

Library funkcije za slučajne brojeve

©2004 Nataša Lavicki-Šatović, Željko Vrba

Slučajni brojevi se često koriste, međutim malo gdje je objašnjeno kako se zapravo generiraju i kako se *ispravno* koriste ugrađene library funkcije za generiranje slučajnih brojeva.

Ovaj članak nastoji razjasniti tu temu (bez ulaženja u teoriju), uz primjere ispravnog i neispravnog korištenja generatora slučajnih brojeva ugrađenog u C library (`rand()` i `srand()` funkcije).

1	<code>rand</code> , <code>srand</code>	1
2	Generiranje slučajnih brojeva u zadanom intervalu	3
2.1	Uniformnost razdiobe	3
3	Razne napomene	5
4	Literatura	5

1 `rand`, `srand`

Funkcija `srand()` postavlja inicijalnu vrijednost za generator slučajnih brojeva koji je implementiran u funkciji `rand()`. Ako se ne koristi `srand()`, `rand()` će izgenerirati uvijek isti niz pseudoslučajnih brojeva.

U ovom programu `rand()` radi s inicijalnom vrijednošću 1.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int i;
    for(i = 0; i < 10; i++)
        printf("%d\n", rand());
    return 0;
}
```

Ako se koristi funkcija `srand` sa vrijednošću 1, `rand()` će izgenerirati isti niz brojeva kao i u prethodni program.

Ako se funkciji `srand()` pri pozivu proslijedi bilo koji broj, `rand()` će pri svakom novom pokretanju programa generirati *isti* niz brojeva. Npr. u ovom programu se `srand(1)` može zamijeniti sa `srand(4)` ili bilo kojim drugim brojem. Pokretanje programa dva puta za redom će ispisati isti niz brojeva.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int i;
    srand(1);
    for(i = 0; i < 10; i++)
        printf("%d\n", rand());
    return 0;
}
```

Sad dolazi na red promjena u programu: ako se stavi `srand()` unutar `for` petlje, vidi se da se uvijek ispisuje isti broj bez obzira da li se `for` petlja izvrši 10 ili 10 000 puta.

To se dešava zbog toga što se, pri svakom novom prolazu petlje, iznova poziva funkcija `srand()` koja pri tome iznova inicijalizira funkciju `rand()`. Nakon toga, `rand()` ispisuje prvi el-

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int i;
    for(i = 0; i < 100000; i++) {
        srand(4);
        printf("%d\n", rand());
    }
}
```

```

    }
    return 0;
}

```

ement (budućeg) niza slučajnih brojeva. Pošto je u primjeru funkciji `srand()` kao argument prosljeđena konstantna vrijednost, prvi element niza koji generira `rand()` će uvijek biti isti broj.

Treba imati na umu da `rand()` ne proizvodi zaista slučajne brojeve, već *izračunava* određeni niz brojeva, a vrijednosti tih brojeva ovise o početnoj vrijednosti sa kojom počinje računati (ta početna vrijednost se eksplicitno zadaje pomoću funkcije `srand()`). Funkcije mogu vratiti samo jednu vrijednost, pa se zato mora koristiti petlja unutar koje se stalno iznova poziva `rand()` ne bi li se dobio čitav niz tih brojeva, ali sve te vrijednosti su određene onom jednom vrijednošću koja je prosljeđena funkciji `srand()`. Pošto se ne čeli dobiti niz istih brojeva, funkciju `srand()` treba staviti izvan petlje u kojoj se poziva `rand()`. Na taj način će `srand()` samo jednom inicijalizirati funkciju `rand()`, a ona će, počevši od te zadane vrijednosti, izračunavati vrijednosti niza (dakle, pseudoslučajne brojeve) onoliko puta koliko puta se u petlji pozove `rand()`.

Kako se ne bi pri svakom pokretanju programa dobivao isti niz brojeva, funkciji `srand()` kao argument se najčešće prosljeđuje rezultat funkcije `time()`. Ova funkcija vraća trenutno (tekuće) vrijeme.¹ Pošto vrijeme nije konstantno, pri svakom novom pokretanju tog programa, niz koji će generirati `rand()` bit će drukčiji² zato što će i inicijalna vrijednost biti uvijek drukčija – vrijeme teče.

U ovom primjeru se upravo tako odvija program (ovo je **ispravan način** korištenja `rand()` i `srand()` funkcija za dobivanje slučajnih brojeva):

U slučaju da se stavi `srand()` unutar petlje i prosljedi `time()` kao argument, dešava se ista stvar kao kad se u istim okolnostima prosljedi kao argument neka konstanta - `srand()` pri svakom prolazu petlje nanovo inicijalizira funkciju `rand()` i ona uvijek ispisuje samo prvi član niza slučajnih brojeva koje bi mogla izgenerirati. Ipak, postoji mala razlika: ako se za inicijalizaciju koristi konstanta i izvrti se petlja 100000 puta, `rand()` će uvijek proizvesti isti broj, ali ako se koristi `time()`, nakon svake sekunde će se promijeniti vrijednost koju generira `rand()`. To će i dalje biti 1. element budućeg pseudo-slučajnog niza, ali pošto vrijeme ipak prolazi, mijenjat će se inicijalna vrijednost, a tako i vrijednost prvog elementa niza kojeg će generirati `rand()`.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
    int i;
    srand((unsigned)time(NULL));
    for(i = 0; i < 10; i++) {
        printf("%d\n", rand());
    }
    return 0;
}

```

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
    int i;
    for(i = 0; i < 100000; i++) {
        srand((unsigned)time(NULL));
        printf("%d\n", rand());
    }
    return 0;
}

```

¹ Na UNIX-u je to vrijeme broj sekundi od ponoći 1.1.1970. godine.

² Rezolucija vremena koju vraća `time()` je 1s. Dakle, ako se unutar manje od jedne sekunde dvaput zaredom pokrene program, dobit će se isti niz brojeva.

Ista vrijednost će se izgenerirati onoliko puta koliko procesor stigne napraviti prolaza prije nego što se promijeni vrijeme (što je poprilično - moj kompjuter stigne oko 1100 puta.³ Bez ispisa bi bilo mnogo više poziva).

2 Generiranje slučajnih brojeva u zadanom intervalu

Funkcija `rand()` vraća cijeli broj u intervalu od 0 do `RAND_MAX`. `RAND_MAX` je konstanta definirana u headeru `<stdlib.h>`, gdje se nalaze i prototipi funkcija `rand()` i `srand()`. U praksi je često potrebno generirati cijele (ili realne) brojeve u nekom drugom intervalu.

Sljedeći način generiranja slučajnih brojeva sa operacijom ostatka (%) **nije dobar**. `rand()` obično koristi LCM⁴ za generiranje slučajnih brojeva. Kod brojeva generiranih tom metodom, bitovi manje težine su “manje slučajni” nego bitovi više težine.⁵

Također, broj koji vrati `rand()` se ne bi trebao “rastavljati” u odvojene cjeline bitova (npr. promatrati kao 4 slučajna bajta). Za svaki slučajni bajt treba ponovo pozvati `rand()`.

Ovo je **preporučeni način** za generiranje slučajnih brojeva u intervalu od 1 do n jer koristi bitove veće težine.

```

/*
   Funkcija vraća slučajni broj u intervalu od 0 do
   n-1 uključivo. Ovaj način generiranja NIJE DOBAR!
*/
int rand_n_bad(int n)
{
    return rand() % n;
}

/*
   Funkcija vraća slučajni broj u intervalu od 1 do n
   uključivo. Ovo je PREPORUCENI način.
*/
int rand_1n(int n)
{
    return 1+(int)(n*(double)rand() / (RAND_MAX + 1.0));
}

```

Za generiranje cijelih slučajnih brojeva u intervalu $[a, b]$ prvo je potrebno generirati realan slučajni broj u intervalu $[0, 1)$ (varijabla f). Nakon toga se jednostavnom aritmetikom broj skalira u željeni interval $[a, b]$.

```

int rand_ab(int a, int b)
{
    double f = rand() / (RAND_MAX+1.0);
    return a + (int)((b-a+1)*f);
}

```

2.1 Uniformnost razdiobe

Gore preporučena metoda za generiranje slučajnih brojeva u zadanom intervalu ne generira *točno* uniformnu razdiobu. Promotrimo slučaj iznimno malog `RAND_MAX`, npr. 9, a želimo generirati cijele slučajne brojeve u intervalu $[0, 6)$. `rand()` vraća vrijednosti od 0 do 9, ukupno 10 njih. Tih 10 vrijednosti pomnožene sa 6 nakon odbacivanja decimalnog dijela daje skup brojeva $\{0, 0, 1, 1, 2, 3, 3, 4, 4, 5\}$. Vidi se da brojevi 2 i 5 imaju dvostruko manju vjerojatnost pojavljivanja nego ostali brojevi.

Zadatak 3.4.1-2 u [1] glasi: ako je mU slučajni cijeli broj između 0 i $m - 1$, kolika je *točna* vjerojatnost da je $[kU] = r$ za $0 \leq r < k$. Naravno, željena vjerojatnost je $1/k$.

³ S time da se to vrijeme velikom većinom troši na ispis na ekran.

⁴ linear congruential method

⁵ Za strogu matematičku pozadinu, čitalac se upućuje na [1].

Stavimo $X = mU$ (m ima ulogu `RAND_MAX`; U je uniformni slučajni broj u intervalu $[0, 1)$). Jednakost $\lfloor kU \rfloor = r$ ekvivalentna je nejednakostima $r \leq kX/m < r + 1$ (U je zamijenjen sa X/m ; ovaj korak se u programu radi kad se rezultat funkcije `rand()` dijeli sa `RAND_MAX+1`). Izoliranjem varijable X slijedi $\frac{m}{k}r \leq X < \frac{m}{k}(r + 1)$. Budući da je X cijeli broj možemo pisati $\lceil \frac{m}{k}r \rceil \leq X < \lceil \frac{m}{k}(r + 1) \rceil$. Vjerojatnost da X upadne u taj interval iznosi

$$\frac{1}{m}(\lceil \frac{m}{k}(r + 1) \rceil - \lceil \frac{m}{k}r \rceil) = 1/k + \epsilon$$

gdje je $|\epsilon| < 1/m$.

U praksi je m dovoljno velik ($\geq 2^{15}$) i navedeno odstupanje od uniformne razdiobe ne predstavlja problem.

Zbog navedenih problema, u [4] dana je funkcija koja vraća slučajne brojeve u intervalu $[0, n)$ po uniformnoj razdiobi. Ovdje navedena implementacija se malo razlikuje od knjige. Detaljna diskusija oko ove funkcije i razlog izmjene može se naći u USENET arhivama [5]. Nedostatak ove implementacije je *sporije izvršavanje*: može biti potrebno nekoliko iteracija petlje da bi se generirao slučajni broj s ispravnom raspodjelom.

```
int nrand(int n)
{
    static const unsigned int rml = RAND_MAX + 1U;
    if (n <= 0 || n > RAND_MAX)
        throw domain_error("nrand() argumetn out of range");
    const unsigned int bucket_size = rml / n;
    int r;
    do r = rand() / bucket_size;
    while (r >= n);
    return r;
}
```

Označimo $b = \lfloor \frac{M+1}{n} \rfloor$ (ovo odgovara varijabli `bucket_size`). Petlja se ponavlja sve dok je $\lfloor X/b \rfloor \geq n$ (X je rezultat koji vrati `rand()` funkcija i nalazi se u intervalu $[0, M]$). Iz ove nejednakosti slijedi $n \leq X/b \Leftrightarrow bn \leq X \Leftrightarrow \lfloor \frac{M+1}{n} \rfloor n \leq X$.

Ostatak pri dijeljenju x mod y može se računati prema formuli $x \text{ mod } y = x - \lfloor x/y \rfloor y$. Stoga se posljednja nejednakost može zapisati kao $X \geq (M + 1) - (M + 1) \text{ mod } n$ - to je uvjet koji mora biti zadovoljen da bi se petlja ponovila. Stoga je *vjerojatnost jednog ponavljanja* petlje jednaka

$$p = \frac{1}{M + 1} ((M + 1) - ((M + 1) - (M + 1) \text{ mod } n)) = \frac{(M + 1) \text{ mod } n}{M + 1}$$

Nas zanima *očekivani broj* ponavljanja petlje. Pridružimo vjerojatnosti p komplementarnu vjerojatnost $q = 1 - p$, tj. vjerojatnost da se petlja *neće* ponoviti. Vjerojatnost izvršavanja petlje *točno* k puta je $p^{k-1}q$. Formirajmo funkciju izvodnicu vjerojatnosti, te njezinu derivaciju:

$$F(z) = qz + pqz^2 + p^2qz^3 + \dots = qz(1 + pz + p^2z^2 + p^3z^3 + \dots) = \frac{qz}{1 - pz}$$

$$F'(z) = \frac{1 - p}{(1 - pz)^2}$$

Očekivanje je jednako $E = F'(1) = \frac{1}{1-p}$. **Tablica 1** daje tipične vrijednosti za M te n za koji nastupa najgori slučaj, kao i vjerojatnost p i očekivani broj ponavljanja petlje E .

M	n	p	E
$2^{15} - 1$	$2^{14} + 1$	0.499969	1.99988
$2^{31} - 1$	$2^{30} + 1$	0.5	2

Tablica 1 Najgori slučaj očekivanja za tipične vrijednosti M

3 Razne napomene

Do sada se podrazumijevalo da se generiraju slučajni brojevi koji imaju *jednoliku razdiobu* (funkcija gustoće vjerojatnosti je konstanta). Za generiranje drugih razdioba, može se pogledati literatura.

Niz brojeva koji generira `rand()` funkcija je *periodičan*, što znači da će se niz nakon nekog broja članova početi ponavljati. C library implementacija rijetko kad specificira period i druge parametre generatora. Stoga se za ozbiljnije primjene gdje treba veća količina slučajnih brojeva (npr. Monte Carlo simulacije) ne bi trebalo oslanjati na C library implementaciju `rand()` funkcije već bi trebalo uzeti kvalitetan generator slučajnih brojeva (neki su dani i u navedenoj literaturi).

Za primjenu u kriptografiji (npr. generiranje ključeva) ovi “jednostavni” generatori slučajnih brojeva (`rand()` te u [1] i [2]) **nisu pogodni** – potreban je *kriptografski siguran* generator slučajnih brojeva. Više o tome, i kriptografiji općenito, može se naći u [3].

4 Literatura

1. D. E. Knuth: The Art of Computer Programming, vol.2: Seminumerical Algorithms, 3rd ed.
2. Numerical Recipes in C: The Art of Scientific Computing. <http://www.nr.com/>
3. Handbook of Applied Cryptography. <http://www.cacr.math.uwaterloo.ca/hac/>
4. Accelerated C++.
5. Usenet grupa `comp.lang.c++.moderated`. Subject: `rand()` from Accelerated C++. From: Seungbeom Kim <musiphil@bawi.org>. Cijela diskusija može se naći na Google arhivi: http://groups.google.com/groups?safe=images&ie=UTF-8&as_umsgid=405A13BC.48F5BE71%40bawi.org&lr=&hl=en

