1. C şi C++ un tur de orizont.

1.1. Structura unui program C foarte simplu

Un *limbaj de programare* reprezintă o interfaţă între *problema de rezolvat* şi *programul de rezolvare.*

Limbajul de programare, prin specificarea unor acţiuni care trebuie executate *eficient* este *apropiat de maşină*. Pe de altă parte, el trebuie să fie *apropiat de problema de rezolvat*, astfel încât soluţia problemei să fie exprimată *direct* şi *concis*.

Trecerea de la specificarea problemei la program nu este directă, ci presupune parcurgerea mai multor *etape*:

- ***analiza şi abstractizarea problemei***. In această etapă se identifică *obiectele* implicate în rezolvare şi acţiunile de transformare corespunzătoare. Ca rezultat al acestei etape se crează un *univers abstract al problemei* (UP), care evidenţiază o mulţime de tipuri de obiecte, relaţiile dintre acestea şi restricţiile de prelucrare necesare rezolvării problemei.
- ***găsirea metodei de rezolvare*** acceptabile, precizând operatorii de prelucrare ai obiectelor din UP.
- ***elaborarea algoritmului de rezolvare***
- ***codificarea algoritmului***

Limbajul C s-a impus în elaborarea programelor datorită:

- uşurinţei de reprezentare a obiectelor cu caracter nenumeric
- capacităţii de reprezentare a obiectelor dinamice
- capacităţii de exploatare a caracteristicilor maşinii de calcul pentru controlul strict al performanţelor programului
- asigurării unei interfeţe transparente cu sistemul de operare al maşinii utilizate.

Limbajul C a fost creat de Dennis Ritchie şi Brian Kernighan şi implementat pe o maşina DEC PDP 11, cu intenţia înlocuirii limbajului de asamblare.. Limbajul are precursori direcţi limbajele BCPL (Richards) şi B (Thompson). Limbajul este folosit ca mediu de programare pentru sistemul de operare UNIX. Limbajul a fost standardizat în 1983 şi 1989.

Limbajul C++ a fost dezvoltat de Bjarne Stroustrup pornind de la limbajul C, începând din anul 1980.

C++ împrumută din Simula 67 *conceptul de clasă* şi din limbajul Algol 68 - *supraîncărcarea operatorilor*.

Dintre noutăţile introduse de C++ menţionăm: *moştenirea multiplă, funcţiile membre statice şi funcţiile membre constante, şabloanele, tratarea excepţiilor, identificarea tipurilor la execuţie, spaţiile de nume, etc*.

Deşi C++ este considerat o extensie a limbajului C, cele două limbaje se bazează pe *paradigme de programare* diferite. Limbajul C foloseşte *paradigma programării procedurale şi structurate*. Conform acesteia, un program este privit ca o mulţime ierarhică de blocuri şi proceduri (funcţii). Limbajul C++ foloseşte *paradigma programării orientate pe obiecte*, potrivit căreia un program este constituit dintr-o mulţime de obiecte care interacţionează.

Elementul constructiv al unui program C este *funcţia*. Un program este constituit dintr-o mulţime de funcţii, declarate pe un singur nivel (fără a se imbrica unele în altele), grupate în *module program*.

O *funcţie* este o secţiune de program, identificată printr-un nume şi parametrizată, construită folosind declaraţii, definiţii şi instrucţiuni de prelucrare. Atunci când este apelată, funcţia calculează un anumit rezultat sau realizează un anumit efect.

Funcţia **`main()`** este prezentă în orice program C. Execuţia programului începe cu **`main()`**. Funcţia **`main()`** poate întoarce un rezultat întreg (**`int`**) sau nici un rezultat (**`void`**). Numai în C este posibil să nu specificăm tipul rezultatului întors de funcţie, acesta fiind considerat în mod implicit **`int`**.

```
/* program C pentru afisarea unui mesaj */
#include <stdio.h>
main(){
  printf("Acesta este primul program in C /n");
}
```

Programul foloseşte un *comentariu*, delimitat prin **/*** şi ***/** care, prin explicaţii în limbaj natural, creşte claritatea programului. Comentariul este constituit dintr-o linie sau mai multe linii, sau poate apare în interiorul unei linii. Nu se pot include comentarii în interiorul altor comentarii.

În C++ se utilizează comentarii care încep cu **//** şi se termină prin sfârşitul de linie.

Linia **#include <stdio.h>** anunţă compilatorul că trebuie să insereze *fişierul antet* **stdio.h**. Acest fişier conţine prototipurile unei serii de funcţii de intrare şi ieşire folosite de majoritatea programelor C. Fişierele antet au prin convenţie extensia .h. Fişierul de inclus este căutat într-o zonă standard de includere, în care sunt memorate fişierele antet ale compilatorului C, dacă numele este încadrat între paranteze unghiulare (**<** şi **>**), sau căutarea se face în zona curentă de lucru, dacă fişierul este încadrat între ghilimele("). Fişierele antet sunt foarte utile în cazul funcţiilor standard de bibliotecă; fiecare categorie de funcţie standard are propriul fişier antet.

```
/* program C pentru afisarea unui mesaj */
#include <stdio.h>
int main(){
  printf("Acesta este primul program in C /n");
  return 0;
}
```

Valoarea întoarsă de funcţia **main()** este în mod obişnuit 0, având semnificaţia că nu au fost întâlnite erori, şi se asigură prin instrucţiunea **return 0**.

Instrucţiunea **printf()** serveşte pentru afişarea la terminal (pe ecran) a unor valori *formatate*.

Faţă de limbajul C, care este considerat un subset, limbajul C++ permite: *abstractizarea datelor, programarea orientată pe obiecte şi programarea generică.*

```
// program C++ pentru afisarea unui mesaj
#include <iostream.h>
void main(void){
  cout << "Primul program C++ \n";
}
```

1.2. Câteva elemente necesare scrierii unor programe C/C++ foarte simple.

1.2.1. Directiva define.

Directiva **#define nume text** este o *macrodefiniţie*. Prelucrarea acesteia, numită *macroexpandare*, înlocuieşte fiecare apariţie a numelui prin textul asociat.

O aplicaţie o reprezintă creerea de *constante simbolice*. De exemplu:

```
#define  PI     3.14159
#define  mesaj  "Bonjour madame"
#define  MAX    100
```

O constantă simbolică astfel definită nu poate fi redefinită prin atribuire.

1.2.2. Tipuri.

Fiecărui nume i se asociază un tip, care determină ce operaţii se pot aplica acelui nume şi cum sunt interpretate acestea. De exemplu:

```
char c='a';     // c este o variabilă caracter initializata cu 'a'
int f(double); //f este o functie de argument real cu rezultat intreg
```

1.2.3. Definiţii şi declaraţii de variabile,

O valoare *constantă* se reprezintă textual (este un *literal*) sau printr-un nume - *constantă simbolică.*

O *variabilă* este un nume (identificator) care desemnează o locaţie de memorie în care se păstrează o valoare.

O variabilă se caracterizează aşadar prin: *nume (adresă)*, *tip* şi *valoare*, atributul valoare putând fi modificat. De exemplu:

```
int   n, p;
char  c;
float eps;
```

O variabilă poate fi *iniţializată* la declararea ei. De exemplu:

```
float eps=1.0e-6;
```

Iniţializarea se face numai o dată, înaintea execuţiei programului.
Variabilele externe şi statice sunt iniţializate implicit la 0.
Pentru o variabilă automatică (declarată în interiorul unui bloc), pentru care există iniţializare explicită, aceasta este realizată la fiecare intrare în blocul care o conţine.

În C++ calificatorul **`const`** aplicat unui nume simbolic arată că acesta nu mai poate fi modificat pe parcursul programului şi reprezintă o constantă. Definirea unei constante presupune şi iniţializarea acesteia.

```
const float pi=3.1415926;
```

O *definiţie* este o construcţie textuală care asociază unui nume o zonă de memorie (un obiect) şi eventual iniţializează conţinutul zonei cu o valoare corespunzătoare tipului asociat numelui.

1.2.4. Atribuirea.

Atribuirea simplă este de forma:**`variabilă = expresie`** şi are ca efect modificarea valorii unei variabile.

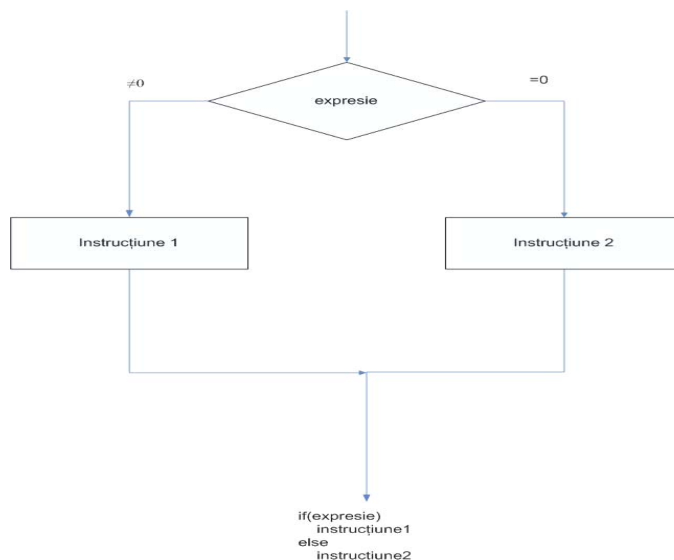
Atribuirea compusă **`a op= b`** reprezintă într-o formă compactă operaţia **`a = a op b`**

Atribuirea multiplă este de forma **`variabilă1 = variabilă2 = … = expresie`** şi iniţializează variabilele, pornind de la dreapta spre stânga cu valoarea expresiei.

Operatorii de incrementare folosiţi în atribuiri au efecte diferite. Astfel:
`a = ++b` este echivalentă cu **`b=b+1; a=b;`** în timp ce:
`a = b++` are ca efect **`a=b; b=b+1;`**

1.2.5. Decizia.

Permite alegerea între două alternative, în funcţie de valoarea (diferită de 0 sau 0) a unei expresii:




```
if (expresie)
   instructiune1;
else
   instructiune2;
```

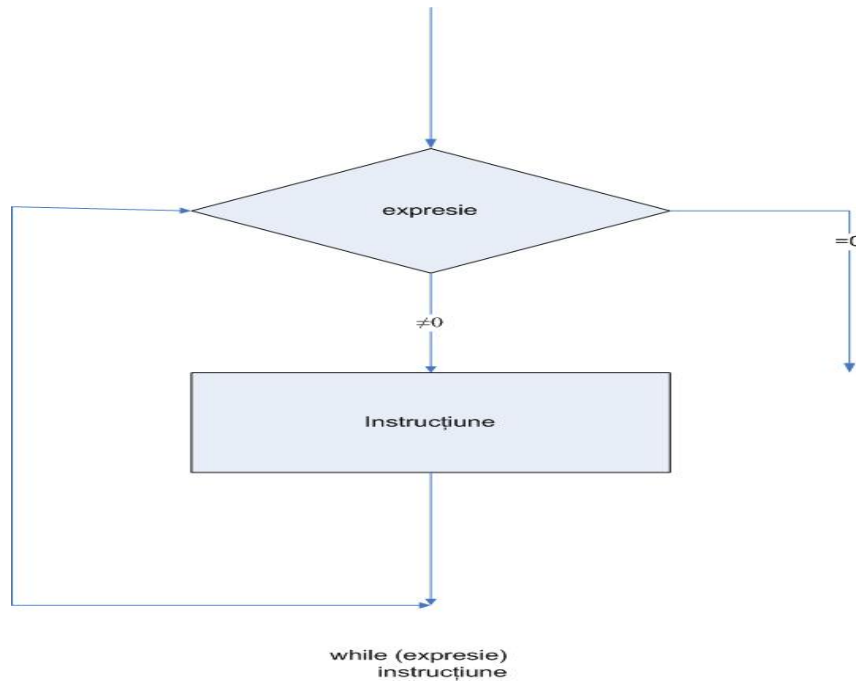
De exemplu:

```
if (a > b)
   max=a;
else
   max=b;
```

1.2.6. Ciclul.

Execută în mod repetat instrucțiunea, cât timp condiția este îndeplinită.

```
while (expresie)
   instructiune;
```



while (expresie)
instrucţiune

De exemplu calculul celui mai mare divizor comun se realizează cu:

```
while (a!=b)
   if (a > b)
      a -=b;
   else
      b -=a;
```

1.2.7. Afişarea valorii unei expresii (descriptori),

Pentru afişarea unei valori la terminal, sub controlul unui format se foloseşte funcția:

```
printf(lista_descriptori, lista_expresii);
```

De exemplu: `printf("pi=%.5f\n", M_PI);`

1.2.8. Citirea valorilor de la terminal.

Pentru citirea unor valori introduse de la terminal, sub controlul unui format se foloseşte funcția:

```
scanf(lista_descriptori, lista_adrese);
```

De exemplu: `scanf("%d%d/n", &a, &b);`

1.3. Structura unui program.

În C nu există noțiunea de program ci aceea de *subprogram funcție*.
Compilatorul recunoaşte noțiunea de *modul* - un ansamblu de variabile şi de funcții.
Un program în C este o funcție cu numele `main()` care se execută întotdeauna prima.
Cea mai simplă structură de program este un modul compus dintr-o singură funcție `main()`:

```
void main (){
    <declaratii locale>
    <instructiuni>
}
```

2. Elementele fundamentale ale limbajului (C / C++).

2.1. Alfabetul limbajului.

Conține setul de caractere ASCII (setul extins 256 caractere).

2.2. Atomi lexicali.

Există următoarele entități lexicale: identificatori, cuvinte cheie, constante, şiruri de caractere, operatori şi separatori.

Spațiile albe în C sunt reprezentate de: spațiu liber (blanc), tabulare orizontală, tabulare verticală, linie nouă, pagină nouă, comentarii. Spațiile albe separă atomii lexicali vecini.

2.2.1. Identificatori.

Identificatorii servesc pentru numirea constantelor simbolice, variabilelor, tipurilor şi funcțiilor.

Sunt formați dintr-o literă, urmată eventual de litere sau cifre. Caracterul de subliniere _ este considerat literă.

Intr-un identificator literele mari si cele mici sunt distincte. Astfel **Abc** si **abc** sunt identificatori diferiți.

Identificatorii pot avea orice lungime, dar numai primele 32 caractere sunt semnificative.

2.2.2. Cuvinte cheie.

Cuvintele cheie sau cuvintele rezervate nu pot fi folosite ca identificatori. Acestea sunt:

```
auto      break     case      char      const     continue
default   do        double    else      enum      extern
float     for       if        int       long      register
return    short     signed    sizeof    static    struct
switch    typedef   union     unsigned  void      volatile
while
```

2.2.3. Literali.

Un literal este o reprezentare explicită a unei valori. Literalii pot fi de mai multe tipuri tipuri: întregi, caractere, reali, enumerări sau şiruri de caractere..

2.2.4 Şiruri de caractere.

Conțin caractere încadrate între ghilimele. În interiorul unui şir ghilimelele pot fi reprezentate prin \” .Un şir de caractere se poate întinde pe mai multe linii, cu condiția ca fiecare sfârşit de linie să fie precedat de \.

Şirurile de caractere în C se termină cu *caracterul nul* `'\0'`. În cazul literalelor şiruri de caractere, compilatorul adaugă în mod automat la sfârşitul şirului caracterul nul.

2.2.5. Comentarii.

Un comentariu începe prin **/*** , se termină prin ***/** şi se poate întinde pe mai multe linii. Comentariile nu pot fi incluse unele în altele (imbricate).

În C++ au fost introduse comentariile pe o singură linie, care încep prin **//** şi au terminator sfârşitul liniei.

2.2.6. Terminatorul de instrucțiune.

Caracterul **;** este folosit ca terminator pentru instrucțiuni şi declarații, cu o singură excepție – după o instrucțiune compusă, terminată prin acoladă, nu mai este necesar terminatorul **;** .

2.2.7. Constante.

Constantele identificatori se obțin folosind directiva `#define` a preprocesorului:

```
#define constantă-identificator      literal sau constantă-identificator
#define constantă-identificator      (expresie-constantă)
```

2.3. **Ciclul de dezvoltare al unui program**

Conține următoarele etape:

1. Definirea problemei de rezolvat.

Analiza problemei cuprinde în afara formulării problemei în limbaj natural, o precizare riguroasă a intrărilor (datelor problemei) şi a ieşirilor (rezultatelor).

De exemplu ne propunem să rezolvăm ecuația de gradul 2: $ax^2+bx+c=0$

Datele de intrare sunt cei 3 coeficienți **a, b, c** ai ecuației, care precizează o anumită ecuație de grad 2.

Rezultatele sunt cele două rădăcini (reale sau complexe) sau celelalte situații particulare care pot apare.

2. Identificarea paşilor necesari pentru rezolvarea problemei începe cu formularea ***modelului matematic*.**

Ca model matematic vom folosi formula:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Formula nu este întotdeauna aplicabilă. Vom distinge următoarele situații:

1. **a=0**, caz în care nu avem de a face cu o ecuație de gradul 2, ci
   - 1.1. este posibil să avem ecuația de gradul 1: **bx+c=0**, cu soluția **x= - c/b** , dacă $b \neq 0$.
   - 1.2. dacă şi **b=0**, atunci
     - 1.2.1. dacă $c \neq 0$, atunci nu avem nici o soluție, în timp ce
     - 1.2.2. dacă şi **c=0**, atunci avem o infinitate de soluții.
2. $a \neq 0$ corespunde ecuației de gradul 2. In acest caz avem alte două situații:
   - 2.1. Formula este aplicabilă pentru rădăcini reale (discriminant pozitiv)
   - 2.2. Pentru discriminant negativ, întrucât nu dispunem de aritmetică complexă, va trebui să efectuăm separat calculele pentru partea reală şi cea imaginară.

3. Proiectarea algoritmului folosind ca instrument **pseudocodul.**

Pseudocodul folosit cuprinde:

- operații de intrare / ieşire:

```
citeşte var1, var2,…
scrie   expresie1, expresie2,…
```

- structuri de control:
  - decizia:

```
dacă expresie atunci
  instrucţiune1;
altfel
  instrucţiune2;
```

  - ciclul:

```
cât timp expresie repetă
  instrucţiune;
```

  - secvența:

```
{ instrucţiune1;
  ...
  instrucţiunen;
}
```

Algoritmul dezvoltat pe baza acestui pseudocod este:

```
reali a,b,c,delta,x1,x2,xr,xi;
citeşte a,b,c;
dacă a=0 atunci
  dacă b≠0 atunci
    scrie –c/b;
  altfel
    dacă c≠0 atunci
       scrie "nu avem nici o solutie";
    altfel
       scrie "o infinitate de solutii";
  altfel
    { delta=b*b-4*a*c;
      dacă delta >= 0 atunci {
         x1=(-b-sqrt(delta))/(2*a);
         x2=(-b+sqrt(delta))/(2*a);
         scrie x1,x2;
       }
       altfel {
         xr=-b/(2*a);
         xi=sqrt(-delta)/(2*a);
         scrie xr,xi;
    }
}
```

4. Scrierea programului folosind un limbaj de programare.

Vom codifica algoritmul descris mai sus folosind limbajul C:

```
#include <stdio.h>
#include <math.h>
void main(void)
{ double a,b,c,delta,x1,x2,xr,xi;
  scanf("%f %f %f",&a,&b,&c);
  if (a==0)
    if (b!=0)
      printf("o singura radacina x=%6.2f\n",-c/b);
    else
      if (c!=0)
        printf("nici o solutie\n");
      else
        printf("o infinitate de solutii\n");
  else
    { delta=b*b-4*a*c;
      if(delta >= 0)
        { x1=(-b-sqrt(delta))/2/a;
          x2=(-b+sqrt(delta))/2/a;
          printf("x1=%5.2f\tx2=%5.2f\n",x1,x2);
        }
      else
        { xr=-b/2/a;
          xi=sqrt(-delta)/2/a;
          printf("x1=%5.2f+i*%5.2f\nx2=%5.2f-i*%5.2f\n",
                  xr,xi,xr,xi);
        }
     }
}
```

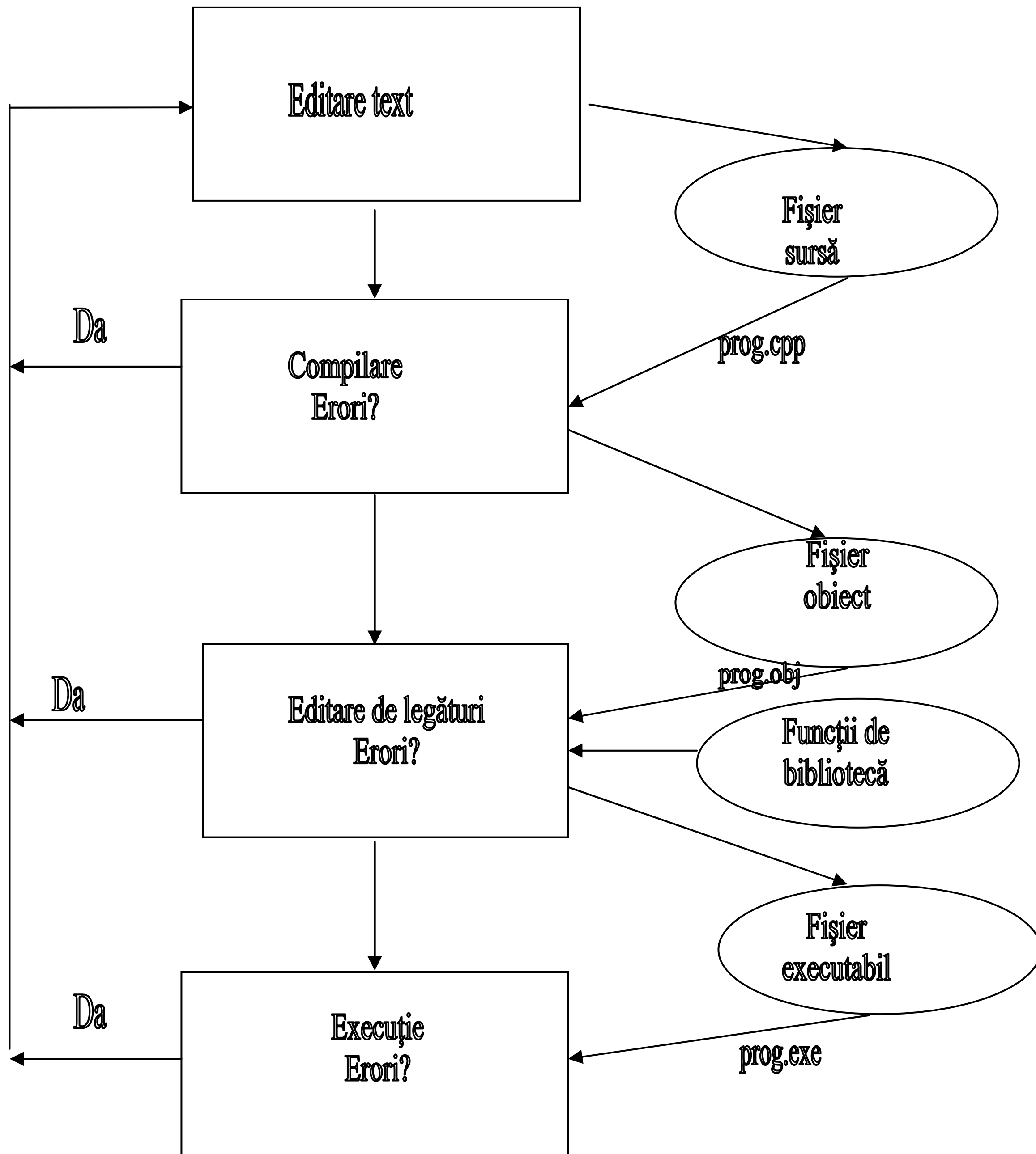


Fig.2.1. Etapele rezolvării unei probleme folosind calculatorul

5. Implementarea programului: editare, compilare, editare de legături, execuţie.

Un *mediu de programare* C conţine:

- *Un editor de text* – folosit pentru creearea şi modificarea codului sursă C
- *Un compilator* – pentru a converti programul C în cod înțeles de calculator.

Procesul de compilare cuprinde:

- o fază de *precompilare* (*macroprocesare*) în care are loc *expandarea macrodefinițiilor* si *compilarea condițională si includerea fisierelor*
- *compilarea propriu-zisă* în urma careia se generează cod obiect

Compilarea din linia de comandă în GCC se face cu:

`gcc -c prog.c` prog.c→prog.o

- *Editor de legături* –leagă la codul obiect anumite *funcții de bibliotecă* (de exemplu intrări/ieşiri) extrase dintr-o bibliotecă de funcții, creindu-se un *program executabil.*

`gcc -o prog prog.o` prog.o→prog.exe

Compilarea şi editarea de legături se poate face printr-o singură comandă:

`gcc -o prog prog.c` prog.c→prog.exe

Opțiunea `-o` permite specificarea fişierului de ieşire. Dacă acesta lipseşte, se consideră în mod implicit ca nume al fişierului executabil `a.out`, Aşadar:

`gcc prog.c` prog.c→a.out

- *Fişiere biblioteci de funcții* –sunt fişiere de funcții gata compilatecare se adaugă (se leagă) la program. Există biblioteci pentru: funcții matematice, şiruri de caractere, intrări/ieşiri.

Biblioteca standard C la execuție (run-time) este adăugată în mod automat la fiecare program; celelalte biblioteci trebuiesc legate în mod explicit.

De exemplu, în GNU C++ pentru a include biblioteca matematică `libm.so` se foloseşte opțiunea –lm:

`gcc -o prog prog.c -lm`

Bibliotecile pot fi:

- *statice* (`.lib` în BorlandC, `.a` în GCC) –codul întregii biblioteci este ataşat programului
- *dinamice* (`.dll` în BorlandC, `.so` în GCC) –programului I se ataşează numai funcțiile pe care acesta le solicită din bibliotecă.

În GCC se leagă în mod implicit versiunea dinamică a bibliotecii; pentru a lega versiunea staticăse foloseşte opțiunea `-static.`

- *Fişiere antet* (*header files*)–datele şi funcțiile conținute în biblioteci sunt declarate în fişiere antet asociate bibliotecilor. Prin includerea unui fişier antet compilatorul poate verifica corectitudinea apelurilor de funcții din biblioteca de funcții asociată (fără ca aceste funcții să fie disponibile în momemtul compilării).

Bibliotecă	Fişier antet
matematică	`math.h`
Siruri de caractere	`string.h`
Intrări/ieşiri	`stdio.h`

Un fişier antet este inclus printr-o directivă cu sintaxa:

```
#include <nume.h>
```

Aceasta caută fişierul antet într-un director special de fişiere incluse (include).

```
#include "nume.h"
```

Caută fişierul antet în directorul current.

Încărcarea şi execuția programului (fişierului executabil .exe) se vface în GCC prin:

```
./ prog
```

6. Depanarea programului (debugging).

Depanarea unui program reprezintă localizarea şi înlăturarea erorilor acestuia.

Cele mai frecvente *erori de sintaxă* se referă la: lipsa terminatorului de instrucţiune ;, neechilibrarea parantezelor, neînchiderea şirurilor de caractere, etc.

Erorile detectate de către *editorul de legături* sunt *referinţele nerezolvate* (apelarea unor funcţii care nu au fost definite, sau care se află în biblioteci care nu au fost incluse).

Cele mai des întâlnite erori la execuţie provin din:

- confuzia între = şi ==
- depăşirea limitelor memoriei allocate tablourilor
- variabile neiniţializate
- erori matematice: împărţire prin 0, depăşire, radical dintr-un număr negative, etc.

Dacă s-a depăşit faza erorilor la execuţie, vom testa programul, furnizându-i date de intrare pentru care cunoaştem ieşirile. Apariţia unor neconcordanţe indică prezenţa unor *erori logice*. Dacă însă, rezultatele sunt corecte, nu avem certitudinea că programul funcţionează corect în toate situaţiile. Prin testare putem constata prezenţa erorilor, nu însă şi absenţa lor.

Desfăşurarea calculelor poate fi controlată *prin execuţie pas cu pas*,sau prin asigurarea unor *puncte de întrerupere* în care să inspectăm starea programului, folosind în acest scop un *depanator* (debugger).

2.4. Întrebări şi probleme.

1. Ce deosebire există între **`'A'`** şi **`"A"`** ?
2. Ce caractere pot fi reprezentate prin secvenţele escape: **`/101`**, **`/012`**.
3. Reprezentaţi şirurile de caractere **`'`** şi **`"`**.