1. O primă privire asupra programelor C++

1.1. Structura unui program

Un prim și cel mai tipic program de început în C++ ar fi cel care afișează pe ecran un mesaj, de genul "Hello World!":

```
1  // primul meu program in C++
2
3  #include <iostream>
4  using namespace std;
5
6  int main ()
7  {
8    cout << "Hello World!";
9    return 0;
10 }
```

```
Hello World!
```

Prima coloană din cele două de mai sus arată codul sursă și a doua coloană arată rezultatul programului compilat și executat. În stânga, numerele reprezintă numerele liniilor de cod – acestea nu fac parte din program ci sunt indicate doar în scop de informaare și pentru a face eventuale referințe la diferitele linii de cod.

Acesta este unul dintre cele mai simple programe care poate fi scris în C++, însă el conține totuși componentele fundamentale pe care orice program C++ le are. Vom analiza linie cu linie codul de mai sus:

```
// primul meu program in C++
```

Aceasta este o linie comentariu. Toate liniile care încep cu două caractere slash (//) sunt considerate comentarii și nu au niciun efect asupra comportamentului programului. Ele conțin scurte explicații sau observații ale programatorului în cadrul codului sursă.

```
#include <iostream>
```

Liniile care încep cu caracterul diez (#) sunt directive pentru preprocessor. Acestea nu reprezintă linii de cod obișnuite cu expresii, ci sunt indicații ce se adresează preprocesorului compilatorului. În acest caz directiva `#include <iostream>` spune preprocesorului să include fișierul standard *iostream*. Acest fișier include declarațiile bibliotecii standard input-output în C++, și ea trebuie inclusă deoarece funcționalitatea ei se va utiliza mai târziu in program.

```
using namespace std;
```

Toate elementele bibliotecii standard C++ sunt declarate în cadrul a ceea ce se numește un namespace, și anume namespace-ul cu numele `std`. Așa că pentru a accesa funcționalitățile acestuia, noi declarăm cu această expresie că vom utiliza aceste entități. Această linie este foarte des utilizată în programele C++ care folosesc biblioteca standard, și practic va fi inclus în majoritatea codurilor sursă pe care le vom scrie.

```
int main ()
```

Această linie corespunde începutului definiției funcției principale, `main`. Funcția `main` este punctul în care orice program C++ își începe executia, indiferent de locația din cadrul codului sursă. Nu contează dacă există alte funcții cu alte nume definite înainte sau după funcția `main` – instrucțiunile conținute în cadrul definiției acestei funcții, `main`, vor fi întotdeauna primele ce vor fi executate în orice program C++. Din acest motiv este obligatoriu ca orice program C++ să aibă o funcție `main`.

Cuvântul `main` este urmat de o pereche de paranteze rotunde (()). Acest lucru se întâmplă deoarece este vorba de o funcție: În C++, ceea ce deosebește o declaratie de functie de alte tipuri de expresii sunt aceste paranteze rotunde care urmează după numele funcției. În anumite cazuri între cele două paranteze se află o listă de parametri ai funcției.

Imediat după aceste paranteze rotunde urmează corpul funcției *main* încadrat de acolade ({}). Tot ceea ce este între acolade este ceea ce face funcția când aceasta se execută.

```
cout << "Hello World!";
```

Această linie este o instrucțiune C++. O instrucțiune reprezintă o expresie simplă sau compusă care produce un efect la execuție. De fapt, această instrucțiune realizează singura acțiune care generează un efect vizibil în acest prim program al nostru. `cout` este numele fluxului de date (engl. *stream*) standard de ieșire în C++, iar efectul instrucțiunii este de a insera o secvență de caractere (în acest caz șirul `"Hello World!"`) în stream-ul standard de ieșire (`cout`, care în general corespunde consolei). `cout` este declarat în fișierul standard `iostream` din cadrul namespace-ului `std`, așa încât acesta este motivul pentru care a trebuit să includem acel fișier cu directiva preprocessor din linia 3 și să declarăm că vom utiliza acel namespace în linia 4.

Observați caracterul punct – virgulă (;) care marchează sfârșitul oricărei instrucțiuni în C++.

```
return 0;
```

Instrucțiunea `return` face să se termine funcția `main`. `return` poate fi urmat de un cod de întoarcere (în exemplul nostru este urmat de codul de întoarcere cu o valoare de zero). Un cod return de 0 pentru funcția `main` este în general interpretat cum că programul a lucrat fără erori și după cum era de așteptat. Acesta este cel mai obișnuit mod de a încheia un program-consolă C++.

Programul a fost scris pe mai multe linii pentru a fi mai lizibil, dar în C sau C++ doar directivele preprocessor trebuie să înceapă fiecare pe o altă linie (și nu se termină cu punct-virgulă). În rest, programul principal l-am fi putut scrie și într-o singură linie:

```
int main () { cout << "Hello World!"; return 0; }
```

Deoarece caracterul punct – virgulă (;) separă practic instrucțiunile una de cealaltă, pur și simplu nu contează dacă scriem mai multe instrucțiuni pe o singură linie sau scriem o instrucțiune pe mai multe linii. Deci am putea scrie de exemplu și următorul program:

```
1  // my second program in C++
2
3  #include <iostream>
4  using namespace std;
5  int main ()
6  {
7    cout <<
8           "Hello World! ";
9    cout <<
10          "I'm a C++ program";
11   return 0;
12 }
```

```
Hello World! I'm a C++ program
```

1.2. Comentarii

Comentariile reprezintă părți ale codului sursă de care compilatorul nu ține seama. Scopul lor este doar de a permite programatorului să comenteze codul-program scris.

C++ permite două moduri de a insera comentarii:

```
1 // comentariu linie
```

```
/* comentariu
      bloc */
```

În primul caz, cunoscut și sub numele de comentariu-linie, se consideră ca și comentariu tot ce urmează scris după perechea de caractere slash (//) și până la sfârșitul respectivei linii.

În al doilea caz, cunoscut sub numele de comentariu-bloc, se consideră ca și comentariu tot ceea ce este scris între caracterele /* și */, cu posibilitatea de a include una sau mai multe linii.

1.3. O primă privire asupra variabilelor

O instrucțiune cum ar fi:

```
x = 5 ;
```

atribuie lui `x` valoarea `5`, după cum nu e greu de ghicit. Dar ce reprezintă de fapt `x`? `x` este o variabilă.

O variabilă în C++ este numele unei zone de memorie care poate fi folosită pentru a stoca o informație. Puteți să vă imaginați variabila ca o cutie poștală în care poți pune sau din care poți scoate o anumită informație. Toate calculatoarele au memorie RAM (Random Access Memory), care este disponibilă pentru a fi utilizată de către programe. Atunci când se declară o variabilă, o anumită bucată din acea memorie este pusă deoparte pentru acea variabilă.

În acest paragraf vom considera doar variabile de tip întreg (`integer`). Un `integer` este un număr întreg, cum ar fi 1, 2, 3, -1, -12, sau 16. O variabilă de tip `integer` este o variabilă care poate memora doar o valoare întreagă.

Pentru a declara o variabilă în general se utilizează o instrucțiune de declarare. De exemplu puteți declara o variabilă x de tip `integer` cu instrucțiunea de mai jos:

```
int x;
```

Când este executată această instrucțiune de către unitatea central de prelucrare (UCP) se va pune deoparte o mică zonă de memorie RAM. Să considerăm de exemplu că variabilei `x` i se asignează locația de memorie 140. Ori de câte ori programul vede valoarea `x` într-o expresie sau într-o instrucțiune știe că trebuie să se uite în locația de memorie 140.

Una din cele mai obișnuite operații efectuate cu variabilele este asignarea. Pentru a face acest lucru se folosește operatorul cu simbolul =. Atunci când UCP execută o instrucțiune cum ar fi `x=5;`, acest lucru se traduce cu "pune valoarea 5 în locația de memorie 140".

Ulterior în programul nostru am putea tipări acea valoare pe ecran utilizând `cout`:

```
cout << x;  // tipareste valoarea lui x  (locatia de memorie 140) la consola
```

In C++, variabilele se mai numesc și **l-values** (se pronunță *ell-values,* și provine de la ***left values***, tradus *valori de stânga*). Orice variabilă (sau **l-value** cum se mai numește) are o adresă de memorie, adresă la care se stochează valoarea ei. Au fost numite **l-values** (adică *valori de stânga*) deoarece sunt singurele valori care pot fi puse în partea stângă a unei instrucțiuni de atribuire. Când facem o atribuire în partea stângă a operatorului de atribuire (care este semnul = în C/C++) trebuie să fie neapărat o valoare de stânga (l-value).

În concluzie, instrucțiuni cum ar fi: `3 = 4;` sau `3 = x;` vor cauza erori la compilare deoarece 3 nu este o valoare de stânga. Valoarea 3 nu are o adresă de memorie, astfel încât nimic nu îi poate fi asignat. 3 înseamnă 3, iar valoarea lui 3 nu poate fi reasignată. În schimb dacă unei variabile

(l-value) i se asignează o valoare, valoarea curentă a variabilei va fi suprascrisă (se va pierde vechea valoare și variabila va reține noua valoare, data în partea dreaptă a operatorului de atribuire, care este semnul =).

Opusul lui **l-values** sunt **r-values** (tradus *valori de dreapta*). O **r-value** se referă la orice valoare care poate fi asignată unei valori de stânga (l-value). **r-values** sunt întotdeauna evaluate pentru a produce o singură valoare. Example de r-values sunt numere (ca de exemplu 3, care evaluat va fi tot 3), variabile (ca de exemplu x, care evaluat va fi numărul asignat ultima dată acestei variabile), sau expresii (ca de exemplu 2+x, care evaluat va fi ultima valoare a lui x plus 2).

Mai jos, câteva exemple de instrucțiuni de asignare, în care se poate vedea cum se evaluează r-values:

```
int y;      // se declara y ca variabila intreaga
y = 4;      // 4 se evalueaza la 4, care este asignat lui y
y = 2 + 5;  // 2 + 5 se evalueaza la 7, care este apoi asignat lui y

int x;      // se declara x ca variabila intreaga
x = y;      // y care este evaluat la 7, este asignat apoi lui x.
x = x;      // x evaluat ca fiind 7, este asignat apoi lui x (inutil!)
x = x + 1;  // x + 1 se evalueaza la 8, care este asignat apoi lui x.
```

Observații importante:

1) Nu există nicio garanție că variabilele unui program vor avea aceeași adresă de memorie de fiecare data când se va rula programul. De exemplu ar fi posibil ca la prima rulare variabilei x să-I fie asignată locația de memorie cu adresa 140. La a doua rulare e posibil ca lui x să-I fie asignată locația de memorie 168.
2) Când unei variabile i se asignează o locație de memorie valoarea din acea locație de memorie este nedefinită (cu alte cuvinte, valoare aflată ultima data acolo va fi încă în acea locație).

Acest lucru poate duce la rezultate interesante, dar și periculoase totodată. Să considerăm următorul program simplu:

```
// #include "stdafx.h" // scoateti comentariul daca lucrati in Visual Studio
#include <iostream>

int main()
{
    using namespace std;    // ne da acces la cout si endl
    int x;                  // se declara o variabila intreaga numita x

    // afiseaza valoarea lui x pe ecran (periculos, deoarece x nu e initializat)
    cout << x << endl;
}
```

În acest caz calculatorul va asigna o locație de memorie neutilizată lui x. Apoi va trimite valoarea aflată în respectiva locație de memorie lui cout, care va afișa valoarea. Dar ce valoare va fi afișată? Nimeni nu știe! Puteți încerca și vedea personal. S-ar putea să vă afișeze o valoare oarecare aflată în locația de memorie pe care i-a asignat-o lui x sau, cu compilatoare Visual Studio mai noi s-ar putea să vă apară un mesaj de eroare.

O variabilă la care nu i s-a asignat nicio valoare se numește **variabilă neinițializată**. Aceste variabile neinițializate sunt foarte periculoase, deoarece pot cauza probleme intermitente (datorită faptului că vor avea valori diferite la fiecare nouă rulare a programului). Acest lucru le face foarte greu de depistat. Majoritatea compilatoarelor moderne afișează avertismente (numite warning în

limba engleză) la compilare dacă au detectat că este folosită o variabilă neinițializată. De exemplu în cazul de mai sus ar putea să apară următorul avertisment:

```
c:\vc2005projects\test\test\test.cpp(11) : warning C4700: uninitialized local
variable 'x' used
```

E bine ca întotdeauna să se asigneze valori variabilelor atunci când acestea sunt declarate. C++ facilitează această inițializare a variabilelor permițând asignarea unei valori pe aceeași linie cu declararea variabilei:

```
1  int x = 0; // se declara variabila intreaga x si i se asigneaza valoarea 0.
```

Acest lucru asigură faptul că variabila respectivă va avea întotdeauna o valoare validă, ușurând depanarea dacă programul nu funcționează corect din alte motive.

Un artificiu folosit de programatorii experimentați este Acela de a asigna variabilei o valoare inițială care să fie în afara domeniului de valori obișnuite pentru respective variabilă. De exemplu, dacă avem o variabilă care să memoreze numărul de abonamente de telefon mobil ale unei persoane, am putea face următoarele:

```
1  int nr_ab = -1;
```

A avea -1 abonamente e un nonsens. Așa încât, dacă mai târziu am face aceasta:

```
1  cout << nr_ab << " abonamente" << endl;
```

și s-ar afișa "-1 abonamente", am putea ști că respectivei variabile nu i s-a asignat niciodată o valoare reală în mod corect.

Regula: Asignați întotdeauna valori variabilelor atunci când le declarați!

Vom discuta despre variabile mai detaliat într-un capitol următor.

cin

cin este opusul lui cout: în timp ce cout afișează date la consolă, cin citește date de la consolă. Având acum o înțelegere de bază a ceea ce sunt variabilele, putem utiliza cin pentru a citi date introduse de utilizator de la consolă pe care să le stocăm apoi variabile.

```
1  //#include "stdafx.h" // scoateti comentariul daca lucrati in Visual Studio
2  #include <iostream>
3
4  int main()
5  {
6      using namespace std;
7      cout << "Introduceti un numar: "; // cereti un numar
8      int x;
9      cin >> x; // citeste numarul de la consola si il memoreaza in x
10     cout << "Ati introdus " << x << endl;
11     return 0;
12 }
```

Încercați să compilați și să rulați acest program. Pentru a vedea ecranul de execuție și după afișarea mesajului `"Ati introdus "` urmat de valoarea introdusă, adăugați încă o linie de program înainte de return `0`, și anume: `system("pause");`

1.4. O primă privire asupra funcțiilor

O **funcție** este o secvență de instrucțiuni destinată să rezolve o anumită sarcină. După cum se știe deja, orice program trebuie să aibă cel puțin funcția numită main(). Majoritatea programelor au însă mai multe funcții, care toate lucrează analog cu main().

Adesea și noi, pe parcursul unei zile, suntem nevoiți să ne întrerupem programul pentru a face temporar altceva. De exemplu, poate citim o carte când ne amintim că trebuie să dăm un telefon. Punem un semn de carte acolo unde am rămas, mergem la telefon și vorbim, după care ne întoarcem și continuăm să citim de unde ne-am întrerupt.

Programele C++ lucrează la fel: Programul va executa secvențial instrucțiunile dintr-o funcție când interceptează apelul unei alte funcții. Un apel de funcție este o expresie care spune unității centrale de prelucrare (UCP) să întrerupă funcția curentă și să execute o altă funcție. UCP-ul "pune un semn de carte" în dreptul punctului curent de execuție și apoi apelează (execută) funcția numită în apelul de funcție. Când se încheie funcția apelată UCP-ul se întoarce în punctul marcat de semn și reia execuția din acel punct. Mai jos este un exemplu de program care arată cum sunt declarate și apelate noile funcții:

```
1  #include <iostream>
2  using namespace std;
3
4  // Declaratia functiei ApelTelefon()
5  void ApelTelefon()
6  {
7       cout << "Dau un telefon!" << endl;
8  }
9
10 // Declaratia functiei main()
11 int main()
12 {
13     cout << "Citesc!" << endl;
14     ApelTelefon(); // apelul functiei ApelTelefon()
15     cout << "Reiau cititul!" << endl;
16     return 0;
17 }
```

```
Citesc!
Dau un telefon!
Reiau cititul!
```

Acest program începe execuția la începutul lui main(), și prima linie care se execută va afișa `Citesc`!. A doua linie din main() este apelul funcției ApelTelefon(). În acest punct se suspendă execuția următoarelor instrucțiuni din main() și UCP sare la ApelTelefon(). Prima (și singura) linie din ApelTelefon() afișează `Dau un telefon!`. Când se încheie ApelTelefon(), funcția apelantă (main()) reia execuția de acolo de unde a întrerupt. Prin urmare, următoarea instrucțiune executată din main() va afișa `Reiau cititul!`.

Funcțiile pot fi apelate de mai multe ori, ca în exemplul de mai jos:

```
1  #include <iostream>
2  using namespace std;
3
4  // Declaratia functiei ApelTelefon()
5  void ApelTelefon()
6  {
7       cout << "Dau un telefon!" << endl;
8  }
9
10 // Declaratia functiei main()
11 int main()
12 {
```

```
Citesc!
Dau un telefon!
Dau un telefon!
Dau un telefon!
Reiau cititul!
```

```
    cout << "Citesc!" << endl;
    ApelTelefon(); // apelul functiei ApelTelefon()
    ApelTelefon(); // apelul functiei ApelTelefon()
    ApelTelefon(); // apelul functiei ApelTelefon()
    cout << "Reiau cititul!" << endl;
    return 0;
}
```

În acest caz programul principal main() este întrerupt de 3 ori de către apelul funcției ApelTelefon().

Însă main() nu este singura funcție care poate apela alte funcții. În următorul exemplu funcția ApelTelefon() va apela o a altă funcție numită ScrieMesaj().

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

// Declaratia functiei ScrieMesaj()
void ScrieMesaj()
{
     cout << "Scriu un mesaj!" << endl;
}
// Declaratia functiei ApelTelefon()
void ApelTelefon()
{
     cout << "Incep sa dau un telefon!" << endl;
     ScrieMesaj();
     ScrieMesaj();
     cout << "Inchei convorbirea!" << endl;
}

// Declaratia functiei main()
int main()
{
    cout << "Citesc!" << endl;
    ApelTelefon(); // apelul functiei ApelTelefon()
    cout << "Reiau cititul!" << endl;
    return 0;
}
```

```
Citesc!
Incep sa dau un telefon!
Scriu un mesaj!
Scriu un mesaj!
Inchei convorbirea!
Reiau cititul!
```

Valori return

Atunci când se încheie execuția funcției main() se returnează o valoare către sistemul de operare (apelantul) prin utilizarea unei instrucțiuni return. Și funcțiile definite de utilizator pot returna o valoare singulară către apelantul lor. Acest lucru se face modificând tipul valorii returnate de funcție în declarația funcției. Un tip de retur **void** înseamnă că funcția nu returnează nicio valoare. Un tip de retur int înseamnă că funcția returnează o valoare integer către apelant.

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

void ReturnNimic()
{
    // Aceasta functie nu returneaza nicio valoare
}

int Return5()
{
    return 5;
}
```

```
5
7
```

```
13 // Declaratia functiei main()
14 int main()
15 {
16     cout << Return5() << endl;
17     cout << Return5() + 2;
18
19     return 0;
20 }
```

Dacă în exemplul de mai sus adăugăm în linia 18 instrucțiunea:

```
18     cout << ReturnNimic();
```

compilatorul va da eroare când se încearcă să se compileze această linie, deoarece funcția ReturnNimic() returnează void și nu este permis a pasa void către cout.

O întrebare care se pune adesea este dacă funcția ar putea returna mai multe valori cu instrucțiunea return. Răspunsul este că nu. Funcțiile pot returna o singură valoare folosind instrucțiunea return. Totuși există căi de a rezolva și această problemă, lucru care se va discuta în paragraful dedicat funcțiilor.

Parametrii

În paragraful anterior ați învățat că o funcție poate returna o valoare apelantului. **Parametrii** se utilizează pentru a permite apelantului să transmită informații către funcție (deci în sens invers față de valorile return)! Acest lucru permite funcțiilor să fie scrise astfel încât să permită realizarea unor sarcini generice, fără a ține cont de valorile specifice utilizate. Transmiterea acestor valori concrete este lăsată în seama apelantului.

Dar cel mai bine puteți înțelege pe baza unui exemplu. Mai jos se vede exemplul extrem de simplu al unei funcții care adună două numere și returnează rezultatul către apelant.

```
1  #include <stdafx.h>
2  #include <iostream>
3
4  // add ia doi parametri de tip integer si returneaza ca rezultat suma acestora
5  // add nu tine cont de valorile concrete pentru x si y
6  int add(int x, int y)
7  {
8      return x + y;
9  }
10
11 int main()
12 {
13     using namespace std;
14     // Apelantul functiei add() este cel care decide valorile exacte pt x si y
15     cout << add(4, 5) << endl; // x=4 si y=5 sunt parametrii
16     return 0;
17 }
```

Când se apelează funcția add() lui x i se asignează valoarea 4 și lui y valoarea 5. Funcția calculează x + y, care dă valoarea, și apoi returnează această valoare apelantului. Această valoare, 9, este apoi trimisă lui cout pentru a fi afișată pe ecran.

La iesire vom avea:

```
9
```

Să privim alte câteva apeluri de funcții():

```
#include <stdafx.h>
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

int multiply(int z, int w)
{
    return z * w;
}

int main()
{
    using namespace std;
    cout << add(4, 5) << endl; // calculeaza 4 + 5
    cout << add(3, 6) << endl; // calculeaza 3 + 6
    cout << add(1, 8) << endl; // calculeaza 1 + 8

    int a = 3;
    int b = 5;
    cout << add(a, b) << endl; // calculeaza 3 + 5

    cout << add(1, multiply(2, 3)) << endl; // calculeaza 1 + (2 * 3)
    cout << add(1, add(2, 3)) << endl; // calculeaza 1 + (2 + 3)
    return 0;
}
```

```
9
9
9
8
7
6
```

Primele trei comenzi de afișare din main() sunt evidente. A patra comandă este de asemenea relativ ușor de urmărit:

```
21  int a = 3;
22  int b = 5;
23  cout << add(a, b) << endl; // calculează 3 + 5
```

În acest caz, add() este apelată cu x = a și y = b. Deoarece a = 3 și b = 5, add(a, b) = add(3, 5), care dă 8.

Să privim prima instrucțiune mai complexă:

```
25  cout << add(1, multiply(2, 3)) << endl; // calculeaza 1 + (2 * 3)
```

Când CPU(unitatea centrală de prelucrare a calculatorului) încearcă să apeleze funcția add(), va asigna x = 1 și y = multiply(2, 3). y nu este un integer, ci este un apel de funcție care trebuie rezolvat. Așa încât înainte ca CPU să apeleze add(), va apela multiply() unde z = 2 și w = 3. multiply(2, 3) produce valoarea 6, care va fi asignată parametrului y al lui add(). Deoarece x = 1 și y = 6, se apelează add(1, 6), care dă 7. Valoarea 7 este transmisă către cout.

Sau, exprimat mai concis (unde simbolul => este folosit pentru a reprezenta evaluarea dată de funcție):
add(1, multiply(2, 3)) => add(1, 6) => 7

Și următoarea instrucțiune pare complicată deoarece unul din parametrii transmiși funcției add() este un alt apel la add().

```
26  cout << add(1, add(2, 3)) << endl; // calculează 1 + (2 + 3)
```

Dar, în mod similar avem:

```
add(1, add(2, 3)) => add(1, 5) => 6
```

Utilizarea eficientă a funcțiilor

Una din cele mai mari provăcări pentru cei începători în ale programării (în afară de învățarea limbajului) este să învețe când și cum să utilizeze funcțiile în mod eficient. Funcțiile oferă o facilitate grozavă, și anume de a sparge programul în părți mult mai mici ușor de mânuit și reutilizabile, care pot fi apoi ușor conectate între ele pentru a realiza o sarcină mai mare și mai complexă. Desfăcând programul în părți mai mici, complexitatea per ansamblu a programului se reduce, ceea ce face ca programul să fie mai ușor de scris și de modificat.

În mod obișnuit, atunci când învățați C++, veți scrie o grămadă de programe care implică 3 pași:

1. Citirea intrărilor de la utilizator
2. Calcularea unei valori din datele de intrare
3. Afișarea valorii calculate

Pentru programe simple citirea intrărilor de la utilizator se poate face în general în main(). Totuși pasul #2 se pretează adesea pentru a utiliza o funcție. Această funcție ar trebui să ia datele de intrare de la utilizator ca parametri și să returneze valoarea calculată. Valoarea calculată poate fi apoi afișată (fie direct în main(), sau printr-o altă funcție dacă valoarea calculată este complexă sau are cerințe speciale de printare).

O regulă foarte bună este aceea ca fiecare funcție să realizeze o singură sarcină. Programatorii începători scriu adesea funcții care combină pașii 2 și 3. Totuși, deoarece a calcula o valoare și a o afișa sunt două sarcini distincte, acest lucru contravine regulii enunțate mai sus. Ideal, o funcție care calculează o valoare ar trebui să returneze valoarea calculată apelantului și să lase apelantul să decidă ce să facă în continuare cu valoarea calculată.