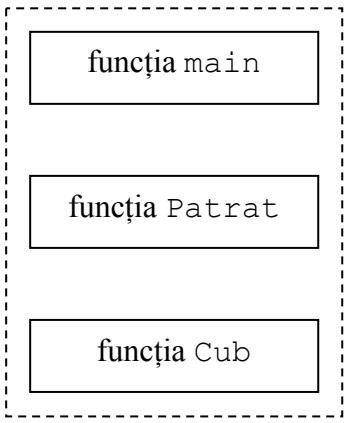
2. Sintaxa şi semantica C++

În acest curs vom vedea care sunt principalele reguli şi simboluri care fac din C++ un limbaj de programare. Vom vedea, de asemenea, ce paşi trebuie parcurşi pentru a scrie un program simplu şi a-l face să ruleze pe un calculator.

2.1 Structura unui program C++

Subprogramele permit scrierea separată a unor părți din program care, apoi, se asamblează în programul final. În C++, subprogramele se numesc *funcții*, iar un program C++ este o colecție de clase şi funcții.



Fiecare program C++ trebuie să conțină o funcție care se numeşte `main`. Aceasta poate fi privită ca o funcție master pentru celelalte funcții din program. Când `main` este programată să execute subprogramul corespunzător functie `Patrat` spunem că `main` *apelează* sau *invocă* funcția `Patrat`.

După ce `Patrat` încheie execuția tuturor instrucțiunilor, ea transmite (sau întoarce) controlul înapoi către funcția `main` care îşi continuă execuția.

Următorul exemplu este un program C++ care este alcătuit din 3 funcții: `main`, `Patrat` şi `Cub`. Detaliile nu sunt importante.

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

int Patrat(int);
int Cub(int);

int main()
{
  cout << "Patratul lui 27 este " << Patrat(27) << endl;
  cout << "si cubul lui 27 este " << Cub(27) << endl;

  return 0;
}

int Patrat(int n)
{
  return n*n;
}

int Cub(int n)
{
  return n*n*n;
}
```

În fiecare dintre funcții, acoladele { şi } marchează începutul şi sfârşitul instrucțiunilor care trebuie executate. Ceea ce se găseşte între acolade se numeşte *corpul funcției* sau *bloc de instrucțiuni*.

Execuția unui program C++ începe întotdeauna cu prima instrucțiune din funcția `main`. În programul de mai sus, aceasta este

```
cout << "Patratul lui 27 este " << Patrat(27) << endl;
```

Aceasta este o instrucțiune care produce afişarea unor informații pe ecranul calculatorului. Detalii vom afla puțin mai târziu, tot în acest capitol. Pe scurt, instrucțiunea tipăreşte două elemente. Primul este mesajul

```
Patratul lui 27 este
```

Cel de-al doilea este o valoare obținută prin apelul (invocarea) funcției Patrat cu valoarea `27`. Această funcție realizează ridicarea la pătrat şi trimite rezultatul, `729`, înapoi către *apelant* (*funcția invocatoare*), adică funcția `main`. Acum `main` continuă execuția tipărind valoarea `729` după care trece la instrucțiunea următoare.

Similar, a doua instrucțiune din funcția `main` tipăreşte mesajul

```
si cubul lui 27 este
```

după care invocă funcția `Cub`. Aceasta întoarce rezultatul `19683`, care este tipărit. Rezultatul final va fi

```
Patratul lui 27 este 729
si cubul lui 27 este 19683
```

Atat `Patrat` cât şi `Cub` sunt exemple de funcții care returnează o valoare. O astfel de funcție transmite o singură valoare către funcția apelant. Cuvântul `int` aflat la începutul primei linii a funcției `Patrat` arată că funcția întoarce o valoare întreagă (un număr întreg).

Să revenim la funcția `main`. Prima ei linie este

```
int main()
```

Cuvântul `int` indică faptul că `main` este o funcție care întoarce o singură valoare, un număr întreg. După tipărirea pătratului şi a cubului lui 27, `main` execută instrucțiunea

```
return 0;
```

pentru a întoarce valoarea 0 către apelant. Dar cine apelează funcția `main`? Răspunsul este: sistemul de operare. Acesta aşteaptă o valoare (exit status) de la `main` după ce aceasta îşi încheie execuția. Prin convenție, valoarea returnată 0 înseamă că totul a decurs OK. O altă valoare (1, 2 etc.) înseamnă că s-a petrecut ceva nedorit.

2.2 Sintaxă şi semantică

Vom începe acum să intrăm în detalii legate de programarea în C++.

Un limbaj de programare este un set de reguli, simboluri şi cuvinte speciale folosite pentru a scrie un program. Regulile sunt valabile atât pentru *sintaxă* (gramatică), cât şi pentru *semantică* (semnificație).

Sintaxa este un set de reguli care definesc exact ce conbinații de litere, numere şi simboluri pot fi folosite într-un limbaj de programare. Nu se acceptă ambiguități. Vom vedea că încălcarea oricărei reguli a limbajului, de exemplu scrierea incorectă a unui cuvânt sau uitarea unei virgule pot genera *erori de sintaxă* (syntax errors) şi codul sursă nu poate fi compilat până la corectarea lor.

Semantica este un set de reguli care determină semnificația instrucțiunilor scrise într-un limbaj de programare.

Şabloane sintactice

În acest curs vom folosi şabloane ca şi exemple generice de construcții sintactice în C++. Cel mai frecvent vom folosi şabloane asemănătoare celui pentru funcția `main`:

Funcția `main`

```
int main()
{
      instrucțiune
      …
}
```

Acest şablon arată că funcția main începe cu cuvântul int urmat de cuvântul main şi o pereche de paranteze rotunde. Prima linie a oricărei funcții numeşte *heading*. Acest heading este urmat de o acoladă care marchează începutul unei liste de instrucțiuni - *corpul funcției*. În final, acolada închisă indică sfârşitul funcției. Cele 3 puncte indică faptul că instrucțiunea poate fi urmată de 0 sau mai multe alte instrucțiuni.

Denumirea elementelor programului: identificatorii

În C++ *identificatorii* sunt nume asociate funcțiilor, claselor sau datelor şi sunt folosite pentru a referi funcții, clase sau date.

Identificatorii sunt alcătuiți din litere (A-Z, a-z), cifre (0-9) şi caracterul underscore (_), dar trebuie să înceapă cu o literă sau cu underscore.

Exemplu

Identificatori corecți: `J9`, `GetData`, `sum_of_squares`

Identificatori incorecți:

`40Hours`	- nu poate începe cu o cifră
`Get Data`	- nu poate conține spațiu
`box-22`	- nu poate conține – pentru că este simbol matematic
`int`	- cuvântul `int` este predefinit în C++

Cuvântul `int` este cuvânt rezervat. Acestea sunt cuvinte care au o utilizare specială în C++ şi nu pot fi utilizate drept identificatori definiți de programator.

Este foarte util ca identificatorii să fie sugestivi şi uşor de citit.

Exemplu

`PRINTTOPPORTION` față de `PrintTopPortion`

2.3 Date şi tipuri de date

De unde ia programul datele de care are nevoie pentru a lucra? Datele sunt păstrate în memoria calculatorului. Fiecare locație are o adresă unică pe care o referim atunci când dorim să stocăm sau să aducem date. Adresa fiecărei locații de memorie este un număr binar. În C++ folosim identificatori pentru a denumi locațiile de memorie. Acesta este unul dintre avantajele limbajelor de programare de nivel înalt: ne eliberează de grija de a gestiona locațiile de memorie la care sunt păstrate datele şi instrucțiunile. În C++ fiecare dată trebuie să fie de un anumit tip. Tipul de dată determină modul în care datele sunt reprezentate în interiorul calculatorului şi operațiile care se pot realiza asupra lor.

C++ defineşte un set de tipuri standard de date, pe care le vom descrie mai jos. De asemenea, programatorul îşi poate defini propriile tipuri de date. Tipurile standard sunt organizate astfel:

- Tipuri simple
  - Tipuri integrale
    - `char`

        - `short`
        - `int`
        - `long`
        - `enum`
    - Tipuri reale
        - `float`
        - `double`
        - `long double`
- Tipuri adresă
    - pointer
    - referință
- Tipuri structurate
    - tablou (array)
    - `struct`
    - `union`
    - `class`

Tipurile integrale

Se numesc aşa pentru că se referă la valori întregi. Despre tipul `enum` nu discutăm în acest capitol. Întregii sunt secvențe de una sau mai multe cifre. Nu se admite virgula. În multe cazuri, semnul – precedă un întreg.

Exemplu

```
22  16  1  -378  -912
```

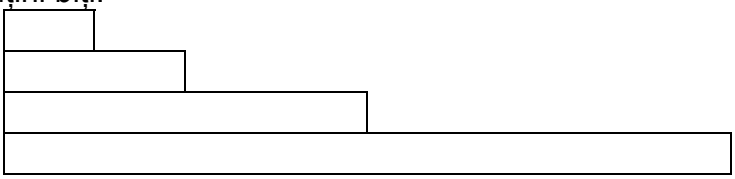
Atunci când cuvântul rezervat unsigned precedă un tip de dată, valoarea întreagă poate fi doar pozitivă sau 0.

Exemplu

```
unsigned int
```

Tipurile de dată `char`, `short`, `int` şi `long` reprezintă întregi de diferite dimensiuni cu mai mulți sau mai puțini biți.

celula de memorie `char`
celula de memorie `short`
celula de memorie `int`
celula de memorie `long`



În general, cu cât sunt mai mulți biți în celula de memorie, cu atât mai mari sunt valorile întregi care pot fi memorate acolo. Dimensiunile acestor celule de memorie sunt dependente de maşină. Pe unele calculatoare, o dată de tip `int` se poate găsi în intervalul -32768 ... +32767, iar pe altele între -2147483648 ... +2147483647. Când programul încearcă să calculeze o valoare mai mare decât valoarea maximă, rezultatul este un *integer overflow*. Un întreg care începe cu cifra 0 este considerat ca fiind scris în baza 8.

Exemplu

`015` este 15_8

Tipul `char`. Acesta este cel mai mic tip de dată care poate fi folosit pentru a reprezenta valori întregi. Se obişnuieşte folosirea acestui tip de dată atunci când se doreşte o economie de memorie şi se folosesc întregi mici. Însă tipul `char`, în mod tipic, se foloseşte pentru a descrie date care constau dintr-un caracter alfanumeric (literă, cifră sau simbol special).

Exemplu

```
'A'  'a'  '8'  '2'  '+'  '$'  ' '
```

Fiecare caracter este cuprins între apostroafe, astfel încât C++ face diferența dintre data de tip caracter `'8'` şi valoarea întreagă `8`, pentru că cele două sunt păstrate în mod diferit în interiorul calculatorului.

Nu sunt obişnuite operații de adunare a caracterului `'A'` cu caracterul `'B'`, de scădere a caracterului `'3'` din caracterul `'8'`, însă aceste caractere se pot compara. Caracterul `'A'` este întotdeauna mai mic decât `'B'`, `'B'` mai mic decât `'C'` etc. Fiecare set de caractere defineşte o astfel de secvență.

Tipurile reale (virgulă mobilă)

Aceste tipuri de date se utilizează pentru a reprezenta numere reale. Numerele reprezentate în virgulă mobilă au parte întreagă şi o parte fracționară, separate de un punct.

Exemplu

```
18.0  127.54  .8  0.57
```

Numărul `0.57` nu este octal. Această regulă este valabilă doar pentru numere întregi.

Aşa cum tipurile integrale din C++ au diferite dimensiuni, la fel se întâmplă şi cu tipurile reale. În ordine crescătoare, acestea sunt `float`, `double` şi `long double`.

Valorile reprezentate în virgulă mobilă pot avea un exponent, ca în notația ştiințifică (un număr este scris ca o valoare înmulțită cu 10 ridicat la putere). În loc de 3.504×10^{12}, în C++ scriem `3.504e12`. „`e`" înseamnă exponent al bazei 10. Numărul dinaintea lui `e` nu trebuie să includă în mod obligatoriu punctul zecimal.

Dintre tipurile de dată reale, cel mai folosit este `float` care, adeseori, este suficient. Valoarea maximă oferită de tipul `float` este, în general, în jur de `3.4e+38`.

Calculatoarele nu pot reprezenta întotdeauna numerele în virgulă mobilă. Datorită faptului că memorarea se face în formă binară, multe valori reale pot fi doar aproximate în acest sistem. De aceea, nu trebuie să ne mire că, pe unele calculatoare, de exemplu `4.8` va fi afişat `4.7999998`, fără a fi vorba de o eroare de programare.

2.4 Declarațiile

Identificatorii pot fi utilizați pentru a denumi constante sau variabile, adică locații de memorie al căror conținut se permite să fie modificat.

Cum spunem calculatorului ce reprezintă un identificator?

Declarația este o instrucțiune care asociază un nume (un identificator) cu o dată, o funcție sau un tip de dată, astfel încât programatorul poate să se refere la acest element prin nume.

Este la fel cum o definiție dintr-un dicționar asociază un nume unei descrieri a elementului la care ne referim prin acest nume.

De exemplu, declarația

```
int empNum;
```

Anunță că `empNum` este numele unei variabile al cărei conținut este de tip `int`. Când declarăm o variabilă, compilatorul alege o locație de memorie şi o asociază cu identificatorul păstrând această asociere la dispoziția programului. Orice identificator dintr-un program trebuie să fie unic în domeniul ei.

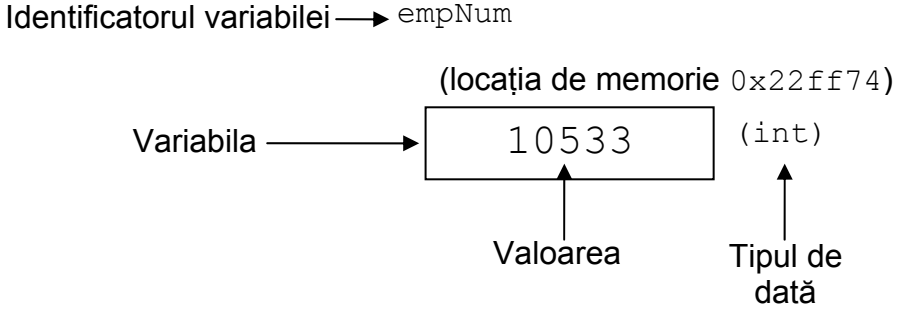
În C++ un identificator trebuie să fie declarat înainte de a fi folosit. Dacă declarăm un identificator ca fiind o constantă şi încercăm mai târziu să îi modificăm valoarea, compilatorul detectează această inconsistență şi semnalează eroare.

Constantele şi variabilele poartă numele generic de *date*.

2.5 Variabilele

Variabila este o locație de memorie referită printr-un identificator şi care păstrează valoarea unei date ce poate fi modificată.

Numele simbolic asociat variabilei se numeşte *identificatorul variabilei* sau *numele variabilei*.



Declararea variabilei înseamnă specificarea atât a numelui ei cât şi a tipului de dată. O declarație se termina întotdeauna cu ;

Exemplu

```
int empNum;
```

Se pot declara mai multe variabile de acelaşi tip într-o singură declarație.

Exemplu

```
int studentCount, maxScore, sumOfScores;
```

Aceasta declație este identică cu:

```
int studentCount;
int maxScore;
int sumOfScores;
```

Declararea fiecărei variabile într-o instrucțiune separată ne permite să adăugăm comentarii pentru înțelegerea declarațiilor la o parcurgere ulterioară a programului.

Exemplu

```
float payRate; //Employee's pay rate
```

Comentariile sunt precedate de `//` şi sunt ignorate de compilator.

2.6 Constantele

Toate numerele, întregi sau reale, sunt constante. La fel, caracterele (cuprinse între `' '`) şi secvențele de caractere sau string-urile, şirurile (cuprinse între `" "`).

Exemplu

```
16  32.3  'A'  "Boys"
```

În C++ ca şi în matematică, o constantă este un element al cărui valoare nu se schimbă niciodată.

Folosim constante ca părți ale unor expresii aritmetice. Putem scrie o instrucțiune care adună constantele `5` şi `6` şi plasează rezultatul în variabila numită `sum`. Numim *valoare literală* (sau literal) orice valoare constantă din program.

O alternativă la literale sunt *constantele simbolice* (sau constante cu nume). Acestea sunt locații de memorie referite printr-un identificator care păstrează date ce pot fi modificate. De exemplu, în loc să folosim literalul `3.14159` folosim constanta simbolică `PI`. Acest lucru face programul mai uşor de citit şi de modificat.

Declararea constantelor în C++ începe cu cuvântul rezervat `const`, iar semnul = se aşează între identificator şi valoarea literală.

Exemplu

```
const char BLANK = ' ';
const float PI = 3.14159;
const int MAX = 20;
```

De regulă, constantele simbolice se scriu cu litere mari pentru a le distinge mai uşor de variabile la citirea programului.

2.7 Acțiuni: instrucțiuni executabile

Asignarea

Valoarea unei variabile poate fi schimbată prin asignare.

Exemplu

```
quizScore = 10;
```

Valoarea `10` este asignată variabilei `quizScore`, adică valoarea `10` va fi stocată în locația de memorie numită `quizScore`. Semantica operatorului de asignare = este „păstrează”, „stochează”. Orice valoare anterioară stocată la acea locație de memorie se pierde, fiind înlocuită de noua valoare.

Într-o instrucțiune de asignare, doar o variabilă poate apărea în stânga operației. Asignarea nu este ca o ecuație matematică (x+y=z+4).

Având declarațiile:

```
int num;
int alpha;
float rate;
char ch;
```

putem face următoarele asignări:

```
alpha = 2856;
rate = 0.36;
ch = 'B';
num = alpha;
```

Asignarea

```
ch = ”Hello”;
```

nu este corectă pentru că `ch` este o variabilă de tip `char` iar `”Hello”` este un string.

Pentru o mai mare lizibilitate a programelor pe care le scriem, vom respecta următoarele reguli:

- Folosim inițiala mică pentru numele de variabile

Exemplu

```
lengthsInYards  hours
```

- Folosim inițiala mare pentru numele de funcții sau clase

Exemplu

```
Cub(27)  MyDataType
```

- Folosim litere mari pentru constante simbolice

Exemplu

```
UPPER_LIMIT  PI
```

Expresia din dreapta operatorului de asignare este evaluată şi valoarea ei este stocată în variabila aflată în stânga operaturului.

O *expresie* este alcătuită din constante, variabile şi operatori.

Exemplu

```
alpha+2  rate-6.0  alpha*num
```

Operatorii admişi într-o expresie depind de tipurile de date ale constantelor şi ale variabilelor din expresie.

Operatorii matematici sunt:

+ Plus unar
- Minus unar
+ Adunare
- Scădere
* Multiplicare
/ Împărțire reală (rezultat real) sau Împărțire întreagă (rezultat întreg)
% Modulo (restul împărțirii)

Operatorii unari folosesc un singur operand. Cei binari folosesc doi operanzi.

Exemplu

```
-54  +259.65  -rate
```

O constantă fără semn este pozitivă.

Împărțirea întreagă este câtul obținut la împărțirea a două numere întregi. Operația modulo reprezintă restul acestei împărțiri şi se aplică doar numerelor întregi.

Exemplu

```
6/2 → 3
7/2 → 3
6%2 → 0
7%2 → 1
```

Împărțirea în virgulă mobilă (reală) generează rezultat real, iar operanzii trebuie să fie reali.

Exemplu

```
7.0/2.0 → 3.5
```

Exemplu

Expresie	Valoare
3+6	9
3.4+6.9	10.3
2*3	6
8.0/-2.0	-4.0
8/9	0
5%2.3	Eroare

Expresie	Valoare
7.0/0.0	eroare
7/0	eroare
7%0	eroare

Pentru că în expresii pot apărea şi variabile, următoarele asignări sunt valide:

```
alpha = num + 6;
num = alpha * 2;
num = num + alpha;
num = 6 % alpha;
```

În cazul instrucțiunii

```
num = num + alpha;
```

valorile lui `num` şi `alpha` sunt adunate, iar rezultatul este păstrat în `num`, înlocuind vechea valoare păstrată aici. Acest exemplu arată diferența dintre egaliatea matematică şi asignare. În matematică

num = num + alpha

este adevărată doar când alpha este 0. Instrucțiunea de asignare

```
num = num + alpha;
```

este validă pentru *orice* `alpha`.

Incrementarea şi decrementarea

Pe lîngă operatorii aritmetici, C++ oferă operatorii de incrementare şi decrementare.

`++` Incrementare
`--` Decrementare

Aceştia sunt operatori unari. Pentru operanzi întregi şi reali, efectul este de adăugare a valorii 1, respectiv de scădere a valorii 1 din operand.

Exemplu

```
int num = 8;
num++; //num va avea valoarea 9
```

Acelaşi efect poate fi obținut prin

```
num = num + 1;
```

Operatorii `++` şi `--` pot fi atât *operatori prefix*:

```
++num;
```

cât şi *operatori postfix*:

```
num++;
```

C++ permite folosirea lui `++` şi `--` în interiorul unor expresii mai lungi:

Exemplu

```
alpha = num++ * 3;
```

Afişarea

Tipărirea rezultatelor se face în C++ folosind o variabilă specială numită `cout` împreună cu *operatorul de inserție* (`<<`).

Exemplu

```
cout << ”Hello”;
```

Această instrucțiune afişează caracterele `Hello` la *dispozitivul standard de ieşire*, de obicei ecranul. Variabila `cout` este predefinită în C++ şi semnifică un *flux de ieşire*. Acesta poate fi imaginat ca o secvență nesfârşită de caractere care merg către dispozitivul de ieşire.

Operatorul de inserție `<<` („put to”) foloseşte 2 operanzi. Cel din stânga este o *expresie flux* (stream) (de exemplu `cout`). În dreapta se află şirul sau expresia al cărei rezultat trebuie afişat.

Exemplu

```
cout << ”The answer is”;
cout << 3 * num;
```

De remarcat faptul că operatorul `<<` arată sensul în care circulă datele: dinspre expresie sau string înspre streamul de ieşire. Operatorul `<<` poate fi folosit de mai multe ori într-o singură instrucțiune de afişare:

Exemplu

```
cout << ”The answer is” << 3 * num;
```

şi rezultatul este acelaşi.

Exemplu

```
int i = 2;
int j = 6;
```

Instrucțiune	Ce se tipăreşte
`cout << i;`	`2`
`cout << ”i = ” << i;`	`i = 2`
`cout << ”j: ” << j << ” i: ” << i;`	`j:6 i:2`

Dacă dorim să afişăm un şir care conține caracterul ” trebuie să plasăm semnul \ înaintea ”:

Exemplu

```
cout << ”Al \”Butch\” Jones”;
```

iar pe ecranul calculatorului va apărea:

```
Al ”Butch” Jones
```

În mod obişnuit, mai multe instrucțiuni de ieşire succesive afişează rezultatele continuu, pe aceeaşi linie:

```
cout << ”Hi”;
cout << ”there”;
```

generează

```
Hithere
```

Pentru a tipări pe linii separate scriem:

```
cout << ”Hi” << endl;
cout << ”there” << endl;
```

şi obținem:

```
Hi
There
```

Identificatorul `endl` („end line”) este un element special din C++: este un manipulator. Este suficient de ştiut acum că `endl` permite terminarea unei linii şi continuarea scrierii pe linia următoare.

2.8 Construirea unui program

Vom vedea cum asamblăm diferitele elemente prezentate până acum pentru a construi un program.

Un program C++ este format din clase şi funcții, una dintre funcții numindu-se obligatoriu `main`. Un program poate conține declarații în afara oricărei funcții. Modelul unui program este:

```
Declarație
...
Definiție de clasă
Definiție de clasă
...
Definiție de funcție
Definiție de funcție
...
```

O definiție de funcție se construieşte după următorul model:

```
Heading
{
    Instrucțiune
    ...
}
```

Vom da exemplu de un program cu o singură funcție, funcția `main`.

```
//*************************************
//Temperaturi.cpp
//Acest program calculeaza temperatura de
//la jumatatea dintre punctul de inghet si
//cel de fierbere a apei
//*************************************

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

```
const float INGHET = 0.0;     //Punctul de inghet al apei
const float FIERBERE = 100.0; //Punctul de fierbere

int main()
{
  float temperaturaMedie;     //Pastreaza rezultatul medierii
                              //dintre INGHET si FIERBERE

  cout << "Apa ingheata la " << INGHET << " grade";
  cout << " si fierbe la " << FIERBERE << " de grade."
       << endl;

  temperaturaMedie = INGHET + FIERBERE;
  temperaturaMedie = temperaturaMedie / 2.0;
  cout << "Temperatura medie este de ";
  cout << temperaturaMedie << " grade." << endl;

  return 0;
}
```

Programul începe cu un comentariu care explică ce face programul. Urmează

```
#include <iostream>
```

care inserează conținutul fişierului `iostream` în programul nostru. El conține informațiile necesare lucrului cu stream-uri de intrare şi de ieşire, ca de exemplu `cout`.

Urmează declararea constantelor `INGHET` şi `FIERBERE`.

Restul programului este definiția funcției `main`. Mai întâi heading-ul urmat de `{}`. Aceste acolade informază compilatorul că `main` este o funcție. Corpul funcției cuprinde declararea variabilei `tempMedie` urmată de o serie de alte instrucțiuni executabile. Funcția `main` returnează `0`.

De notat aranjarea liniilor de program, spațierea şi folosirea comentariilor.

Blocuri (instrucțiuni compuse)

Corpul unei funcții este un exemplu de bloc:

```
{
    Instrucțiune
    ...
}
```

Un bloc conține 0 sau mai multe instrucțiuni cuprinse între `{}`. O instrucțiune se termină obligatoriu cu `;` Există şi instrucțiunea vidă:

```
;
```

Obişnuim ca instrucțiunile dintr-un bloc să le deplasăm puțin spre dreapta pentru claritatea programului. De asemenea, putem opta pentru gruparea instrucțiunilor prin separarea grupurilor cu rânduri libere.

2.9 Preprocesorul C++

Imaginați-vă că sunteți în rolul compilatorului de C++ şi vi se prezintă următorul program:

```
int main()
```

```
{
  cout << "Happy Birthday" << endl;
  return 0;
}
```

Recunoaşteți identificatorul `int` ca fiind cuvânt rezervat C++ şi `main` ca fiind numele unei funcții care trebuie să existe în mod obligatoriu. Dar `cout` şi `endl`? Nu au fost declarate ca şi variabile sau constante şi nu sunt cuvinte rezervate. Veți afişa următorul mesaj de eroare:

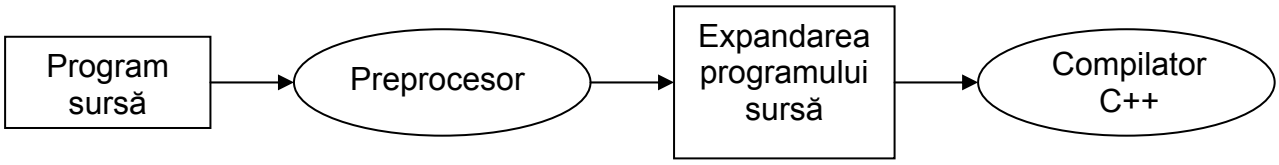
```
In function 'int main()':
Line 3: 'cout' undeclared
Line 3: 'endl' undeclared
```

Pentru a corecta eroarea, programatorul trebuie să insereze la începutul programului linia

```
#include <iostream>
```

Ea spune că întreg conținutul fişierului `iostream` va fi inserat în program. El conține declarațiile lui `cout` şi `endl`.

Instrucțiunea `#include` nu este interpretata de compilatorul C++, ci de un program numit *preprocesor*. El acționează ca un filtru şi precede faza de compilare. O linie care începe cu **#** se numeşte *directivă de preprocesare*.



La preprocesare sunt posibile următoarele acțiuni:

- Includerea altor fişiere în fişierul care urmează să fie compilat
- Definirea constantelor simbolice şi a macrourilor (macrourile sunt specifice limbajului C şi nu le vom prezenta)
- Compilarea condițională

Directiva de preprocesare `#include`

Această directivă de preprocesare produce includerea în codul sursă a unei copii a fişierului specificat. Ea are două forme.

Prima formă este cea în care se folosesc semnele `< >` ca în directiva

```
#include <iostream>
```

Acestea indică faptul că de referim la un fişier din biblioteca standard iar preprocesorul caută acel fişier în *directorul standard* pentru includere.

În varianta

```
#include "consum.dat"
```

Ghilimelele arată că fişierul `consum.dat` se găseşte în acelaşi director cu fişierul sursă.

Directiva de preprocesare `#define`: constante simbolice

Prin directiva de preprocesare `#define` se pot defini *constante simbolice*.

Exemplu

```
#define PI 3.14159
```

Proprocesorul va înlocui toate aparițiile lui `PI` din textul care urmează declarației cu valoarea `3.14159`. Dacă dorim să modificăm valoarea constantei, aceasta poate fi

modificată prin înlocuirea valorii din directiva de preprocesare. Diferențele dintre constantele definite prin cuvântul cheie `const` şi constantele definite prin directive de preprocesare sunt că primele au un tip de dată şi că sunt vizibile în faza de debugging. Odată înlocuită o constantă de către preprocesor, doar valoarea sa va mai fi vizibilă.

Directiva de preprocesare

```
#define DEBUG
```

în care lipseşte valoarea asociată constantei şterge orice apariție a identificatorului `DEBUG` din fişierul sursă. Identificatorul rămâne definit şi poate fi testat prin directivele `#if defined` sau `#ifdef`.

Există 5 constante simbolice predefinite:

Constanta simbolică	Descrierea
`__LINE__`	Numărul liniei curente din fişierul sursă
`__FILE__`	Numele fişierului sursă
`__DATE__`	Data la care a fost compilat fişierul sursă
`__TIME__`	Ora la care a fost compilat fişierul sursă
`__STDC__`	Constanta întreagă 1

Directiva de preprocesare `#undef` aplicată unei constante simbolice sau unui macro definite prin `#define` realizează ştergerea definiției.

Compilarea condițională

Compilările condiționate permit programatorilor să controleze execuția directivelor de preprocesare şi a compilării programului sursă. Compilările condiționale pot fi realizate prin folosirea directivelor de preprocesare `#ifndef` şi `#ifdef`.

Codul

```
#ifndef NULL
  #define NULL 0
#endif
```

verifică dacă `NULL` a fost deja definită în program, iar daca daca nu, o defineşte.

Compilarea condițională se foloseşte de regulă pentru debugging. Se folosesc instrucțiuni de afişare care tipăresc valorile unor variabile şi care confirmă fluxul corect al programului. Aceste afişări care nu mai sunt necesare dupa ce programul a fost corectat sunt, de regulă, încadrate de directive condiționale de preprocesare.

Exemplu

```
#ifdef DEBUG
  cout << "Variabila x = " << x << endl;
#endif
```