3. Expresii aritmetice, apeluri de funcții şi ieşiri

În acest curs vom discuta în detaliu despre scrierea expresiilor aritmetice şi despre formatarea ieşirilor. Vom vedea, de asememea, cum putem folosi bibliotecile de funcții – funcții scrise anterior şi care fac parte din orice sistem C++.

3.1 Expresii aritmetice

Vom studia modul în care pot fi scrise expresii care conțin mai mult de un operator şi care folosesc operanzi care au diferite tipuri de dată.

Reguli de precedență

Expresiile aritmetice sunt formate din constante, variabile, operatori şi paranteze. Ordinea în care sunt realizate operațiile este stabilită conform *regulilor de precedență*. Cele 5 operații aritmetice de bază şi parantezele sunt ordonate în felul următor:

Cea mai înaltă precedență

```
( )
*  /  %
+  -
```

Cea mai scăzută precedență

Exemplu

```
tempMedie = INGHET + FIERBERE / 2.0
```

În acest exemplu, mai întâi se efectuează împărțirea `FIERBERE / 2.0`, iar apoi rezultatul este adunat cu `INGHET`.

Folosind parantezele, se poate schimba ordinea de evaluare a expresiei.

Exemplu

```
tempMedie = (INGHET + FIERBERE) / 2.0
```

Mai întâi sunt evaluate subexpresiile din paranteze, iar apoi urmăm precedența operatorilor.

Atunci când apar în aceeaşi expresie mai mulți operatori cu aceeaşi precedență, *ordinea de grupare* sau *asociativitatea* este de la stânga la dreapta. Expresia

```
int1 - int2 + int3
```

este echivalentă cu

```
(int1 - int2) + int3
```

şi nu cu

```
int1 - (int2 + int3).
```

Un alt exemplu:

```
(float1 + float2) / float1 * 3.0
```

Se evaluează mai întâi parantezele, apoi rezultatul se împarte la `float1`, iar în final se realizează multiplicarea cu `3.0`.

Conversii implicite şi explicite

Valorile întregi şi cele reale sunt stocate în mod diferit în memorie. Modelul din memorie al biților care reprezintă constanta 2 nu arată ca modelul din memorie al biților care reprezintă constanta 2.0. Problema este ce se întâmplă când folosim un întreg şi un real în aceeaşi expresie sau într-o asignare.

Instrucțiuni de asignare

Dacă facem declarațiile

```
int unInt;
float unFloat;
```

atunci variabila `unInt` poate păstra doar valori întregi, iar variabila `unFloat` doar valori în virgulă mobilă. Instrucțiunea de asignare

```
unFloat =12;
```

pare că încarcă valoarea întreagă `12` în variabila `unFloat`. Însă calculatorul refuză să stocheze altceva decât valori de tip float în variabila `unFloat`. Compilatorul inserează, în acest caz, două noi instrucțiuni care mai întâi convertesc valoarea `12` în `12.0` şi apoi stochează `12.0` în variabila `unFloat`. Această transformare automată a unei valori dintr-un tip de dată în alt tip de dată se numeşte *conversie implicită* (*type coercion*, *forțare de tip*).

Instrucțiunea

```
unInt = 4.8;
```

provoacă de asemenea o forțare de tip. Când un număr real este asignat unei variabile întregi, partea fracționară este trunchiată. Ca rezultat, lui `unInt` i se asignează valoarea `4`.

Adeseori, în conversiile implicite sunt implicate expresii întregi. Păstrarea rezultatului unei expresii cu rezultat de tip întreg într-o variabilă reală (float) nu provoacă pierderi de informație. Stocarea rezultatului unei expresii reale într-o variabilă întreagă conduce la trunchierea părții fracționare.

Pentru a clarifica programul şi pentru a evita erorile putem folosi *conversia explicită* (*type casting*). În C++o operație de cast constă din precizarea tipului de dată pe care dorim să îl aibă rezultatul urmat, între paranteze, de expresia pe care dorim să o convertim.

Exemplu

```
unFloat = float(3 * unInt + 2);
unInt = int(5.2 / unFloat – altFloat);
```

Următoarele două instrucțiuni produc rezultate identice:

```
unInt = unFloat + 8.2;
unInt = int(unFloat + 8.2);
```

Diferența constând în claritatea programului şi eliminarea erorilor de la compilare.

Scrierea expresiilor aritmetice

Până acum am vorbit doar despre combinarea diferitelor tipuri de dată în operația de asignare. Este, de asemenea, posibilă combinarea datelor de diferite tipuri în expresii.

Exemplu

```
unInt * unFloat
4.8 + unInt – 3
```

Întotdeauna, când într-o expresie apar variabile de tip întreg şi variabile de tip float apar conversii implicite după cum urmează:

1. Întregul este forțat temporar la o valoare reală
2. Se efectuează operația
3. Rezultatul este real

Să analizăm a doua instrucțiune din exemplul anterior, în care varianta `unInt` conține valoarea `2`. Operatorul + are operanzi de tipuri diferite, de aceea valoarea lui `unInt` este forțată la `2.0`. Această conversie este temporară şi nu afectează

valoarea 2 stocată în unInt. Se efectuează adunarea, iar rezultatul este 6.8. Scăderea are de asemenea, doi operanzi de tipuri diferite: 6.8 şi 3. Valoarea 3 este forțată la 3.0, se execută scăderea şi rezultatul este numărul real 3.8.

În interiorul expresiilor se pot folosi conversiile explicite de tip pentru a reduce riscul de apariție al erorilor şi pentru claritate:

Exemplu

```
float(unInt) * unFloat
4.8 + float(unInt – 3)
```

Conversiile explicite de tip nu se fac, însă, doar pentru claritate.

Să calculăm media mai multor numere. Suma lor este stocată în sum si numărul lor este stocat în count. Avem următoarele declarații:

```
int sum;
int count;
float average;
```

Valoarea medie se găseşte astfel:

```
average = sum / count; //eroare
```

Dacă sum este 6.0 şi count este 80, rezultatul va fi 0.0. De ce? Expresia din dreapta operatorului = conține doi operanzi întregi. În această situație, împărțirea este de tip întreg, deci rezultatul este 0. Apoi, rezultatul este convertit la valoarea reală 0.0 înainte de a fi stocat în average. Pentru a corecta rezultatul, modificăm ultima instrucțiune astfel:

```
average = float(sum) / float(count);
```

Împărțirea va fi reală, iar rezultatul va fi 0.75.

Până acum ne-am referit doar la tipurile int şi float. Conversiile se pot aplica şi valorilor char, short sau double.

3.2 Apeluri de funcții şi biblioteci de funcții

Apeluri de funcții

În capitolul anterior am văzut un program care conținea trei funcții: main, Patrat şi Cub. Toate trei returnau cate o valoare. Patrat si Cub returnează valori către funcțiile apelante, iar main întoarce o valoare către sistemul de operare.

În instrucțiunea

```
cout << ” si cubul lui 27 este ” << Cub(27) << endl;
```

secvența Cub(27) este un *apel de funcție* sau *invocare de funcție*. Calculatorul opreşte temporar execuția funcției main şi porneşte funcția Cub. Când Cub îşi încheie execuția tuturor instrucțiunilor, calculatorul revine la main din punctul în care a fost oprită.

În apelul funcției Cub, numărul 27 se numeşte *parametru* sau *argument*. Parametrii creează posibilitatea unei funcții să lucreze cu diferite valori. Astfel, putem scrie

```
cout << Cub(4);
cout << Cub(16);
```

Modelul sintactic al unui apel de funcție este

```
NumeleFuncției(ListăDeParametri)
```

Lista de parametri este mecanismul prin care funcțiile comunică una cu cealaltă.

Unele funcții, de exemplu `Patrat` sau `Cub`, au un singur parametru în lista de parametri. Alte funcții, de exemplu `main`, nu au niciun parametru în listă. Există funcții cu doi, trei sau mai mulți parametri în listă, separați prin virgulă.

Funcțiile care întorc o valoare pot fi utilizate în expresii în acelaşi fel în care se folosesc constantele sau variabilele. Valoarea calculată de funcție înlocuieşte apelul funcției în expresie.

Exemplu

```
unInt = Cub(2) * 10; //unInt va pastra valoarea 80
```

Într-o expresie, un apel de funcție are cea mai mare precedență.

Considerații referitoare la apelurile de funcții:

- Apelurile de funcții sunt folosite în expresii. Nu apar ca instrucțiuni de sine-stătătoare;
- Funcția calculează o valoare (un rezultat) care poate fi folosit apoi într-o expresie;
- Funcția întoarce exact un rezultat – nu mai multe, nici mai puține.

Funcția `Cub` aşteaptă să i se dea, să i se *transmită* un parametru de tip int. Dacă primeşte un parametru de tip float, compilatorul realizează o forțare implicită a tipului de dată.

Exemplu

`Cub(6.9)` calculează 6^3 şi nu 6.9^3

Până acum am folosit doar constante literale ca şi parametri ai funcției `Cub`. Aceştia, însă, pot fi şi variabile sau constante simbolice şi, în general, expresii având un tip potrivit cu cel al parametrului.

În instrucțiunea

```
alfa = Cub(int1 * int1 + int2 * int2);
```

expresia care reprezintă lista de parametri este evaluată prima, şi numai după aceea rezultatul este transmis funcției. De exemplu, dacă `int1` conține `3` şi `int2` conține `5`, atunci parametrul transmis funcției `Cub` este `34`.

O expresie din lista de parametri a funcției poate include şi apeluri de funcții. Putem rescrie apelul precedent folosind funcția `Patrat`:

```
alfa = Cub(Patrat(int1) + Patrat(int2));
```

Biblioteci de funcții

Anumite calcule, cum ar fi rădăcina pătrată, sunt foarte des foloste în programe. De aceea, limbajul C++ include o *bibliotecă standard* care este o colecție de funcții prescrise care realizează anumite operații.

Fişierul header	Funcția	Tipul parametrilor	Tipul rezultatului	Rezultatul
`<stdlib.h>`	`abs(i)`	`int`	`Int`	Valoarea absolută a lui `i`
`<math.h>`	`cos(x)`	`double`	`double`	Cosinusul lui `x` (`x` în radiani)
`<math.h>`	`fabs(x)`	`double`	`double`	Valoarea absolută a lui `x`
`<math.h>`	`pow(x, y)`	`double`	`double`	x^y. Dacă `x=0.0`, `y` trebuie să fie pozitiv. Dacă `x<0.0`, `y` trebuie să fie întreg

Pentru a folosi o bibliotecă de funcții, trebuie să plasăm directiva `#include` la începutul programului, specificând fişierul header dorit.

Exemplu

Funcții void

Până acum am discutat despre funcții care întorc o valoare. Dacă studiem următoarea definiție de funcție, observăm că ea începe cu cuvântul `void` în loc de `int` sau `double`:

```
void Calcul(...)
{
     ...
}
```

Acesta este un exemplu de funcție care nu întorcea nicio valoare către funcția apelantă. Ea realizează doar o acțiune şi apoi revine- Acestea sunt *funcții care nu întorc nicio valoare* sau *funcții void*. În multe limbaje de programare aceste funcții se mai numesc şi *proceduri*.

Spre deosebire de funcțiile care întorc o valoare, acestea se apelează într-o singură instrucțiune de sine-stătătoare:

Exemplu

```
Calcul(plataPeOra, ore);
```

Din punctul de vedere al apelantului, aceste funcții arată ca o comandă:

```
ExecutaAsta(x, y, z);
FaAsta();
```

3.3 Formatarea ieşirilor

Formatarea ieşirilor unui program înseamnă modul în care se poate controla apariția pe ecran sau la imprimantă a rezultatelor programelor. Dacă variabilele `i`, `j` şi `k` au valorile `15`, `2` şi `6`, atunci instrucțiunea

```
cout << ”Rezultate: ” << i << j << k;
```

produce şirul de caractere

```
Rezultate: 1526
```

Fără spații între numere, rezultatul este dificil de interpretat.

Spațierea verticală

Am văzut deja că pentru aceasta se foloseşte manipulatorul `endl`. O secvență de instrucțiuni de instrucțiuni de ieşire continuă să scrie pe linia curentă până când `endl` termină linia.

Să vedem ce afişează următoarele instrucțiuni:

```
cout << ”Formatarea ” << endl;
cout << endl;
cout << ”iesirilor. ” << endl;
```

Prima instrucțiune produce afişarea pe ecran a şirului de caractere `Formatarea`, iar `endl` provoacă trecerea pe rândul următor. Următoarea instrucțiune produce o nouă trecere pe rândul următor a cursorului. A treia instrucțiune tipăreşte cuvântul `iesirilor` şi termină linia. Rezultatul este:

```
Formatarea

iesirilor.
```

Instrucțiunile de mai sus sunt echivalente cu :

```
cout << ”Formatarea ” << endl << endl;
cout << ”iesirilor. ” << endl;
```

sau cu

```
cout << ”Formatarea ” << endl << endl << ”iesirilor. ” << endl;
```

sau cu

```
cout << ”Formatarea ” << endl << endl
     << ”iesirilor. ” << endl;
```

Ultimul exemplu arată că o instrucțiune C++ poate fi scrisă pe mai multe linii. Compilatorul urmăreşte apariția semnului `;` şi nu sfârşitul fizic al liniei.

Inserarea spațiilor într-o linie

Pentru a controla spațierea orizontală se obişnuieşte introducerea unor spații suplimentare. Pentru a preveni afişarea numerelor 15, 2 şi 6 în forma

```
Rezultate: 1526
```

putem tipări câte un singur caracter (constantă tip `char`) între numere:

```
cout << ”Rezultate: ” << i << ’ ’ << j << ’ ’ << k;
```

Această instrucțiune va afişa:

```
Rezultate: 15 2 6
```

Dacă dorim afişarea unor spații mai mari, putem opta pentru folosirea şirurilor constante care conțin spații:

```
cout << ”Rezultate: ” << i << ”    ” << j << ”   ” << k;
```

Această instrucțiune afişează:

```
Rezultate: 15   2  6
```

Pentru a produce ieşirea:

```
 * * * *
* * * * *
```

putem folosi următoarele instrucțiuni:

```
cout << ” * * * * * ” endl;
cout << ”* * * * *” << endl;
```

Pentru ca spațiile să fie tipărite pe ecran, ele *trebuie* incluse între apostrafe sau ghilimele. Remarcăm că instrucțiunea:

```
cout << ’*’ <<       ’*’;
```

are următorul efect:

```
**
```

pentru că spațiile care sunt în afara apostroafelor nu sunt luare în considerare la tipărire.

Manipulatori

Am folosit de multe ori până acum manipulatorul `endl` pentru a termina o linie afişată pe ecran. În C++, un manipulator este o entitate care se comportă ca o funcție, dar se foloseşte ca o dată. Ca funcție el produce o acțiune, iar ca dată poate fi plasat într-o serie de operații de inserție:

```
cout << unInt << endl << unFloat;
```

Manipulatorii se folosesc numai în instrucțiuni de intrare sau de ieşire. Bibliotecile standard C++ oferă o serie întreagă de manipulatori, iar noi vom studia trei dintre ei: `endl`, `setw` şi `setprecision`.

Pentru a îl folosi pe `endl` trebuie să includem fişierul header `iostream`. Pentru ceilalți manipulatori trebuie să includem, în plus, şi fişierul header `iomanip`.

Exemplu

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
 int unInt = 2;
 cout << setw(5) << unInt;
}
```

Manipulatorul `setw` (*set width*) permite stabilirea numărului de coloane folosite pentru următoarea afişare. El se aplică doar numerelor şi string-urilor, nu şi datelor de tip `char`. Parametrul lui `setw` este o expresie întreagă numită *specificație a dimensiunii câmpului*. Numărul de coloane stabilite pentru afişare se numeşte *câmp*. Data afişată va fi aliniată la dreapta, iar pozițiile câmpului rămase astfel libere vor fi umplute cu spații.

Exemplu

```
int a = 33;
int b = 7132;

cout << setw(4) << a
     << setw(5) << b
     << setw(7) << "Salut";
```

␣␣33␣7132␣␣Salut

câmpurile au fost completate cu spații; acestea au fost marcate prin ␣

```
cout << setw(1) << a
     << setw(4) << b
     << setw(5) << "Salut";
```

337132Salut

câmpurile au fost mărite automat la dimensiunea datei afişate

Stabilirea dimensiunii câmpului afectează doar următorul element afişat. După aceea, dimensiunea este resetată la 0, ceea ce înseamnă că dimensiunea va fi extinsă la atâtea coloane câte sunt necesare.

Exemplu

```
int a = 33;
int b = 7132;

cout << "Salut"
     << setw(5)
     << a << b;
```

afişează

```
Salut   337132
```

La specificarea dimensiunii câmpului pentru numerele reale, trebuie să ținem cont că punctul zecimal ocupă şi el o poziție. Valoarea `4.85` ocupă 4 coloane, nu 3.

Exemplu

```
float x = 4.85;

cout << setw(4) << x << endl        4.85
     << setw(6) << x << endl          4.85
     << setw(3) << x << endl;       4.85
```

Există câteva observații care trebuie făcute în legătură cu afişarea numerelor reale.

1. Numerele foarte mari sunt afişate implicit în formă ştiințifică.

   Exemplu

`123456789.5` este afişat `1.23457+008`

2. Dacă numărul afişat este întreg, nu se va tipări ca număr real.

   Exemplu

   `95.0` este afişat `95`

Pentru a evita aceste formate implicite, înaintea afişării oricărui număr real trebuie să includem următoarele două instrucțiuni:

```
cout.setf(ios::fixed, ios::floatfield);
cout.setf(ios::showpoint);
```

Aceste instrucțiuni folosesc noțiuni mai avansate de C++ care nu pot fi explicate acum în detaliu. `setf` este o funcție `void` asociată stream-ului `cout` (punctul dintre cout şi `setf` este strict necesar). Prima instrucțiune ne asigură că numerele reale vor fi tipărite în formă zecimală şi nu ştiințifică. Cea de-a doua instrucțiune specifică faptul că punctul zecimal va fi tipărit întotdeauna, chiar şi pentru numere întregi. Momentan vom utiliza aceste instrucțiuni în această formă. Aceste setări rămân valabile până la o nouă modificare a lor.

Adeseori dorim să controlăm numărul de zecimale afişate, de exemplu pe 12.8 să îl tipărim `12.80` sau pe 16.38753 să îl tipărim `16.39`. Pentru aceasta trebuie să folosim manipulatorul `setprecision`.

Exemplu

```
cout << setprecision(3) << x;
```

Parametrul lui setprecision stabileşte numărul de zecimale cu care va fi tipărit un număr real.

Exemplu

```
float x = 310.0;
cout.setf(ios::fixed, ios::floatfield);
cout.setf(ios::showpoint);
cout << setw(10)                                  ␣␣␣␣310.00
     << setprecision(2) << x;
cout << setw(7)                                   310.00000
     << setprecision(5) << x;
x=4.827;
cout << setw(6)                                   ␣␣4.83
     << setprecision(2) << x;
```