Računske vježbe 8

Programiranje I

1. Napisati program kojim se definiše struktura **Complex**, koja predstavlja kompleksni broj. Kompleksni broj se sadrži od realnog i imaginarnog dijela. Demonstrirati sabiranje dva kompleksna broja i štampati rezultat.

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

struct Complex {
    double real;
    double imag;
};

void print_complex(struct Complex);

int main()
{
    struct Complex complex1;
    struct Complex complex2 = {.real = 1.22, .imag = 0.24};
    complex1.real = 0.17;
    complex1.imag = 0.23;

    printf("Realni i imaginarni dio prvog kompleksnog broja: ");
    print_complex(complex1);

    printf("\nRealni i imaginarni dio drugog kompleksnog broja: ");
    print_complex(complex2);
    // sada cemo preko pokazivaca zadati complex3
    struct Complex *complex3;
    complex3 = (struct Complex *)malloc(sizeof(struct Complex));
    // vrsimo sabiranje dva kompleksna broja
    (*complex3).real = complex1.real + complex2.real;
    complex3->imag = complex1.imag + complex2.imag;
    // (*complex).real je isto sto i complex->real, pojednostavljena sintaksa
    printf("\nZbir dva kompleksna broja je: ");
    print_complex(*complex3);
}

void print_complex(struct Complex complex)
{
    /*
    Uocavamo da se strukture mogu prenositi po vrijednosti, tj. citava struktura,
    sa svim svojim podacima, se prenosi po vrijednosti.
    */
    printf("(%lf, %lf)", complex.real, complex.imag);
}
```

Na primjeru kompleksnih brojeva se jasno uočava motiv za uvođenjem struktura. Očigledno je da sa do sada demonstriranim tipovima podataka nije moguće na valjan način modelovati kompleksni broj. Struktura predstavlja kompozitni tip podatka sačinjen od primitivnih tipova podataka, ali potencijalno i od drugih struktura. Promjenljive koje sačinjavaju strukturu fizički su grupisane u jedan memorijski blok predstavljen zajedničkim imenom. Promjenljive unutar strukture nazivaju se podacima članovima. U slučaju kompleksnog broja, podaci članovi su realni i imaginarni dio, a pristupamo im pomoću imena kompleksnog broja koje može biti proizvoljno:

```
complexNumber.real = 2.84;
```

Bez uvođenja strukture, rad sa kompleksnim brojevima bi nam bio gotovo pa nemoguć. Recimo da želimo da simuliramo sabiranje dva kompleksna broja. Bez strukture bismo morali uvesti 4 promjenljive: real1, imag1, real2, imag2. Šta bismo radili u slučaju da moramo sabrati 4 kompleksna broja ili proizvoljan broj kompleksnih brojeva? Možda bismo mogli definisati niz pa u parovima čuvati realne i imaginarne djelove, ali je već jasno da je ovaj pristup jako nepraktičan. Kao i na svaki drugi podatak, može da se definiše i pokazivač na podatak strukturnog tipa. Ovo, dalje, znači da preko ovog pokazivača možemo da zauzmemo memoriju:

```
struct Complex *complex3;
complex3 = (struct Complex *)malloc(sizeof(struct Complex));
```

što smo u slučaju *complex3* i uradili. U slučaju pokazivača, podacima članovima pristupamo sa:

```
(*complex3).real \\ ili
complex3.real
```

gdje je `->` operator pristupa podacima strukture preko pokazivača. Sa njegove lijeve strane nalazi se pokazivač, a sa desne strane identifikator podatka člana. Podaci strukturnog tipa mogu biti argumenti funkcija. U našem slučaju napisali smo jednu prostu funkciju koja štampa proslijeđeni kompleksni broj. Za vježbu implementirati preostale operacije nad kompleksnim brojevima i to u vidu funkcija. Takođe, napisati funkciju kojoj se prosljeđuje pokazivač na strukturu i koja štampa kompleksan broj.
Nismo bili u potpunosti iskreni kada smo rekli da bez uvođenja strukture ne bismo mogli raditi sa kompleksnim brojevima. C99 uvodi tip podatka complex. Da bismo ga koristili potrebno je uključiti biblioteku **complex.h**. Standard C99 definiše sljedeće kompleksne tipove: float complex, double complex i long double complex. Imaginarni broj se označava sa I. Više na strani 101 u knjizi.

2. Napisati program kojim se učitava niz studenata definisanih strukturom **Student**, koja sadrži ime studenta i niz njegovih ocjena. Potrebno je realizovati funkciju **grade_average**, kojoj se kao argument prosljeđuje formiran niz struktura tipa student, a kao rezultat je potrebno odrediti i odštampati pojedinačne prosječne ocjene studenata i njihovu ukupnu prosječnu ocjenu.

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

struct Student
{
    char name[30];
    int grades[30];
    int numGrades;
};

double grade_average(struct Student *, int);

int main()
{
    struct Student *arr;
    int length, i, j;
    double average;
    printf("Unesite duzinu niza studenata: ");
    scanf("%d", &length);
    arr = (struct Student *)malloc(length * sizeof(struct Student));
    if(arr == NULL) exit(1);
    printf("Unesite podatke o studentima\n");
    for(i = 0; i < length; i++)
    {
        printf("Ime studenta: ");
        scanf("%s", arr[i].name);
        printf("Broj ocjena: ");
        scanf("%d", &arr[i].numGrades);
        printf("Ocjene: ");
        for(j = 0; j < arr[i].numGrades; j++)
            scanf("%d", &arr[i].grades[j]);
    }
    average = grade_average(arr, length);
    printf("Prosjek svih ocjena je %lf", average);
}

double grade_average(struct Student *arr, int length)
{
    int i, j;
    double average, totalAverage = 0;
    for(i = 0; i < length; i++)
    {
        average = 0;
        for(j = 0; j < arr[i].numGrades; j++)
            average += arr[i].grades[j];
        average /= arr[i].numGrades;
        totalAverage += average;
        printf("Prosjek ocjena studenta %s je %lf.\n", arr[i].name, average);
    }
    return totalAverage / length;
}
```

Struktura **Student** je nešto složenija od strukture **Complex**. Sada kao podatke članove imamo niz ocjena

i ime koje je predstavljeno stringom. Niz studenata smo deklarisali kao pokazivač te dinamički zauzeli memoriju:

```
arr = (struct Student *)malloc(length * sizeof(struct Student));
```

što je identično kao u slučaju nizova primitivnih tipova. Funkciji koja računa prosjek prosljeđujemo pokazivač na niz studenata. Za svakog studenta računamo njegov prosjek, a nakon toga ga dodajemo na ukupni prosjek koji na kraju dijelimo sa brojem studenata odnosno dužinom niza.

3. Elementi liste su strukture koje u sebi sadrže po jedan string. Napisati funkciju koja određuje da li su ti stringovi leksikografski uređeni u rastući poredak.

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

struct node
{
    char word[20];
    struct node *next;
};

int ordered(struct node *);

int main()
{
    int i, n;
    struct node *current, *previous, *head;
    puts("Unesite broj elemenata liste:");
    scanf("%d", &n);
    puts("Unesite elemente liste:");
    for(i = 0; i < n; i++)
    {
        current = (struct node *)malloc(sizeof(struct node));
        scanf("%s", current->word);
        current->next = NULL;
        if(i == 0) head = current; // cuvamo glavu
        else previous->next = current;
        previous = current;
    }
    if(ordered(head) == 1)
        puts("Rijeci su uredjene u rastuci poredak.");
    else
        puts("Rijeci nisu uredjene u rastuci poredak.");
    printf("%s", head->word); // glava se nije promijenila
}

int ordered(struct node *p)
{
    while(p->next != NULL)
    {
        if(strcmp(p->word, p->next->word) > 0) return 0;
        p = p->next;
    }
    return 1;
}
```

Lista predstavlja sekvencu povezanih struktura koje ćemo nazivati čvorovima. U čvorovima liste su upisani neki proizvoljni podaci. Za pamćenje narednog elementa liste koristićemo samoreferentne strukture,

odnosno one strukture koje za član imaju pokazivač na strukturu istog tipa. Ovaj član ćemo nazivati **next** i on će pokazivati na naredni element u listi ako ima tog elementa, odnosno na **NULL** ako ga nema tj. ako smo stigli do kraja (kraj liste se naziva rep liste). Rad sa listom podrazumijeva da imamo sačuvan pokazivač na prvi element (glavu) liste. Listu povezujemo na sljedeći način. Biće nam potrebne dvije pomoćne promjenljive za tekući i prethodni element. Za proizvoljnu dužinu liste, pomoću *for* ciklusa prvo dinamički alociramo memoriju za tekući element. Nakon toga upišemo podatke članove i kažemo da next tekućeg elementa pokazuje na NULL. U slučaju prve iteracije čuvamo glavu. Nakon toga, postavljamo da prethodni element pokazuje na tekući. U narednim iteracijama prethodnom elementu (čiji next pokazivač pokazuje na NULL) postavljamo next na novi tekući element. U poslednjoj iteraciji tekući element će za next imati NULL i on će postati rep. Naredbom:

```
previous = current;
```

činimo da prethodni i tekući pokazivači pokazuju na jedan te isti memorijski objekat. Međutim, već u narednoj iteraciji kada dinamički alociramo memoriju za novi element:

```
current = (struct node *)malloc(sizeof(struct node));
```

razdvajamo ova dva pokazivača u memoriji jer *current* sada pokazuje na nešto drugo. Na ovaj način mi gubimo eksplicitan pristup proizvoljnom elementu liste, ali čuvanjem glave možemo uvijek doći do njega. Funkciji koja provjerava da li su riječi leksikografski uređene prosljeđujemo glavu liste. Kroz listu prolazimo *while* ciklusom sve dok naredni element nije NULL, odnosno idemo do pretposljednjeg. Uočite da smo u lokalnoj pokazivačkoj promjenljivoj **p** čuvali tekući element. Naredbom:

```
p = p->next;
```

nećemo izgubiti glavu :) već ćemo reći da pokazivačka promjenljiva p sada pokazuje na naredni element!