Osnove inženjerske informatike II.

Uvod u programiranje

Petlje i grananja

K. F. & V. B.

> The class of problems capable of solution by the machine can be defined fairly specifically. They are those problems which can be solved by human clerical labour, working to fixed rules, and without understanding.
>
> A. Turing (1946.)

> Electronic computers are intended to carry out any definite rule of thumb process which could have been done by a human operator working in a disciplined but unintelligent manner.
>
> A. Turing (1950.)

Većina je kompjutorskih programa namijenjena prevođenju skupa podataka koji opisuju problem u drugi, opisu rješenja primjereniji skup podataka. U inženjerskim i znanstvenim primjenama to se obično svodi na prevođenje jednoga skupa brojeva u drugi. Prvi skup podataka čini ulazne, a drugi izlazne podatke programa.

Napisat ćemo program koji par ulaznih podataka — realni broj x i cijeli broj n — „prevodi" u broj x^n i ispisuje ga.

Zadatak se može podijeliti u tri gotovo neovisne cjeline:

1. učitati realni broj x i cijeli broj n,
2. izračunati x^n,
3. ispisati dobivenu vrijednost.

Koraci 1. i 3. razmjerno su jednostavni, pa ih odmah možemo prevesti u programski kôd:

1.
```
cout << "x -> ";
cin >> x;
cout << "n -> ";
cin >> n;
```

3.
```
cout << "y = " << y <<  endl;
```

Broj x upisujemo u varijablu `x`, a broj n u varijablu `n`. Za rezultat, koji je realni broj, uveli smo varijablu `y`. Za zapis cijelih brojeva obično se upotrebljavaju varijable i konstante tipa ***int***, a za prikaz realnih brojeva varijable i konstante tipa ***double***. Varijabla `n` je, prema tome, tipa ***int***, dok su varijable `x` i `y` tipa ***double***. Znamo da svaku varijablu treba prije upotrebe definirati, čime joj pridružujemo tip i osiguravamo potreban memorijski prostor:

```
double x;
int n;
```

(varijablu `y` definirat ćemo kasnije).

Za upis ulaznih podataka — vrijednosti koje pridružujemo varijablama `x` i `n` — upotrijebili smo standardni ulazni tôk *cin* (povezan s tipkovnicom) i operator $>>$, a za ispis rezultata standardni izlazni tôk *cout* i operator $<<$. Na početku programa treba stoga uključiti standardnu datoteku zaglavljâ iostream te učiniti vidljivima nazive koje ćemo trebati.

Zasad naš program izgleda ovako:

```
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()  {
  double x;
  int n;

  cout << "x -> ";
  cin>> x;
  cout<< "n -> ";
  cin >> n;

  y ← x^n

  cout << "y = " << y << endl;
  return 0;
}
```

Redak

$$\boldsymbol{y} \leftarrow x^n$$

nije kôd u jeziku *C++*, nego takozvani *pseudo–kôd*, koji u prikazima i opisima zamjenjuje stvarni programski kôd čitljivo i manje–više sažeto naznačavajući operacije koje tek treba prevesti u programski jezik: izračunanu vrijednost x^n pridružit ćemo varijabli $\boldsymbol{y}$. Kako pseudo–kôd teži matematičkoj egzaktnosti i preciznosti izraza, znakom $\leftarrow$ naglašava se nesimetričnost operacije pridruživanja: izraz s lijeve strane (*lvalue*) znaka $\leftarrow$ označava *memorijsku lokaciju* u koju se pohranjuje *vrijednost* izraza s njegove desne strane (*rvalue*). Ti izrazi ne mogu zamijeniti mjesta, ne možemo napisati $x^n \leftarrow \boldsymbol{y}$ (pa čak ni $x^n \rightarrow \boldsymbol{y}$). Iako se u jezicima *C*/*C++* i *Fortran* rabi jednostavniji[1], ali simetrični znak =, ne smijemo zaboraviti razliku između značenja izraza s njegove lijeve i desne strane. (Posebno: pridruživanje nije jednakost; jednakost se ispituje operatorom ==.)

Preostaje nam, dakle, još samo da napišemo kôd za izračunavanje vrijednosti x^n. Znamo da je

$$x^n = \underbrace{1 \times x \times x \times \cdots \times x}_{n \text{ množenja}} = \underbrace{(\cdots((1 \times x) \times x) \cdots \times x) \times x}_{n \text{ množenja}}$$

pa je odgovarajući algoritam, izražen kao pseudo–kôd:

$\boldsymbol{y} \leftarrow 1{,}0$
for $\boldsymbol{i} \leftarrow 1$ **to** n **step** 1 **do**
    $\boldsymbol{y} \leftarrow y \cdot x$

Varijabli $\boldsymbol{y}$ prvo pridružujemo vrijednost $1{,}0$, a potom n puta množimo vrijednost trenutno pridruženu varijabli $\boldsymbol{y}$ s vrijednošću varijable $\boldsymbol{x}$ te dobiveni umnožak pohranjujemo u $\boldsymbol{y}$; primjerice, ako je $n = 4$:

[1] U programskom bi se kôdu znak $\leftarrow$ mogao zamijeniti, recimo, sa <– ili, kao u *Pascal*u, sa :=.

i	y
	1,0
1	$1{,}0 \cdot x = x$
2	$x \cdot x = x^2$
3	$x^2 \cdot x = x^3$
4	$x^3 \cdot x = x^4$

S pomoću varijable ***i*** brojimo provedena množenja — njezina se vrijednost mijenja od 1 do n u svakom se koraku povećavajući za 1. Petlja **for** je naredba za upravljanje tôkom izvođenja programa koja omogućava ponavljanje neke operacije ili niza operacija unaprijed određeni broj puta; ta naredba postoji u većini programskih jezika (u *Fortran*u se, doduše, naziva `DO`).

Prevedemo li prethodni pseudokôdni odlomak u jezik *C++*, dobit ćemo:

```
double y = 1.0;
for (int i = 1; i <= n; i = i + 1)
  y = y * x;
```

Varijablu `y` *definiramo* (kako je y realni broj, njezin je tip ***double***) i odmah je *inicijaliziramo* pridružujući joj vrijednost 1,0. Sintaksa petlje **for** možda je malo zapetljanija negoli u drugim programskim jezicima. U oblim su zagradama tri izraza razdvojena točkama sa zarezima (`;`): prvi je izraz (***int*** `i = 1`) *definicija* i *inicijalizacija* brojača `i`, drugi je izraz (`i <= n`) *uvjet* za izvođenje petlje (dok je vrijednost varijable `i` manja ili jednaka vrijednosti varijable `n`, petlja se izvodi), a u trećem se izrazu *mijenja vrijednost* brojača — u našem se slučaju ta vrijednost povećava za 1 (`i = i + 1`). Budući da brojač `i` poprima cjelobrojne vrijednosti od 1 do n, prirodno je da bude tipa ***int***. Redak koji slijedi iza upravljačkoga retka sadrži naredbu koja se izvršava u petlji, takozvano *tijelo petlje*.

Jezici *C* i *C++* često omogućuju jezgrovitije pisanje negoli neki drugi jezici (katkad je ipak upitno povećava li to čitljivost kôda). Tako se za povećavanje vrijednosti varijable `i` za 1 mogu upotrijebiti prefiksni ili sufiksni unarni operatori `++`: `++i` ili `i++`. Upravljački redak petlje **for** bit će tada

```
for (int i = 1; i <= n; ++i)
```

Treba li sadržaj varijable `i` povećati za neku drugu vrijednost k ($k \neq 1$), umjesto `i = i + k` možemo pisati i `i += k`. Takva sintaksa odgovara našem načinu razmišljanja i izražavanja: obično kažemo „povećali smo i za k”, a ne „uzeli smo vrijednost iz `i`, dodali joj k i rezultat strpali natrag u `i`”.

Taj se sintaktički oblik može primijeniti za većinu operatora **op**: ***lvalue*** **op**$=$ *expr* znači isto što i ***lvalue*** $=$ *lvalue* **op** $(expr)$. Uočite zagrade oko *expr*; izraz `y *= x + w` znači `y = y * (x + w)`, a ne `y = y * x + w`.

U „algoritmu” koji smo upotrijebili za izračunavanje n–te potencije broja x množenje se izvodi n puta. Uvest ćemo sada znatno učinkovitiji, ali, naravno, i složeniji algoritam, a njegova će nam programska realizacija omogućiti opis još dviju upravljačkih naredbi.

Ako je n paran, $x^n = x^{2\frac{n}{2}} = x^{n/2} \cdot x^{n/2} = (x^{n/2})^2$; time se broj množenja upola smanjuje. Ponovo, ako je $n/2$ paran, $x^n = ((x^{n/4})^2)^2$. I tako dalje. Ukratko, ako je $n = 2^k$ za neki pozitivni cijeli broj k,

$$x^n = x^{2^k} = \underbrace{(\cdots(x^2)^2\ \cdots)^2}_{k \text{ puta}}.$$

Ako pak n nije paran, $n/2$ nije cijeli broj; no, tada je $(n-1)/2$ cijeli broj, pa je $x^n = x \cdot (x^{(n-1)/2})^2$. Isto vrijedi za $n/2^i$ u i-tom koraku. Iz tih zaključaka možemo izvesti poboljšani algoritam u kojem se množenje izvodi najviše $2\log_2 n$ puta; zapisan kao pseudo–kôd:

$\boldsymbol{y} \leftarrow 1$
while $n > 0$ **do**
    if $\mathrm{odd}\,(n)$ **then**
        $\boldsymbol{y} \leftarrow y \cdot x$
    $\boldsymbol{x} \leftarrow x \cdot x$
    $\boldsymbol{n} \leftarrow n$ **div** 2

odd () je funkcija koja ispituje je li njezin argument neparan broj, a **div** je operator cjelobrojnoga dijeljenja. (Pretpostavljamo da nam početne vrijednosti n i x više neće trebati te da ih smijemo mijenjati.)

Prijevodom u jezik *C++* dobivamo:

```
double y = 1.0;
while (n > 0) {
  if (n % 2 == 1)
    y = y * x;
  x = x * x;
  n /= 2;
}
```

Umjesto petlje **for** upotrijebili smo petlju **while**. I ta petlja postoji u mnogim drugim programskim jezicima, a upotrebljava se obično u slučajevima u kojima broj ponavljanja nije unaprijed poznat. Izraz u oblim zagradama iza ključne riječi **while** je *uvjetni izraz*; petlja se izvodi tako dugo dok je taj izraz istinit. Za razliku od petlje **for**, u kojoj se sva tri izraza neophodna za upravljanje — inicijalizacija varijabli koje se pojavljuju u uvjetnom izrazu, sam uvjetni izraz i promjena vrijednosti tih varijabli — navode u zagradama uz ključnu riječ, u petlji **while** se u zagradama navodi samo uvjetni izraz; varijable treba inicijalizirati prije petlje, a promjenom njihovih vrijednosti u tijelu petlje treba omogućiti promjenu logičke vrijednosti uvjetnoga izraza. U našem se primjeru varijabla `n` inicijalizira pri upisu, a njezinu vrijednost mijenjamo dijeljenjem sa 2 u posljednjem retku tijela petlje (kako su i varijabla `n` i konstanta 2 tipa ***int***, riječ je o cjelobrojnom dijeljenju).

Dok se u petlji **for** izvršavala samo jedna naredba, sada se u svakom prolazu kroz petlju izvode tri naredbe. Grupa naredbi koje u stanovitom smislu čine cjelinu, poput tijela petlje, zatvara se u vitičaste zagrade i naziva *složenom naredbom* ili *blokom*; cijeli se blok može smatrati jednom naredbom (ali, iza vitičaste zagrade koja označava završetak bloka ne dolazi točka sa zarezom).

U tijelu petlje susrećemo se s još jednom kontrolnom naredbom: naredbom **if**. Ta naredba omogućava *uvjetno izvođenje* operacija — one će se izvesti samo ako je neki uvjet ispunjen.

U našem primjeru ispitujemo je li trenutna vrijednost varijable `n` neparna. Operator `%` daje ostatak pri cjelobrojnom dijeljenju. Ako je broj neparan, ostatak pri dijeljenju sa 2 mora biti 1. Nemojte zaboravite, u jeziku *C++* je operator za ispitivanje jednakosti operator ==, dok je = operator pridruživanja. Operatora `%` ima viši prioritet od operatora == ; prvo se izračunava `n % 2`, a tek potom dobiveni rezultat uspoređuje sa 1. Ako je vrijednost u `n` neparna (dakle, ako je uvjetni izraz naredbe **if** istinit),

izvršit će se naredba `y = y * x;`. Ako je pak ta vrijednost neparna (uvjetni izraz nije istinit), navedena se naredba jednostavno preskače te se odmah prelazi na sljedeću naredbu (`x = x * x;`).

Do sada smo prešutno pretpostavljali da je n pozitivan broj. Ako je n jednak nuli, uvjetni izraz `n > 0` petlje **`while`** nije istinit, pa se petlja neće izvesti ni jednom; program i tada daje ispravan odgovor: 1,0, jer je $x^0 = 1 \ \forall x \neq 0$. (U prethodnoj inačici programa s petljom **`for`**, uvjetni izraz `i <= n`, nakon inicijalizacije `i = 1`, također nije istinit, pa se niti u tu petlju ne ulazi.)

No, ako je n negativan ($n = -m$, za m pozitivan), kontrolni izraz također neće biti istinit, ali odgovor 1,0 nije točan: znamo da je $x^{-m} = \dfrac{1}{x^m} = \left(\dfrac{1}{x}\right)^m$. U obje inačice programa moramo stoga prije petlje dodati

```
if (n < 0) {
  x = 1 / x;
  n = -n;
}
```

Jesmo li ispitali sve mogućnosti? Za eksponent n jesmo, no što je s bazom x? Problemi će nastati ako su i x i n jednaki nuli te ako je x jednak nuli, a n manji od nule. U prvom slučaju rezultat, $0{,}0^0$, nije definiran, dok je u drugom slučaju $\left(\dfrac{1}{0{,}0}\right)^n = \infty$. Uz to, za svaki je n veći od nule $0{,}0^n = 0{,}0$ pa ne treba ništa računati. Sve tri mogućnosti za $x = 0{,}0$ obuhvatit ćemo jednom naredbom **`if`**, koju treba dodati prije prethodne:

```
if (x == 0.0)
  if (n > 0) {
    cout << "y = 0.0" << endl;
    return 0;
  }
  else {
    if (n == 0)
      cout << "y = NaN" << endl;
    else
      cout << "y = Infinity" << endl;
    return 1;
  }
```

Unutar te naredbe ugniježđena je naredba **`if`**...**`else`**, a unutar nje je ugniježđena još jedna naredba **`if`**...**`else`**. Dok naredbu **`if`** upotrebljavamo kada se neka operacija smije izvesti samo ako je neki uvjet ispunjen (a u protivnom se preskače), s pomoću naredbe **`if`**...**`else`** omogućavamo odabir jedne od dviju operacija, ovisno o tome je li neki uvjet ispunjen ili nije. Kažemo da naredbe **`if`** i **`if`**...**`else`** omogućavaju *grananje* — odabir jedne od dviju mogućnosti: prva naredba, hoće li se neka operacija izvesti ili ne, a druga, izvođenje jedne od dviju mogućih operacija.

Kao što smo već rekli, ako je $x = 0{,}0$, moramo razlikoviti tri mogućnosti za vrijednost n. Ispitujemo prvo je li $n > 0$; ako jest, izvest će se takozvana grana **then** — blok iza ključne riječi **`if`** i uvjeta. (U jezicima *C* i *C++* nema ključne riječi `then`; ona se podrazumijeva. No, uvjetni izraz mora biti, kao i u naredbi **`while`**, u oblim zagradama.) Akoli pak n nije veći od nule, izvest će se grana

else — blok iza ključne riječi **`else`**. Za n su ostale još dvije mogućnosti pa je potrebna još jedna naredba **`if`**...**`else`**. Budući da do nje možemo doći samo ako n nije veći od nule, u njezinu ćemo granu **else** ući samo ako je n manji od nule. U obje se njezine grane izvodi po jedna naredba pa vitičaste zagrade nisu potrebne. Grane pak prethodne/srednje naredbe **`if`** sadrže po dvije naredbe pa moraju biti obuhvaćene vitičastim zagradama.

Obje grane završavaju naredbama **`return`**, što znači da se prekida izvođenje programa. Ako n nije veći od nule, rezultat ili nije definiran (`NaN` — *Not a Number*, dakle, nešto što nije broj) ili je beskonačan. Oba slučaja smatramo greškama pa operacijskom sustava vraćamo vrijednost 1 kao poruku da nešto nije bilo u redu.

I na kraju, sve rečeno možemo uklopiti u cjeloviti program. Prije nego što okrenete stranicu, pokušajte ga sami sastaviti.

```
#include <iostream>

using namespace std;

int main() {
  double x;
  int n;

  cout << "x -> ";
  cin >> x;
  cout << "n -> ";
  cin >> n;

  if (x == 0.0)
    if (n > 0) {
      cout << "y = 0.0" << endl;
      return 0;
    }
    else {
      if (n == 0)
        cout << "y = NaN" << endl;
      else
        cout << "y = Infinity" << endl;
      return 1;
    }

  if (n < 0) {
    x = 1 / x;
    n = --n;
  }

  double y = 1.0;
  while (n > 0) {
    if (n % 2 == 1)
      y = y * x;
    x = x * x;
    n /= 2;
  }

  cout <<  "y = " <<  y <<  endl;
  return 0;
}
```