Osnove inženjerske informatike II.

Uvod u programiranje

Funkcije

K. F. & V. B.

The art of programming is the art of organising complexity.
E. W. Dijkstra

1. Pozivi

Složene i zapletene zadatke obično rješavamo tako da ih razbijemo na niz jednostavnijih, više ili manje neovisnih zadataka, koje možemo zasebice rješavati. Rastavljanje se može nastaviti — često se podzadaci rastavljaju na još jednostavnije zadatke. Takva je *hijerarhijska dekompozicija* zamršena zadatka na one jednostavnije, lakše rješive, jedno od najjačih misaonih oruđa u svladavanju složenosti jer omogućava, s jedne strane, razumijevanje zadatka na raznim razinama cjelovitosti, potpunosti i podrobnosti opisa te, s druge strane, razumijevanje i rješavanje pojedinih podzadataka neovisno o drugima na istoj razini.

Program za potenciranje koji smo napisali u odjeljku *Petlje i grananja* neposredno se na potpuno prirodan način raspao na tri međusobno gotovo neovisne cjeline:

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

Razbijanje zadatka na lakše rješive cjeline moralo bi se, radi preglednosti, čitljivosti i lakšega razumijevanja programa, odraziti u njegovoj *hijerarhijskoj strukturi*. U proceduralnim je programskim jezicima, poput *Fortran*a, *Pascal*a ili *C*-a, oblikovanje funkcija osnovni mehanizam hijerarhijskoga strukturiranja programskog kôda; u jeziku *C*++ to je jedan od osnovnih mehanizama (znamo, naime, da *C*++ osim proceduralne podržava i neke druge programske paradigme koje također omogućuju strukturiranje kôda). Većini bi podzadataka trebale odgovarati *funkcije* — imenovane programske cjeline namijenjene rješavanju određenih, većih ili manjih zadataka. Sam bi naziv funkcije trebao biti ključem njezina značenja i njezine svrhe:

```
int main()  {
  double x;
  int n;

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

  double y = power (x, n);

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

Gotovo je odmah jasno što ovaj programčić radi. Ako pak ne znamo što radi program na kraju odjeljka *Petlje i grananja*, morat ćemo se poprilično potruditi da to iz programskoga kôda otkrijemo.

Naravno, dio kôda koji izračunava x^n mora biti uklopljen u funkciju `power()`. *C*/*C*++ programi obično se sklapaju od takvih razmjerno kratkih, neovisno napisanih i testiranih funkcija. Funkcije na višoj razini hijerarhije pozivaju funkcije s niže razine; na najvišoj je razini funkcija **`main`**`()`.

Dobro testirana funkcija za koju se sa stanovitom pouzdanošću (a u jednostavnijim slučajevima čak i sa sigurnošću) može reći da ispravno radi ono čemu je namijenjena, često se, zajedno s drugim srodnim i dobro testiranim funkcijama, sprema u biblioteku i upotrebljava kao „crna kutija"— dovoljno je znati *što* radi, nije pretjerano bitno *kako* to radi. Tada se ta funkcija u istom programu, pa i u različitim programima, može pozvati po volji mnogo puta s različitim vrijednostima kao argumentima (dio kôda koji nije zatvoren u funkciju trebalo bi svaki puta prepisivati, vjerojatno uz promjenu nazivâ varijabli).

Upotreba funkcije naziva se *pozivom funkcije*. Funkcije u programskim jezicima dijele mnoge sličnosti s matematičkim pojmom funkcije, ali vidjet ćemo da postoje i neke razlike. Formalno, u jezicima *C* i *C++* poziv funkcije je *izraz* u kojem se operator `()` primjenjuje na naziv funkcije, primjerice:

```
power (x, n)
```

Između oblih zagrada od kojih je operator `()` sastavljen navode se, slično kao u matematici, vrijednosti koje prenosimo u funkciju. Te vrijednosti nazivamo *funkcijskim argumentima*. U našem primjeru to su realni broj x i cijeli broj n, to jest, vrijednosti pohranjene u varijablama `x` i `n`. (Katkad ćemo jednostavnosti i sažetosti radi, iako ne baš korektno, kazati da su argumenti funkcije varijable `x` i `n`.) Argumenti funkcije `power()` mogu, naravno, biti i vrijednosti nekih drugih dviju varijabli:

```
power (y, m)
```

a mogu to biti i dvije doslovno navedene konstantne vrijednosti:

```
power (3.14, 13)
```

Vrijednost koju funkcija izračunava (i „vraća" naredbom **`return`**) je *vrijednost funkcije*. (Rekli smo da je poziv funkcije izraz; vrijednost funkcije ujedno je i *vrijednost* tog *izraza*.) S tom ćemo vrijednošću najčešće uraditi nešto korisno; možemo je, recimo, pridružiti nekoj varijabli:

```
z = power (x, n);
```

prenijeti u neku drugu funkciju ili u aritmetički izraz:

```
sqrt (power (x, 2)) + power (y, n)
```

ili ispisati:

```
cout << n << "th power of " << x << " is "<< power (x, n) << endl;
```

Ugniježđeni pozivi funkcija, poput

```
sin (sqrt (power (x, n)))
```

izvode se, kao što iz matematičkoga iskustva i očekujemo, „iznutra prema van". To znači da se unutarnja funkcija — funkcija koja se nalazi najdublje između parova zagrada (u primjeru je to `power()`) — izračunava prva. Njezina se vrijednost tada kao argument prenosi u sljedeću, najbližu vanjsku funkciju (ovdje `sqrt()`). I tako dalje: vrijednost funkcije `sqrt()` prenosi se u funkciju `sin()`.

Napomenut ćemo još da u *C*-u i *C++*-u funkcija može, ali ne mora biti definirana tako da vraća vrijednost.[1] Funkcije koje ništa ne vraćaju upotrebljavaju se zbog popratnih učinaka koje izazivaju

[1] U *Pascal*u se formalno razlikuju funkcije i procedure, a u *Fortran*u funkcije i potprogrami. Za razliku od funkcija, procedure i potprogrami ne vraćaju vrijednosti.

(takvih „funkcija" u matematici nema). Samo zbog popratnih učinaka mogu se pozivati i funkcije koje imaju vrijednost; u tom se slučaju njihova vrijednost jednostavno zanemaruje.

2. Definicija

Vrijeme je da definiramo našu funkciju `power()`.

Varijabla u funkciji koja prihvaća argument naziva se *parametrom funkcije*.[2] Parametri funkcije navode su u njezinu zaglavlju. *Zaglavlje* svake funkcije, pa i naše funkcije `power()`,

```
double power (double x, int n)
```

sastavljeno je od tri dijela:

- naziva funkcije: `power`,
- deklaracijâ i, ujedno, definicijâ parametara, između oblih zagrada iza naziva funkcije: `(double x, int n)`,
- tipa vrijednosti funkcije, ispred njezina naziva: **`double`**.

Tipovi parametara funkcije određuju njezinu domenu, a tip vrijednosti njezinu kodomenu. Može se reći da je navedeno zaglavlje prijevod u jezik *C++* matematičke definicije

$$\text{power} \ : \ \mathbb{R} \times \mathbb{Z} \ \rightarrow \ \mathbb{R}$$

(ali ne smijemo zaboraviti razliku između skupova $\mathbb{R}$ i $\mathbb{Z}$ i tipova **`double`** i **`int`**). Dakle, kao i odgovarajuća matematička funkcija, funkcija `power()` uređenom paru brojeva (od kojih je prvi tipa **`double`**, a drugi tipa **`int`**) pridružuje jedan određeni broj (tipa **`double`**).

Funkcijski parametri su varijable koje će se upotrebljavati u tijelu funkcije. Definicije parametara stoga su definicije varijabli. Sintaktička je razlika ponajprije u tome što definicije parametara ne završavaju točkama sa zarezima nego se odvajaju zarezima. Akoli ih je više istoga tipa, taj tip treba navesti uz svaki parametar:

```
double some_func (double x, y)          // syntax error
double some_func (double x, double y)   // OK
```

To je dodatna razlika: znamo da se u jednoj deklaracijskoj naredbi može navesti više „običnih" varijabli;

```
double x, y;
```

isto je što i

```
double x;
double y;
```

Nema li funkcija parametara, ostaju samo prazne zagrade:

```
int main()
```

[2] Argumenti se katkada nazivaju *aktualnim parametrima* ili *aktualnim argumentima*, a parametri *formalnim parametrima* ili *formalnim argumentima*.

Zaglavlje je početak definicije funkcije. Iza zaglavlja slijedi njezino tijelo:

result–type *function–name* (*parameter–definition–list*$_{opt}$) {
    statement–sequence$_{opt}$
}

Tijelo funkcije sadrži niz naredbi koje propisuju kako funkcija radi ono čemu je namijenjena. Tijelo funkcije je prema definiciji jezika složena naredba, pa se zatvara u vitičaste zagrade — čak i kada tijelo sadrži samo jednu naredbu, ona mora biti u vitičastim zagradama. Štoviše, kao što indeks *opt* (kratica od *optional* — neobvezan, prepušten izboru) uz *statement–sequence* nagoviješta, tijelo funkcije može biti prazno; takva funkcija neće biti pretjerano korisna:

```
void do_nothing_and_return_nothing() {}
```

ali je definicija jezika dopušta kao teorijsku mogućnost.

Cijela definicija funkcije `power()` ispisana je na sljedećoj stranici. Kao što vidite, kôd u tijelu funkcije sličan je kôdu koji smo napisali u odjeljku *Petlje i grananja*. Ipak, postoje neke bitne razlike, a potrebna su i dodatna pojašnjenja:

(1) Podaci u funkciju ne ulaze upisom iz ulaznoga tôka nego preko parametara `x` i `n`. Rekli smo već da su parametri nove varijable; pri pozivu funkcije pridružuju im se argumenti. Tako se pri pozivu

```
power(y, m)
```

vrijednosti varijabli `y` i `m` pridružuju parametrima/varijablama `x` i `n`; možemo reći i da je `x` *lvalue*, dok je `y` *rvalue*. Stoga vrijednosti parametara `x` i `n` možemo mijenjati bez straha da ćemo promijeniti vrijednosti varijabli `y` i `m` koje bi nam i nakon poziva funkcije mogle zatrebati. To vrijedi i onda kada argument i parametar imaju iste nazive, kao u pozivu

```
power(x, n)
```

Riječ je o dvije različite varijable, o dvije varijable s različitim dosezima. Pojam *dosega naziva* označava dio kôda u kojem se naziv, s deklariranim značenjem, može upotrebljavati. Parametar je *lokalna varijabla pozvane funkcije*, dok je argument *lokalna varijabla funkcije koja poziva* (ili globalna varijabla). Iako je parametar, strogo govoreći, definiran izvan tijela funkcije, njegov je doseg od { kojom tijelo započinje do } kojom završava. Lokalne su varijable nedostupne izvan tijela funkcije, a u njezinu tijelu skrivaju vanjske varijable istoga naziva.

(2) Izračunana vrijednost se ne ispisuje, nego se naredbom

```
return y;
```

vraća kao vrijednost funkcije. Naglašavamo, vraća se *vrijednost*, a ne varijabla u kojoj se ta vrijednost nalazi; ta je varijabla lokalna, pa je po izlasku nedostupna.

Slično tome, ako su $x = 0{,}0$ i $n > 0$, izvođenje program se ne prekida nakon ispisivanja rezultata, nego se, kao vrijednost funkcije, vraća vrijednost $0{,}0$:

```
return 0.0;
```

Tipovi vrijednosti koje se naredbama **`return`** vraćaju iz funkcije moraju biti jednaki tipu vrijednosti funkcije. U stvari, pravilo nije tako strogo: umjesto 0.0 mogli smo napisati 0. Prevodilac bi tada „cjelobrojnu” vrijednost prešutno pretvorio u „realnu”.

```
#include <iostream>
#include <limits>
#include <cstdlib>


double power (double x, int n) {
  if (x == 0.0)
    if (n > 0)
      return 0.0;
    else {
      bool iec559 = std::numeric_limits<double>::is_iec559;
      if (n == 0) {
        if (iec559)
          return std::numeric_limits<double>::quiet_NaN();
        else {
          std::cerr << "y = NaN" << std::endl;
          std::exit (1);
        }
      }
      else {
        if (iec559)
          return std::numeric_limits<double>::infinity();
        else {
          std::cerr << "y = Infinity" << std::endl;
          std::exit (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;
  }

  return y;
}
```

(3) Obrada slučajeva u kojima su $x = 0{,}0$ i $n \leq 0$ sada je nešto složenija.

Dijeljenja s nulom (općenitije, izračunavanja koja daju neizmjerne vrijednosti) i izračunavanja čiji rezultati nisu definirani gotovo su uvijek posljedice pogrešaka u logici programa ili u ulaznim podacima, no u nekim se slučajevima izvođenje programa ipak može ili smije nastaviti. To će ovisiti o kontekstu u kojem se takvo izračunavanje pojavljuje. No, o kontekstu u kojem je naša funkcija pozvana mi, njezini autori, obično ne znamo ništa pa je prekid izvođenja programa ponešto pregrubo rješenje problema takvih „zabranjenih" operacija. Odluku o nastavku ili prekidu izvođenja treba prepustiti, ako je to moguće, funkciji koja poziva funkciju `power()`.

IEEE Standard for Binary Floating–Point Arithmetic (*ANSI/IEEE 754–1985*, poznat i kao *IEC 60559:1989* ili *IEC 559*) skupove realnih brojeva prikazivih vrijednostima tipova **`float`**, **`double`** i **`long double`** proširuje „vrijednostima" `±Infinity` i `NaN` (*Not a Number*). Ako je zapis realnih brojeva usklađen s tim standardom, tada `Infinity` i `NaN` mogu biti „rezultati" operacija $1/0{,}0$ i $0{,}0^0$.

Svojstva tipova podataka namijenjenih zapisu brojeva sadržana su u *standardnom predlošku* **`numeric_limits`**<> (deklariranom u zaglavnoj datoteci limits). Zasad nećemo (i ne možemo) podrobnije objašnjavati što su predlošci; ukratko i neformalno, predložak je nešto što je parametrizirano tipom ili tipovima. Za funkcije se može reći da su parametrizirane vrijednostima: pozivamo ih tako da njihovim parametrima pridružimo vrijednosti; predloške pak upotrebljavamo tako da njihovim parametrima pridružimo tipove. Prema definiciji jezika naziv tipa se navodi između znakova < i >. Tako ćemo za pristup numeričkim svojstvima tipa **`int`** pisati **`numeric_limits`**<**`int`**>, za pristup svojstvima tipa **`float`** pisat ćemo **`numeric_limits`**<**`float`**>, dok ćemo za pristup svojstvima tipa **`double`** napisati **`numeric_limits`**<**`double`**>. Pridruživanjem tipa i sam predložak postaje tipom.

Je li zapis brojeva tipa **`double`** usklađen sa standardom *IEC 559* utvrđujemo tako da ispitamo vrijednost člana *`is_iec559`* tipa **`numeric_limits`**<**`double`**>. (Pojedine članove nekih cjelina izdvajamo s pomoću operatora `::`. Već smo ga upoznali pri navođenju naziva iz standardne biblioteke, poput **`std::`***`cout`*.) Ako jest, vrijednost naše funkcije može biti `Infinity` ili `NaN`. Zapise tih „vrijednosti", kao vrijednosti tipa **`double`**, dobivamo pozivima funkcija *`infinity`*`()` i *`quiet_NaN`*`()`, također sadržanima u **`numeric_limits`**<**`double`**>.

Ako pak zapis brojeva nije usklađen sa *IEC 559*, nakon ispisa „rezultata" ipak prekidamo izvođenje programa. (Prekid izvođenja možemo izbjeći primjenom mehanizma *baratanja iznimkama*, no njegov bi nas opis odveo predaleko.) Tôk *`cerr`* je standardni izlazni tôk namijenjen ispisu poruka o pogreškama; najčešće je, kao i *`cout`*, povezan s „prozorom" u kojemu se program izvodi. Za prekid izvođenja programa ne možemo upotrijebiti naredbu **`return`**; ona nas iz funkcije `power()` vraća u funkciju koja ju je pozvala. Stoga pozivamo standardnu funkciju *`exit`*`()`. Ta funkcija, deklarirana u zaglavnoj datoteci cstdlib, izvodi veliko pospremanje (kakvo bi izveo program pri normalnom završetku), prekida izvođenje programa i okolini koja je pokrenula program vraća, kao izvještaj o uspješnosti izvođenja, vrijednost svog argumenta.

I još jedna napomena: funkciju ne možete definirati unutar definicije neke druge funkcije — sve su funkcije *globalne*; za razliku od *Pascal*a, u jezicima *C* i *C++* ne postoje *lokalne funkcije*.

3. Deklaracija

Kada prevodilac naiđe na poziv funkcije `power()`, mora znati kako interpretirati poziv — kojega tipa moraju biti argumenti i kojeg je tipa njezina vrijednost. Važna je uloga prevodioca i provjera je li funkcija ispravno upotrijebljena. Primjerice, pozovete li funkciju s

```
power ("Goodbye Pork Pie Hat", 2);
```

prevodilac će ispisati nešto poput

```
error:  incompatible type for argument 1
        of 'double power(double, int)'
```

ili poput

```
error:  cannot convert 'const char*' to 'double'
        for argument 1 to 'double power(double, int)'
```

A pri pozivu

```
power (2);
```

ispisat će nešto nalik na

```
error:  too few arguments to function 'double power(double, int)'
```

No, ne mora svaki poziv s neodgovarajućim tipovima argumenata biti pogreška:

```
power (2, 4);
```

Budući da poznaje tipove parametara, prevodilac će izvršiti prešutnu pretvorbu vrijednosti tipa ***int*** (prvi argument, 2, je konstanta tipa ***int***) u vrijednost tipa ***double***. Pretvorba vrijednosti tipa ***int*** u vrijednost tipa ***double*** je sigurna pretvorba jer se ne gubi točnost zapisa broja; takva se pretvorba naziva i unapređenjem tipa vrijednosti. No, pretvorba ne mora biti sigurna:

```
power (2, 3.99999);
```

Konstanta 3.99999, tipa ***double***, bit će prešutno pretvorena u cjelobrojnu vrijednost 3. Pristojni će vas prevodilac upozoriti:

```
warning: passing 'double' for argument 2 to 'double power(double, int)'
```

ali standard jezika to ne traži. Upozorenje (*warning*) nije pogreška (*error*), prevodilac vam samo želi reći da još jednom razmislite što radite — jeste li to zaista htjeli. Želite li ga ušutkati, tip vrijednosti možete izričito pretvoriti:

```
power (2, int (3.99999));
```

Kao što već znate, prevodiocu *deklaracijom* kazujemo kako treba interpretirati neki naziv.

Ako ste definiciju vaše funkcije smjestili prije funkcije koja je poziva (u našem primjeru funkcije **`main()`**), sama će ta definicija, točnije, zaglavlje funkcije u njoj, poslužiti kao deklaracija — doseg naziva funkcije počinje neposredno iza njezinog zaglavlja.[3] Ako se pak definicija vaše funkcije nalazi iza funkcije koja je poziva (ili u nekoj drugoj datoteci ili u nekoj biblioteci), treba je prije prvoga poziva deklarirati:

[3] To znači da funkcija može biti *rekurzivna* — može pozivati samu sebe.

```
// deklaracija funkcije:
double power (double x, int n);

int main() {
  // ...
  double y = power (2.0, 4);
  // ...
}

// definicija funkcije:
double power (double x, int n) {
  // ...
}
```

(// ... označava dijelove kôda koji nas trenutno ne zanimaju).

Deklaracija funkcije je sastavljena od njezina zaglavlja i točke za zarezom koja prevodiocu govori: to je sve, definicija je negdje drugdje (ako prevodilac ne nađe definiciju u istoj datoteci, stavit će njezin naziv na popis naziva koje mora razriješiti povezivač pretraživanjem ostalih ciljnih datoteka i biblioteka).

U deklaraciji je sadržano sve što prevodilac treba znati o funkciji: njezin naziv, broj, tipove i redoslijed parametara te tip njezine vrijednosti. Nazivi parametara nisu bitni, pa se mogu izostaviti:

```
double power (double, int);
```

Moguća su, prema tome, dva oblika deklaracije funkcije:

result–type function–name (*parameter–definition–list*$_{opt}$) ;

result–type function–name (*parameter–type–list*$_{opt}$) ;

(Navedu li se nazivi parametara, njihov doseg prestaje odmah iza znaka ;.)