Structura unui program, variabile, instructiuni (citiri, scrieri)

Dezvoltarea unui program cu mediul de lucru Code::Blocks

Așa cum am spus în lecția anterioară, un program presupune prelucrarea unor date de intrare în scopul obținerii rezultatelor dorite, datele de ieșire. Pe lângă date, despre care am început să discutăm, alt element important sunt instrucțiunile. Ele reprezintă pașii care dorim să se execute în procesul de trecere de la datele de intrare la cele de ieșire. Dar, înainte să discutăm despre primele instrucțiuni, să vedem care este structura unui program în C/C++.

Un program este scris într-un fișier text care poate avea una dintre extensiile `c` sau `cpp`. Dacă extensia este `c` programul va trebui să conțină doar elemente ale limbajului C.

Trebuie respectată strict sintaxa limbajului. Altfel, în etapa de analiză a codului orice eroare va face ca procesul să se încheie.

Pentru început vă propun următoarea structură a unui fisier cu extensia `cpp`. Începem așa pentru că liniile de mai jos vor apărea în toate programele pe care le vom realiza în această etapă.

```
#include <iostream>
using namespace std;
...
int main() {
    ...
}
```

Despre linia `#include <iostream>` sunt de spus următoarele: în afară de operatori, instrucțiuni și alte elemente care fac parte din nucleul limbajului, odată cu dezvoltarea acestuia s-au pus la dispoziția programatorului diverse elemente uzuale (pentru ca acesta să nu mai fie nevoit să le rescrie). Atunci când dorim să folosim un astfel de element trebuie să anunțăm biblioteca în care acesta este definit. Vom vedea mai departe că biblioteca `iostream` o anunțăm (includem) pentru a putea folosi `cin` și `cout` cu ajutărul cărora citim și afișăm date. Alt exemplu este cu biblioteca `cmath`, pe care o anunțăm în program cu o linie de forma `#include <cmath>` în scopul utilizării diverselor funcții de calcul matematic (ex: `sqrt`, `log` ...).

Iată, în continuare, despre linia `using namespace std;`

Dacă ea ar lipsi, numele instrucțiunilor de intrare și de ieșire nu ar putea fi scrise în program direct: `cin`, `cout` ci sub forma `std::cin` respectiv `std::cout`. Iată o explicație. Identificatorii sunt grupați în *spații de nume*, iar cele două instrucțiuni au spațiul de nume `std`. De exemplu, am putea avea o altă bibliotecă, în care să avem definit un identificator `cin` cu altă semnificație. Dacă am include ambele biblioteci în același program și am folosi `cin`, la care dintre identificatori ar trebui să înțeleagă compilatorul că ne referim? Evident că ar fi o situație ambiguă și atunci, soluția de a evita asta ar fi să precedăm identificatorul de spațiul său de nume conform sintaxei de mai sus. Totuși, într-un program în care identificatorul apare doar într-o bibliotecă, ar fi incomod să îl scriem mereu precedat de spațiul de nume. Cu această linie de cod spunem că dacă un identificator nu are scris spațiul de nume de către programator, să se încerce automat plasarea în fața acestuia a lui `std`.

După cele două linii de cod am scris puncte de suspensie. Asta înseamnă că este liber să scriem acolo diverse elemente. La acest moment al drumului nostru vom scrie declarații de variabile.

Variabile

Acum este momentul să revenim la date și să discutăm despre *variabile*. Pe ele trebuie să ni le imaginăm ca fiind zone continue de octeți din memoria RAM cărora la declarare li se asociază un *nume* și un anumit *tip de date*.

În zona rezervată variabilei se va stoca o valoare de un anumit *tip*. Vom vedea că variabila are o valoare inițială dar acolo putem pune ulterior alte valori, de regulă prin instrucțiuni de citire sau atribuire.

Tipul de date este cel care indică natura informației pe care o stochează variabila precum și numărul de octeți care se vor rezerva în memorie pentru ea..

Ca natură de informație putem avea: numere întregi și numere reale. Pentru ambele categorii avem mai multe tipuri disponibile, diferența fiind doar numărul de octeți ocupați de variabilă (și implicit intervalul de valori posibile).

Tipurile întregi pe care le vom introduce acum sunt: `short`, `int` și `long long`.

Tipurile reale sunt `float` și `double`.

Declararea adică anunțarea în program a variabilelor necesită sintaxa:

```
tip nume;
    sau
tip nume1,
nume2;
```

Exemple:

```
int a;
long long b, c, d;
double x;
```

Numele unei variabile trebuie să respecte regula de scriere în limbaj a *identificatorilor* și anume: succesiune de litere ale alfabetului (mari, mici), cifre și caracterul undeline (_), dar să nu înceapă cu o cifră. Identificatorii sunt așadar nume, atribuite de programator sau predefinite, pentru diverse elemente de limbaj (nume de variabile, nume de funcții, nume de tipuri de date ...).

Limbajul C/C++ este unul *case sensitive*, adică se face diferență între caracterele literă mare și cele literă mică. Acest lucru este valabil nu numai la scrierea identificatorilor dar și pentru celelalte elemente. Sunt și câteva excepții, pe care le vom puncta la momentul potrivit (scrierea constantelor întregi hexazecimale, scrierea constantelor reale ...).

Exemple

`a`	Nume corect de variabilă
`A`	Nume corect de variabilă, dar diferit față de cel de pe linia anterioară
`b1`	Corect
`x 1`	Incorect (nu poate apărea spațiu într-un nume de variabilă)
`x 1`	Corect
`_12`	Corect
`A+2`	Incorect (nu poate apărea caracterul + într-un nume de variabilă)
`aNa`	Corect
`ana`	Corect, nume diferit față de cel de pe linia anterioară

`1x`	Incorect, numele unei variabile poate conține cifre dar nu poate să înceapă cu cifră

Iată în continuare o analiză mai detaliată a modului în care numărul de octeți rezervați în memorie influențează valorile posibile pentru o variabilă. Această secțiune poate fi sărită pentru moment de către cei care tocmai iau contact cu limbajul și se recomandă reveni la ea după ce se capătă ceva încredere.

Variabilele de tip `short` permit stocarea de valori întregi și ocupă doi octeți. Adică `16` biți. Cei `16` biți reprezintă `16` locații elementare din RAM, fiecare dintre ele putând avea fie valoarea `0` fie valoarea `1`. Așadar, sunt 2^{16} moduri distincte de a forma secvențe de `0` și de `1` de lungime `16`, adică tot atâtea valori posibile pentru o variabilă de tip `short`. Aceste configurații sunt atât pentru valori negative cât și pentru valori pozitive (jumătate pentru fiecare). Deducem că valorile posibile pentru o variabilă de tip `short` sunt între -2^{15} și $2^{15}-1$. Mai exact între `-32768` și `32767`. (diferența de `1` între valorile absolute ale minimului și maximului poate fi explicată acum prin prezența în interval a valorii `0`).

Tipul `short` poate fi precedat de cuvântul `unsigned` (lucru valabil și la celelalte două tipuri de date întregi prezentate) obținând un nou tip de date, numit `unsigned short`. Datele de tip `unsigned short` ocupă de asemenea `16` biți, dar toate configurațiile reprezintă valori pozitive. Astfel, aceste sunt cuprinse între `0` și $2^{16}-1$, adică între `0` și `65535`.

Discuția de mai sus este valabilă și pentru celelalte tipuri întregi. Iata, în tabelul de mai jos, rezultatul:

Tipul de date	Memorie ocupată	Valori posibile	
`short`	`2` octeți (16 biți)	$-2^{15}...2^{15}-1$	`-32768 .. 32767`
`unsigned short`	`2` octeți (16 biți)	$0...2^{16}-1$	`0..65535`
`int`	`4` octeți (32 biți)	$-2^{31}...2^{31}-1$	± `2` miliarde și `100` milioane (aprox.)
`unsigned int`	`4` octeți (32 biți)	$0...2^{32}-1$	`0` .. `4` miliarde și `200` milioane (aprox.)
`long long`	`8` octeți (64 biți)	$-2^{63}...2^{63}-1$	Numere cu semn cu până la `18` cifre
`unsigned long long`	`8` octeți (64 biți)	$0...2^{64}-1$	Numere pozitive cu pâna la `19` cifre

Observați că pentu tipurile de date mari am scris doar o valoare aproximativă garantată.

Indiferent care dontre tipurile de mai sus este cel folosit,, sunt valabile regulile de aplicare învățate pentru operatorii de date întregi.

Vom discuta pe larg despre modul de reprezentare internă a datelor în zonele de RAM ce le sunt rezervate la capitolul dedicat operatorilor pe biți.

Cele două tipuri reale pe care le vom utiliza sunt `float` și `double`. Despre datele reale este important de reținut că unele dintre ele sunt memorate cu aproximație. Iată și justificarea: datele de tip `float` ocupă patru octeți de mamorie iar cele de tip `double` ocupă `8` octeți. Valoarea `1.0/3` este una reală cu o infinitate de zecimale. Evident, că într-o cantitate finită de memorie nu poate fi memorată o astfel de valoare.
Diferența principală dintre cele două tipuri este, ca și la tipurile întregi, intervalul de valori posibile. Aici această diferență apare și la precizie, adică la numărul de zecimale care se poat stoca. Indiferent care este tipul de date real utilizat, se respectă regulile de aplicare a operatorilor la date reale, studiate anterior.

Înainte de a da câteva exemple care să fixeze cele spuse, vom extinde o definiție care a apărut în lecția despre operatori, anume cea de *expresie*.

Spuneam că o expresie este o construcție corectă în limbaj care are o valoare finală. Așadar, pot fi incluse aici și variabilele. Deci:
- o constantă este o expresie;
- o variabilă este o expresie;
- orice combinare corectă de operatori cu alte expresii este o expresie;

În exemplele următoare, considerăm că variabilelor declarate li s-au dat anterior valori (vom învăța chiar în această lecție că unul dintre modurile prin care putem face asta este citirea).

Fie declarațiile
`short a, b;`
`int c, d;`
`double e, f;`	

`a + b - f`	Este o expresie corectă, de tip real (double)
`c%d`	Este o expresie corectă, de tip întreg
`(a+d)/3`	Expresie corectă de tip întreg
`e/3`	Expresie corectă, de tip real
`e%3`	Incorect sintactic, se încearcă aplicarea operatorului `%` la un operand real

Atunci când un programator declară o variabilă, el trebuie să gândească așa: doresc să mi se rezerve în RAM, la executare, un loc în care să stochez o valoare. Și prin declarare obțin asta. Rămâne ca, prin ce scriu în continuare în program, să folosesc util acel spațiu alocat.

În exemplul de program de mai sus, apar punctele de suspensie în două locuri. În ambele locuri este permisă declararea de variabile. Diferența este că dacă le declarăm sus, acestea vor avea valoarea inițială `0`. Dacă le declarăm în interiorul funcției `main`, acestea nu vor fi neapărat `0` la început (ele au o valoare inițială, pentru că prin rezervarea lor a zonei de memorie, ele preiau valoarea care se formează cu configurația valorilor biților acelei zone).
Să revenim la structura programului, și anume la codul următor:

```
int main() {
    ...
}
```

Ne-am referit anterior la aceasta prin funcția `main`. Ce reprezintă în C++ termenul de funcție vom discuta mai târziu, pentru că necesită puțin intrarea în detalii și nu aici este momentul.

Zona dintre acoladele funcției `main` este locul în care se descriu pașii programului. Spuneam că acolo se pot declara variabile, dar că acestea se pot declara și înainte. Instrucțiunile se pot scrie însă numai în interior. Deci aici se descriu pașii programului. Acestea se vor executa strict în ordinea în care apar.

În această lecție vom discuta despre două dintre instrucțiunile limbajului: citirea și scrierea.

Instrucțiunea de citire

Este cea prin care se preiau datele de intrare.

Sintaxa:

```
cin >> variabila;
  sau
cin >> variabila1 >> variabila2;
```

`cin` este un cuvânt cheie. Este semnul că dorim să scriem o instrucțiune de citire. `>>` este separatorul, se pune înaintea fiecărei variabile care apare la citire (observăm că într-o singură instrucțiune putem cere o dată, două sau chiar mai multe date). La final este obligatoriu să se scrie catacterul `;` Observăm că drept argumente pentru `cin` avem doar variabile.

Modul de executare

Când se ajunge la o citire programul va aștepta până când utilizatorul va introduce date pentru fiecare variabilă, iar datele introduse se vor stoca, în ordine, în zona de memorie rezervată variabilei corespunzătoare.

În momentul introducerii, după fiecare valoare se poate tasta enter spațiu sau tab (acestea se mai numesc și caractere albe) dar la final trebuie tastat obligatoriu enter.

De exemplu, interpretarea executării celei de-a doua instrucțiuni de citire de mai sus este: introdu două numere întregi, iar primul se va memora în `variabila1` și al doilea în `variabila2`.

Iată câteva exemple:

Fie declarațiile
`int a, b;`
`double c, d;`	

`cin>>a;`	Corect, se așteaptă introducerea de la tastatură a unui număr întreg.
`cin>>a>>b;`	Corect, se așteaptă introducerea de la tastatură a două numere întregi.
`cin>>a;` `cin>>b;`	Corect, se așteaptă introducerea de la tastatură a două numere întregi. Deci, este vorba de aceași lucru ca mai sus, pentru că instrucțiunile se execută în ordinea în care apar.
`cin>>a>>c;`	Corect, se așteaptă introducerea a două numere, unul întreg și unul real.
`cin>>a>>b>>c>>d;`	Corect, se așteaptă introducerea a patru numere, primele două întregi și ultimele două reale.
`cin>>a>>1;`	Incorect sintactic, al doilea argument nu este o variabilă.
`cin>>a+b;`	Incorect sintactic, argumentul nu este o variabilă.

Instrucțiunea de scriere (de afișare)

Este cea prin care se tipăresc pe ecran rezultatele programului (datele de ieșire).

Sintaxa:

```
cout << expresie << "mesaj";
```

Observăm că de data asta `cout` este cuvântul "semn" că este vorba despre o afișare iar separatorul este `<<` (la `cin` era `>>`). Drept argumente nu mai este obligatoriu să avem variabile (ca la `cin`) ci orice produce o valoare (adică o expresie). De asemenea, putem tipări orice mesaj (dar trebuie scris între ghilimele). Vom vedea mai târziu că aceste mesaje sunt de fapt tot expresii și anume *constante șir de caractere*. La final trebuie pus caracterul `;` ca și în alte cazuri, ca sepataror de instrucțiuni și declarații (până acum am mai indicat nevoia de a pune `;` la `using namespace std`, declarații de variabile și citire).

Modul de executare

Se afișează pe ecran rezultatele expresiilor (deci mai întâi acestea se evaluează) și/sau conținuturile mesajelor (în ordinea în care apar).

La afișare trebuie să ne imaginăm că există pe ecran un cursor imaginar care indică locul unde se va face următoarea afișare, iar după afișare acest cursor este deplasat după data de ieșire afișată (pe același rând). Dacă dorim afișare pe rând nou nu este seficient să scriem o instrucțiune de afișare nouă ci trebuie să tipărim, parte a unui mesaj la `cout`, un caracter special ce reprezintă codificarea lui enter (`\n`, scris între ghilimele). În detaliu despre aceste coduri de caractere, la lecția dedicată tipului de date `char`.

Iată câteva exemple:

Fie declarațiile

`int a, b;` presupunem că, în momentul executării fiecărei bucăți de cod de mai jos, variabila `a` are valoarea `1` iar `b` are valoarea `2`.	
Secvență de cod	Ce se tipărește pe ecran
`cout<<a;`	`1`
`cout<<a<<b;`	`12`
`cout<<a;` `cout<<b;`	`12` Observăm că nu este nicio diferență față de rezultatul de la codul anterior.

`cout<<a<<"\n";`
`cout<<b;`
	`1`
	`2`
	Observăm că al doilea număr este pe rând nou întrucât între ele se tipărește și un enter.

`cout<<a<<" "<<b;`	`1 2` Observăm că între cele două numere s-a tipărit și un spațiu.
`cout<<a+b;`	`3` Se evaluează expresia argument al lui cout și se tipărește pe ecran rezultatul ei.
`cout<<"suma este"<<a+b;`	`suma este3`
`cout<<"suma este"<< a+b;`	`suma este3`
`cout<<"suma este "<<a+b;`	`suma este 3` Observăm că aici, înainte de `3` este un spațiu pentru că el a fost parte a mesajului. În exemplul anterior spațiul a fost pus doar între separator și expresia `a+b` și nu are efect pe ecran.

`cout<<"Suma este\n"<<a+b;`	`Suma este`
	`3`
	Observăm că putem scrie codul finalului de rând și ca parte a unui mesaj care conține și alte caractere.

`cout<<"a+b"<<a+b;`	`a+b3` Se tipărește mesajul `a+b` și apoi rezultatul expresiei `a+b`.
`cout>>a;`	Eroare de compilare pentru că nu s-a folosit separatorul >>
`Cout<<a;`	Eroare de compilare, C++ este *case sensitive* și cuvântul cout a fost scris cu inițială mare.

Citirea datelor de intrare din fișiere în loc de tastatură și scrierea datelor de ieșire în fișiere în loc de ecran.

Și aceasta este o secțiune pe care cei care sunt la primele lecții de info o pot sări. După ce se capătă ceva experiență (de exemplu după ce se lucrează câteva probleme inclusiv cu structuri repetitive) se recomandă revenirea.

La modul de lucru prezentat până acum datele de intrare se preiau de la tastatură iar rezultatele se scriu pe ecran. În acest fel la datele de ieșire avem acces doar imediat după terminarea programului, în fereastra de consolă (vezi secțiunea de mai jos din acest material, dedicată lucrului cu mediul Code::Blocks).

În programe complexe volumul de date de ieșire poate fi mare iar acestea trebuie memorate pentru a putea le folosi ulterior. Acest lucru este posibil prin scrierea acestor date în fișiere în loc de ecran.

De asemenea, programele complexe pot avea un volum mare de date de intrare iar acestea nu ar putea fi introduse de la tastatură ci trebuie preluate dintr-un fișier. Acest fisier poate fi chiar cel cu datele de ieșire ale altui program etc. Aici arătăm modalitatea extrem de simplă de a lucra cu fișiere de intrare/ieșire pornind de la lucrul cu consola (tastatură și monitor - despre care tocmai am vorbit).

Plecăm de la următoarea problemă simplă: Se citesc de la tastatură două numere de maxim `4` cifre. Să se afișeze pe ecran `3` valori: pe primul rând, separate prin spațiu, suma și diferența numerelor citite iar pe al doilea rând produsul lor. Iată mai jos soluția:

```
#include <iostream>
using namespace std;
int a, b;
int main () {
    cin>>a>>b;
    cout<<a+b<<" "<<a-b<<"\n";
    cout<<a*b;
    return 0;
}
```

Trecem acum la următorul enunț: Pentru problema anterioară cele două numere se află scrise pe un rând în fișierul `date.in` (sau fiecare pe câte un rând) și se cere calcularea acelorași rezultate, dar scrierea lor, în același mod, în fișierul `date.out`.

Soluția este următoarea (am scris în partea stângă codul prezentat mai sus și în partea dreaptă codul de la noul enunț):

```
#include <iostream>
using namespace std;
int a, b;


int main () {
    cin>>a>>b;
    cout<<a+b<<" "<<a-b<<"\n";
    cout<<a*b;
    return 0;
}
```

```
#include <fstream>
using namespace std;
int a, b;
ifstream fin ("date.in");
ofstream fout("date.out");
int main () {
    fin>>a>>b;
    fout<<a+b<<" "<<a-b<<"\n";
    fout<<a*b;
    return 0;
}
```

Pentru a rula programul din dreapta trebuie ca în prealabil să ne asigurăm că există fișierul `date.in`, în același folder cu sursa, și el conține cele două numere așa cum le vedem când le introducem de la tastatură. De exemplu putem crea fișierul absolut în același mod în care creăm în proiect fișierul sursă (vezi finalul acestui material, dedicat modului de lucru cu CodeBlocks).

În această problemă, la lansarea în executare nu ni se va cere nimic de la tastatură și nu vedem rezultate pe ecran. Va trebui să căutăm în același folder fișierul `date.out` și acolo vom găsi valorile tipărite de program. Dacă fișierele de intrare trebuie create înainte de executarea programului, cele de ieșire se creează automat în timpul rulării. Mai mult, dacă un fișier de ieșire deja există înainte de rulare, după noua executare a programului el se recreează, adică vechiul conținut este înlocuit cu datele scrise la executarea curentă a programului.

Să analizăm acum diferențele din cod. În primul rând se folosește biblioteca `fstream`. Celelalte două linii scrise bold au rolul de a asocia fișierele fizice de pe hard disk cu variabile de program. Linia `ifstream fin(”date.in”);` trebuie interpretată astfel: se declară o variabilă numită `fin`, de tip `ifstream` și la

declarare aceasta se asociază cu fișierul cu numele "`date.in`", care se află în același folder cu sursa `.cpp` și pe care îl va reprezenta în program.

Numele `fin` poate fi orice identificator corect dar noi am ales `fin` pentru a păstra o ușoară asociere cu ceea ce face `cin` când este vorba despre tastatură. Tipul `ifstream` este recunoscut în urma includerii bibliotecii `fstream` și semnificația sa este: *input file stream*. Din fișierele declarate astfel se poate doar citi. Observați că ulterior în program folosim `fin` ca și cum am fi folosit `cin`, dar datele se preiau din fișierul asociat cu `fin` în loc de tastatură.

Cele prezentate mai sus despre citire sunt valabile la afișare: `ofstream` (*output file stream*) este tipul de date folosit când vorbim despre fișierele în care doar vom scrie. Numele ales de noi, `fout`, se folosește ca și `cout` pentru ecran și rezultatele se scriu în fișierul asociat, absolut după aceleași reguli ca în cazul scrierii pe ecran. De exemplu, structura fișierului `date.out` va fi la terminarea executării programului următoarea: pe prima linie două numere separate prin spațiu iar pe a doua linie un număr (cu semnificația din enunț).

Un principiu corect de programare este să încheiem fișierul de intrare cu instrucțiunea `fin.close();` iar folosirea fișierului de ieșire cu `fout.close();`. Omiterea acestor instrucțiuni la scrierea programului folosind Code Blocks nu provoacă însă erori. Astfel, standard corect, programul de mai sus este:

```
#include <fstream>
using namespace std;
int a, b;
ifstream fin ("date.in");
ofstream fout("date.out");
int main () {
    fin>>a>>b;
    fout<<a+b<<" "<<a-b<<"\n";
    fout<<a*b;
    fin.close();
    fout.close();
    return 0;
}
```

Comentarii

Un alt element important al limbajului este reprezentat de comentarii. Acestea fac parte din fișierul sursă însă nu sunt luate în calcul la compilare. Sunt în general două motive pentru folosirea comentariilor:
- dorim să scriem o explicație despre o anumită bucată de cod, în scopul reamintirii cu ușurință a rolului acesteia.
- dorim să înlocuim o bucată de cod cu alta dar vrem să păstrăm cumva vechea variantă pentru a putea reveni la ea.

Sunt două tipuri de comentarii, *de linie* și *block*.

Comentariile de linie si introduc prin două caractere `//`. Ceea ce urmează pe linia curentă după ele nu este luat în calcul la compilare.

Comentariile block încep cu combinația `/*` și se termină cu `*/` Ele se pot întinde pe mai multe linii.

Iată un exemplu de program care ne cere să introducem două numere naturale și apoi afișează suma lor. Au fost inserate în cod comentarii din ambele tipuri.

```
#include <iostream>
```

```
using namespace std;
int a, b; // aici am declarat doua variabile
int main () {
    cin>>a>>b;
    /* mai sus este o instructiue de citire
    iar pe linia urmatoare este una de afisare
    */
    cout<<a+b;
}
```

Observație. Comentariul de rând poate fi introdus și prin trei caractere slash, consecutive.

Indentarea codului

În programul de mai jos se observă că instrucțiunile dintre acoladele funcției main sunt scrise cumva mai în interior. Ele sunt în fapt indentate cu un caracter tab. Vom vedea că este o regulă nescrisă de a structura codul astfel (scrierea cu un tab mai la dreapta pentru o bucată de cod subordonată cumva alteia), dar vom vorbi mai în detaliu despre asta odată cu prezentarea instrucțiunii `if`.

Modul de realizare a unui program C/C++ folosind mediul de dezvoltare Code::Blocks

Minimul necesar pentru a realiza un program în limbajul C/C++ este să avem pe calculator limbajul și un editor simplu de texte. Putem scrie sursa programului chiar în *Notepad*. Apoi, urmează etapa de analiză sintactică a sursei și, în caz că acest lucru este promovat (sursa respectă regulile de sintaxă) se trece la generarea codului executabil (fișierul cu extensia exe care nu mai este într-un format pe care să îl înțeleagă omul ci conține codificarea a ceea ce este necesar la executare). Pentru generarea codului executabil este suficient să se ruleze în linia de comandă a sistemului de operare (în Windows aceasta se poate obține prin utilitarul *cmd*) un fișier al limbajului care face acest lucru (de regulă *gcc.exe* sau *g++.exe*) care să aibă ca parametru fișierul sursă cu extensia `cpp`.

Acest mod de lucru este ceea ce au avut inițial la dispoziție programatorii, însă ulterior s-au dezvoltat utilitare care să realizeze totul într-un mod mai prietenos, lăsând programatorului doar grija dezvoltării sursei.

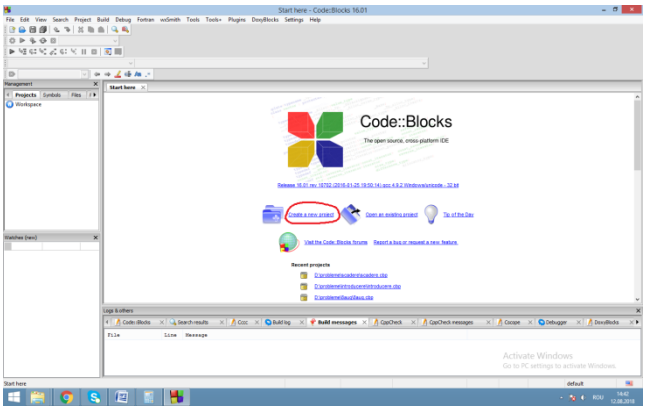
Code::Blocks pune la dispoziție un editor pentru scrierea sursei (care oferă diverse opțiuni de dezvoltare), generarea codului executabil prin simpla apăsare a unei taste, dar și posibilitate de depanare (rulare pas cu pas a programului și urmărirea în acest timp a valorilor variabilelor, în scopul depistării mai ușor a erorilor logice).

Presupunem acum că avem instalată o versiune recentă de Code::Blocks (exemplele sunt date folosind versiunea Code::Blocks 16.01).

Pasul 1.

Lansăm în executare Code::Blocks

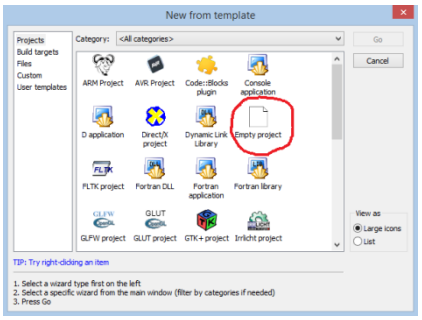
Interfața arată ca mai jos:



Pasul 2.

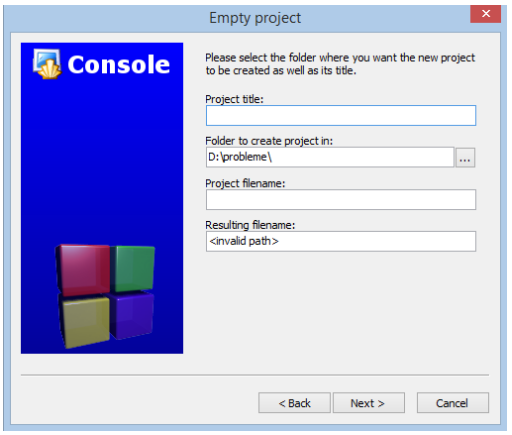
Selectăm *Create a new project*

Obținem:



Pasul 3.

Selectăm *Empty project*, apoi *Next* și obținem:



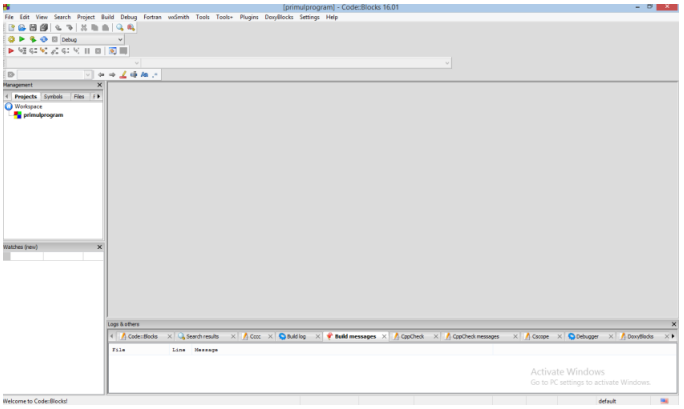
La prima rulare (imediat după instalare) câmpul *Folder to create project in* este vid și trebuie completat cu numele unui folder de pe hard diskul local unde doriți să se salveze programul curent. Odată stabilit acest folder, la rulările ulterioare va apărea implicit (în exemplu, pe calculatorul meu, acest folder este D:\probleme).

Câmpul *Project title* trebuie completat cu un nume care să respecte regulile de numire a folderelor din sistemul de operare pe care lucrați. Pentru început, programele ce le vom scrie au un singur fișier (sursa, fișierul cu extensia `c` sau `cpp`). Ulterior vom vedea că un program poate fi format din mai multe fișiere (alte fișiere `cpp`, fișiere cu date de unde să se preia datele de intrare sau unde să se scrie datele de ieșire), fișiere cu diverse resurse necesare programului (imagini, sunete) etc. Toate cele necesare unui program trebuie grupate în

același *proiect* și în acest câmp noi completăm numele proiectului. Pe hard disk, în folderul stabilit anterior se va vrea un altul, cu numele proiectului curent.

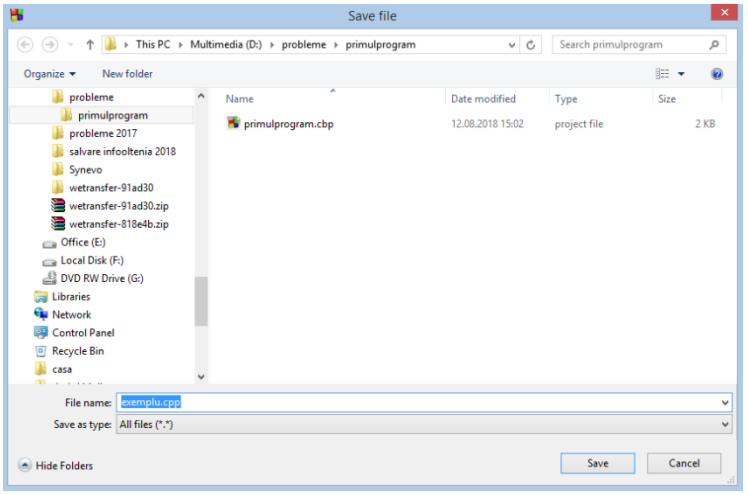
În acest exemplu eu voi crea un proiect cu numele *primulprogram*.

Se apasă *Next*, apoi *Finish* și se obține:



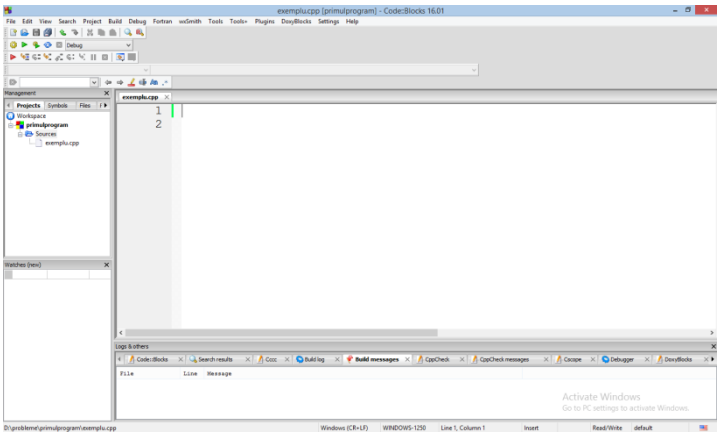
Pasul 4

Acum trebuie creat un fișier sursă în care să scriem programul. Pentru aceasta se selectează, în ordine, următoarele opțiuni: *File->New->Empty File->Yes*. Se obține o căsuță de dialog în care se va completa numele fișierului sursă.



Numele trebuie să respecte regulile de numire a fișierelor dar extensia trebui să fie obligatoriu `c` sau `cpp`.

Eu am ales numele `exemplu.cpp`. Se apasă *save*, apoi *ok* și acum proiectul este gata pentru a scrie sursa.

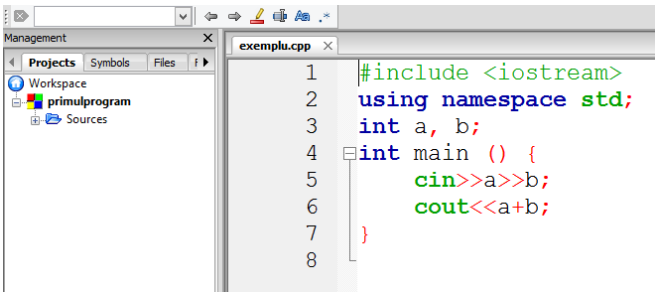


În fereastra din stânga-sus (cu numele *Management*) la eticheta *projects* trebuie să apară numele proiectului (așa cum l-ați stabilit, *primulprogram*) iar dacă desfacem lista de elemente ce i se subordonează (click pe + din dreptul lui) vom obține un subelement numit *Sources* iar acestuia trebuie să i se subordoneze fișierul cu numele stabilit (`exemplu.cpp`).

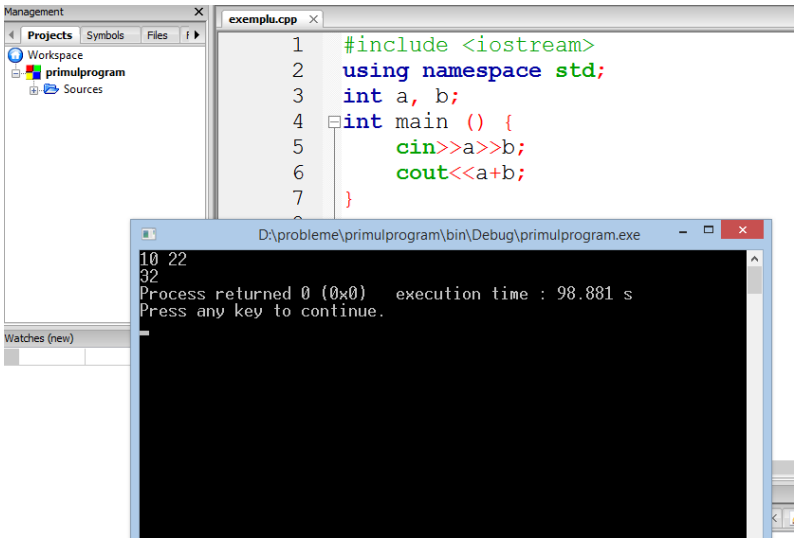
Editățm, în continuare următorul program:

```
#include <iostream>
using namespace std;
int a, b;
int main () {
     cin>>a>>b;
     cout<<a+b;
}
```

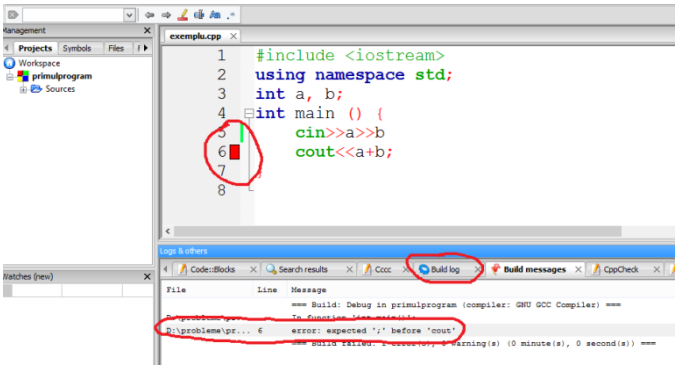
Interfața va arăta:



Programul de mai sus este unul corect sintactic, așa că ne așteptăm ca el să ruleze. Prin apăsare tastei *F9* (Echivalentă cu a selecta opțiunea *Build and Run* din meniul *Build*) programul va fi analizat și, fiind corect, va fi generat codul executabil și va fi lansat în executare. Apare așa zisa *fereastră de consolă,* cea care permite interacțiunea cu utilizatorul. După ce acesta introduce două numere întregi și apasă enter se va obține rezultatul tot în fereastra consolă. Iată:



Dacă la încercarea de rulare sursa nu ar fi fost scrisă corect din punct de vedere sintactic, ar fi apărut eroare de compilare semnalată atât printr-un indicator în editor în jurul locului cu prima eroare găsită cât și în fereastra Build Messages poziționată de regulă sub fereastra editor.



În acest caz programatorul trebuie să corecteze eroarea de sintaxă și apoi să încerce iarăși executarea. Se observă că locul în care este indicată eroarea în sursă este de regulă imediat după poziția erorii (aici este la începutul rândului ce urmează locului unde lipsește caracterul ;).

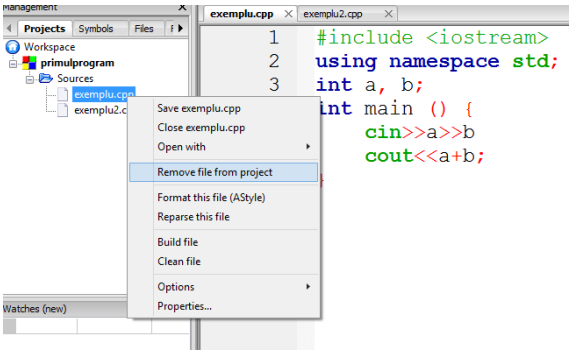
Dacă dorim să redeschidem un proiect la care am mai lucrat anterior, avem la dispoziție opțiunea *Open an existing project* din fereastra ce apare odată cu deschiderea Code::Blocks (apoi căutăm folderul proiectului și încărăm fișierul cu numele proiectului și extensia *cbp*). O altă variantă ar fi să urmărim dacă în partea de jos a aceleiași ferestre, la secțiunea *Recent projects* se află o opțiune cu proiectul care ne interesează.

Realizarea unui program nou se poate realiza în mai multe moduri:

- O primă varintă ar fi să închidem proiectul curent (Click dreapta pe numele proiectului în fereastra de *Management*, din stânga-sus și alegem *Close project*). Apoi putem urma pașii descriși pentru a crea un proiect nou.

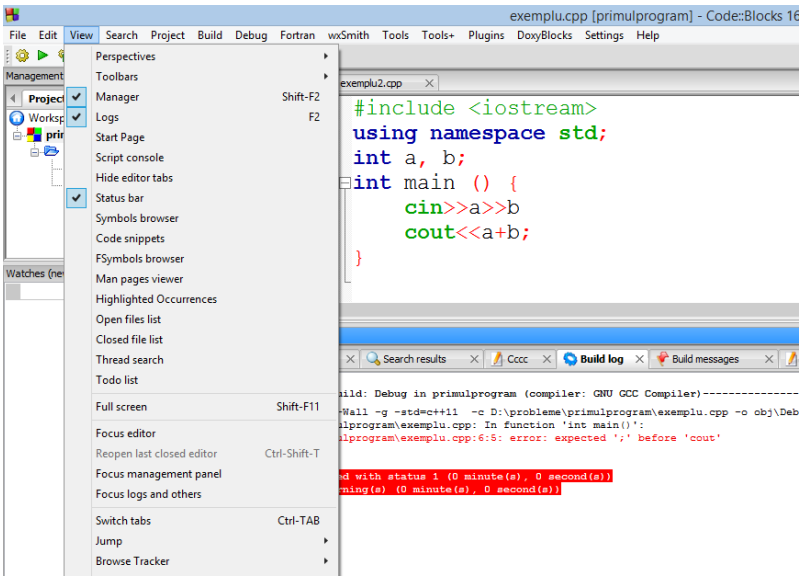
- O a doua variantă este să creăm un proiect nou lăsându-l deschis și pe cel curent. Pentru asta selectăm *File->New->Project* și urmăm pașii descriși mai înainte pentru a crea proiectul nou. În fereastra de *management* vor fi vizibile ambele, dar unul singur este cel curent (care se va rula dacă apăsăm *F9*, de exemplu) și acesta este afișat îngroșat.

- A treia variantă este să scriem un program nou în același proiect. Nu discutăm de păstrarea aceluiași fișier sursă căruia să îi rescriem conținutul, evident că este posibil acest lucru dar de regulă ne dorim să păstrăm și vechiul program. Așadar, în proiectul în care avem deja o sursă vom crea un al doilea fișier sursă. Să presupunem că îl numim `exemplu2.cpp` Acesta va apărea la secțiunea *sources* alături de cel scris inițial. Dacă scriem noul program în acesta, ne vom confrunta cu următoarea situație: Avem două fișiere care conțin funcția `main` în același proiect. Care se va executa ? Evident că este o situație ambiguă și nu se va crea fișierul executabil, acest lucru indicându-ni-se ca o eroare de compilare. Pentru aceasta avem posibilitatea de a elimina celălalt fișier din proiect. Click dreapta pe numele fișierului pe care dorim să îl eliminăm din proiect (în fereastra *Management*) și selectăm *Remove file from project*.



Efectul este că fișierul eliminat nu mai figurează în proiect, însă el rămâne pe hard disk în folderul proiectului și astfel dispunem în continuare de codul scris (chiar îl putem reinclude în proiect, pentru aceasta putem urma pașii: click dreapta pe numele proiectului și selectăm *Add files*, urmând să selectăm ceea ce dorim să includem la proiect).

În meniul *View* sunt câteva opțiuni utile pentru a ne configura modul în care arată interfața utilitarului Code::Blocks.



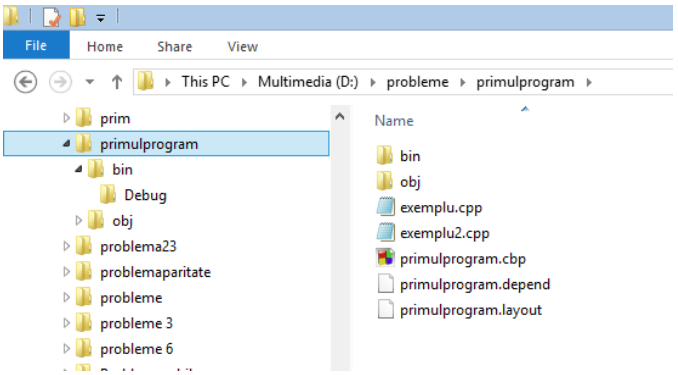
Observăm că implicit avem fereastra cu fișierul sursă în cea mai mare zonă a ecranului, fereastra de *Management* în colțul din stânga sus și ferestrele *Logs*, cele cu informații despre compilare, executare etc, în partea de jos. Dacă dorim să revenim la acest mod de organizare a interfeței Code::Blocks (în cazul în care din diverse motive aceasta s-a modificat), vom selecta *Perspectives->Code::Blocks default*.

Pentru a reafișa doar fereastra *Management* bifăm opțiunea corespunzătoare din meniul *View*. Pentru a reafișa ferestrela *Logs*, bifăm, de asemenea, opțiunea cprespunzătoare, așa cum se vede mai sus.

Pe parcurs vom reveni cu informații noi despre mediul Code::Blocks, inclusiv despre modul de lucru *pas cu pas*.

Spuneam că odată cu crearea unui proiect se crează și un folder cu diferite fișiere despre programul curent.

Iată un exemplu de structură a acestui folder.



Observăm că se găsesc în folder fișierele sursă scrise în cadrul acestui proiect. Este, de asemenea, fișierul `primulprogram.cbp`, cel pe care l-am selectat când am dorit să reîncărcăm ulterior creării, proiectul. Încă un lucru interesant: în folderul *bin*, se află un folder *debug*, iar în acesta un fișier numit `primulprogram.exe`. Acesta este executabilul programului generat din ultima sursă la care s-a apăsat `F9` și a reușit compilarea. Așadar, simplu dublu click pe acest fișier are ca efect rularea programului.

Probleme rezolvate

1. Scrieți un program care cere să introducem de la tastatură două numere și care afișează, pe un singur rând, separate prin câte un spațiu, suma, diferența și produsul numerelor. Cele două numere sunt naturale de maxim patru cifre.

```
#include <iostream>
using namespace std;
int a, b;
int main () {
    cin>>a>>b;
    cout<<a+b<<" "<<a-b<<" "<<a*b;
}
```

Fiind garantat că numerele date și cele obținute pe parcurs au maxim nouă cifre, tipul `int` este suficient pentru a memora atât valorile lor cât și ale rezultatelor.

2. Scrieți un program care cere să introducem de la tastatură un număr și afișează suma ultimelor două cifre ale sale (numărul dat are între două și nouă cifre).

```
#include <iostream>
using namespace std;
int n;
int main () {
    cin>>n;
    cout<<n%10 + n/10%10;
}
```

3. Scrieți un program care cere să introducem de la tastatură un număr și afișează produsul dintre prima și a treia cifră a sa considerând cifrele numerotate de la dreapta (numărul dat are între trei și nouă cifre).

```
#include <iostream>
using namespace std;
int n;
int main () {
    cin>>n;
    cout<<n%10 * (n/100%10);
}
```

Observăm plasarea între paranteze a subexpresiei care obține cifra zecilor, altfel, operatorii fiind toți de aceeași prioritate, s-ar fi evaluat de la stânga la dreapta și nu s-ar fi obținut rezultatul corect.

4. Scrieți un program care cere să introducem de la tastatură un număr și care afișează suma cifrelor impare ale sale. Se garantează că numărul introdus are exact două cifre.

```
#include <iostream>
using namespace std;
int n;
int main () {
    cin>>n;
    cout<<(n%10)*(n%2) + n/10 * (n/10%2);
}
```

Observăm că dacă `c` este o variabilă care memorează un număr de o cifră, expresia `c%2` are valoarea `0` dacă este pară cifra și respectiv `1` în caz că este impară cifra.

5. Scrieți un program care cere să introducem de la tastatură două numere întregi și care afișează media aritmetică a lor. Se garantează că numerele introduse au maxim nouă cifre.

```
#include <iostream>
```

```
using namespace std;
int a, b;
int main () {
    cin>>a>>b;
    cout<<(a+b)/2.0;
}
```

Observăm că rezultatul poate fi real (când suma numerelor este impară) și atunci este obligatoriu să scriem constanta `2` ca pe una reală, altfel operatorul `/` între două valori întregi ar fi trunchiat rezultatul.

6. Scrieți un program care cere să introducem de la tastatură un număr natural și care afișează valoarea pătratului ultimei cifre precedată de mesajul "`Rezultatul este:`"

```
#include <iostream>
using namespace std;
int a;
int main () {
    cin>>a;
    cout<<"Rezultatul este:"<<(n%10) *
(n%10);
}
```

Temă

Întrebări și exerciții

1. Ce este un *tip de date*?

2. Precizați două tipuri de date reale și trei tipuri de date înregi.

3. Ce este o *variabilă*?

4. Care este sintaxa instrucțiunii de citire ?

5. Care este modul de executare a instrucțiunii de citire ?

6. Care este sintaxa instrucțiunii de afișare ?

7. Care este modul de executare a instrucțiunii de afișare ?

8. Ce este un identificator? Ce reguli trebuie respectate când scriem un identificator ?

9. Ce reprezintă comentariile în limbajul de programare C/C++?

10. Indicați două modalități de a scrie comentarii.

Probleme

1. Scrieți un program care citește un număr de trei cifre și afișează suma cifrelor sale.

Exemplu:

Date de intrare	Date de ieșire
304	7

2. Scrieți un program care citește un număr de trei cifre și afișează media aritmetică a cifrelor sale.

Date de intrare	Date de ieșire
103	1.333333

3. Scrieți un program care citește două numere reale `a` și `b` ce reprezintă coeficienții unei ecuații de gradul I de forma `ax+b = 0`. Se garantează că `a` va fi dat nenul. Afișați soluția acestei ecuații.

Date de intrare	Date de ieșire
4 8	-2

4. Scrieți un program care cere două numere ce reprezintă lungimea și lățimea unui dreptunghi și care afișează valoarea perimetrului dreptunghiului, apoi, pe rândul următor valoarea ariei.

Date de intrare	Date de ieșire
2 7	18 14

5. Scrieți un program care cere să introducem numărul de secunde în care un mobil a parcurs distanța de `1Km` și care afișează viteza medie (în metri pe secundă) cu care s-a deplasat mobilul.

Date de intrare	Date de ieșire
120	8.333333