

Programiranje 2

Predavanje 12 - pretprocesor, standardna biblioteka, mjerenje vremena

Matej Mihelčić

Prirodoslovno-matematički fakultet
Matematički odsjek

31. svibnja 2023.



Prije prevođenja izvornog koda u objektni ili izvršni kod, izvršavaju se pretprocesorske naredbe. Svaka linija izvornog koda koja započinje znakom `#` (osim u komentaru) čini jednu pretprocesorsku naredbu. Pretprocesorska naredba završava krajem linije, a ne znakom `;`. Opći oblik pretprocesorske naredbe je:

```
#naredba parametri
```

Pretprocesorske naredbe nisu sastavni dio jezika C stoga ne podliježu sintaksi jezika.

Primjeri pretprocesorskih naredbi: `#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 kod, na mjestu `#include` naredbe. Ta datoteka smije imati svoje pretprocesorske naredbe koje će se također obraditi.

Ovisno o načinu navođenja `ime_datoteke`, pretprocesor će tražiti datoteku:

- u direktoriju (folderu) u kojem se nalazi program, ako je ime navedeno unutar navodnika "",
- na mjestu određenom operacijskim sustavom za sistemske datoteke, ako je ime navedeno između znakova `<>`.

Naredba #define

Naredba `#define` ima oblik: `#define ime tekst_zamjene`. Ona definira simbol `ime` kao objekt. Ako je zadan i `tekst_zamjene` onda `ime` postaje *makro-naredba*. Pretprocesor će od mjesta `#define` do kraja datoteke, svaku pojavu imena `ime` zamijeniti tekстом `tekst_zamjene`. Do zamjene neće doći unutar znakovnih nizova (string konstanti), unutar para dvostrukih navodnika `"` i unutar susjednog para znakova `/*` i `*/`.

Parametrizirane makro-naredbe

U parametriziranoj makro-naredbi, simboličko ime i tekst_zamjene sadrže formalne argumente. `#define ime(argumenti) tekst_zamjene`. Argumenti se pišu u zagradama `()` i odvajaju zarezom. Prilikom poziva makro-naredbe, formalni argumenti se zamjenjuju stvarnim argumentima navedenim u pozivu. Zamjena se vrši tako da se tekst argumenta zamijeni tekстом stvarne vrijednosti navedene u pozivu. Makro-naredba je efikasnija od funkcije (jer nema prijenosa argumenata), ali može izazvati neželjene efekte.

Makro naredba za nalaženje većeg od dva argumenta je:

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

A i B su formalni argumenti.

Parametrizirane makro-naredbe

Pretprocesor će naredbu:

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

zamijeniti naredbom:

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

Formalni argumenti (parametri) A i B zamijenjeni su stvarnim argumentima $a1$ i $a2$ (supstitucija teksta).

Naredba:

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

će biti zamijenjena naredbom:

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

Razlika makro naredbe i funkcije

Sličnost makro-naredbe i funkcije može zavarati. Kod makro-naredbe nema kontrole tipa argumenata. Argumenti se doslovno supstituiraju (tekstualno), što može izazvati neželjene efekte.

Ako makro-naredbu `max` pozovemo na sljedeći način:

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

nakon supstitucije dobivamo:

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

Posljedica: veća varijabla bit će inkrementirana dva puta!

Zaglavlje `<stdio.h>` sadrži neke makro-naredbe (npr. `getchar`, `putchar`). Funkcije u `<ctype.h>` su također uglavnom izvedene kao makro-naredbe.

Makro-naredbe i prioritet operacija

Različite makro-naredbe za kvadriranje argumenta:

```
1 #include <stdio.h>
2 #define SQ1(x) x*x
3 #define SQ2(x) (x)*(x)
4 #define SQ3(x) ((x)*(x))
5
6 int main(void){
7     printf("%d\n", SQ1(1+1));
8     printf("%d\n", 4/SQ2(2));
9     printf("%d\n", 4/SQ3(2));
10    return 0; }
```

Izlaz programa je:

3
4
1

Nakon poziva SQ1(1+1) se dogodi supstitucija

$$1 + 1 * 1 + 1 = 1 + 1 + 1 = 3.$$

Nakon poziva 4/SQ2(2) se dogodi supstitucija

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

Nakon poziva 4/SQ3(2) se dogodi supstitucija

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

Samo treća makro naredba daje korektan rezultat u svim slučajevima. Zato se kod definicije parametriziranih makro-naredbi koristi puno zagrada.

Naredba #define s više linija teksta

U #define naredbi, tekst_zamjene obuhvaća tekst do kraja linije. Ako želimo da ime bude zamijenjeno s više linija teksta, moramo koristiti obratnu kosu crtu (\) na kraju svakog reda, osim posljednjeg. Oznaka \ označava da se red teksta nastavlja na početku sljedećeg.

Makro naredbu za inicijalizaciju polja možemo definirati:

```
1 #define INIT(polje, dim) for(int i=0;\n2 i < (dim); ++i) \n3 (polje)[i] = 0.0;
```

Naredba #undef

Definicija nekog imena može se poništiti korištenjem #undef naredbe. Nakon naredbe #undef ime, simbol ime više nije definiran.

Pretpostavimo da želimo redefinirati konstantu M_PI koja reprezentira vrijednost od π u tipu double.

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

Provjeru je li simbol ime definiran ili ne možemo napraviti naredbama #ifdef odnosno #ifndef.

Uvjetno uključivanje koda — #if, #endif

Pretprocesorske naredbe `#if`, `#endif`, `#else`, `#elif` služe za uvjetno uključivanje (ili isključivanje) pojedinih dijelova programa. Uvjetno uključivanje koda naredbama `#if`, `#endif` ima oblik:

```
1 #if uvjet
2     blok naredbi
3 #endif
```

Ako je uvjet ispunjen, tada će blok naredbi između `#if uvjet` i `#endif` biti uključen u izvorni kod, koji će biti predan prevoditelju za prevođenje. Ako uvjet nije ispunjen, blok neće biti uključen (prevoditelju će biti proslijeđen kod bez tog bloka koda).

`#if` nije zamjena za `if` naredbu jezika C, već radi na razini teksta izvornog koda.

Uvjetno uključivanje koda — #if, #endif

Uvjet u #if naredbi je konstantan cjelobrojni izraz (0 - laž, $\neq 0$ - istina). Najčešća svrha uključivanja/isključivanja je uključiti neku datoteku zaglavlja, ako neki simbol ili preprocesorska varijabla nije bila definirana ranije. Provjeru definiranosti možemo izvršiti naredbom `defined(ime)` koja vraća 1 ako je ime definirano, a 0 ako nije. Kod provjere nedefiniranosti imena možemo koristiti i operator negacije (!), `!defined(ime)`.

Provjeru nedefiniranosti imena za uključivanje odgovarajuće datoteke zaglavlja `datoteka.h` možemo napraviti naredbama:

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

uz definiranje imena `__datoteka.h__` unutar datoteke `datoteka.h` naredbom: `#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, stoga postoje kraći oblici #ifdef i #ifndef.

Prethodnu provjeru možemo napisati u obliku:

```
1 #ifndef __datoteka.h__  
2     #include "datoteka.h"  
3 #endif
```

uz definiranje imena __datoteka.h__ unutar datoteke datoteka.h naredbom: #define __datoteka.h__.

Složene `if` naredbe za uključivanje ili isključivanje pojedinih dijelova koda (u pretprocesoru) grade se pomoću naredbi `#else` i `#elif`. `#else` ima isto značenje kao `else` u C-u, dok `#elif` ima isto značenje kao `else if`.

Treba imati na umu da su `#else` i `#elif` naredbe koje rade na nivou teksta programa!

Uvjetno uključivanje koda - Primjer

Kod izrade koda koji treba raditi na raznim operacijskim sustavima, testiramo koji je operacijski sustav u pitanju, kroz ime (simbol) `SYSTEM` da bi se uključilo ispravno zaglavlje.

```
1  #if SYSTEM == SYSV
2      #define DATOTEKA "sysv.h"
3  #elif SYSTEM == BSD
4      #define DATOTEKA "bsd.h"
5  #elif SYSTEM == MSDOS
6      #define DATOTEKA "msdos.h"
7  #else
8      #define DATOTEKA "default.h"
9  #endif
10 #include DATOTEKA
```


Uvjetno uključivanje koda - 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 koristimo standardni simbol `DEBUG`.

```
1  ...  
2  scanf("%d", &x);  
3  #ifdef DEBUG  
4  printf("Debug: x=%d\n", x); // testiranje  
5  #endif
```

Svi standardni prevoditelji imaju `-Dsimbol` opciju koja omogućava da se simbol definira na komandnoj liniji.

Definiranje simbola kod prevođenja

Pretpostavimo da je program koji sadrži prikazani dio koda smješten u `prog.c`. Tada će prevođenje naredbom: `cc -o prog prog.c` proizvesti program u kojem ispis varijable `x` nije uključen. Prevođenje naredbom `cc -DDEBUG -o prog prog.c` će stvoriti izvršni kod koji uključuje `printf` naredbu jer je simbol `DEBUG` definiran.

Mogućnost `-Dsymbol` ima i `Code::Blocks`, ali je nužno napraviti projekt.

Standardna C biblioteka sadrži niz funkcija, tipova i makro naredbi. Pripadne deklaracije nalaze se u sljedećim standardnim zaglavljima:

```
<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>
```

Postoji i zaglavlje `complex.h`.

Matematičke funkcije u <math.h>

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

- $\sin(x)$ - $\sin x$
- $\cos(x)$ - $\cos x$
- $\tan(x)$ - $\operatorname{tg} x$
- $\operatorname{asin}(x)$ - $\arcsin(x) \in [-\frac{\pi}{2}, \frac{\pi}{2}]$, $x \in [-1, 1]$
- $\operatorname{acos}(x)$ - $\arccos(x) \in [0, \pi]$, $x \in [-1, 1]$
- $\operatorname{atan}(x)$ - $\operatorname{arctg}(x) \in [-\frac{\pi}{2}, \frac{\pi}{2}]$
- $\operatorname{atan2}(y, x)$ - (x, y) su koordinate točke u ravnini, vraća $\operatorname{arctg} \frac{y}{x} \in [-\pi, \pi]$ jer prepoznaje kvadrant. Mjeri kut između x osi i polupravca koji kreće od ishodišta i prolazi kroz točku (x, y) .
- $\sinh(x)$ - $\operatorname{sh} x$
- $\cosh(x)$ - $\operatorname{ch} x$

Matematičke funkcije u `<math.h>`

- $\tanh(x)$ - `th x`
- $\exp(x)$ - `ex`
- $\log(x)$ - `ln x`, $x > 0$
- $\log_{10}(x)$ - `log10x`, $x > 0$
- $\text{pow}(x, y)$ - `xy` - greška ako $x = 0$ i $y \leq 0$ ili $x < 0$ i y nije cijeli broj
- $\text{sqrt}(x)$ - `√x`, $x \geq 0$
- $\text{ceil}(x)$ - `⌈x⌉`, tipa `double`, najmanji cijeli broj $\geq x$
- $\text{floor}(x)$ - `⌊x⌋`, tipa `double`, najveći cijeli broj $\leq x$
- $\text{fabs}(x)$ - `|x|`
- $\text{ldexp}(x, n)$ - `x · 2n`

- *frexp*(*x*, *int * exp*) - ako je $x = y \cdot 2^n$, uz $y \in [\frac{1}{2}, 1 >$, vraća *y*, a eksponent *n* sprema na memorijsku lokaciju na koju pokazuje *exp*.
- *modf*(*x*, *double * ip*) - rastavlja *x* na cjelobrojni i razlomljeni dio, oba istog predznaka kao *x*. Razlomljeni dio vrati, a cjelobrojni dio spremi na memorijsku lokaciju na koju pokazuje *ip*.
- *fmod*(*x*, *y*) - realni (floating-point) ostatak dijeljenja x/y , istog znaka kao *x*. Ako je $y = 0$, rezultat ovisi o implementaciji.

Razlika atan i atan2

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

$\text{atan}(x)$ vraća $\text{arctg } x \in [-\frac{\pi}{2}, \frac{\pi}{2}]$, a $\text{atan2}(y, x)$ interpretira argumente kao koordinate točke (x, y) u ravnini i vraća $\text{arctg} \frac{y}{x} \in [-\pi, \pi]$ jer prepozna je kvadrant u kojem je točka.

```
1 #include <stdio.h>  
2 #include <math.h>  
3 int main(void) {  
4     double x = -1.0, y = -1.0;  
5     printf("%f\n", atan(y / x)); // 0.785398  
6     printf("%f\n", atan2(y, x)); // -2.356194  
7     return 0; }
```

Točni rezultati su $\pi/4$ i $-3\pi/4$.

Funkcije floor i ceil

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

Rezultat ispisa sljedećeg dijela koda:

```
1 printf("%g\n", floor(5.2));  
2 printf("%g\n", floor(-5.2));  
3 printf("%g\n", ceil(5.2));  
4 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 , gdje 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 $\text{ostatak} = x - ((\text{int})(x/y)) \cdot y$.

Sljedeći odsječak koda ispisuje:

```
1 printf("%g, \u00a0", fmod(5.2, 2.6)); //0
2 printf("%g, \u00a0", fmod( 5.57, 2.51)); //0.55
3 printf("%g, \u00a0", fmod( 5.57, -2.51)); //0.55
4 printf("%g, \u00a0", fmod(-5.57, 2.51)); //-0.55
5 printf("%g\n", fmod(-5.57, -2.51)); //-0.55
```

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

Funkcija frexp

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

Funkcija rastavlja broj x na binarnu mantisu i binarni eksponent.

```
1 double x = 8.0;  
2 int exp_2;  
3 printf("%f, ", frexp(x, &exp_2)); //0.500000,  
4 printf("%d\n", exp_2); //4
```

Funkcije exp, log, log10 i pow

```
1 double exp(double x);  
2 double log(double x);  
3 double log10(double x);  
4 double pow(double x, double y);
```

Odsječak koda:

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

ispisuje:

22

22

Neke funkcije iz <stdlib>

Datoteka zaglavlja <stdlib.h> sadrži funkcije:

- qsort - QuickSort algoritam za općenito sortiranje niza,
- bsearch - Binarno traženje zadanog podatka u već sortiranom nizu,

U pozivu ovih funkcija kao stvarni argument moramo zadati funkciju za uspoređivanje podataka u nizu.

Funkcija za usporedbu, nazovimo ju `comp`, prima pokazivače na dva objekta koje treba usporediti i vraća rezultat tipa `int`.

```
1 int comp(const nesto *a, const nesto *b);
```

Rezultat određuje međusobni poredak objekata `*a` i `*b` (povratna vrijednost kao kod funkcije `strcmp` za usporedbu stringova).

Funkcija `qsort` sortira niz uzlazno, prema zadanom poretку. Za obrnuti (*silazni*) poredak, treba zamijeniti predznak rezultata funkcije `comp`.

Funkcija qsort

Funkcija za sortiranje niza QuickSort algoritmom:

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

Dio programa za uzlazno sortiranje stringova:

```
1 char rjecnik[3][20] = {"po", "ut", "sri"};  
2 int i;  
3 qsort(rjecnik, 3, 20, strcmp);  
4 for (i = 0; i < 3; ++i)  
5     puts(rjecnik[i]);
```

Funkcija bsearch

Funkcija za binarno traženje zadanog podatka u sortiranom nizu (prema poretку zadanom funkcijom comp):

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

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

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

Pripadni indeks možemo izračunati aritmetikom pokazivača, samo treba paziti na tipove.

Korektno baratanje s tipovima

Pošto smo funkciji `bsearch` proslijedili funkciju `strcmp` koja kao parametre prima `char *` umjesto `void *`, dobit ćemo upozorenje prevoditelja.

Problem možemo popraviti stvaranjem nove funkcije korektnog prototipa unutar koje, uz pretvorbu tipova, pozovemo funkciju `strcmp`.

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

Poziv funkcije `qsort` glasi:

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

Analogno, povratnu vrijednost funkcije `bsearch` treba eksplicitno pretvoriti u `char *`.

Korektno baratanje s tipovima

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

Kod sortiranja rječnika zamjenama pokazivača treba paziti na tipove u uspoređi.

Elegantnije i čitljivije rješenje dobijemo pretvorbom tipa funkcije u zadani pri pozivu funkcija `qsort` i `bsearch`.

```
1 qsort(rjecnik, 3, 20,  
2 (int (*)(const void*, const void*)) strcmp);
```

Možemo uvesti i novi tip za funkciju:

```
1 typedef int (*Comp_f) (const void*, const void*);  
2 qsort(rjecnik, 3, 20, (Comp_f) strcmp);
```


Sortiranje polja cijelih brojeva možemo implementirati:

```
1 typedef int (*Comp_f) (const void*, const void*);
2
3 int main(void) {
4     int i, polje[4] = {1, 3, -4, 2};
5     qsort(polje, 4, sizeof(int),
6         (Comp_f) usporedi); /* Cast funkcije */
7     for (i = 0; i < 4; ++i)
8         printf("%d\n", polje[i]);
9     return 0; }
```

Funkcija usporedi — za uzlazno sortiranje

Funkcija usporedi za uspoređivanje cijelih brojeva (uzlazno):

```
1 int usporedi(const int *p_a, const int *p_b){  
2  /* Nije dobro: return *p_a - *p_b;  
3  jer rezultat ne mora biti korektno prikaziv! */  
4    if (*p_a < *p_b)  
5        return -1;  
6    else if (*p_a > *p_b)  
7        return 1;  
8    else  
9        return 0; }
```

Funkcije rand i srand

Funkcije za generiranje slučajnih cijelih brojeva:

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

Funkcija rand() vraća pseudo-slučajni cijeli broj u rasponu od 0 do RAND_MAX, s tim da RAND_MAX mora biti barem $2^{15} - 1 = 32767$ (16-bitni int bez negativnih brojeva).

Funkcija srand(seed) postavlja sjeme za generator pseudo-slučajnih brojeva na zadanu vrijednost seed. Standardno sjeme je 1, ako ga ne postavimo sami.

```
1 unsigned int seed; int i;  
2 printf("%d\n", RAND_MAX); /* 32767 */  
3 scanf("%u", &seed);  
4 srand(seed);  
5 for(i = 1; i <= 10; ++i) printf("□%6d\n", rand());
```

Funkcije rand i srand

Generatori pseudoslučajnih brojeva se često koriste kod heurističkih algoritama za brzo ali nepotpuno pretraživanje velikih prostora rješenja. Često se koriste i kod implementacije raznih igara itd.

Funkciju srand često koristimo u kombinaciji s funkcijom time iz zaglavlja time.h.

Povećanje raspona pseudoslučajnih brojeva na dvostruki kvadratni (ako je prikaziv) možemo postići:

```
1 int randint(void) {  
2   return RAND_MAX * rand() + rand(); }
```

Raspon 0 do $(RAND_MAX + 1)^2 - 1$ možemo dobiti:

```
1 int randint(void) {  
2   static int RAND_BASE = RAND_MAX + 1  
3   return RAND_BASE * rand() + rand(); }
```

Funkcije rand i srand

Raspon možemo povećati koristeći operator pomaka:

```
1  int randint(void) { /* int od 0 do 2^30 - 1 */  
2      return ( rand() << 15 ) + rand(); }
```

Skaliranje nenegativnog slučajnog cijelog broja na zadani interval $[l, u]$:

```
1  int randint_interval(int l, int u) {  
2      return l + randint() % (u - l + 1); }
```

Za dobivanje uniformno distribuiranih realnih brojeva iz intervala $[0, 1]$, bitno je znati RANDINT_MAX (za naš randint = $2^{30} - 1$):

```
1  double rand_double() {  
2      return (double) randint() / RANDINT_MAX; }
```

Napomena: postoje puno bolji generatori pseudo-slučajnih brojeva od navedenih.

U datoteci zaglavlja <time.h> deklarirani su tipovi i funkcije za manipulaciju danima, datumima i vremenom. Detaljnije ćemo opisati smo funkcije za vrijeme s ciljem stvaranja jednostavne štoperice za vrijeme izvršavanja pojedinih dijelova programa.

Deklarirana su dva aritmetička tipa za prikaz vremena: a) `time_t` za prikaz stvarnog kalendarskog vremena i b) `clock_t` za prikaz procesorskog vremena.

Stvarno vrijeme - funkcija time

Stvarno kalendarsko vrijeme (razlika od danog trenutka do 01.01.1970, 00 : 00) mjeri se funkcijom `time_t time(time_t *tp)`; u sekundama. Funkcija vraća navedenu razliku ili `-1` ako vrijeme nije dostupno. Ako pokazivač `tp` nije `NULL` onda se izlazna vrijednost sprema i u `*tp`.

Što pericu za mjerenje vremena izvršavanja dijela programa implementiramo kao razliku vremena dobivenog funkcijom `time` na kraju i na početku izvođenja zadanog dijela programa. Razliku možemo izračunati funkcijom: `double difftime(time_t time2, time_t time1)`; koja vraća `time2 - time1` izraženu u sekundama.

Primjer — mjerenje realnog vremena

Radimo jednostavni program za testiranje mjerenja vremena. Program treba trajati dovoljno dugo da možemo izmjeriti vrijeme izvršavanja, zato koristimo dvije cjelobrojne petlje. Pozivamo funkciju `rand()` $5 \cdot 10^9$ puta.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  int main(void){
5      int i, j;
6      time_t t1, t2;
7      time(&t1); /* Moze i t1 = time(NULL); */
8      for (i = 1; i < 100000; ++i)
9          for (j = 1; j < i; ++j)    rand();
10     time(&t2); /* Moze i t2 = time(NULL); */
11     printf("%g\n", difftime(t2, t1));
12     return 0; } //44s Intel(R) Core(TM) i7-10750H
```


Procesorsko vrijeme - funkcija clock

Procesorsko vrijeme mjeri se u broju otkucaja procesorskog sata. Funkcija za očitavanje procesorskog vremena je: `clock_t clock(void)`; . Funkcija vraća procesorsko vrijeme (u broju otkucaja sata) od početka izvršavanja programa ili `-1` ako vrijeme nije dostupno. Simbolička konstanta `CLOCKS_PER_SEC` sadrži broj otkucaja procesorskog sata u jednoj sekundi. Štopericu implementiramo po istom principu razlike vremena.

Pretvaranje u sekunde realiziramo funkcijom `dsecnd`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 double dsecnd (void) {
6     return (double)( clock() ) / CLOCKS_PER_SEC;}
```

Procesorsko vrijeme - funkcija clock

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

10
11      t1 = dsecnd();
12      for (i = 1; i < 100000; ++i)
13          for (j = 1; j < i; ++j)
14              rand();
15      t2 = dsecnd();
16      time = t2 - t1;
17      printf("%g\n", time);
18      return 0; } //44.903s,
19      //Intel(R) Core(TM) i7-10750H
```

Vrijeme izvršavanja QuickSort-a

Mjerimo vrijeme izvršavanja algoritma qsort iz <stdlib.h> na slučajno generiranom polju od 10^7 cijelih brojeva.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #define MAXN 10000000 /* 107 */
5  int x[MAXN];
6
7  double dsecnd (void) {
8      return (double)( clock() ) / CLOCKS_PER_SEC; }
9  //slučajni broj iz [0,230-1]
10 int randint(void) {
11     return ( rand() << 15 ) + rand(); }
```

Vrijeme izvršavanja QuickSort-a

```
12 int intcomp(const int *p_a, const int *p_b){
13     if (*p_a < *p_b) return -1;
14     else if (*p_a > *p_b) return 1;
15     else return 0; }
16
17 typedef int (*Comp_fun) (const void*, const void*);
18
19 int main(void){
20     int n = MAXN; /* Broj elemenata u polju. */
21     int i;
22     double start, stop;
23     for (i = 0; i < n; ++i)
24         x[i] = randint();
25
26     start = dsecnd();
27     qsort(x, n, sizeof(int), (Comp_fun) intcomp);
28     stop = dsecnd();
```

Vrijeme izvršavanja QuickSort-a

```
28     for (i = 1; i < n; ++i)
29         if (x[i-1] > x[i])
30             printf("Greska na [%d, %d]\n", i-1, i);
31
32     printf("n=%8d, vrijeme=%9.3f[s]\n",
33           n, stop - start);
34     return 0; }
```

Vrijeme izvršavanja: 1.67s na Intel(R) Core(TM) i7-10750H uz MinGW64 na Windows-u 11.