5 Variabile şi atribuire

5.1 Evaluarea şi refolosirea expresiilor

Funcţiile din C, asemenea celor din matematică, au ca principal scop efectuarea unor prelucrări, datele fiind transmise obişnuit ca parametri. Uneori însă funcţiile folosesc în calcule rezultate intermediare. O funcţie de calcul al rădăcinilor unei ecuaţii de gradul II de exemplu, ar calcula întâi discriminantul, selectând apoi una din trei variante depinzând de semnul acestuia. Pentru aceasta trebuie efectuate două decizii, iar codul ar avea structura:

```
if (discriminant > 0) {
  tipăreşte soluţie 1
  tipăreşte soluţie 2
} else if (discriminant == 0)
  tipăreşte soluţia unică
else printf("nu are solutie reala\n");
```

În schema de mai sus, fie că înlocuim *discriminant* cu apelul unei funcţii care îi calculează valoarea, fie direct cu expresia respectivă, calculul său va fi repetat la fiecare apariţie (inclusiv în calculul celor două soluţii distincte). Dincolo de ineficienţă, *duplicarea de cod* (repetarea în program a aceluiaşi fragment, aici expresia `discrim(a, b, c)` sau `b*b - 4*a*c`) este ea însăşi dăunătoare, ducând la cod mai complicat şi mai greu de întreţinut, cu potenţiale erori dacă ulterior modificările nu se operează consistent pe toate fragmentele duplicate.

O soluţie ilustrată anterior e scrierea unei funcţii care să ia ca parametri `a`, `b`, şi discriminantul deja calculat `delta`. Aceasta ar urma să fie apelată din funcţia care ia ca parametri `a`, `b`, şi `c`, aşa cum s-a cerut.

```
void printsol(double a, double b, double delta);
// scrisa in capitolul anterior, similar cu schema de mai sus
void solve_eq2(double a, double b, double c)
{
  printsol(a, b, c, b*b - 4*a*c);
}
```

Astfel, expresia pentru discriminant apare şi e calculată doar o dată, la apel, valoarea fiind referită apoi în funcţie prin numele parametrului. Totuşi e nenaturală scrierea unei funcţii auxiliare pentru această problemă simplă.

Am întâlnit situaţia şi la funcţia `readint`, care nu are parametri, însă transmite un prim caracter citit cu `getchar()` funcţiei auxiliare `readint_c` care continuă citirea după cum acesta e semn sau nu. Aici *nu* putem repeta apelul `getchar()` în locul fiecărei utilizări a parametrului `c`, deoarece apeluri repetate ar consuma şi returna caractere *diferite* de la intrare.

Dăm încă un exemplu, pentru a insista asupra modului în care se face transmiterea parametrilor în limbajul C, *prin valoare*.

Scriem o funcţie care calculează puterea (naturală) a unui număr real prin înjumătăţirea succesivă a exponentului, după relaţia de recurenţă:

$$x^n = \begin{cases} 1 & n = 0 \\ x \cdot (x^{n/2})^2 & n \text{ impar} \\ (x^{n/2})^2 & n \text{ par} \end{cases}$$

Implementând direct după definiţia recursivă, cu o funcţie auxiliară pentru calculul lui x^2, obţinem un cod eficient, în care numărul de apeluri la funcţia putere este logaritmic în n (datorită înjumătăţirii exponentului).

```
double sqr(double x) { return x*x; }
double pow2s(double x, unsigned n) {
  return n == 0 ? 1 : n % 2 ? x * sqr(pow2s(x, n/2))
                : sqr(pow2s(x, n/2));
}
```

Dacă însă nu folosim funcţia `sqr` şi scriem direct înmultirea celor două jumătăţi egale în corpul funcţiei, putem constata (prin introducerea unei tipăriri la fiecare apel) că se fac apeluri repetate pentru acelaşi exponent, funcţia devenind mai ineficientă chiar decât cea simplă cu n înmulţiri.

pentru exponent $n = 3$:

```
double pow2(double x, unsigned n) {
  printf("exponent %u\n", n);
  return n == 0 ? 1
   : n % 2 == 0 ? pow2(x, n/2) * pow2(x, n/2)
   : x * pow2(x, n/2) * pow2(x, n/2);
}
```

```
exponent 3
exponent 1
exponent 0
exponent 0
exponent 1
exponent 0
exponent 0
```

În a doua variantă, apar două apeluri *distincte* `pow2(x, n/2)`, care sunt efectuate şi evaluate independent, fără refolosirea rezultatului. Prima variantă e diferită, deoarece la apelul `sqr(pow2s(x, n/2))` *nu* se expandează expresia din corpul funcţiei `sqr` în `pow2s(x, n/2) * pow2s(x, n/2)` . Transmiterea parametrilor în C se face *prin valoare*, funcţiile nu lucrează cu substituţii de expresii sau variabile. În momentul apelului la `sqr` se calculează *valoarea* a lui `pow2s(x, n/2)`, şi aceasta e transmisă ca parametru. Funcţia lucrează cu *valori numerice*, fără a cunoaşte expresiile din care acestea au provenit.

5.2 Variabile şi declararea lor

Avem nevoie de echivalentul conceptului de variabilă auxiliară din matematică, altfel spus, de posibilitatea de a asocia valoarea unei expresii cu un *nume*, prin care se poate apoi referi valoarea fără a fi necesară recalcularea ei. În limbajul C, aceasta se poate face prin *declararea* de *variabile*. O variabilă este un obiect caracterizat printr-un *nume* (un *identificator*, la fel ca şi numele de funcţii sau de parametri) şi o *valoare* de un anumit *tip*.

Declaraţia de variabile are sintaxa similară cu a parametrilor de funcţie: tipul urmat de numele variabilei şi `;` . Se pot însă declara mai multe variabile separate prin virgulă, fără a repeta numele tipului, iar fiecare poate fi *iniţializată* cu valoarea unei expresii care e evaluată la momentul declaraţiei (aceasta fiind chiar motivaţia dată mai sus pentru introducerea de variabile).
Simplificat, doar cu cele amintite până în prezent, sintaxa unei declaraţii e:

declaraţie ::= *tip* *lista-decl-init* `;`
lista-decl-init ::= *decl-init* | *lista-decl-init* `,` *decl-init*
decl-init ::= *identificator* | *identificator* `=` *expresie*

Astfel, secvenţa `int sgn = 1, x1, x2;` declară trei variabile întregi, prima iniţializată, iar celelalte două nu. O variabilă declarată fără initializare într-o funcţie are o valoare *nedeterminată*; la fel sunt şi rezultatele programului care o foloseşte înainte de a-i da o valoare. Asemenea erori nu apar dacă scriem întotdeauna iniţializarea cu expresia a cărei valoare vrem de fapt s-o folosim!

Declarând o variabilă, rescriem mai natural funcţia de citire a unui întreg:

```
int readint(void) {       // readnat, readnat_rc definite anterior
  int c = getchar();    // declaratie cu initializare
  return c == '-' ? - readnat() : c == '+' ? readnat() : readnat_rc(0, c);
}
```

De asemenea, cu discriminantul ca variabilă, scriem mai simplu soluţia ecuaţiei de gradul II, fără a mai fi nevoie de o funcţie auxiliară:

```
#include <math.h>       // pentru declaratia functiei sqrt
void solve_eq2(double a, double b, double c) {  // cere a != 0
  double delta = b*b - 4*a*c;
  if (delta > 0) {
    printf("solutia 1: %f\n", (-b - sqrt(delta))/2/a);
    printf("solutia 2: %f\n", (-b + sqrt(delta))/2/a);
  } else if (delta == 0) printf("solutie unica: %f\n", -b/2/a);
  else printf("nu are solutie\n");
}
```

Putem scrie declaraţii în cadrul oricărei instrucţiuni compuse, în particular în corpul unei funcţii. Variabilele declarate astfel se numesc *variabile locale* (discutăm ulterior declaraţiile de *variabile globale*, plasate în afara funcţiilor).

Următoarele noţiuni sunt importante pentru utilizarea variabilelor locale:
– *domeniul de vizibilitate* este porţiunea din program unde se poate folosi identificatorul declarat (e recunoscut ca numele unei variabile de tipul precizat). Se extinde din momentul declarării şi până la sfârşitul blocului respectiv.
– *durata de memorare* reprezintă porţiunea din execuţia programului pentru care se păstrează în memorie valoarea variabilei. Se extinde de asemenea din momentul parcurgerii declaraţiei şi până la părăsirea blocului respectiv.
– *iniţializarea* se face la *fiecare parcurgere* a declaraţiei.

Expresia din iniţializare e deci *reevaluată* la fiecare parcurgere a declaraţiei în execuţia programului, şi poate produce deci rezultate diferite: astfel, fiecare apel la funcţia `readint` produce o nouă evaluare a apelului `getchar()` care va returna *alt* caracter de la intrare. Variabila `c` nu poate fi folosită în afara funcţiei şi valoarea ei se pierde între două apeluri succesive.

Standardul ANSI C (1989) şi corespondentul său ISO din 1990 permiteau declaraţii doar înaintea tuturor instrucţiunilor dintr-un bloc. Standardul C99 defineşte un bloc ca secvenţă arbitrară de declaraţii şi instrucţiuni. Astfel putem declara şi iniţializa o variabilă chiar în punctul din program în care e nevoie de ea pentru a memora valoarea unei expresii.

În exemplele de până acum, valoarea variabilelor nu e modificată; ele sunt folosite doar pentru referirea la valoarea expresiei din momentul iniţializării. Aceasta corespunde cu limbajele de programare funcţionale, unde se foloseşte termenul de *legătură* (*binding*) între numele variabilei şi valoarea expresiei. Fundamental, această noţiune e diferită de *mutabilitate*, proprietatea de a putea modifica valoarea de care e legată o variabilă. În C, care este un limbaj imperativ, nu se face o astfel de distincţie, şi valoarea unei variabile poate fi modificată prin operaţia de *atribuire* (cu acelaşi simbol, `=` ca şi iniţializarea în declaraţie, deşi sunt două noţiuni conceptual şi sintactic diferite).

Altfel exprimat, în C nu privim variabila conceptual ca un nume la care se asociază prin legătură o valoare, ci operaţional ca nume asociat cu o locaţie de memorie a cărei conţinut poate fi modificat de oricâte ori.

Excepţie fac variabilele declarate cu *calificatorul de tip* `const` înaintea numelui de tip, de exemplu: `const double e = 2.71828;` . Nu e permisă modificarea valorii unui obiect astfel declarat, iar compilatorul va semnala ca eroare eventuale atribuiri la acesta în program.

5.3 Atribuirea

Operaţia de atribuire modifică o valoare memorată. Ea are sintaxa:

lvalue `=` *expresie*

Termenul *lvalue*, valoare care se poate afla la stânga (*left-hand*) unei atribuiri înseamnă o expresie care desemnează un *obiect*, adică în limbajul C, o zonă de memorie care conţine o valoare. Această categorie include variabilele şi elemente de limbaj prezentate ulterior (elemente de tablou, referiri prin adrese).

Atribuirea este o operaţie, iar `=` un operator binar, cu doi operanzi. *Efectul* operaţiei este evaluarea *ambilor* operanzi (evaluarea celui din stânga determină *care* e obiectul atribuit), şi atribuirea valorii expresiei din dreapta la obiectul din stânga. *Valoarea* întregii expresii de atribuire este chiar valoarea atribuită.

În C, atribuirea este o *expresie* şi nu o instrucţiune, şi poate apare oriunde sintaxa cere o expresie, inclusiv ca parte a altei expresii. Sigur, cel mai frecvent ea e folosită independent, urmată de `;` în instrucţiunea-expresie, de exemplu: `x = a + 5;`. Ea poate fi folosită însă şi în partea dreaptă a altei atribuiri: `x1 = x2 = -b/(2*a)`. Operatorul de atribuire e *asociativ la dreapta*, deci `x2` e atribuit cu `b/(2*a)`, iar `x1` e atribuit cu valoarea expresiei `x2 = -b/(2*a)`, care conform definiţiei e valoarea membrului drept. Deci `x1` şi `x2` sunt atribuiţi cu aceeaşi valoare, după cum sugerează şi sintaxa.

Pe de altă parte, atribuirea nu reprezintă o expresie *lvalue*: expresia de atribuire are noua *valoare* a obiectului atribuit, dar nu îl reprezintă pe acesta. Ca atare, nu putem folosi o atribuire în partea *stângă* a altei atribuiri.

Atenţie! O eroare frecventă e folosirea atribuirii `=` în loc de comparaţie `==` sau invers; codul rezultat poate fi corect sintactic, dar se comportă altfel decât intenţionat. Folosind comparaţia în loc de atribuire apar instrucţiuni-expresie de tipul `x == y + z;` care nu au nici un efect (rezultatul comparaţiei nu e folosit); multe compilatoare vor semnala un avertisment.

Mai frecventă e greşeala de a scrie `=` în loc de `==` . O decizie de forma `if (x = 5)` e corectă sintactic, atribuirea fiind o expresie! Valoarea ei e cea a expresiei atribuite, aici nenulă, deci "condiţia" va fi întotdeauna adevărată, iar `x` e atribuit neintenţionat! Similar, `x = 0` rezultă într-o condiţie falsă. Şi aici, compilatorul poate avertiza la folosirea atribuirii ca principal operator într-o decizie; intenţia programatorului e însă mai greu de evaluat.

Astfel, sintagma `if (`*var* `=` *func*`( ))` e validă şi chiar folosită în programe. Ea apare când rezultatul funcţiei trebuie memorat pentru folosire ulterioară, dar în acelaşi timp testat pentru validitate (un rezultat nul fiind considerat invalid, convenţie standard pentru tipul *adresă*). Expresia de atribuire din test se consideră adevărată doar dacă e nenulă, deci codul de pe prima ramură a deciziei e executat doar pentru un rezultat valid. Totuşi, compararea explicită `if ((`*var* `=` *func*`( )) != 0)` chiar dacă semantic redundantă, e mai clară şi exprimă mai bine intenţia codului.

5.4 Valoare şi efect lateral

Citirea şi scrierea (de la intrarea/la ieşirea standard, sau mai general din/în fişiere), şi atribuirea unui obiect memorat sunt principalele cazuri în care un program modifică starea mediului său de execuţie, acţiune numită *efect lateral*. Denumirea e consistentă cu abordarea funcţională pe care am introdus-o la început: principalul scop al unei funcţii e calculul rezultatului, iar eventualele modificări sunt secundare (laterale) fluxului principal de computaţie.

În stilul *imperativ* de programare practicat în limbajul C, efectul lateral are însă rolul central: principala instrucţiune elementară e instrucţiunea-expresie, reprezentată de atribuire şi apelurile funcţiilor de intrare/ieşire. Expresia care formează o astfel de instrucţiune nu e evaluată pentru valoarea sa, ci pentru efectul lateral. Dat fiind că orice expresie (mai puţin apelul unei funcţii `void`) are o valoare, iar unele au şi efect lateral, care poate modifica inclusiv valorile unor variabile folosite în evaluarea expresiei, se pune întrebarea *când* are loc efectul lateral în raport cu evaluarea expresiei.

Limbajul C nu impune restricţii privind ordinea de evaluare a operanzilor în expresii, cu excepţia celor implicite în definirea operatorilor condiţional şi de secvenţiere, şi particularităţii operatorilor `&&` şi `||` . La fel, momentul exact în care au loc efectele laterale e lăsat la latitudinea implementării, şi e constrâns doar prin definirea unor *puncte de secvenţă* (*sequence points*), unde toate efectele laterale produse de evaluările anterioare trebuie să fi produs. Asemenea puncte de secvenţă se află după fiecare instrucţiune, înainte de un apel de funcţie (după evaluarea parametrilor), înainte de revenirea dintr-o funcţie de bibliotecă, după evaluarea primului operand la `&&` şi `||`, etc. Înţelegerea detaliilor privind ordinea de evaluare şi efectele laterale e importantă pentru a evita programe cu efect dependent de implementare sau nedefinit.

5.5 Operatorii de pre- şi postincrementare

Unul din cele mai frecvente cazuri particulare de atribuire e incrementarea sau decrementarea unei variabile folosite ca şi contor, de exemplu `i = i + 1` sau `i = i - 1` . Limbajul C oferă operatorii unari `++` şi `--` pentru scrierea mai concisă a acestor expresii. Ei sunt tot operatori de atribuire şi merită o atenţie deosebită datorită folosirii lor frecvente şi existenţei a două variante diferite, *prefix* şi *postfix*: `++`*lvalue* respectiv *lvalue*`++` . Discutăm incrementarea, toate observaţiile fiind valabile pentru decrementare, cu schimbarea lui `+1` în `-1`.

Aplicat ca operator prefix sau postfix, `++` are *acelaşi efect lateral*: incrementarea obiectului referit de expresie (în particular: variabilei) cu valoarea `+1` *corespunzătoare tipului respectiv* (această precizare, aparent evidentă pentru tipuri numerice e foarte importantă pentru adrese, discutate ulterior).

Valoarea expresiei de incrementare este însă diferită în cele două cazuri. Pentru operatorul *postfix*, valoarea întregii expresii e cea a operandului *înainte* de incrementare, în timp ce pentru operatorul *prefix*, valoarea expresiei e cea a operandului *după* incrementare. Precizând printr-un exemplu, secvenţa

```
int n = 0; printf("%d ", n++); printf("%d ", n);
```

va tipări `0 1`, dar următoarea secvenţă pereche va tipări `1 1` :

```
int n = 0; printf("%d ", ++n); printf("%d ", n);
```

În ambele cazuri, al doilea număr tipărit este `1`, pentru că *efectul* celor două operaţii asupra lui `n` este acelaşi: incrementarea. Diferă primul număr tipărit, care este *valoarea* expresiei de incrementare, adică valoarea lui `n` înainte şi respectiv după incrementare, conform definiţiei date. Deosebirea se reţine uşor deoarece poziţia operatorului faţă de expresie sugerează relaţia în timp între atribuire şi evaluarea expresiei. Putem considera că atât `++n` cât şi `n++` au valoarea lui `n`, dar în primul caz atribuirea se efectuează înainte de evaluarea expresiei, iar in al doilea caz după. Corespunzător, se folosesc denumirile *preincrementare* şi *postincrementare*, şi la fel pentru decrementare.

Folosite ca instrucţiuni-expresie, ambele au acelaşi *efect* (deoarece nu se foloseşte *valoarea* expresiei). Dimpotrivă, folosite în alte expresii, ele diferă datorită *valorii* lor, ca în exemplul tipăririi.

Încheiem ilustrând cu un exemplu aspectele legate de ordinea de evaluare şi efecte laterale. O exprimare *greşită* a intenţiei de incrementare este `i = i++`. Această expresie are un efect nedefinit asupra variabilei `i`, deoarece conţine *două* efecte laterale: cel de incrementare şi cel de atribuire, iar ordinea lor de aplicare nu e restricţionată de limbaj. Presupunând că valoarea iniţială a lui `i` este 2, valoarea lui `i++` este tot 2. Ca urmare a expresiei, pe de o parte `i` urmează a fi incrementat, pe de altă parte atribuit cu 2. În funcţie de ordinea în care se aplică cele două modificări, valoarea finală a lui `i` poate fi 2 sau 3.
La fel, pentru `i = ++i`, efectele laterale sunt incrementarea şi atribuirea cu 3 (valoarea lui `++i` dacă `i` era 2), iar valoarea finală a lui `i` poate fi 3 sau 4 (dacă e incrementat după atribuire). Ambele expresii prezentate au efect nedefinit în C şi compilatorul le semnalează printr-un mesaj de avertisment.

Nedefinit e şi efectul tipăririi `int i=2; printf("%d %d", i++, ++i);` deoarece limbajul C *nu precizează* ordinea de evaluare a argumentelor unei funcţii, şi nici ordinea între evaluarea expresiilor şi aplicarea efectelor laterale. Evaluarea celor două expresii şi aplicarea efectelor în diverse ordini produc:

`2 4` *eval-1 efect-1* (`i = 3`) *eval-2 efect-2* (`i = 4`)
`3 3` *eval-2 efect-2* (`i = 3`) *eval-1 efect-1* (`i = 4`)
`2 3` *eval-1 eval-2 efect-1* (`i = 3`) *efect-2* (`i = 4`)

În concluzie, trebuie evitate expresii în care o variabilă afectată printr-un efect lateral apare şi în altă subexpresie, iar ordinea de evaluare a celor două expresii e neprecizată, deoarece în acest caz efectul global e nedefinit şi deci eronat. Mai general, efectele laterale în expresii complexe trebuie tratate cu multă atenţie, preferând descompunerea în expresii mai simple.

5.6 Operatori de atribuire compuşi

În practică apar des actualizări ale unei variabile printr-o operaţie elementară (adunare, înmulţire, etc.) faţă de valoarea curentă, de exemplu: `x = x - 10` sau `n = n / q`. Limbajul C oferă o formă prescurtată pentru expresii de forma *lvalue* = *lvalue op expresie* şi anume *lvalue op*= *expresie*
Exemplele de mai sus se rescriu astfel: `x -= 10` şi `n /= q` .
Există operatori compuşi pentru cei 5 operatori aritmetici: `+=`, `-=`, `*=`, `/=`, `%=` şi alţii pentru prelucrări pe biţi care vor fi prezentaţi la capitolul corespunzător. În toate cazurile, operatorul compus este *un singur element lexical*, fără spaţiu între operatorul de atribuire `=` şi cel care-l precede.

5.7 Un calculator pentru expresii simple

Scriem un program care citeşte de la intrare o expresie formată din numere întregi, operatori aritmetici şi paranteze, şi calculează valoarea ei. Definim întâi recursiv forma expresiilor acceptate:

expresie ::= *termen* | *expresie* `+` *termen* | *expresie* `-` *termen*
termen ::= *factor* | *termen* `*` *factor* | *termen* `/` *factor*
factor ::= *număr-natural* | `+` *factor* | `-` *factor* | `(` *expresie* `)`

În această definiţie, *expresie* şi *termen* sunt similar structurate, diferă doar operatorii multiplicativi, respectiv aditivi. Pentru *expresie*, de exemplu, calculul recursiv va avea permanent o valoare curentă, iniţializată prin apelul la *termen*. Pe măsură ce se citesc construcţii de tipul ±*termen*, ele intră în calculul noii valori curente, care e transmisă ca parametru în noul apel recursiv.

Deci, o variantă de bază, care nu acceptă decât operatorii + şi -, fără paranteze, ar avea structura:

```
int expr(int e) {       // apelam cu expr(readnat())
  int c = getchar();
  return c == '+' ? expr(e + readnat()) :
         c == '-' ? expr(e - readnat()) : e;
}
```

Pornind de la acest schelet de bază, scriem programul complet ţinând cont de următoarele observaţii:
– *expresie* nu e formată direct din numere naturale, ci din *termeni*, deci apelul `readnat` va fi inlocuit cu un apel către funcţia (tot recursivă) care citeşte şi calculează valoarea unui termen
– *termen* e construit similar cu *expresie*, din factori
– un factor poate fi o expresie (complexă) între paranteze. În acest caz, se apelează recursiv funcţia pentru *expresie*, iar la revenire se verifică existenţa parantezei închise
– acceptăm oricâte spaţii în expresie, care trebuie consumate înainte de fiecare operator sau factor
– tipărim un mesaj dacă acolo unde e necesar un operand nu apare un număr natural

```
#include <ctype.h>
#include <stdio.h>
int skipsp(int c) { return isspace(c) ? skipsp(getchar()) : c; }
int skipspace(void) { return skipsp(getchar()); }
unsigned readnat_rc(unsigned r, int c) {
  return isdigit(c) ? readnat_rc(10*r + (c-'0'), getchar()) : (ungetc(c, stdin), r);
}
int readnat_c(int c) {
  if (isdigit(c)) return readnat_rc(0, c);
  else { printf("lipseste numar\n"); return 0; }
}
int expr();
int factor() {
  int c = skipspace();
  if (c == '+') return factor();
  else if (c == '-') return - factor();
  else if (c == '(') {
    int res = expr();
    if ((c = skipspace()) != ')') printf("lipseste )\n");
    return res;
  } else return readnat_c(c);
}
int term2(int f) {
  int c = skipspace();
  if (c == '*') return term2(f * factor());
  else if (c == '/') return term2(f / factor());
  else { ungetc(c, stdin); return f; }
}
int term(void) { return term2(factor()); }
int expr2(int t) {
  int c = skipspace();
  if (c == '+') return expr2(t + term());
  else if (c == '-') return expr2(t - term());
  else { ungetc(c, stdin); return t; }
}
int expr(void) { return expr2(term()); }
int main(void) {
  printf("%d\n", expr());
  return 0;
}
```