UNIVERZITET METROPOLITAN BEOGRAD

CS130 - C/C++ PROGRAMSKI JEZIK

Strukture, Unije, Upravljanje memorijom

Lekcija 05

PRIRUČNIK ZA STUDENTE

CS130 - C/C++ PROGRAMSKI JEZIK

Lekcija 05

STRUKTURE, UNIJE, UPRAVLJANJE MEMORIJOM

UVOD

Ova lekcija treba da ostvari sledeće ciljeve:

Objekti koje susrećemo u svakodnevnom životu su uglavnom veoma složeni, pa ih je teško opisati pomoću samo jedne promenljive. Vrednost svake od osobina mora biti smeštena u posebnu promenljivu, a promenljive moraju biti različitog tipa podatka.

Pojedinačne informacije koje opisuju osobine objekata, kao što su informacije o preduzećima ili zaposlenima se u jeziku C često grupišu u takozvane slogove odnosno strukture (records, struct). **Slogovi su korisnički definisani tipovi podataka** koji su pogodni za bolju organizaciju, predstavljanje, i naravno čuvanje informacija o sličnim objektima.

U C-u osim struktura postoji još jedan od korisničkih tipova podataka, a to su unije. 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.

Poglavlje 1

Strukture

DEFINICIJA STRUKTURE

Struktura je skup jedne ili više promenljih, koje mogu biti različitih tipova, grupisanih zajedno radi lakše manipulacije. U cilju definisanja strukture koristi se ključna reč struct

C jezik nema klase ali zato sadrži strukture koje obezbeđuju pogodan način za definisanje objekata.

Za definisanje strukture se koristi ključna reč ***struct***. Na ovaj način se definiše novi tip podatka. Osnovni format definicije strukture je:

```
struct [naziv strukture]
{
   definicija clana strukture;
   definicija clana strukture;
   ...
   definicija clana strukture;
} [jedna ili više strukturna promenljiva];
```

Nakon ključne reči ***struct*** navodi se naziv strukture (opciono, može i da se ne navede), a nakon toga, između vitičastih zagrada, opis njenih članova (ili polja).

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.

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 je dat primer deklarisanja strukture *Books*:

```
struct Books
{
   char  title[50];
   char  author[50];
```

```
   char  subject[100];
   int   book_id;
} book;
```

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;
  struct Date published;
};
```

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, o čemu će uskoro biti više reči:

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

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>
struct Song song1, song2;

strcpy( song1.title, "Havana Club" );
strcpy( song1.composer, "Ottmar Liebert" );

song1.duration = 251;  // Vreme trajanja pesme.
song1.published.year = 1998;
```

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 imati istu vrednost kao i odgovarajući članovi strukturne promenljive *song1*.

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( "Naslov knjige : %s\n", book.title);
   printf( "Autor knjige : %s\n", book.author);
   printf( "Tema knjige : %s\n", book.subject);
   printf( "ID knjige : %d\n", book.book_id);

   return 0;
}
```

U nastavku je dat još jedan primer deklaracije:

```
typedef struct Song Song_t;
```

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.

NIZOVI STRUKTURA

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

Ako je potrebno imati podatke o imenima i broju dana i meseci u godini, najbolje je kreirati strukturu *mesec* koja sadrži *brojDana* i *ime*:

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

i koristiti niz ovakvih struktura:

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

Dakle, 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 navedenoj 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)
```

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.

Neka je za prethodni primer koji smo imali definisana funkcija na sledeći način:

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

Štampanje podataka o knjizi je moguće izvršiti korišćenjem:

```
printBook( Book1 );
```

gde bi definicija funkcije *printBook* imala sledeći oblik:

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

Ukoliko je parametar funkcije strukturnog tipa, onda se sadržaj stvarnih argumenata kopira u fiktivne parametre funkcije koja se poziva.

Poglavlje 2

Strukture i pokazivači

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, pri čemu je pokazivač element strukture, 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 ispravno, 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_strukturi
```

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
{
 // Deklaracija novog zapisa.
 struct Song song;
 // Pokazivac na sledeci zapis.
 struct Cell *pNext;
};
```

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

```
struct_pointer->title;
```

Pogledajmo ponovo primer gde smo izvršili deklaraciju promenljivih za rad sa mužičkim numerama korišćenjem pokazivača (kod 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 veći prioritet od indirektnog operatora (*).

```
#include <string.h>
Song_t song1,*pSong; // Jedan objekat tipa Song_t,
   // i jedan pokazivac na tip Song_t.

//...
```

```
*pSong = &song1;
if ( (*pSong).duration > 180 )
  printf( "Pesma %s traje duze od 3 min.\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 .). Stoga je moguće ***if*** iskaz u prethodnom primeru napisati na sledeći način:

```
if ( pSong->duration > 180 )
  printf( "Pesma %s traje duze od 3 min.\n",
          pSong->title );
```

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:

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

Funkcija se iz glavnog programa poziva na sledeći način:

```
/* odstampati informacije o Book1 prosledjivanjem adrese od Book1 */
printBook( &Book1 );
```

dok bi definicija funkcije korišćenjem pokazivača izgledala ovako:

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

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

UGNJEŽDENE STRUKTURE I POKAZIVAČI

U jednoj 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:

```
struct Date
{
   int month, day, 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 {
   char name[20];
   int height;
   struct 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 radi uštede memorijskog prostora. Funkcija za 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>

void main () {
   struct Person p1;
   setValues(&p1);
   printf("Stampanje podataka o osobi\n");
   getValues(&p1);
}

void setValues(struct Person* pers)  {
   printf("Unesi ime osobe: ");
   scanf("%s", pers->name);
   printf("Unesi visinu u incima: ");
   scanf("%d",&pers->height);

   printf("Unesi m, d, g datuma rodjenja: ");
   scanf("%d%d%d", &pers->bDay.month,
    &pers->bDay.day, &pers->bDay.year);
```

```
}

void getValues(const struct Person* pers)  {
   printf("Ime osobe: %s\n", pers->name);
   printf("Visina osobe: %d\n", pers->height);
   printf("Datum rodjenja osobe: %d/%d/%d",
    pers->bDay.month, pers->bDay.day, pers->bDay.year);
}
```

Poglavlje 3

Unije i Polja bitova

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 podataka 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. Deklaracija unije se izvršavana na sledeći način:

```
union [naziv unije]
{
   definicija clana unije;
   definicija clana unije;
   ...
   definicija clana unije;
} [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 karaktera. 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.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

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( "Velicina memorije koju zauzimaju podaci : %d\n", sizeof(data));

   return 0;
}
```

Rezultat programa biće:

```
Velicina memorije koju zauzimaju podaci : 20
```

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];
};

void main( )
{
   union Data data;

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

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

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 odštampan 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>

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 Programiranje");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

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 (npr. vrednost ***z***) ili o studentu (npr. 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:

```
void 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.

POLJA 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;
op.vrsta_okvira = PUN;
```

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Poglavlje 4

Upravljanje memorijom

UVOD U UPRAVLJANJE DINAMIČKOM 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

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.

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), za razliku od dela memorije koji se zove stek u kome se čuvaju podaci za obične, tj. statičke promenljive.

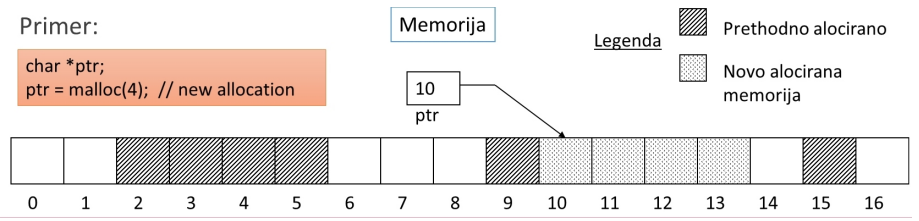
Na Slici 4.1 je dat spisak osnovnih funkcija za upravljanje memorijom u C-u.

	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 4.1.1 Osnovne funkcije za upravljanje memorijom u C-u [6]

Proces alociranja prostora je dat na Slici 4.2.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.



Slika 4.1.2 Proces dinamičkog alociranja prostora [izvor: autor].

FUNKCIJE ZA DINAMIČKO REZERVISANJE MEMORIJE

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

Ukoliko unapred znate veličinu niza koga ćete koristiti da bi ste rešili neki problem, onda su stvari jasne i onda možete koristiti statički niz:

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

Uzmimo sada u obzir slučaj gde nemamo predstavu o tome kolika će biti dužina teksta koju želimo da smestimo u memoriji. U ovom slučaju nam je potreban jedan pokazivač tipa *char* preko koga ćemo u datom trenutku moći da alociramo neophodnu veličinu memorije za odgovarajući tekstualni sadržaj. 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.

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>

void main()
{
   char name[100];
   char *description;

   strcpy(name, "Marko");

   description = malloc( 200 * sizeof(char) );
   if( description == NULL )
   {
    fprintf(stderr, "Greska - nemoguce je alocirati memoriju\n");
   }
   else
   {
    strcpy( description, "Marko, 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 bismo mogli u slučaju da smo unapred deklarisali niz fiksne dužine.

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, tu memoriju treba osloboditi pozivom funkcije free:

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

Poziv *free(ptr)* oslobađa dinamički alociran blok memorije na adresi na koju pokazuje *ptr*, pri čemu je neophodno da *ptr* pokazuje na blok memorije koji je alociran pozivom funkcije *malloc* ili *calloc*. Pritom se ne sme koristiti nešto što je već oslobođeno niti se sme dva puta oslobađati ista memorija.

Ukoliko neki dinamički alociran blok nije oslobođen ranije, on će biti oslobođen pri završetku rada programa, zajedno sa svom drugom memorijom koja je dodeljena programu.

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 konstataciju, treba voditi računa i o tome da se vrednost pokazivača ne promeni da pokazuje na neku adresu.

Ukoliko je potrebno smanjiti ili povećati veličinu već alociranog memorijskog bloka onda je to moguće uraditi korišćenjem funkcije realloc:

```
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.

Funkcija *realloc* kao rezultat vraća **NULL** pokazivač ukoliko nije mogla da odvoji memorijski prostor dužine *size* bajtova. U tom slučaju funkcija neće da oslobodi stari memorijski blok niti da obriše njegov sadržaj.

Funkcija *realloc* vraća pokazivač tipa ***void**** na realociran blok memorije.

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.

UPOTREBA FUNKCIJA REALLOC() 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

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>

void main()
{
```

```
   char name[100];
   char *description;

   strcpy(name, "Marko Markovic");

   description = malloc( 30 * sizeof(char) );
   if( description == NULL )  {
    fprintf(stderr, "Greska - nemoguce je alocirati memoriju\n");
   }
   else  {
    strcpy( description, "Marko je student Metropolitana.");
   }

   description = realloc( description, 100 * sizeof(char) );
   if( description == NULL ) {
    fprintf(stderr, "Greska - nemoguce je alocirati memoriju\n");
   }
   else {
    strcat( description, "Ona je u 10. klasi");
   }

   printf("Name = %s\n", name );
   printf("Description: %s\n", description );

   free(description);
}
```

4.1 Rad sa strukturama i funkcijama

DINAMIČKO ALOCIRANJE I STRUKTURE

Struktura takođe 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;

void main()
{
```

```
    RECORD *ptr_one;

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

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

    printf("Prva vrednost: %d\n",(*ptr_one).i);
    printf("Druga vrednost: %f\n", (*ptr_one).PI);
    printf("Treca vrednost: %c\n", (*ptr_one).A);

    free(ptr_one);
}
```

Kao što možemo videti iz primera, prvo definišemo pokazivač *ptr_one* na tip *RECORD*. Zatim, primenom dinamičkog alociranja, odvajamo prostor u hip memoriji za objekat a adresu tog objekat dodeljujemo pokazivaču *ptr_one*:

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

U nastavku vršimo izmenu osobina objekta korišćenjem već opisane konstrukcije:

```
(*ptr_one).i
```

gde se prvo vrši dereferenciranje pokazivača:

```
(*ptr_one)
```

i dobijamo mogućnost da pristupimo vrednosti na toj adresi, a zatim primenom operatora tačka (.) možemo da pristupimo bilo kom članu zapisa, bilo da je to *i*, *PI* ili *A*.

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

Rezultat funkcije može biti pokazivač, i veoma 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();
void main (void)
{
    char* str;
    str = setName();
    printf("%s",str);
    free(str);
```

```
}
char* setName (void)
{
    char* name;
    name = malloc(80*sizeof(char));
    printf("Unesite vase ime: ");
    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 u stvari 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 nepredvidivog rezultata.

Poglavlje 5

Vežbe

KORIŠĆENJE STRUKTURA KOD GEOMETRIJSKIH OBLIKA (9 MIN)

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);
    printf("y koordinata tacke a je %d\n", a.y);
    printf("x koordinata tacke b je %d\n", b.x);
    printf("y koordinata tacke b je %d\n", b.y);

    printf("Obim trougla je %f\n", circumference(triangle, 3));
    printf("Obim kvadrata je %f\n", circumference(square, 4));
    printf("Pov. trougla: %f\n",
        Heron(triangle[0], triangle[1], triangle[2]));
    printf("Pov. kvadrata: %f\n",
        area(square, sizeof(square)/sizeof(struct point)));
}
```

POKAZIVAČI I STRUKTURE (9 MIN)

Primer koji sledi pokazuje osnovne osobine sprezanja struktura i pokazivača

```
#include <stdio.h>

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

    Test lista1, lista2;
    Test *p_lista;
```

```
    lista1.element1 = 5;
    lista1.p_element2 = &lista1.element2;
    *lista1.p_element2 = 0.25;

    p_lista=&lista2;
    p_lista->element1 = 2;  /* ili: (*p_lista).element1 = 2; */

    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;  */

    printf("elementi strukture lista1: %10d %10.2f\n",
        lista1.element1, lista1.element2);
    printf("elementi strukture lista2: %10d %10.2f\n",
        lista2.element1, lista2.element2);
}
```

PRIMENA STRUKTURA, POKAZIVAČA I FUNKCIJA (9 MIN)

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;  /* 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);
}
```

UPOTREBA FUNKCIJA CALLOC() I REALLOC() KOD NIZOVA (9 MIN)

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>

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

    printf ("Unesite iznos: ");
    scanf ("%d",&a);

    ptr_data = (int*) calloc ( a,sizeof(int) );
    if (ptr_data==NULL)
    {
        printf ("Greska u alociranju zeljenog prostora");
        exit (1);
    }

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

    printf ("Izlaz: ");
    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>

void main ()
{
    int * buffer;

    /*alociranje (odvajanje) inicijalnog memorijskog bloka*/
    buffer = (int*) malloc (10*sizeof(int));
    if (buffer==NULL)
    {
        printf("Greska u alociranju prostora!");
        exit (1);
    }

    /*odvoji (alociraj) dodatni deo memorije koriscenjem realloc*/
    buffer = (int*) realloc (buffer, 20*sizeof(int));
    if (buffer==NULL)
    {
        printf("Greska u realociranju prostora!");
        //Osloboditi inicijalni memorijski blok.
        free (buffer);
        exit (1);
    }
    free (buffer);
}
```

VRAĆANJA NIZA IZ FUNKCIJE KORIŠĆENJEM POKAZIVAČA (9 MIN)

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);
}
```

Poglavlje 6

Zadaci za samostalni rad

ZADACI ZA SAMOSTALNO VEŽBANJE

Na osnovu materijala sa predavanja i vežbi uraditi samostalno sledeće zadatke:

Zadatak 1. Napisati program koji učitava niz od *n* elemenata strukture ličnost, koja sadrži sledeća polja: ime, adresa, dan rođenja, mesec rođenja i godina rođenja. Uraditi sledeće (20 min):

- Unete elemente prikazati na ekranu.
- Prikazati osobe koje su u horoskopu rak.
- U programu se vrši izbor jednog horoskopskog znaka i prikazuju se sve osobe koje su rođene u tom znaku.

Zadatak 2. Kreirati strukturu *Datum*, a zatim je koristiti u strukturi *Licnost* koja treba da sadrži podatke: ime, adresa, datum. Prikazati osobu koja je rođena 1995. godine a najmlađa (od svih ostalih rođenih 1995. godine). (20 min)

Zadatak 3. Napisati program kojim se korišćenjem funkcije (5 min):

```
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.

DODATNI ZADACI ZA SAMOSTALNI RAD

Koristeći materijal sa predavanja i vežbi, rešiti sledeće zadatke

Zadatak 1. Napisati program kojim se učitava niz struktura deklarisan sa (20 min):

```
struct licnost osoba[MAXOS],*pok;
```

i ispisuje na dva načina. Prvo korišćenjem elemenata niza, a zatim korišćenjem pokazivača koji se inicijalizuje adresom niza struktura.

Zadatak 2. 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. (20 min)

Zadatak 3. Formirati niz od *N* elemenata, od kojih je prvi jednak *M*, a svaki sledeći član je jednak zbiru kubova prethodnog elementa niza. Pretpostavimo da su elementi niza poređani u krug i da se izbacuje svaki *K*-ti počevši brojanje od prvog elementa. Odštampati redni broj (indeks) i vrednost elementa niza koji se izbacuje *L*-ti po redu. Raditi isključivo sa dinamički alociranim nizovima (*malloc*, *free*). (20 min)

Poglavlje 7

Domaći zadatak

PRAVILA ZA DOMAĆI ZADATAK

Detaljno proučiti pravila za izradu domaćih zadataka

Svaki student dobija od asistenta sopstvenu kombinaciju domaćeg zadatka.

Onlajn studenti bi trebalo mejlom da se najave, kada budu želeli da krenu sa radom na predmetu i prikupljanjem predispitnih obaveza.

Odgovarajući Visual Studio (NetBeans ili CodeBlocks) projekat koji predstavlja rešenje domaćeg zadatka smestiti u folder CS130-DZ05-Ime-Prezime-BrojIndeksa. Zipovani folder CS130-DZ05-Ime-Prezime-BrojIndeksa poslati predmetnom asistentu (lazar.mrkela@metropolitan.ac.rs) u mejlu sa naslovom (subject)CS130-DZ05, inače se neće računati.

Studenti iz Niša predispitne obaveze predaju asistentima u Nišu (sofija.ilic@metropolitan.ac.rs, mihajlo.vukadinovic@metropolitan.ac.rs, i uros.lazarevic@metropolitan.ac.rs).

Student tradicionalne nastave ima 7 dana, od dana kada je dobio mail sa domaćim zadatkom, da uradi i pošalje rešenje za maksimalan broj poena. Ukoliko student pošalje domaći nakon tog roka, najviše može da ostvari 50% od maksimalnog broja poena.

Studenti onlajn nastave imaju rok da predaju rešene domaće zadatke 10 dana pre termina ispita u ispitnom roku u kome polažu CS130 C/C++ programski jezik.

Vreme izrade: 1,5h.

Zaključak

REZIME

Na osnovu utvrđenog gradiva možemo zaključiti sledeće:

Struktura je korisnički definisan tip podatka koji 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 strukture.

Deklaracija strukture počinje ključnom rečju ***struct*** za kojom sledi ime strukture. Deklarisanje strukture je u stvari deklarisanje novog tipa podatka. Stoga je neophodno deklarisati strukturnu promenljivu koja će predstavljati instancu te strukture.

Struktura može biti prosleđena funkciji kroz listu argumenata. Za razliku od imena niza, ime strukture ne predstavlja adresu. Stoga, da biste 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 prosleđivanju po adresi potrebno manje memorije nego kod prosleđivanja 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.

U C-u je moguće dinamički 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 pojava se naziva curenje memorije (eng. memory leak). Da bi se izbeglo curenje memorije, neophodno je dealocirati odvojenu memoriju korišćenjem funkcije *free*.

REFERENCE

Korišćena literatura

[1] Jeff Kent , C++: Demystified: A Self-Teaching Guide, McGraw-Hill/Osborne, 2004.

[2] Nenad Filipović, Programski jezik C, Tehnički fakulet u Čačku, Univerzitet u Kragujevcu, 2003.

[3] Milan Čabarkapa, C - Osnovi programiranja, Krug, Beograd, 2003.

[4] Filip Maric, Predrag Janicic, Osnove programiranja kroz programski jezik C, Univerzitet u Beogradu, 2014.

[5] Paul J Deitel, Harvey Deitel, C - How to program, 7th edition, Pearson, 2013.

[6] http://www.tutorialspoint.com/cprogramming/index.htm

[7] http://www.codingunit.com/category/c-tutorials

[8] http://www.learncpp.com/