

Programiranje 2

12. predavanje

Saša Singer

`singer@math.hr`

`web.math.pmf.unizg.hr/~singer`

PMF – Matematički odsjek, Zagreb

Sadržaj predavanja

● Pretprocesor:

- Naredba `#include`.
- Naredba `#define`.
- Parametrizirana `#define` naredba.
- Uvjetno uključivanje.

● Pregled standardne C biblioteke.

- Standardne datoteke zaglavlja `*.h`.
- Matematičke funkcije iz `<math.h>`.
- Neke funkcije iz `<stdlib.h>`.
- Mjerenje vremena — neke funkcije iz `<time.h>`.

Informacije

Konzultacije (službeno):

🕒 petak, 12–14 sati, ili — po dogovoru.

Programiranje 2 je u kolokvijskom razredu C3.

🕒 Drugi kolokvij: četvrtak, 23. 6. 2016., u 15 sati.

🕒 Popravni kolokvij: ponedjeljak, 5. 9. 2016., u 15 sati.

Uputa: “izbjegnite” popravni — obavite to ranije!

Ne zaboravite, “žive” su i domaće zadaće na adresi

<http://degiorgi.math.hr/prog2/ku/>

Dodatni bodovi “čekaju na vas”.

Informacije — rok za zadaće, podsjetnik, usmeni

Rok za predaju zadaća je

- 🕒 dan drugog kolokvija, do početka kolokvija.

Aplikacija se tada “zatvara za javnost” — bodovi su konačni.

Napomena o podsjetniku = “šalabahteru” na službenom webu:

- 🕒 Nemojte ga nositi na kolokvij — to je zabranjeno!

- 🕒 Dobit ćete ga na kolokviju.

Pogledajte unaprijed što sve piše tamo, da se lakše snađete!

Ako netko poželi izaći na usmeni (nakon rezultata kolokvija),

- 🕒 dogovor za usmeni je u vrijeme uvida u kolokvije.

Pretprocesor

Općenito o pretprocesoru

Prije prevođenja izvornog kôda u objektni ili izvršni kôd izvršavaju se pretprocesorske naredbe.

- Svaka linija izvornog kôda koja započinje znakom # (osim u komentaru) predstavlja jednu pretprocesorsku naredbu.
- Pretprocesorska naredba završava krajem linije, a ne znakom ;.

Opći oblik pretprocesorske naredbe je

```
#naredba parametri
```

One nisu sastavni dio jezika C, te ne podliježu sintaksi jezika.

Neke od pretprocesorskih naredbi su: #include, #define, #undef, #if, #ifdef, #ifndef, #elif, #else, #endif.

Naredba `#include`

Naredba `#include` može se pojaviti u dva oblika:

```
#include "ime_datoteke"
```

ili

```
#include <ime_datoteke>
```

U oba slučaja

- 🔴 pretprocesor **briše** liniju s `#include` naredbom
- 🔴 i uključuje **sadržaj datoteke** `ime_datoteke` u izvorni kôd, na mjestu `#include` naredbe.

Ta datoteka **smije** imati **svoje** pretprocesorske naredbe, koje će se, također, “obraditi” (= uključi tekst, “prođi kroz njega”).

Naredba #include (*nastavak*)

Gdje se traži navedena datoteka?

Ako je `ime_datoteke` navedeno unutar navodnika `"`,

- 📍 onda pretprocesor traži datoteku u direktoriju u kojem se nalazi izvorni program.

Ako je `ime_datoteke` navedeno između znakova `< >`,

- 📍 to signalizira da se radi o sistemskoj datoteci (kao na pr. `stdio.h`), pa će pretprocesor tražiti datoteku na mjestu određenom operacijskim sustavom.

Naredba #define

Naredba `#define` ima oblik

```
#define ime tekst_zamjene
```

i **definira** simbol `ime` kao “živi” objekt.

Ako je zadan i `tekst_zamjene` (ne mora), onda `ime` postaje tzv. “**makro-naredba**”. U tom slučaju, pretprocesor će

- od mjesta `#define` naredbe, **do kraja datoteke**, svaku pojavu imena `ime` zamijeniti tekстом `tekst_zamjene`.

Do **zamjene neće doći** unutar

- znakovnih nizova** (string konstanti), tj. unutar para **dvostrukih** navodnika `"`,
- komentara**, tj. unutar susjednog para znakova `/*` i `*/`.

Parametrizirane makro–naredbe

U parametriziranoj makro–naredbi,

- simboličko ime i tekst_zamjene sadrže tzv. formalne argumente.

```
#define ime(argumenti) tekst_zamjene
```

- Argumenti se pišu u zagradama () i odvajaju zarezom.

Prilikom poziva makro–naredbe, ovi formalni argumenti se

- zamjenjuju stvarnim argumentima navedenim u pozivu.

Zamjena ide na razini teksta — “tekst, umjesto teksta”!

Makro–naredba je efikasnija od funkcije, jer u njoj nema prijenosa argumenata, ali može izazvati neželjene efekte.

Primjer parametrizirane makro-naredbe

Primjer. Makro-naredba za nalaženje **većeg** od dva argumenta

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

Ovdje su **A** i **B** **formalni** argumenti.

Ako se u kôdu pojavi naredba

```
x = max(a1, a2);
```

pretprocesor će je zamijeniti naredbom

```
x = ((a1) > (a2) ? (a1) : (a2));
```

Formalni argumenti (parametri) **A** i **B** **zamijenjeni** su **stvarnim** argumentima **a1** i **a2** — na razini **teksta**.

Primjer parametrizirane makro–naredbe (nast.)

Ako na drugom mjestu imamo naredbu

```
x = max(a1 + a2, a1 - a2);
```

ona će biti zamijenjena s

```
x = ((a1 + a2) > (a1 - a2) ? (a1 + a2) : (a1 - a2));
```

Napomena. Sličnost makro–naredbe i funkcije može **zavarati**.

- 🔴 Kod makro–naredbe **nema kontrole tipa** argumenata.
- 🔴 Supstitucija argumenata je “**doslovna**” na razini **teksta**, što može izazvati **neželjene** efekte.

Razlika makro–naredbe i funkcije

Ako makro–naredbu `max` pozovemo na sljedeći način

```
x = max(i++, j++);
```

nakon supstitucije dobivamo

```
x = ((i++) > (j++) ? (i++) : (j++));
```

Posljedica: jedna od varijabli `i`, `j` neće biti inkrementirana samo jednom (kao kod poziva funkcije) — **veća** varijabla bit će inkrementirana **dva** puta(!), a rezultat `x` ovisi o prevoditelju.

Napomena. Neke “funkcije” deklarirane u `<stdio.h>` su, zapravo, makro–naredbe — na primjer, `getchar` i `putchar`. Isto tako, funkcije u `<ctype.h>` izvedene su, uglavnom, kao makro–naredbe.

Makro-naredbe i prioritet operacija

Primjer. Različite makro-naredbe za **kvadriranje** argumenta.

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

```
#define SQ1(x) x*x
```

```
#define SQ2(x) (x)*(x)
```

```
#define SQ3(x) ((x)*(x))
```

```
int main(void) {  
    printf("%d\n", SQ1(1+1));  
    printf("%d\n", 4/SQ2(2));  
    printf("%d\n", 4/SQ3(2));  
    return 0;  
}
```

Makro-naredbe i prioritet operacija (nastavak)

Ovaj program ispisuje:

3

4

1

Objašnjenje.

● SQ1: `#define SQ1(x) x*x`

`x = 1+1` i doslovnom supstitucijom u SQ1 dobivamo

$$1+1*1+1 = (\text{prioritet!}) = 1+1+1 = 3$$

● SQ2: `#define SQ2(x) (x)*(x)`

`x = 2` i supstitucijom u `4/SQ2(2)` dobivamo

$$4/(2)*(2) = 4/2*2 = 2*2 = 4$$

Makro–naredbe i prioritet operacija (nastavak)

● SQ3: `#define SQ3(x) ((x)*(x))`

`x = 2` i supstitucijom u `4/SQ3(2)` dobivamo

$$4/((2)*(2)) = 4/(2*2) = 4/4 = 1$$

Zato se u parametriziranim makro–naredbama koristi **gomila** zagrada — da se osigura **korektan prioritet** operacija!

Tek zadnji **SQ3** korektno daje **kvadrat** argumenta u **svim** slučajevima, a to se htjelo!

Naredba `#define` i više linija teksta

U `#define` naredbi, `tekst_zamjene` ide

od kraja imena koje definiramo, do kraja linije.

Ako želimo da `ime` bude zamijenjeno s više linija teksta,

moramo koristiti obratnu kosu crtu (`\`) na kraju svakog reda, osim posljednjeg.

Kao i prije, `\` znači da se red nastavlja na početku sljedećeg.

Primjer. Makro za inicijalizaciju polja možemo definirati ovako (nije jako čitljivo i bolje je `ne` koristiti):

```
#define INIT(polje, dim) for(int i=0;\n                i < (dim); ++i) \n                (polje)[i] = 0.0;
```

Naredba #undef

Definicija nekog imena može se **poništiti** korištenjem **#undef** naredbe. Nakon naredbe

```
#undef ime
```

simbol **ime** više **nije definiran** (“nije živ”).

Primjer. Redefinicija konstante **M_PI** (vrijednost π u **double**)

```
#include <math.h>
    /* math.h definira M_PI kao 3.14... */
#undef M_PI
#define M_PI (4.0*atan(1.0))
```

Provjeru je li simbol **ime** **definiran** ili **ne**, možemo napraviti naredbama **#ifdef**, odnosno, **#ifndef** (v. malo kasnije).

Uvjetno uključivanje kôda — #if, #endif

Pretprocesorske naredbe `#if`, `#endif`, `#else`, `#elif` služe za **uvjetno uključivanje** (ili **isključivanje**) pojedinih dijelova programa.

Uvjetno uključivanje kôda naredbama `#if`, `#endif` ima oblik:

```
#if uvjet
    blok naredbi
#endif
```

Ako je **uvjet ispunjen**, onda će

- blok naredbi između `#if uvjet` i `#endif` biti **uključen** u izvorni kôd — koji ide prevoditelju na prevođenje.

Ako **uvjet nije** ispunjen, taj blok **neće** biti uključen, tj. u prevoditelj ide tekst **bez** tog bloka kôda.

Uvjetno uključivanje kôda (nastavak)

Oprez: ovaj `#if` radi na nivou **teksta** izvornog kôda i **nije zamjena** za `if` naredbu u samom jeziku C.

- Uvjet koji se pojavljuje u `#if` naredbi je **konstantan cjelobrojni izraz**. **Nula** se interpretira kao **laž**, a svaka vrijednost **različita od nule** kao **istina**.
- Najčešća svrha uključivanja/isključivanja je **uključiti** neku **datoteku zaglavlja**, ako neko **ime nije** bilo **definirano** ranije (ime = simbol, ili pretprocesorska “varijabla”).

Za provjeru **definiranosti** nekog **imena** (simbola) služi izraz

```
defined(ime)
```

koji daje **1** — ako je **ime** **definirano**, a **0** — ako **nije**.

Uvjetno uključivanje kôda (nastavak)

Najčešće se koristi provjera “je li neko **ime nedefinirano**”. Tu smijemo koristiti i operator **negacije** !

```
!defined(ime)
```

Primjer. Provjera **nedefiniranosti** nekog “**privatnog imena**” za **uključivanje** odgovarajuće datoteke zaglavlja **datoteka.h**

```
#if !defined(__datoteka.h__)  
    #include "datoteka.h"  
#endif
```

s tim da se na **početku datoteka.h** definira “**privatno ime**”

```
#define __datoteka.h__
```

Naredbe `#ifdef` i `#ifndef`

To je standardna tehnika kojom se izbjegava višestruko uključivanje `.h` datoteka (i potencijalna beskonačna rekurzija).

Konstrukcije `#if defined` i `#if !defined` se vrlo često pojavljuju u praksi, pa postoje “pokrate” u obliku naredbi  `#ifdef` i `#ifndef` (respektivno).

Usput, zagrade oko imena simbola (`ime`) nisu obavezne.

Primjer. Prethodnu provjeru možemo napisati u obliku

```
#ifndef __datoteka.h__  
    #include "datoteka.h"  
#endif
```

a na početku `datoteka.h` se definira ime `__datoteka.h__`.

Naredbe `#else` i `#elif`

Složene `if` naredbe za uključivanje ili isključivanje pojedinih dijelova kôda (u pretprocesoru) grade se pomoću naredbi `#else` i `#elif`.

- Značenje `#else` je isto kao značenje `else` u C-u.
- Značenje `#elif` je isto kao značenje `else if`.

Oprez: sve ove naredbe rade na nivou teksta programa i

- nisu zamjena za odgovarajuće naredbe u samom jeziku C!

Uvjetno uključivanje kôda — Primjer 1

Primjer. Za kôd koji treba raditi na **raznim** operacijskim sustavima, testira se koji je **operacijski** sustav u pitanju, kroz ime (ili simbol) **SYSTEM**, da bi se uključilo **ispravno** zaglavlje.

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

Uvjetno uključivanje kôda — Primjer 2

Primjer. U fazi **razvoja** programa **korisno** je ispisivati razne **međurezultate**, za kontrolu **korektnosti** izvršavanja programa. U **završnoj** verziji programa, sav taj suvišan ispis treba **eliminirati**. Za to se koristi standardni simbol **DEBUG**.

```
...
scanf("%d", &x);
#ifdef DEBUG
    printf("Debug: x = %d\n", x); // testiranje
#endif
```

Pitanje. Može li se **definirati** neki **simbol** (poput **DEBUG**) “izvana” — prilikom **prevođenja** programa?

- 🔴 Svi **standardni** prevoditelji imaju **-Dsimbol** opciju koja dozvoljava da se **simbol** **definira** na **komandnoj** liniji.

Definiranje simbola kod prevođenja

Primjer. Pretpostavimo da je program koji sadrži prikazani dio kôda smješten u `prog.c`. Tada će prevođenje naredbom

```
cc -o prog prog.c
```

proizvesti program u koji ispis varijable `x` nije uključen.

Međutim, prevođenje naredbom

```
cc -DDEBUG -o prog prog.c
```

dat će izvršni kôd koji **uključuje** `printf` naredbu, jer je simbol `DEBUG` definiran.

Napomena. Ovu mogućnost `-Dsimbol` ima i `Code::Blocks`, ali **samo** kad se radi s **projektima**. Potražite i probajte!

Standardna C biblioteka

Općenito o zaglavljima

Standardna C biblioteka sadrži

● niz funkcija, tipova i makro-naredbi.

Pripadne deklaracije nalaze se u sljedećim standardnim zaglavljima (redom kao u KR2):

<assert.h>	<float.h>	<math.h>	<stdarg.h>
<stdlib.h>	<ctype.h>	<limits.h>	<setjmp.h>
<stddef.h>	<string.h>	<errno.h>	<locale.h>
<signal.h>	<stdio.h>	<time.h>	

Matematičke funkcije u <math.h>

Konvencija: `x` i `y` su tipa `double`, a `n` je tipa `int`. Sve funkcije kao rezultat vraćaju `double`.

Funkcija	Značenje
<code>sin(x)</code>	$\sin x$
<code>cos(x)</code>	$\cos x$
<code>tan(x)</code>	$\operatorname{tg} x$
<code>asin(x)</code>	$\arcsin x \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right], \quad x \in [-1, 1]$
<code>acos(x)</code>	$\arccos x \in [0, \pi], \quad x \in [-1, 1]$
<code>atan(x)</code>	$\operatorname{arctg} x \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$
<code>atan2(y, x)</code>	(x, y) koordinate točke u ravnini — vraća $\operatorname{arctg} \frac{y}{x} \in [-\pi, \pi]$, jer prepoznaje kvadrant

Matematičke funkcije u `<math.h>` (nastavak)

Funkcija	Značenje
<code>sinh(x)</code>	$\operatorname{sh} x$
<code>cosh(x)</code>	$\operatorname{ch} x$
<code>tanh(x)</code>	$\operatorname{th} x$
<code>exp(x)</code>	e^x
<code>log(x)</code>	$\ln x, \quad x > 0$
<code>log10(x)</code>	$\log_{10} x, \quad x > 0$
<code>pow(x, y)</code>	x^y — greška ako $x = 0$ i $y \leq 0$, ili $x < 0$ i y nije cijeli broj
<code>sqrt(x)</code>	$\sqrt{x}, \quad x \geq 0$

Matematičke funkcije u <math.h> (nastavak)

Funkcija	Značenje
<code>ceil(x)</code>	$\lceil x \rceil$, u double formatu najmanji cijeli broj $\geq x$
<code>floor(x)</code>	$\lfloor x \rfloor$, u double formatu najveći cijeli broj $\leq x$
<code>fabs(x)</code>	$ x $
<code>ldexp(x, n)</code>	$x \cdot 2^n$
<code>frexp(x, int *exp)</code>	ako je $x = y \cdot 2^n$, uz $y \in [\frac{1}{2}, 1)$, vraća y , a eksponent n sprema u <code>*exp</code> . Ako je $x = 0$, onda je $y = n = 0$.

Matematičke funkcije u `<math.h>` (nastavak)

Funkcija	Značenje
<code>modf(x, double *ip)</code>	rastavlja x na cjelobrojni i razlomljeni dio, oba istog predznaka kao x . Razlomljeni dio vrati, a cjelobrojni dio spremi u <code>*ip</code> .
<code>fmod(x, y)</code>	realni (floating-point) ostatak dijeljenja x/y , istog znaka kao x . Ako je $y = 0$, rezultat ovisi o implementaciji.

Razlika atan i atan2

Razlika između funkcija

```
double atan(double x);  
double atan2(double y, double x);
```

atan(x) vraća

• $\arctg x \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right],$

atan2(y, x) interpretira argumente (paziti na poredak!)

• kao **koordinate** točke (x, y) u ravnini i vraća

• $\arctg \frac{y}{x} \in [-\pi, \pi],$ (radi i za $x = 0$, pita **prije** dijeljenja!)

jer prepoznaje **kvadrant** u kojem je točka (x, y) .

Razlika atan i atan2 (nastavak)

Primjer. Program

```
#include <stdio.h>
#include <math.h>
main(void) {
    double x = -1.0, y = -1.0;
    printf("%f\n", atan(y / x));
    printf("%f\n", atan2(y, x));
    return 0; }
```

ispisuje

0.785398

-2.356194

Točni rezultati su $\pi/4$, $-3\pi/4$ (treći kvadrant).

Funkcije floor i ceil

Uočite da funkcije za “najmanje” i “najveće” cijelo

```
double floor(double x);  
double ceil(double x);
```

vraćaju rezultat tipa `double`, a ne `int`.

Primjer. Rezultat ispisa sljedećeg dijela kôda

```
printf("%g\n", floor(5.2));  
printf("%g\n", floor(-5.2));  
printf("%g\n", ceil(5.2));  
printf("%g\n", ceil(-5.2));
```

je: 5, -6, 6, -5 (zbog `%g` formata).

Funkcija fmod

Funkcija

```
double fmod(double x, double y);
```

vraća

- “realni” ostatak pri dijeljenju x/y ,
- s tim da ostatak ima isti predznak kao x .

Princip je isti kao kod cjelobrojnog dijeljenja,

$$x = \text{cjelobrojni kvocijent} \cdot y + \text{ostatak},$$

s tim da je

$$|\text{ostatak}| < |y|,$$

ili

$$\text{ostatak} = x - ((\text{int})(x/y)) \cdot y.$$

Funkcija fmod (nastavak)

Primjer. Sljedeći odsječak kôda

```
printf("%g, ", fmod(5.2, 2.6));  
printf("%g, ", fmod( 5.57,  2.51));  
printf("%g, ", fmod( 5.57, -2.51));  
printf("%g, ", fmod(-5.57,  2.51));  
printf("%g\n", fmod(-5.57, -2.51));
```

ispisuje

0, 0.55, 0.55, -0.55, -0.55

zbog $(\text{int})(5.57/2.51) = 2$ i $5.57 = 2 \cdot 2.51 + 0.55$, itd.

Funkcija frexp

Funkcija

```
double frexp(double x, int *exp);
```

rastavlja broj x na binarnu “**mantisu**” i binarni **eksponent**.

Primjer. Sljedeći odsječak kôda

```
double x = 8.0;
int exp_2;
printf("%f, ", frexp(x, &exp_2));
printf("%d\n", exp_2);
```

ispisuje

```
0.500000, 4
```

Funkcije exp, log, log10 i pow

```
double exp(double x);  
double log(double x);  
double log10(double x);  
double pow(double x, double y);
```

Primjer. Sljedeći odsječak kôda

```
printf("%g\n", log(exp(22)));  
printf("%g\n", log10(pow(10.0, 22.0)));
```

ispisuje

22

22

Neke funkcije iz <stdlib>

Datoteka zaglavlja <stdlib.h> ima nekoliko vrlo korisnih funkcija. Na primjer:

- 📌 `qsort` — QuickSort algoritam za općenito sortiranje niza podataka,
- 📌 `bsearch` — Binarno traženje zadanog podatka u (sortiranom) nizu.

U ovim funkcijama možemo (zapravo, moramo) sami zadati

- 📌 funkciju za uspoređivanje podataka u nizu.

Funkcija qsort

Funkcija za **sortiranje** niza **QuickSort** algoritmom:

```
void qsort(void *base, size_t n, size_t size,  
           int (*comp) (const void *, const void *));
```

Primjer.

```
int main(void) {  
    char rjecnik[3][20] = {"po", "ut", "sri"};  
    int i;  
  
    qsort(rjecnik, 3, 20, strcmp);  
    for (i = 0; i < 3; ++i) puts(rjecnik[i]);  
    return 0; }
```

Funkcija bsearch

Funkcija za **binarno traženje** zadanog podatka u **sortiranom** nizu:

```
void *bsearch(const void *key, const void *base,  
              size_t n, size_t size,  
              int (*comp) (const void *, const void *));
```

Vraća **pokazivač** na **nađeni** podatak (ako ga ima), ili **NULL**.

Primjer.

```
printf("%s\n",  
       bsearch("ut", rjecnik, 3, 20, strcmp));
```

Korektno baratanje s tipovima

Na **ovako** napisan program (v. **qs_1.c**) compiler se “**buni**” na nekoliko mjesta, upozorenjem

🔴 zbog **miješanja** tipova **char *** i **void ***.

Program **radi** korektno, ali... **nije lijepo!**

Popravak. Funkciju **strcmp** za usporedbu stringova treba “**zatvoriti**” u funkciju **korektnog prototipa** — nazovimo ju **usporedi**, a iznutra **pretvorimo** tipove pokazivača:

```
int usporedi(const void *a, const void *b)
{
    return strcmp((char *) a, (char *) b);
}
```

Korektno baratanje s tipovima (nastavak)

Poziv funkcije `qsort` onda glasi:

```
qsort(rjecnik, 3, 20, usporedi);
```

Sasvim analogno, povratnu vrijednost funkcije `bsearch` treba propisno “`castati`” u `char *`

```
printf("%s\n",  
      (char *) bsearch("ut", rjecnik, 3, 20, usporedi));
```

Na popravljani program (v. `qs_1a.c`) compiler “`nema teksta`”.

Napomena. Kod sortiranja rječnika zamjenama pokazivača, a ne stringova (kao `ovdje`), treba paziti na tipove u `usporedi`!

Definiranje kriterija sortiranja

Primjer. Sortiranje polja cijelih brojeva (v. `qs_2.c`, `qs_2a.c`).

```
int main(void)
{
    int i, polje[4] = {1, 3, -4, 2};

    qsort(polje, 4, sizeof(int), usporedi);

    for (i = 0; i < 4; ++i)
        printf("%d\n", polje[i]);

    return 0;
}
```

Funkcija usporedi

Funkcija **usporedi** za **uspoređivanje** **cijelih** brojeva:

```
int usporedi(const void *a, const void *b)
{
    int na = *((int *) a), nb = *((int *) b);
    // int na = * (int *) a, nb = * (int *) b;

    if (na == nb)
        return 0;
    else if (na > nb)
        return 1;
    else
        return -1;
}
```

Funkcije rand i srand

Funkcije za generiranje “slučajnih” cijelih brojeva:

```
int rand(void);  
void srand(unsigned int seed);
```

Funkcija `rand()` vraća

- 📌 tzv. “pseudo-slučajni” cijeli broj u rasponu od 0 do `RAND_MAX`, s tim da `RAND_MAX` mora biti barem $2^{15} - 1 = 32767$ (tj. 16-bitni `int`).

Funkcija `srand(seed)`

- 📌 postavlja tzv. “sjeme” za generator “pseudo-slučajnih” brojeva na zadanu vrijednost `seed`.

Standardno “sjeme” je 1, ako ga ne postavimo sami!

Funkcije rand i srand (nastavak)

Primjer.

```
unsigned int seed; int i;

printf("%d\n", RAND_MAX);    /* 32767 */

scanf("%u", &seed);
srand(seed);

for (i = 1; i <= 10; ++i)
    printf(" %6d\n", rand());
```

Mjerenje vremena

Datoteka zaglavlja <time.h>

U datoteci zaglavlja <time.h> deklarirani su **tipovi** i **funkcije** za manipulaciju

- **anima**, **datumima** i **vremenom**.

Detaljnije ćemo opisati samo funkcije za **vrijeme**, s ciljem:

- kako napraviti jednostavnu “štopericu” za **vrijeme izvršavanja** pojedinih dijelova programa.

Za početak, deklarirana su **dva aritmetička** tipa za prikaz vremena:

- **time_t** — za prikaz stvarnog **kalendarskog** vremena,
- **clock_t** — za prikaz **procesorskog** vremena.

Razlog za **dva** različita tipa = **različite** jedinice!

Stvarno vrijeme — funkcija time

Stvarno kalendarsko vrijeme mjeri se

- u nekim “standardnim” jedinicama za vrijeme (recimo, sekundama — nije bitno).

Funkcija za “očitanje” trenutnog vremena je:

```
time_t time(time_t *tp);
```

Vraća

- trenutno vrijeme, ili `-1`, ako vrijeme nije dostupno.

Ako pokazivač `tp` nije `NULL`,

- onda se izlazna vrijednost sprema i u `*tp`.

To je katkad zgodno napraviti — radi preglednosti.

Razlika vremena — funkcija difftime

Stvarna **vrijednost** rezultata **nije** naročito korisna (uključivo i jedinice). Najčešće je to

- broj sekundi proteklih od nekog “**nultog**” trenutka!

Za “štopericu” **realnog** vremena treba nam samo

- **razlika** vremena: vrijeme na kraju — vrijeme na početku.

Tome služi funkcija

```
double difftime(time_t time2, time_t time1);
```

koja vraća

- razliku **time2 — time1** izraženu u **sekundama**.

Primjer — mjerenje realnog vremena

Primjer. Treba izabrati neki posao koji **dovoljno dugo** traje, da nešto i vidimo. Zato koristimo **dvije** cjelobrojne petlje, da dobijemo “**kvadratnu**” složenost.

🕒 Za **demo**, radimo oko $5 \cdot 10^9$ poziva funkcije **rand**.

Cijeli **program** za ovaj eksperiment je (v. **time_1.c**)

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

int main(void)
{
    int i, j;
    time_t t1, t2;
```

Primjer — mjerenje realnog vremena

```
time(&t1);    /* Moze i t1 = time(NULL); */

for (i = 1; i < 100000; ++i)
    for (j = 1; j < i; ++j)
        rand();

time(&t2);    /* Moze i t2 = time(NULL); */

printf("%g\n", difftime(t2, t1));

return 0;
}
```

Na mom računalu (uz Intelov C), rezultat je 161 (sekundi).

Procesorsko vrijeme — funkcija clock

Procesorsko vrijeme mjeri se

u broju tzv. “otkucaja” procesorskog sata.

Funkcija za “očitanje” procesorskog vremena je:

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

Vraća

procesorsko vrijeme (u broju otkucaja sata) od početka izvršavanja programa, ili `-1`, ako vrijeme nije dostupno.

Dodatno, simbolička konstanta `CLOCKS_PER_SEC` sadrži

broj “otkucaja” procesorskog sata u jednoj sekundi.

Primjena u “štoperici” ide po istom principu razlike vremena.

Primjer — mjerenje procesorskog vremena

Primjer. Pretvaranje u **sekunde** realiziramo funkcijom **dsecnd**.
Cijeli **program** za ovaj eksperiment je (v. **time_2.c**)

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

double dsecnd (void) {
    return (double)( clock( ) ) / CLOCKS_PER_SEC;
}

int main(void)
{
    int i, j;
    double t1, t2, time;
```


Primjer — mjerenje procesorskog vremena

```
t1 = dsecnd( );

for (i = 1; i < 100000; ++i)
    for (j = 1; j < i; ++j)
        rand();

t2 = dsecnd( );
time = t2 - t1;

printf("%g\n", time);

return 0;
}
```

Rezultati za dva izvršavanja su 160.687, 160.718 (sekundi).