
#error Compilation Error: A C++ compiler is required to
compile this program!
#endif
```

Makro __cplusplus je uobičajen makro za većinu C kompajlera. Tako da, ukoliko makro nije definisan u toku kompajliranja, onda se kompajliranje prekida i poruka (Compilation Error: A C++ compiler is required to compile this program!) biće prikazana korisniku.

❑ **Direktiva #line**

Direktiva #line se koristi da se promeni vrednost __LINE__ i __FILE__ makroa. (Napomena: ime fajla je opciono) Makroi __LINE__ i __FILE__ predstavljaju linije odnosno fajl iz koga se čitaju podaci.

```
#line 50 "myfile.cpp"
```

Ovo će postaviti __LINE__ na 50 i __FILE__ na myfile.cpp.

❑ **Direktiva #pragma**

Direktiva #pragma se koristi da se izvrši ukidanje određene poruke o grešci, upravljanje debagiranje hipa i steka, itd. Ovo su direktive specifične za kompajler (stoga treba proveriti dokumentaciju kompajlera da bi videli kako ovo radi)
Na primer, iskaz "#pragma inline" ukazuje da će assembly kod biti dostupan u izvornom kodu.
Ukoliko direktiva nije dostupna u implementaciji iskaz će biti ignorisan.

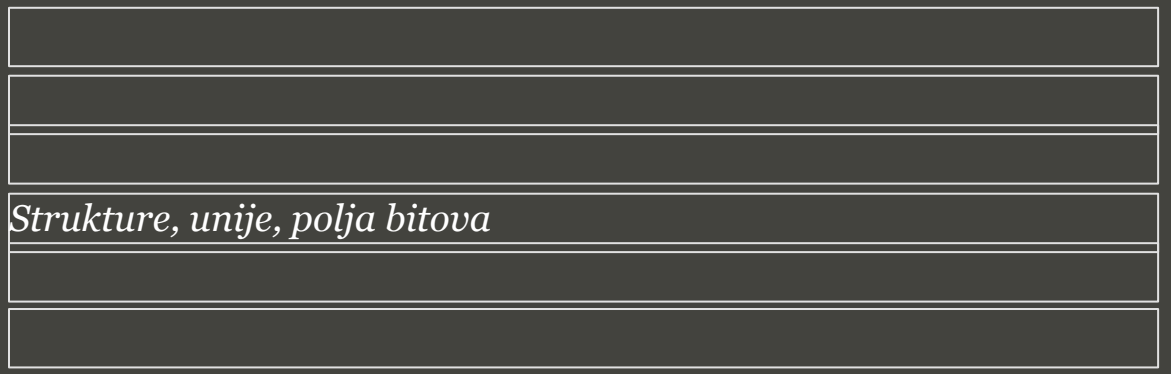
Predefinisani makroi

ANSI C definiše veliki broj makroa. Iako su svi dostupni u fazi programiranja, predefinisane makroe ne treba direktno modifikovati. U nastavku je dat demonstrativni primer:

```
#include <stdio.h>
main()
{
   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );
   printf("ANSI :%d\n", __STDC__ );
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Vežba - Strukture, Unije, Polja bitova



- *Primer. Strukture i typedef*
- *Primer. Korišćenje struktura kod geometrijskih oblika*
- *Primer. Pokazivači na strukture*
- *Primer. Pokazivači i strukture*
- *Primer. Pokazivači na strukture unutar struktura*
- *Primer. Pokazivači na strukture unutar struktura*

07

PRIMER. STRUKTURE I TYPEDEF

U nastakvu je dat primer korišćenje direktive typedef kod struktura u cilju lakšeg i kraćeg zapisa naziva novog korisničkog tipa podatka

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

typedef int ceo_broj ;
typedef struct point POINT;

struct point
{
          int x;
          int y;
};

void main()
{
          ceo_broj x = 3;
          POINT a;
          printf("x = %d\n", x);
          a.x = 1; a.y = 2;
          printf("sizeof(struct point) = %d\n", sizeof(POINT));
          printf("x koordinata tacke a je %d\n", a.x);
          printf("y koordinata tacke a je %d\n", a.y);
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER. KORIŠĆENJE STRUKTURA KOD GEOMETRIJSKIH OBLIKA

Napisati C program koji izračunava obim i površinu trougla i kvadrata u koordinatnoj ravni.

Prvi deo koda

```
#include <stdio.h>
#include <math.h>
struct point
{
            int x;
            int y;
};

float segment_length(struct point A, struct point B)
{
            int dx = A.x - B.x;
            int dy = A.y - B.y;
            return sqrt(dx*dx + dy*dy);
}

float Heron(struct point A, struct point B, struct
point C)
{
            float a = segment_length(B, C);
            float b = segment_length(A, C);
            float c = segment_length(A, B);
            float s = (a+b+c)/2;
            return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

Nastavak koda:

```
float circumference(struct point polygon[], int num)
{
            int i;
            float o = 0.0;
            for (i = 0; i<num-1; i++)
            o += segment_length(polygon[i],
polygon[i+1]);
            o += segment_length(polygon[num-1],
polygon[0]);
            return o;
}

float area(struct point polygon[], int num)
{
            float a = 0.0;
            int i;
            for (i = 1; i < num -1; i++)
            a += Heron(polygon[0], polygon[i],
polygon[i+1]);
            return a;
}

void main()
{
            struct point a, b = {1, 2}, triangle[3];
            struct point square[4] = {{0, 0}, {0,
1}, {1, 1}, {1, 0}};
            a.x = 0; a.y = 0;
            triangle[0].x = 0; triangle[0].y = 0;
            triangle[1].x = 0; triangle[1].y = 1;
            triangle[2].x = 1; triangle[2].y = 0;
            printf("sizeof(struct point) = %ld\n",
sizeof(struct point));
            printf("x koordinata tacke a je %d\n",
a.x);
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER. POKAZIVAČI NA STRUKTURE

U nastavku je dat primer korišćenja pokazivača na strukture, pri čemu se pokazivač koristi kao argument funkcije

```
#include <stdio.h>
typedef struct point
{
            int x, y;
} POINT;
void get_point_wrong(POINT p)
{
            printf("x = ");
            scanf("%d", &p.x);
            printf("y = ");
            scanf("%d", &p.y);
}
void get_point(POINT* p)
{
            printf("x = ");
            scanf("%d", &p->x);
            printf("y = ");
            scanf("%d", &p->y);
}

void main()
{
            POINT a = {0, 0};
            printf("get_point_wrong\n");
            get_point_wrong(a);
            printf("a: x = %d, y = %d\n", a.x, a.y);
            printf("get_point\n");
            get_point(&a);
            printf("a: x = %d, y = %d\n", a.x, a.y);
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER. POKAZIVAČI I STRUKTURE

Primer koji sledi pokazuje osnovne osobine sprezanja struktura i pokazivača.

```
#include <stdio.h>

void main(void)
{
            typedef struct {              /* struktura podataka Test */
                        int element1;
                        float element2;
                        float *p_element2;      /* pokazivac na float u strukturi */
            } Test;

            Test lista1, lista2;          /*  strukturne promenljive */
            Test *p_lista;                /* pokazivac na strukturu Test */

            lista1.element1 = 5;          /* direktna dodela vrednosti */

            /* dodela adrese pokazivacu i posredna dodela vrednosti */
            lista1.p_element2 = &lista1.element2;
            *lista1.p_element2 = 0.25;

            /* pokazivacu se dodeljuje adresa strukturne promenljive lista2 */
            p_lista=&lista2;
            /* pristup element strukture preko pokazivaca na struktutru */
            p_lista->element1 = 2;  /* ili:     (*p_lista).element1 = 2; */

            /* upotreba pokazivaca u strukturi preko pokazivaca na struktutru */
            p_lista->p_element2 = &p_lista->element2;
            *p_lista->p_element2 = 0.75;
            /* ili:
            (*p_lista).p_element2 = &(*p_lista).element2;
            *(*p_lista).p_element2 = 0.75;
            */
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER. POKAZIVAČI NA STRUKTURE UNUTAR STRUKTURA

Jedan od najmoćnijih alata u C programima su pokazivači na strukture unutar struktura. Specijalan slučaj je kada struktura sadrži pokazivač na istu strukturu, tzv. rekurzivne strukture.

```
/*   pokazivaci na strukture u strukturama - liste
struktura
a) formiranje liste struktura koje sadrze podatke o tackama
u ravni
b) opciono dodavanje, brisanje i pregeled elemenata lista
*/
#include <stdio.h>
#include <stdlib.h>          /* malooc() */
#include <ctype.h>           /* toupper()*/

typedef struct {
             int x,y;
} XY;

typedef struct ST_tacka{
             int id;
             XY koord;
             struct ST_tacka *sledeci;     /* pointer na
isti tip strukture */
} Tacka;

void  main(void)
{
             Tacka  iniT, *pocetakT, *krajT, *radT, *novT,
*pomT;
             int brojac=0, id;
             char slovo;

             /* inicijalizovanje pointera */
             radT=&iniT;
             pocetakT=&iniT;
             pocetakT->sledeci=&iniT;

             /* opcije programa */
izbor:
             printf("unesi slovo T-tacka L-linija S-spisak
B-brisi K-kraj ");
```

Ovaj programerski "kompleks" omogućava formiranje ulančenih struktura podataka jer pokazivači na strukture unutar strukture pokazuju na sledeću ili prethodnu strukturu istog tipa. Prednost u odnosu na standardne nizove je u tome što broj članova ovakvih lanaca ne mora biti unapred definisana, a sam lanac struktura može biti razgranat (unutar strukture je više pokazivača, a zavisno od uslova neki od njih pokazuje na nastavak) ili kružan (pokazivač u zadnjoj strukturi pokazuje na prvu strukturu lanca). Posebna pogodnost ovakvog programerskog pristupa je veoma jednostavan postupak umetanja struktura u postojeći lanac, što se postiže jednostavnim promenama pokazivača u dvema strukturama između kojih se umeće novi član lanca. Sloboda u formiranju lanaca struktura potiče od mogućnosti dinamičkog upravljanja memorijom. Za te potrebe koriste se funkcije malloc() i free(). Primeri koji slede demonstrira opisane programerske tehnike, a tipični su za programe iz oblasti kompjuterske grafike (razne CAD aplikacije). Iako su krajnje pojednostavljeni (zastupljna su samo dva grafička elementa: tačka i linija u ravni), pruža pravu osnovu za nadgradnju grafičkih aplikacija ili programa koji koriste veće liste povezanih podataka npr. baze podataka.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER. POKAZIVAČI NA STRUKTURE UNUTAR STRUKTURA

Jedan od najmoćnijih alata u C programima su pokazivači na strukture unutar struktura. Specijalan slučaj je kada struktura sadrži pokazivač na istu strukturu, tzv. rekurzivne strukture.

```
 fflush(stdin);
 slovo=getchar();
 /* switch grananje */
 switch(toupper(slovo))
 {
 case 'T':               /* dodavanje tacke u listu
*/
  novT=(Tacka *)malloc(sizeof(Tacka)); /* memorija
*/
  if(novT != NULL)
  {
   novT->sledeci = radT->sledeci;
   radT->sledeci = novT;
  }
  novT->id=++brojac;
  printf("unesi koordinate tacke x,y\t");
  scanf("%d,%d", &novT->koord.x, &novT->koord.y);
  radT=novT;
  krajT=novT;
  break;
 case 'L':
  printf("NIJE PODRZANO\n");
  break;
 case 'S':               /* prikaziavanje sadrzaja
lista */
  /* sadrzaj liste tacaka */
  radT=pocetakT;
  while(radT->sledeci != pocetakT)
  {
   radT=radT->sledeci;
   printf("tacka id=%d x=%d y=%d\n",
    radT->id, radT->koord.x, radT->koord.y);
  }
  break;
 case 'K':   return;     /* prekid izvrsavanja
programa */
```

```
case 'B':               /* brisanje nekog entiteta */
  printf("unesi id-tacke\t");
  scanf("%d", &id);

  /* brisanje tacke */
  radT=pocetakT;
  while(radT->sledeci != pocetakT)
  {
   pomT=radT;
   radT=radT->sledeci;
   if(radT->id == id)
   {
    pomT->sledeci = radT->sledeci;
    free(radT); /* oslobadjanje memorije */
    radT=krajT;
    goto izbor;
   }
  }
  break;
 default : printf("uneto je pogresno slovo!\n");
 }
 goto izbor;
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER. KORIŠĆENJE STRUKTURA, POKAZIVAČA I FUNKCIJA

Napraviti strukturu Osoba koja sadrži podatke: ime, prezime i broj godina. Zatim napisati funkciju koja štampa ime na standardni izlaz a koristi pokazivač na strukturu kao argument.

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

struct Osoba
{
    char prezime[20];
    char ime[20];
    int  godina;
};

struct Osoba osoba;                                  /* eksterno definisanje strukture */
void PrikaziIme(struct Osoba *p);  /* prototip funkcije */

int main(void)
{
    struct Osoba *st_ptr;     /* pokazivac na strukturu */
    st_ptr = &osoba;                     /* postavi pokazivac na strukturu osoba */

    strcpy(osoba.prezime, "Marko");
    strcpy(osoba.ime, "Markovic");
    printf("%s ", osoba.ime);
    printf("%s ", osoba.prezime);
    osoba.godina = 63;

    PrikaziIme(st_ptr);          /* prosledi pokazivac funkciji */
    return 0;
}

void PrikaziIme(struct Osoba *p)
{
    printf("%s ", p->ime);      /* p pokazuje na strukturu */
    printf("%s ", p->prezime);
    printf("%d ", p->godina);
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Vežba – Dinamičko upravljanje memorijom

Upravljanje memorijom, funkcije za upravljanje memorijom

- *Primer. Generisanje nizova proizvoljne dužine*
- *Primer. Upotreba funkcije realloc()*
- *Primer. Upotreba funkcija calloc() i realloc() kod nizova*
- *Primer. Vraćanja niza iz funkcije korišćenjem pokazivača*
- *Funkcija vraća pokazivač na dinamički alociranu memoriju – Primer1*

08

PRIMER. GENERISANJE NIZOVA PROIZVOLJNE DUŽINE

Napisati program koji unosi niz proizvoljne dimenzije i nalazi najveci element

Verzija 1

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
            int n, *a, i, max;
            printf("Unesi dimenziju niza : ");
scanf("%d", &n);
            a = (int*) malloc(n*sizeof(int));
            if (a == NULL)
            {
                        printf("Greska : Nema
dovoljno memorije!\n");
                        return 1;
            }

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

            // PRONALAZENJA MAKSIMUMA NIZA - Sami
dodajte kod???
            free(a);
            return 0;
}
```

Verzija 2

```
#include <stdio.h>
#include <stdlib.h>
int* CreateIntArray(int n)
{
            return (int*) malloc(n*sizeof(int));
}

int main()
{
            int n, *a, i, max;
            printf("Unesi dimenziju niza : ");
scanf("%d", &n);
            a = CreateIntArray(n);
            if (a == NULL)
            {
                        printf("Greska : Nema
dovoljno memorije!\n");
                        return 1;
            }
            /* Nadalje a koristimo kao obican niz */
            free(a);
            return 0;
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER. UPOTREBA FUNKCIJE REALLOC()

Program demonstrira niz kome se velicina tokom rada povećava korišćenjem funkcije realloc()

```
#include <stdio.h>
#include <stdlib.h>
#define KORAK 10
int main()
{
          int* a = NULL;
          int duzina = 0, alocirano = 0;
          int n, i;
          do
          {
                    printf("Unesi ceo broj (-1 za kraj): "); scanf("%d", &n);
                    if (duzina == alocirano)
                    {
                              alocirano = alocirano + KORAK;
                              a = realloc(a, alocirano*sizeof(int));
                    }
                    a[duzina++] = n;
          } while (n != -1);

          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]);
          free(a);
          return 0;
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER. UPOTREBA FUNKCIJA CALLOC() I REALLOC() KOD NIZOVA

Programi u nastavku demonstriraju upotrebu funkcija calloc() i realloc() kod dinamički alociranih nizova

Primer. Upotreba calloc() funkcije za alociranje niza

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

int main ()
{
            int a,n;
            int * ptr_data;

            printf ("Enter amount: ");
            scanf ("%d",&a);

            ptr_data = (int*) calloc ( a,sizeof(int)
);
            if (ptr_data==NULL)
            {
                        printf ("Error allocating
requested memory");
                        exit (1);
            }

            for ( n=0; n<a; n++ )
            {
                        printf ("Enter number #%d:
",n);
                        scanf ("%d",&ptr_data[n]);
            }

            printf ("Output: ");
            for ( n=0; n<a; n++ )
                        printf ("%d ",ptr_data[n]);

            free (ptr_data);
```

Primer. Realociranje memorije korišćenjem realloc()

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

int main ()
{
            int * buffer;

            /*get a initial memory block*/
            buffer = (int*) malloc (10*sizeof(int));
            if (buffer==NULL)
            {
                        printf("Error allocating
memory!");
                        exit (1);
            }

            /*get more memory with realloc*/
            buffer = (int*) realloc (buffer,
20*sizeof(int));
            if (buffer==NULL)
            {
                        printf("Error reallocating
memory!");
                        //Free the initial memory
block.
                        free (buffer);
                        exit (1);
            }
            free (buffer);
            return 0;
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER. VRAĆANJA NIZA IZ FUNKCIJE KORIŠĆENJEM POKAZIVAČA

Napisati funkciju u kojoj se vrši učitavanje dinamičkog niza. Napisati i glavni program u kome se vrši štampanje članova niza na standardni izlaz

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

int* GenerisiNiz(int n)
{
          return (int*)malloc(n*sizeof(int));
}

int* UnosNiza(int n)
{
          int *x,i;
          x = GenerisiNiz(n);
          for(i=0;i<n;i++) scanf("%d",&x[i]);
          return x;
}

void main()
{
          int i,n,*a;
          printf("Unesi dimenziju niza ");
          scanf("%d",&n);

          a=UnosNiza(n);
          for(i=0;i<n;i++) printf("%5d",a[i]);
                    printf("\n");
          free(a);
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

FUNKCIJA VRAĆA POKAZIVAČ NA DINAMIČKI ALOCIRANU MEMORIJU – PRIMER1

*Napisati program koji koristi dinamički niz **str** u koji smešta rečenicu "Dinamicka alokacija memorije"*

Program treba da koristi funkciju preko pokazivača sa ciljem da odredi poslednje slovo u rečenici, i na kraju dobijeni rezultat štampa na ekran

```
#include <stdio.h>
#include <string.h>  /* strcpy() */
#include <stdlib.h>  /* *malloc(), free(), EXIT_FAILURE, EXIT_SUCCESS
*/

char *kraj(char *str);  /* prototip korisnicke funkcije */

int main(void)
{
	char *str;

	/* alociranje memorije za string */
	if ((str = (char *) malloc(100)) == NULL)
	{
		printf("Nema dovoljno memorije. Stop.\n");
		return EXIT_FAILURE;  /* neregularan prekid programa */
	}

	/* string dobija sadrzaj */
	strcpy(str, "dinamicko rezervisanje memorije");

	printf("string: %s\n", str);
	printf("poslednje slovo je: %c\n", *kraj(str));

	/* oslobadjanje alocirane memorije */
	free(str);

	return EXIT_SUCCESS;
}
```

Funkcija *ime_funkcije vraća pokazivač na deklarisani tip tip. Pravi primer za ovakve funkcije je bibliotečka funkcija *malloc(). Ova funkcija služi za dinamičko alociranje memorije računara (rezervisanje memorije za vreme rada računara). Njen parametar je veličina zahtevanog prostora u bajtovima. Ukoliko nema prostora koji se zahteva ovom funkcijom rezultat funkcije je NULL (pokazivač na nulu). Ako je rezervisanje uspešno izvedeno rezultat je pokazivač na početak memorijskog bloka koji je rezervisan. Ovaj pokazivač je tzv. generičkog tipa (void), pa se vrši njegova eksplicitna konverzija uz pomoć cast-opertaora u pokazivač na zahtevani tip (u ovom primeru char). Komplement funkcije malloc() je free() koja oslobađa (dealocira) prethodno rezervisanu memoriju.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

FUNKCIJA VRAĆA POKAZIVAČ NA DINAMIČKI ALOCIRANU MEMORIJU – PRIMER2

Dinamički alocirana promenljiva može da bude rezultat funkcije. Rezultat se u tom slučaju preko pokazivača vraća pozivaocu funkcije

U nastavku je dat primer funkcije getline() koja čita liniju po liniju teksta sa standardnog ulaza i smešta tekst u dinamički alociran segment memorije. Maksimalna dužina linije koja može biti smeštena se prosledjuje kao parametar funkcije. Funkcija ima i deo koda koji oslobađa bilo koji deo memorije koji nije zauzet. Rezultat funkcije je pokazivač na prvi karakter učitane linije teksta.

```
char *getline( unsigned int len_max )
{
  char *linePtr = malloc( len_max+1 );       // Reserve storage for
"worst case."
  if ( linePtr != NULL )
  {
    // Read a line of text and replace the newline characters with
    // a string terminator:
    int c = EOF;
    unsigned int i = 0;
    while ( i < len_max && ( c = getchar( ) ) != '\n' && c != EOF )
      linePtr[i++] = (char)c;
    linePtr[i] = '\0';

    if ( c == EOF && i == 0 )      // If end-of-file before any
    {                              // characters were read,
       free( linePtr );            // release the whole buffer.
       linePtr = NULL;
    }
    else                           // Otherwise, release the unused
portion.
       linePtr = realloc( linePtr, i+1 );     // i is the string
length.
  }
  return linePtr;
}
```

U nastavku je dat primer kako je moguće pozvati funkciju getline() iz nekog drugog bloka ili glavnog programa

```
char *line;
if (( line = getline(128) ) != NULL )   // If we can read a line,
{
  /* ...  */                           // process the line,
  free( line );                        // then release the
buffer.
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Vežba – Pretprocesorske naredbe

Pretprocesor, pretprocesorske naredbe

- *Primer. Upotreba pretprocesorskih direktiva #define i #include*
- *Primer. Upotreba pretprocesorskih direktiva za uslovno prevođenje:*
- *Primer. Upotreba direktiva za uslovno prevođenje i #undef*
- *Primer. Upotreba pretprocesorske direktive #pragma*

09

PRIMER. UPOTREBA PRETPROCESORSKIH DIREKTIVA #DEFINE I #INCLUDE

Kao što smo već u prvoj lekciji pomenuli naredba define se koristi za definisanje konstanti koje možemo koristiti u našem programu

```
#include <stdio.h>

#define height 100
#define number 3.14
#define letter 'A'
#define letter_sequence "ABC"
#define backslash_char '\?'

void main()
{
   printf("value of height    : %d \n", height );
   printf("value of number : %f \n", number );
   printf("value of letter : %c \n", letter );
   printf("value of letter_sequence : %s \n", letter_sequence);
   printf("value of backslash_char  : %c \n", backslash_char);
}
```

Rezulat programa je:

```
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER. UPOTREBA PRETPROCESORSKIH DIREKTIVA ZA USLOVNO PREVOĐENJE:

U nastavku su data dva primera koja ilustruju korišćenje direktiva #ifdef, #ifndef, #else i #endif za uslovno prevođenje

Primer programa za upotrebu direktiva #ifdef, #else i #endif

```
#include <stdio.h>
#define RAJU 100

int main()
{
   #ifdef RAJU
   printf("RAJU is defined. So, this line will be
added in " \
          "this C file\n");
   #else
   printf("RAJU is not defined\n");
   #endif
   return 0;
}
```

Izlaz programa je:

```
RAJU is defined. So, this line will be added in this C file
```

Primer programa za upotrebu direktiva #ifndef i #endif:

```
#include <stdio.h>
#define RAJU 100
int main()
{
   #ifndef SELVA
   {
      printf("SELVA is not defined. So, now we are
going to " \
             "define here\n");
      #define SELVA 300
   }
   #else
   printf("SELVA is already defined in the
program");

   #endif
   return 0;
}
```

Izlaz programa je:

```
SELVA is not defined. So, now we are going to define here
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER. UPOTREBA DIREKTIVA ZA USLOVNO PREVOĐENJE I #UNDEF

Deo koda nakon klauzule “if” se uključuje u kod ukoliko je #if uslov tačan, u suprotnom se uključuje deo koda koji ide iza #else. Naredba #undefine poništava definiciju postojećeg makroa.

Primer upotrebe direktiva #if, #else i #endif

```
#include <stdio.h>
#define a 100
int main()
{
   #if (a==100)
   printf("This line will be added in this C file
since " \
          "a \= 100\n");
   #else
   printf("This line will be added in this C file
since " \
          "a is not equal to 100\n");
   #endif
   return 0;
}
```

Izlaz programa je:

```
This line will be added in this C file since a = 100
```

Primer upotrebe direktive #undef

```
#include <stdio.h>

#define height 100
void main()
{
   printf("First defined value for height    :
%d\n",height);
   #undef height           // undefining variable
   #define height 600     // redefining the same for new
value
   printf("value of height after undef \&
redefine:%d",height);
}
```

Izlaz programa:

```
First defined value for height : 100
value of height after undef & redefine : 600
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER. UPOTREBA PRETPROCESORSKE DIREKTIVE #PRAGMA

Direktiva #pragma se koristi da se neka funkcija pozove pre i posle main funkcije C programa.

```
#include <stdio.h>

void function1( );
void function2( );

#pragma startup function1
#pragma exit function2

int main( )
{
   printf ( "\n Now we are in main function" ) ;
   return 0;
}

void function1( )
{
   printf("\nFunction1 is called before main function call");
}

void function2( )
{
   printf ( "\nFunction2 is called just before end of " \
            "main function" ) ;"
}
```

Izlaz programa je:

```
Function1 is called before main function call
Now we are in main function
Function2 is called just before end of main function
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER. KORIŠĆENJE MAKROA UNUTAR DRUGOG MAKROA

U nastavku je dat program koji štampa tabelu vrednosti funkcija:

```
#include <stdio.h>
#include <math.h>        // Prototypes of the cos( ) and exp( ) functions.

#define PI          3.141593
#define STEP        (PI/8)
#define AMPLITUDE   1.0
#define ATTENUATION 0.1              // Attenuation in wave propagation.
#define DF(x)       exp(-ATTENUATION*(x))
#define FUNC(x)    (DF(x) * AMPLITUDE * cos(x))  // Attenuated oscillation.
// For the function display:
#define STR(s)  #s
#define XSTR(s) STR(s)            // Expand the macros in s, then stringify.

int main( )
{
           double x = 0.0;

           printf( "\nFUNC(x) = %s\n", XSTR(FUNC(x)) );   // Print the function.

           printf("\n %10s %25s\n", "x", STR(y = FUNC(x)) );     // Table header.
           printf("-----------------------------------------\n");
           for ( ; x < 2*PI + STEP/2;  x += STEP )
                      printf( "%15f %20f\n", x, FUNC(x) );

           return 0;
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Zadaci za samostalan rad

Strukture, unije, upravljanje memorijom, direktive pretprocesora

- *Zadaci iz struktura, upravljanja memorijom i pretprocesora*

10

ZADACI IZ STRUKTURA, UPRAVLJANJA MEMORIJOM I PRETPROCESORA

U nastavku su dati zadaci koji mogu biti korišćeni za samostalnu vežbu:

Napisati program koji učitava niz od n elemenata strukture ličnost, koja sadrži sledeća polja: ime, adresa, dan rođenja, mesec rođenja i godina rođenja. Unete elementa prikazati na ekranu.

Napisati program koji učitava niz od n elemenata niza strukture ličnost, koja sadrži sledeća polja: ime, adresa, dan rođenja, mesec rođenja i godina rođenja. Prikazati osobe koje su u horoskopu **rak**.

Napisati program koji učitava niz od n elemenata niza strukture ličnost, koja sadrži sledeća polja: ime, adresa, dan rođenja, mesec rođenja i godina rođenja. U programu se vrši izbor jednog horoskopskog znaka i prikazuju se sve osobe koje su rođene u tom znaku

Napisati program kojim se korišćenjem funkcije: void citaj(struct licnost *osoba) učitavaju podaci za dve osobe, zatim adresa strukturne promenljive u kojoj su podaci starije osobe predaje pokazivačkoj promenljivoj **s,** i ispisuju elementi strukture na koju ova promenljiva pokazuje.

Napisati program kojim se učitava niz struktura deklarisan sa: struct licnost osoba[MAXOS],*pok; i ispisuje na dva načina. Prvo korišćenjem elemenata niza, a zatim korišćenjem pokazivača koji se inicijalizuje adresom niza struktura.

Napisati funkciju int ortonormirana(int** A, int n) kojom se proverava da li je zadata kvadratna matrica A dimenzije n x n ortonormirana. Za matricu ćemo reći da je ortonormirana ako je skalarni proizvod svakog para različitih vrsta jednak 0, a skalarni proizvod vrste sa samom sobom. Funkcija vraća 1 ukoliko je matrica ortonormirana, 0 u suprotnom. Napisati glavni program u kome se učitava dimenzija kvadratne matrice n, a zatim i elementi i pozivom funkcije se utvrđuje da li je matrica ortonormirana. Maksimalna dimenzija matrice nije unapred poznata.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Zaključak

12

O STRUKTURAMA

Na osnovu utvrđenog gradiva možemo zaključiti sledeće:

Objekti u svakodnevnom životu su uglavnom dosta kompleksni tako da je njihove karakteristike teško opisati korišćenjem samo jedne promenljve. Osim toga, osobine objekata su uglavnom različitog tipa pa stoga korišćenje niza nije dobra opcija. Rešenje je da se koriste strukture kako bi se opisali kompleksi objekti. Struktura je korisnički definisan tip podatka koja omogućava da se upakuju povezane promenljive u jednu celinu, čak i kada su promenljive različitog tipa podatka. Promenljive koje su deo strukture se nazivaju članice ili elementi strukrure.

Deklaracija strukture počinje sa ključnom rečju *struct* za kojom sledi ime strukture. Struktura, kao i funkcija, ima svoje telo uokvireno otvorenom i zatvorenom vitičastom zagradom({ }). Medjutim, za razliku od funkcije, nakon zatvorene vitičaste zagrade mora da se navede tačka-zapeta(;). Telo strukture sadrži deklaracije njenih članova.

Deklarisanje strukture je ustvari deklarisanje novog tipa podatka. Stoga je neophodno deklarisati stukturnu promenljivu koja će predstavljati instancu te strukture. Korišćenjem operatora tačka (.) koji sledi za imenom instance strukture vrši se pristup njenim članovima, bilo da je potrebno da se čita ili izmeni vrednosti promenljive koja je član strukture.

Struktura može biti prosleđena funkciji kroz listu argumenata. Za razliku od imena niza, ime strukture ne predstavlja adresu. Stoga, da bi ste promenili vrednost članova strukture u funkciji neophodno je da strukturu prosledite po adresi. Međutim, čest je slučaj da se struktura prosledi funkciji po adresi iako se njen sadržaj unutar funkcije ne menja. Razlog je taj što je pri prosledjivanju po adresi potrebno manje memorije nego kod prosledjivanja po vrednosti, kod koga se vrši kopiranje sadržaja strukture. Ukoliko kod prosleđivanja po adresi želite da sprečite promenu sadržaja strukture unutar funkcije, u tom slučaju koristite ključnu reč const ispred naziva instance u listi argumenata.

Struktura može biti ugnježdena unutar druge strukture, kao na primer: promenljiva koja predstavlja rođendan u okviru strukture Osoba može da bude tipa Datum, koji je takođe strukturni tip podatka.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

O UPRAVLJANU MEMORIJOM I DIREKTIVAMA PRETPOCESORA

Možemo da izvedemo sledeći zaključak:

U C-u je moguće dinamičko alocirati deo memorije u toku izvršavanja programa. Deo memorije u kome se alocira dinamička memorija se naziva hip, za razliku od promenljivih koje se deklarisu na steku u vreme kompajliranja programa. Neohodno je koristiti pokazivače u kombinaciji sa funkcijama malloc i calloc da bi se izvršilo dinamičko alociranje memorije, i poželjno je da pokazivač bude istog tipa kao i deo memorije koji se odvaja.

Životni vek dinamički alocirane promenljive traje koliko i program u kome je deklarisana. Međutim, ukoliko pre završetka programa pokazivač koji pokazuje na dinamički alociranu promenljivu izađe iz opsega važenja onda vi više nemate pristup dinamički alociranoj promenljivoj. Stoga, dinamički alocirana promenljiva zauzima memorijski prostor kome se ne može pristupiti. Ova pojave se naziva curenje memorije (eng. memory leak). Da bi se izbeglo curenje memorije, neophodno je dealocirati odvojenu memoriju korišćenjem funkcije free().

Rezultat funkcije može biti pokazivač. U tom slučaju, pokazivač može da pokazuje ili na statičku promenljivu ili na dinamički alociranu promenljivu deklarisanu u okviru funkcije, nikako na lokalnu promenljivu.

Programski jezik C omogućuje određene olakšice ako se koristi preprocesor kao prvi korak u prevođenju. Pretprocesiranje, dakle, predstavlja samo pripremnu fazu, pre kompilacije. Suštinski, pretprocesor vrši samo jednostavne operacije nad tekstualnim sadržajem programa i ne koristi nikakvo znanje o jeziku C.

Sve pretrocesorske naredbe počinju znakom taraba (#). Najbitnije pretprocesorske direktive se koriste za: uključivanje fajlova zaglavlja, makrozamenu i uslovno prevođenje.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.