CONŢINUTURI Bacalaureat

1. Algoritmi

1.1. Noţiunea de algoritm, caracteristici

1.2. Date, variabile, expresii, operaţii

1.3. Structuri de bază (liniară, alternativă şi repetitivă)

1.4. Descrierea algoritmilor (programe pseudocod)

2. Elementele de bază ale unui limbaj de programare (Pascal sau C, la alegere)

2.1. Vocabularul limbajului

2.2. Constante. Identificatori

2.3. Noţiunea de tip de dată. Operatori aritmetici, logici, relaţionali

2.4. Definirea tipurilor de date

2.5. Variabile. Declararea variabilelor

2.6. Definirea constantelor

2.7. Structura programelor. Comentarii

2.8. Expresii. Instrucţiunea de atribuire

2.9. Citirea/scrierea datelor

2.10. Structuri de control (instrucţiunea compusă, structuri alternative şi repetitive)

3. Subprograme predefinite

3.1. Subprograme. Mecanisme de transfer prin intermediul parametrilor

3.2. Proceduri şi funcţii predefinite

4. Tipuri structurate de date //(tipul pointer)

4.1. Tipul tablou

4.2. Tipul şir de caractere

– operatori, proceduri şi funcţii predefinite pentru: citire, afişare, concatenare, căutare, extragere, inserare, eliminare şi conversii (şir □ valoare numerică)

4.3. Tipul înregistrare

5. Fişiere text

5.1. Fişiere text. Tipuri de acces

5.2. Proceduri şi funcţii predefinite pentru fişiere text

6. Algoritmi elementari

6.1. Probleme care operează asupra cifrelor unui număr

6.2. Divizibilitate. Numere prime. Algoritmul lui Euclid

6.3. Şirul lui Fibonacci. Calculul unor sume cu termenul general dat

6.4. Determinare minim/maxim

6.5. Metode de ordonare (metoda bulelor, inserţiei, selecţiei, numărării)

6.6. Interclasare

6.7. Metode de căutare (secvenţială, binară)

6.8. Analiza complexităţii unui algoritm (considerând criteriile de eficienţă durata de executare şi spaţiu de memorie utilizat)

7. Subprograme definite de utilizator

7.1. Proceduri şi funcţii

– declarare şi apel

– parametri formali şi parametri efectivi

– parametri transmişi prin valoare, parametri transmişi prin referinţă

– variabile globale şi variabile locale, domeniu de vizibilitate

7.2. Proiectarea modulară a rezolvării unei probleme

8. Recursivitate

8.1. Prezentare generală
8.2. Proceduri şi funcţii recursive
9. Metoda backtracking (iterativă sau recursivă)
9.1. Prezentare generală
9.2. Probleme de generare. Oportunitatea utilizării metodei backtracking
10. Generarea elementelor combinatoriale
10.1. Permutări, aranjamente, combinări
10.2. Produs cartezian, submulţimi
11. Grafuri
11.1. Grafuri neorientate
– terminologie (nod/vârf, muchie, adiacenţă, incidență, grad, lanț, lanț elementar, ciclu, ciclu elementar, lungime, subgraf, graf parţial)
– proprietăţi (conex, componentă conexă, graf complet, hamiltonian, eulerian)
– metode de reprezentare (matrice de adiacenţă, liste de adiacenţă)
11.2. Grafuri orientate
– terminologie (nod/vârf, arc, adiacenţă, incidenţă, grad intern şi extern, drum, drum elementar, circuit, circuit elementar, lungime, subgraf, graf parţial)
– proprietăţi (tare conexitate, componentă tare conexă)
– metode de reprezentare (matrice de adiacenţă, liste de adiacenţă)
11.3. Arbori
– terminologie (nod, muchie, rădăcină, descendent, descendent direct/fiu, ascendent, ascendent direct/părinte, frați, nod terminal, frunză)
– metode de reprezentare în memorie (matrice de adiacenţă, liste ”de descendenţi”, vector ”de taţi”)

I. Algoritmi

1.1. Noțiunea de algoritm, caracteristici

Ce este un algoritm?

Conceptul de algoritm nu este nou. Termenul algoritm derivă din numele unui matematician persan, *Abu Jafar Mohammed ibn Musa al Khowarizmil* (cca. 825 e.n.), care a scris o carte cunoscută sub denumirea latină de „Liber algorithmi".

Matematicienii Evului Mediu înţelegeau prin algoritm o regulă pe baza căreia se efectuau calcule aritmetice. Ulterior, termenul de algoritm a circulat într-un sens restrâns, exclusiv în domeniul matematicii. O dată cu dezvoltarea calculatoarelor cuvântul algoritm a dobândit o semnificaţie aparte, astfel încât astăzi gândirea algoritmică s-a transformat, dintr-un instrument specific matematicii, într-o modalitate fundamentală de abordare a problemelor în diverse domenii.

Un algoritm reprezintă o metodă de rezolvare a problemelor de un anumit tip.

A rezolva o problemă înseamnă a obţine, pentru anumite date de intrare, rezultatul problemei (date de ieşire):

Algoritmul este constituit dintr-o succesiune de operaţii care descriu, pas cu pas, modul de obţinere a datelor de ieşire, plecând de la datele de intrare. Se pot descrie algoritmi pentru rezolvarea problemelor din orice domeniu de activitate.

Scopul elaborării algoritmului unei probleme este acela de a scrie un program într-un anumit limbaj de programare. Dar dacă avem de-a face cu o problemă mai complexă, înainte de a scrie programul este bine să scriem paşii algoritmului; în acest scop avem la dispoziţie o formă foarte accesibilă de reprezentare a algoritmilor, şi anume pseudocodul.

De exemplu, orice reţetă de bucătărie poate fi considerată un algoritm prin care, plecând de la materiile prime, obţinem printr-o succesiune finită de operaţii produsul finit.

Exemplul 1:
Presupunând că dispunem de un aragaz, o tigaie, 2 ouă, sare şi 2oo ml ulei, să pregătim ochiuri.
Date de intrare: ouă, ulei, sare.
Date de ieşire: ochiuri.
Pas 1: Se pune tigaia pe foc.
Pas 2: Se toarnă uleiul în tigaie.
Pas 3: Aşteptăm până când se încinge uleiul.
Pas 4: Spargem cu îndemânare ouăle în tigaie.
Pas 5: Aşteptăm până când ouăle se rumenesc.
Pas 6: Dacă nu ţinem regim, adăugăm sare.

Observăm că am descris o succesiune de paşi, prin care, pentru orice ,,date" de intrare (ulei, sare, ouă), obţinem rezultatul dorit (ochiuri). Fiecare pas constă din operații culinare specifice, care se execută în ordinea în care sunt specificate.

Exemplul 2

Să analizăm un alt exemplu, cu care suntem familiarizaţi de la matematică: rezolvarea ecuaţie de gradul I de forma : ax+b=0, cu a,b ϵ R.

Date de intrare: a, b ϵ R

Date de ieşire: x ϵ R, soluţia ecuaţiei, sau un mesaj adecvat

Pas 1: citeşte datele de intrare a şi b numere reale

Pas 2: dacă a≠0 atunci

scrie "soluţia ecuaţiei este ", x← -b/a

altfel

dacă b=0 atunci

scrie "infinitate de soluţii"

altfel

scrie "relaţie matematică imposibilă"

Această succesiune de paşi rezolvă ecuaţia de forma specificată pentru orice valori ale datelor de intrare, prin urmare este un algoritm.

Proprietăţi caracteristice ale algoritmilor

Exemplele precedente generează în mod firesc două întrebări:

1. *Pentru orice problemă există un algoritm de rezolvare?*
   Răspunsul este NU! Există probleme pentru care se poate demonstra că nu există algoritmi de rezolvare, dar şi probleme pentru care nici nu s-a demonstrat că nu admit o metodă de rezolvare algoritmică, dar nici nu s-a descoperit soluţia algoritmică.
2. *Orice succesiune de paşi reprezintă un algoritm?*
   Din nou, răspunsul este NU! Pentru a fi un algoritm, secvenţa trebuie să îndeplinească condiţiile:
   - ✓ *Rezolvabilitate* ~ algoritmul să aibă cel puţin o soluţie;
   - ✓ *Claritate* ~ la fiecare moment, operaţia care urmează a fi executată este unic determinată definită şi realizabilă (adică poate fi efectuată la momentul respectiv, cu mijloacele disponibile);
   - ✓ *Generalitate* (universalitate) ~ o secvenţă de paşi reprezintă un algoritm de rezolvare a unei probleme dacă obţine date de ieşire (rezultate) pentru orice date de intrare specifice problemei (adică să rezolve o întreagă clasă de probleme, nu una particulară);
   - ✓ *Finitudine* ~ rezultatele problemei se obţin după un număr finit de paşi;
   - ✓ *Eficienţa* ~ algoritmul să rezolve problema într-un număr cât mai mic de paşi pentru a se obţine un timp de execuţie cât mai mic alături de un consum mic de memorie.

În concluzie, deşi nu putem defini cu rigurozitate noţiunea de algoritm, putem descrie mai detaliat această noţiune astfel:

Un **algoritm** este constituit dintr-o succesiune clară şi finită de operaţii realizabile, efectuate mecanic, care pornind de la un set de date de intrare conduc în timp finit la obţinerea unui set de date de ieşire (rezultate).

Etapele rezolvării unei probleme

Rezolvarea unei probleme constituie un proces complex, care comportă mai multe etape.

1. Analiza problemei în scopul stabilirii datelor de intrare, precum şi a rezultatelor pe care trebuie să le obţinem prin rezolvarea problemei.
2. Elaborarea unui algoritm de rezolvare a problemei.
3. Implementarea algoritmului într-un limbaj de programare.
4. Verificarea corectitudinii algoritmului propus.
   Un prim pas constă în testarea programului pe diverse seturi de date de test. Seturile de date de test trebuie elaborate cu atenţie, astfel încât să acopere, pe cât posibil, toate variantele de execuţie a algoritmului, inclusiv situaţii de excepţie, şi să verifice dacă fiecare subproblemă a problemei date este rezolvată corect (dacă este posibil, se va testa separat fiecare modul de program).
5. Analiza complexităţii algoritmului.
   În general, există mai mulţi algoritmi de rezolvare a unei probleme date. Pentru a alege cel mai bun algoritm, trebuie să analizăm aceşti algoritmi în scopul determinării eficienţei lor şi, pe cât posibil, a optimalităţii lor.
   Eficienţa unui algoritm se evaluează din două puncte de vedere:
   a. Din punctul de vedere al spaţiului de memorie necesar pentru memorarea valorilor variabilelor care intervin în algoritm (complexitate spaţiu);
   b. Din punctul de vedere al timpului de execuţie (complexitate timp).

1.2 Date, variabile, expresii, operaţii

Definiţie: O **dată** este orice entitate cu care poate opera calculatorul.

Orice algoritm lucrează cu date: date de intrare (datele pe care trebuie să le primească un algoritm din exterior), date de ieşire (datele pe care trebuie să le furnizeze algoritmul în exterior), precum şi date de manevră (date temporare, necesare algoritmului pentru a obţine datele de ieşire pe baza datelor de intrare).

Datele cu care lucrează algoritmii pot fi clasificate din mai multe puncte de vedere. O primă clasificare a datelor, în funcţie de posibilitatea de a-şi modifica valoarea, este:

- **Constante** - date care nu îşi modifică valoarea; de exemplu: 10, 3 .14, ”sir de caractere”, ‘A’, fals (0).
- **Variabile** - date care îşi modifică valoarea. O variabilă poate fi referită printr-un nume (o succesiune de litere, cifre şi liniuţă de subliniere, primul caracter fiind obligatoriu literă sau liniuţă de subliniere) şi are asociată o valoare. Numele unei variabile nu se schimbă pe parcursul algoritmului, dar valoarea acesteia se poate modifica.

Pentru a cunoaşte natura valorilor care pot fi asociate variabilelor precum şi a operaţiilor permise cu acestea, variabilele trebuie declarate înainte de a fi utilizate.

În funcţie de valoarea lor, datele pot fi clasificate astfel:

1) Date numerice - au ca valori numere (naturale, întregi sau reale);
2) Date alfabetice - au ca valori caractere sau şiruri de caractere;
3) Date logice - au valoarea adevărat sau fals (1 sau 0).

Expresii

O expresie este constituită dintr-o succesiune de operanzi, conectaţi prin operatori. Un operand poate fi o constantă, o variabilă, sau o expresie încadrată între paranteze rotunde. Operatorii desemnează operaţiile care se execută asupra operanzilor. Operatorii care pot fi utilizaţi într-o expresie depind de tipul operanzilor (numerici întregi, numerici reali, caractere, şiruri de caractere sau logici).

Evaluarea unei expresii presupune calculul valorii expresiei, prin înlocuirea valorilor variabilelor care intervin ca operanzi în expresie şi efectuarea operaţiilor specificate de operatori.

Categorii de operatori:

Operatori aritmetici

Operatorii aritmetici definesc o operaţie aritmetică şi pot fi clasificaţi astfel:

1. *Operatori aritmetici multiplicativi:* * (înmulţire), /(împărţire), % (restul împărţirii întregi).

Operatorul de împărţire (/) are un efect diferit, în funcţie de tipul operanzilor. Dacă ambii operanzi sunt întregi, se face împărţire întreagă (se obţine ca rezultat un număr întreg, care este câtul împărţirii primului operand la cel de-al doilea).

Exemplu:

```
întreg a,b
a←7; b←2
scrie a/b  =>  expresia a/b are valoarea 3.
```

Dacă cel puţin unul dintre operanzi este real, se face împărţire reală (se obţine ca rezultat un număr real).

Exemplu:

```
real a,b
a←7; b←2
scrie a/b  => expresia a/b are valoarea 3.5.
```

Operatorul % se poate aplica numai asupra operanzilor întregi.

2. *Operatori aritmetici aditivi:* + (adunare) şi -(scădere).

Operatorii aritmetici aditivi şi multiplicativi sunt **binari** (acționează asupra a doi operanzi). Operatorii aritmetici se pot aplica numai operanzilor numerici.

Rezultatul evaluării unei expresii aritmetice este numeric (întreg sau real, în funcţie de operanzi şi operatori).

Operatori relaționali

Operatorii relaţionali descriu relaţia de ordine sau de egalitate dintre cei doi operanzi: < (mai mic), > (mai mare), ≤ (mai mic sau egal), ≥ (mai mare sau egal), = (egal), ≠ (diferit).
Operatorii relaţionali sunt operatori binari şi se pot aplica numai operanzilor numerici, logici (fals < adevărat) şi de tip caracter (ordinea caracterelor fiind determinată de codurile lor ASCII).
Valoarea unei expresii relaţionale este întotdeauna de tip logic (deci poate fi adevărat sau fals).

Operatori logici

Operatorii logici se pot aplica operanzilor logici. Valoarea unei expresii logice este de tip logic.
Operatorii logici definesc o operaţie logică: negaţie logică - !; conjuncţie logică - *şi*; disjuncţie logică - *sau*. Operatorul ! este unar, operatorii ***şi, sau*** sunt operatori binari. Efectul acestor operatori este cel uzual. Îi reamintim în tabelul următor:

x	y	!x	x sau y	x şi y
Fals (0)	Fals (0)	Adevărat (1)	Fals (0)	Fals (0)
Fals (0)	Adevărat (1)	Adevărat (1)	Adevărat (1)	Fals (0)
Adevărat (1)	Fals (0)	Fals (0)	Adevărat (1)	Fals (0)
Adevărat (1)	Adevărat (1)	Fals (0)	Adevărat (1)	Adevărat (1)

Evaluarea unei expresii
În procesul de evaluare a unei expresii se respectă regulile de bază, învăţate la matematică (în primul rând se evaluează expresiile dintre parantezele rotunde; apoi se execută operaţiile în ordinea priorităţii lor; dacă există mai multe operaţii cu aceeaşi prioritate, se execută în ordine, în funcţie de asociativitatea lor).

Prioritatea operatorilor este: (1 fiind considerată prioritatea maximă):

Prioritate	Operatori	Simbol	Asociativitate
1.	Negaţia logică	!	de la dreapta la stânga
2.	Aritmetici multiplicativi	*, /, %	de la stânga la dreapta
3.	Aritmetici aditivi	+, -	de la stânga la dreapta
4.	Relaţionali	<, >, ≤, ≥, =, ≠	de la stânga la dreapta
5.	Conjuncţie logică	şi	de la stânga la dreapta
6.	Disjuncţie logică	sau	de la stânga la dreapta

1.3 Structuri de bază (liniară, alternativă şi repetitivă)

Principiile programării structurate

Creşterea complexităţii aplicaţiilor a impus la începutul anilor ‘70 apariţia unei noi paradigme în programare: programarea structurată. Scopul era de a dezvolta noi tehnici de programare care să permită dezvoltarea unor programe fiabile, uşor de elaborat în echipă, uşor de depanat, de întreţinut şi de reutilizat.

Un prim principiu al programării structurate este ***modularizarea***. Pentru proiectarea unor aplicaţii complexe, este necesară descompunerea problemei care trebuie rezolvată în subprobleme relativ independente, pentru fiecare dintre

aceste subprobleme scriindu-se module de program mai simple. Fiecare modul efectuează un set de prelucrări specifice şi este relativ independent de celelalte module, cu care comunică prin intermediul unui set de parametri, care constituie interfaţa.

Avantajele sunt multiple. Cum la orice firmă se lucrează în echipă, modulele de program pot fi implementate de mai mulţi programatori.

Modificarea unui modul nu afectează celelalte module. Fiecare modul poate fi implementat, testat, depanat, modificat, independent de celelalte.

Un alt principiu fundamental este ***structurarea datelor şi a prelucrărilor***. Programatorul are posibilitatea de a-şi grupa datele în colecţii, organizate după anumite reguli, denumite structuri de date.

Prelucrările asupra datelor sunt structurate separat. Conform teoremei de structură a lui **Bohm - Jacopini**, orice prelucrare poate fi descrisă prin compunerea a trei structuri fundamentale: **structura liniară** (secvenţială), **structura alternativă** şi **structura repetitivă**.

1.4 Reprezentarea algoritmilor în pseudocod

Pentru ca o secvenţă de operaţii să constituie un algoritm, ea trebuie să fie clară, adică la orice moment operaţia care urmează a fi executată trebuie să fie unic determinată, definită şi realizabilă (să poată fi efectuată la momentul respectiv, cu mijloacele disponibile). Apare întrebarea: care sunt operațiile definite, cu ajutorul cărora să putem descrie algoritmi?

Este nevoie de o metodă universală de reprezentare a algoritmilor, ulterior fiecare programator având posibilitatea de a implementa algoritmii în limbajul pe care îl preferă.

De-a lungul timpului s-au impus două modalităţi de reprezentare a algoritmilor: **schemele logice** şi **limbajele de tip pseudocod**.

Schemele logice constituie o metodă de reprezentare grafică, foarte sugestivă, dar cu o serie de dezavantaje: se dă o egală importanţă componentelor principale ca şi detaliului, prin urmare schemele logice devin deosebit de stufoase şi greu de urmărit; pentru aplicaţiile mai complexe, când este necesară modularizarea, este practic imposibil de pus în evidenţă legăturile dintre module în cadrul schemei logice. Din acest motiv, treptat s-a impus o altă metodă de reprezentare a algoritmilor: pseudocodul.

Un limbaj de tip pseudocod este un ansamblu de convenţii, respectate în mod sistematic, care definesc operațiile permise (denumite şi instrucţiuni) pentru reprezentarea algoritmilor.

Un limbaj pseudocod se prezintă sub formă de text şi se bazează pe nişte aşa-numite cuvinte cheie. Fiecare cuvânt cheie identifică în mod unic un anumit tip de acţiune.

Acţiunile algoritmului se reprezintă în pseudocod prin ceea ce numim instrucţiuni. Ansamblul cuvintelor cheie împreună cu regulile care trebuie respectate în folosirea lor, alcătuiesc ceea ce numim sintaxa limbajului pseudocod.

Există o mare diversitate de limbaje pseudocod. Practic, fiecare programator îşi poate proiecta propriul pseudocod, definind cuvintele cheie ale acestuia şi impunând nişte reguli de sintaxă.

Structura secvenţială

Declararea datelor
Sintaxa: variabila tip;

La începutul oricărui algoritm, vom preciza datele de intrare, datele de ieşire, eventualele date de manevră precum şi tipul acestora. Înainte de a utiliza orice variabilă, după cum am precizat şi anterior, o vom **declara**, precizând **numele** şi **tipul** ei. O variabilă nu poate fi declarată de mai multe ori în acelaşi algoritm.
Exemple
x real;
c caracter;
i întreg;

Operaţia de citire
Sintaxa: **citeşte** $variabila_1$, $variabila_2$, ..., $variabila_n$;
Efect: Prin operaţia de citire (denumită şi operaţie de intrare) se preiau succesiv valori de la tastatură şi se asociază, în ordine, variabilelor specificate.

Operaţia de scriere
Sintaxa: **scrie** $expresie_1$, $expresie_2$, ..., $expresie_n$;
Efect: Operaţia de scriere (denumită şi operaţie de ieșire) presupune evaluarea în ordine a expresiilor specificate şi afişarea pe ecran a valorilor lor pe aceeaşi linie.

Operaţia de atribuire
Sintaxa: variabila ← expresie;
Efect: se evaluează expresia, apoi se atribuie valoarea expresiei variabilei din membrul stâng.

Observaţii
1. Pentru claritate, putem insera într-un algoritm comentarii, mici texte explicative. Începutul unui comentariu este marcat de succesiunea de caractere /* iar sfârşitul comentariului este marcat de */.

Parcurgerea instrucţiunilor în secvenţă, în ordinea specificării lor, reprezintă o structură liniară (secvenţială).

Structura alternativă
dacă expresie **atunci**
    instrucţiune_1
[**altfel**
    instrucţiune_2]
sf. dacă

parantezele drepte indică faptul că ramura de altfel e opţională

Efect:
Se evaluează expresia.

Dacă valoarea expresiei este adevărat, atunci se execută instrucţiune_1. Dacă valoarea expresiei este fals, se execută instrucţiune_2 dacă ramura de *altfel* există iar în caz contrar nu se execută nimic.

Observaţii
Atât pe ramura atunci, cât şi pe ramura altfel este permisă executarea unei singure instrucţiuni. În cazul in care este necesară efectuarea mai multor operaţii, acestea se grupează într-o singură instrucţiune compusă.
Instrucţiunea *dacă…atunci* permite executarea unei singure instrucţiuni, în funcţie de valoarea unei expresii, deci permite selectarea condiţionată a unei alternative. Această instrucţiune implementează în pseudocod structura alternativă.
Aplicaţii: Modulul unui număr, Verificarea parităţii unui număr, Rezolvarea ecuaţiei de gradul II.

Structura repetitivă

În funcţie de numărul de paşi executaţi există 2 tipuri de structuri repetitive:

- ✓ cu număr necunoscut de paşi; care în funcţie de locul în care apare condiţia de repetiţie se clasifică în:
  - ➢ instrucţiune repetitivă condiţionată anterior
  - ➢ instrucţiune repetitivă condiţionată posterior
- ✓ cu număr cunoscut de paşi

Structură repetitivă cu număr necunoscut de paşi:

- **Instrucţiunea repetitivă condiţionată anterior**

cât timp expresie **execută**
    instrucţiune
sf. cât timp
Efect:
Pas 1: se evaluează expresia;
Pas 2: dacă valoarea expresiei este fals(0), se iese din instrucţiunea *cât timp…execută*;
daca valoarea expresiei este adevărat, se execută instrucţiunea, apoi se revine la Pas 1.

Observaţii
Instrucţiunea se execută repetat, cât timp valoarea expresiei este adevărat (1). Pentru ca ciclul să nu fie infinit, este obligatoriu ca instrucţiunea care se execută să modifice cel puţin una dintre variabilele care intervin în expresie, astfel încât aceasta să poată lua valoarea fals(0).
Dacă expresia are de la început valoarea fals(0), instrucţiunea nu se execută nici măcar o dată.

- **Instrucţiunile repetitive condiţionate posterior**

execută
instrucţiune
cât timp expresie;
Efect:
Pas 1: se execută instrucţiunea;
Pas 2: se evaluează expresia;
Pas 3:dacă valoarea expresiei este fals(0) se iese din instrucţiunea repetitivă; dacă valoarea expresiei este adevărat(1), se revine la Pas 1.

repetă
instrucţiune
până când expresie;
Efect:
Pas 1: se execută instrucţiunea;
Pas 2: se evaluează expresia;
Pas 3:dacă valoarea expresiei este adevărat(1) se iese din instrucţiunea repetitivă; dacă valoarea expresiei este fals(0), se revine la Pas 1.

Observaţii:

1. Instrucțiunea *execută...cât timp* se execută repetat, cât timp valoarea expresiei este adevărat(1). Pentru ca ciclul să nu fie infinit, este obligatoriu ca instrucţiunea care se execută să modifice cel puţin una dintre variabilele care intervin în expresie, astfel încât aceasta să poată lua valoarea fals(0).
2. Instrucțiunea *repetă...până când* se execută repetat, cât timp valoarea expresiei este fals(0). Pentru ca ciclul să nu fie infinit, este obligatoriu ca instrucţiunea care se execută să modifice cel puţin una dintre variabilele care intervin în expresie, astfel încât aceasta să poată lua valoarea adevărat(1).
3. Deoarece evaluarea expresiei în ambele repetitive se face după execuţia instrucţiunii, instrucţiunea se execută cel puţin o dată.

Pentru ca instrucţiunea *cât timp…execută* să fie echivalentă cu instrucţiunea *execută…cât timp* este necesară verificarea în prealabil a condiţiei (expresiei logice) astfel:

```
cât timp expresie execută
   instrucţiune
sf. cât timp
```

```
dacă expresie atunci
  execută
    instrucţiune
  cât timp expresie;
sf. dacă
```

Pentru ca instrucţiunea *execută… cât timp* să fie echivalentă cu instrucţiunea *repetă…până când* este necesară negarea condiţiei (expresiei logice) astfel:

```
execută
   instrucţiune
cât timp expresie
```

```
repetă
     instrucţiune
până când !(expresie);
```

Structură repetitivă cu număr cunoscut de paşi:

- **Instrucţiunea repetitivă cu număr cunoscut de paşi**

Pentru contor←expresie$_1$, expresie$_2$,[pas] **execută**
    Instrucţiune
Sf. pentru
Efect:
Pas 1: Se evaluează *expresie$_1$*.
Pas 2: Se atribuie variabilei *contor* valoarea expresiei *expresie$_1$*.
Pas 3. Se evaluează *expresie$_2$*.
Pas 4: Dacă valoarea variabilei *contor* este mai mare/mică (în funcție de valoarea *pas, adică pozitivă sau negativă*) decât valoarea expresiei *expresie$_2$*, atunci se iese din instrucțiunea repetitivă. Dacă valoarea variabilei *contor* este mai mică/mare sau egală cu valoarea expresiei *expresie$_2$*, atunci se execută instrucțiune şi apoi se incrementează/decrementează (se modifică valoarea contorului cu pas) valoarea variabilei *contor*, după care se revine la Pas 3.

Observații

1. Dacă variabila *pas* lipsește se consideră implicit 1.
2. Instrucțiunea repetitivă cu număr cunoscut de pași poate fi simulată cu ajutorul celorlalte două instrucţiuni repetitive.

Executarea repetată a unei instrucţiuni, controlată de valoarea unei expresii, reprezintă o structură repetitivă.

II. Elemente de bază ale limbajului C/C++

2.0 Noţiuni introductive

Limbajul de programare este mijlocul de comunicare între utilizator şi calculator. Pentru a defini limbajul de programare se au în vedere 3 aspecte:
Sintaxa = reprezintă totalitatea regulilor care trebuie respectate pentru definirea elementelor limbajului;
Semantica = defineşte semnificaţia construcţiilor sintactic corecte;
Pragmatica = defineşte modul de utilizare a elementelor limbajului.
Implementarea unui algoritm într-un limbaj de programare se numeşte **program.**

Evoluţia limbajelor de programare
O clasificare a limbajelor de programare în funcţie de nivel:
- de nivel scăzut (cele pentru care programatorul trebuie să cunoască modul de funcţionare a procesorului – lucrează cu regiştrii procesorului);
- de nivel înalt (necesită cunoştinţe tehnice de nivel minim);

Exemplu de limbaj de programare de nivel scăzut: limbajele de asamblare.
Exemple de limbaje de nivel mediu şi înalt:

- 1955 **FORTRAN** (Formula Translation) – destinat aplicaţiilor tehnico-ştiinţifice cu caracter numeric
- 1960 **ALGOL** (Algorithmic Language) este primul limbaj definit riguros, cu o sintaxă complet formalizată; folosit şi astăzi de proiectanţii de limbaje de programare
- 1960 **COBOL** (Common Business Oriented Language) – destinat aplicaţiilor economice
- 1971 – **PASCAL** – conceput de Niklaus Wirth pentru studenţi (ca să-şi însuşească rapid „arta programării”)
- 1972 – **C** – conceput de Brian Kernighan şi Dennis Richie ca limbaj cu destinaţie universală
- 1980 – Bjarne Stroustup publică specificaţiile limbajului **C++** (extensie a limbajului C destinată programării pe obiecte (POO))
- 1995 – James Gostling a publicat specificaţiile limbajului **Java** (are ca obiectiv principal portabilitatea, el fiind independent de maşina pe care rulează)

Evoluţia este şi la nivel de programare Internet şi grafică.

2.1 Vocabularul limbajului

Vocabularul limbajului este alcătuit din: *setul de caractere, identificatori, separatori şi comentarii.*

Setul de caractere
În C/C++ setul de caractere acceptat este cel al codului ASCII;
- codul ASCII standard (codifică de la 0-127)
- codul ASCII extins (codifică de la 128-255) caractere grafice şi semigrafice

Există 256 de coduri ASCII.
Exemple:
‘a’=097

‘A’=065
‘0’=048
Pentru a scrie în limbaj de programare un caracter utilizând codul ASCII :
Alt (din dreapta) + codul ASCII al caracterului.
Exemplu: Alt+179 scrie caracterul grafic ’|’

Identificatorii sunt întâlniţi şi sub denumirea de “nume” şi au rolul de a desemna nume de variabile, constante , funcţii, etc.
Din punct de vedere sintactic identificatorii sunt o succesiune de **litere, cifre şi liniuţe de subliniere** dintre care prima trebuie să fie literă sau liniuţă de subliniere.
Ex: nume corect de variabilă: a, a3, _56, nr_cuv
nume greşit de variabilă: 3aq, nr cuv
Observaţii:
- un identificator poate să aibă orice lungime, semnificative sunt primele 31 caractere.
- C/C++ este *case-sensitive* (face distincţie între literele mari şi mici; adică un identificator *a* diferă de identificatorul *A*)
- identificatorii ar trebui să aibă denumirea în conformitate cu scopul utilizării lor.

O categorie specială de identificatori este reprezentată de **cuvintele rezervate** sau **cuvinte cheie** ale limbajului.
Din punct de vedere sintactic cuvintele cheie sunt identificatori utilizaţi numai în scopul pentru care au fost creaţi.

Exemplu:
if (cuvânt cheie care marchează începutul instrucţiunii alternative în C)
while (cuvânt cheie care marchează începutul instrucţiunii repetitive în C)
Observaţie: este eroare de sintaxă utilizarea cuvintelor cheie în alt scop decât cel pentru care au fost create

Separatorii
Sunt utilizaţi pentru a separe unităţile sintactice între ele;
Există mai multe tipuri de separatori:
- separatori standard: spaţiul (‘ ’), tab(‘\t’), enter (new line ‘\n’)
- delimitatori: operatorul virgulă (‘,’)
- separatori speciali: ‘;’(este utilizat la finalul unei instrucţiuni), ‘’(apostroafe) şi “” (utilizaţi pentru constantele de tip caracter şi şir de caractere)

Comentariile
Reprezintă o succesiune de caractere pe care compilatorul (cel care transcrie codul sursă în limbaj maşină) nu le ia în considerare.
Există în C/C++ două tipuri de comentarii:
- pe mai multe linii cu marcajul /*…*/
- pe o singură linie (de la marcaj până la sf. de linie) cu marcajul //

Sunt utilizate pentru a creşte gradul de lizibilitate a programului şi ajută utilizatorii multiplii la înţelegerea programului.

2.2 Constante

Variabilele - sunt date a căror valoare poate fi modificată pe parcursul execuţiei programului.
Constantele - sunt date ce nu pot fi modificate în cadrul programului. *Constantele întregi* sunt numere întregi exprimate in:
zecimal - succesiuni de cifre zecimale (ex: 100)
octal - succesiuni de cifre octale precedate de 0 (ex: 024)
hexazecimal - succesiuni de cifre hexazecimale precedate de 0x sau 0X
(ex: 0xFF1) ; baza 16 are cifre 0..9 şi litere A..F unde A=10…F=15

Constantele reale pot fi specificate în notaţia uzuală sau în format exponenţial (ştiinţific). În forma uzuală cuprind partea întreagă şi partea zecimală, separate de caracterul **.**(punct). În format exponenţial se specifică în plus un exponent al lui 10, precedat de **e** sau **E**. În acest caz valoarea numărului se obţine înmulţind numărul (corespunzător construcţiei din faţa literei **e/E**) cu 10 la puterea specificată de exponent.

Constantele caracter sunt constituite din caractere încadrate între apostroafe. Se pot construi *secvenţe escape* (formate din caracterul *backslash* '\' urmat de codul ASCII al caracterului - în baza 8 sau x şi codul ASCII în baza 16)
Exemple:
'\65' este '5' în baza 8
'\x35' este '5' în baza 16
Unele caractere negrafice au asociate secvenţe escape speciale:

Secvenţă escape	Caracter
'\b'	Caracterul **backslash** (deplasează cursorul pe ecran cu o poziţie la stânga)
'\t'	Caracterul **tab** orizontal
'\n'	Caracterul **newline** (determină trecerea cursorului la linie nouă)
'\a'	Caracterul **alarm** (generează un sunet)
'\\'	Caracterul **backslash**
'\''	Caracterul **apostrof**
'\"'	Caracterul **ghilimele**

Constantele şir de caractere sunt constituite dintr-o succesiune de caractere încadrate între ghilimele. Sunt reprezentate intern prin codurile ASCII ale caracterelor şi terminate cu '\0' (terminatorul şirurilor de caractere sau caracterul NULL).
O constantă simbolică este o constantă desemnată printr-un identificator. Poate fi predefinită sau definită de utilizator.
Exemple de constante simbolice predefinite în C/C++:
- MAXINT cu valoarea 32767 *(# define MAXINT 32767*)
- MAXLONG cu valoarea 2147483647

Exemple de constante şir de caractere definite de utilizator:
"Acesta este un sir"

2.3 Noţiunea de tip de dată. Operatori aritmetici, logici, relaţionali

O **dată** este orice entitate cu care poate opera calculatorul.

Orice limbaj de programare dispune de un set de tipuri de date predefinite numite şi tipuri de date standard

Un **tip de date** este format din mulţimea valorilor pe care le pot lua datele de tipul respectiv, modul de reprezentare în memorie precum şi operaţiile care se pot aplica datelor de tipul respectiv.

Observaţie:

Mulţimea valorilor unui anumit tip de date reprezintă **constantele** tipului respectiv.

Operatori aritmetici, logici, relaţionali

Operatorii aritmetici : +, -, *, /, %

Operatorul	Tipul operaţiei
+, -	Semne (operatori unari)
*, /, %	De multiplicitate (operatori binari)
+,-	Aditivi (operatori binari)

Observaţii:

Operatorul '%' se poate aplica numai datelor de tip întreg.

Dacă operatorul '/' se aplică cel puţin unui operand de tip real atunci rezultatul este real (idem pentru operatorii +,-,*)

Exemplu:

```
int a=5,b=2;
cout<<a/b;   // 2
cout<<(float)a/b;  // 2.5
cout<<5/2.0;   //2.5
```

Operatorii logici sunt de două tipuri: logici globali şi logici pe biţi (sunt operatori binari toţi în afară de negaţia logică şi pe biţi care sunt unari)

- Operatori logici globali (&&, ||, !)

&&	0	!0=1
0	0	0
!0=1	0	1

||	0	!0=1
0	0	1
!0=1	1	1

!	
0	1
!0=1	0

Observaţie: Operatorii logici globali se pot aplica datelor de orice tip standard, rezultatul fiind întotdeauna logic.

Exemplul 1:

```
int a=5,b=3;
if(a>0)
   cout<<a; //5
else
   cout<<b;
```

Exemplul 2:

```
int a=5,b=3;
if (a)//⇔ a!=0
   cout<<a; //5
else
    cout<<b;
```

Exemplul 3:

```
int a=5,b=3;
if(a&&b)
  cout<<a+b;
//8
else
  cout<<a-b;
```

Exemplul 4:

```
int a=5,b=3;
if(a>0 && b<0) //1&&0==0
  cout<<a+b;
else
  cout<<a-b; //2
```

Exemplul 5:

```
int a=5,b=3;
if(a>0   ||   b<0)
//1||0==1
  cout<<a+b;  //8
else
  cout<<a-b;
```

Exemplul 6:

```
int a=4;
cout<<(a<0)<<"
"<<!a<<"  "  <<
(a==4); // 0 0 1
```

- Operatori logici pe biţi (&, |, ~, ^)

şi pe biţi

&	0	1
0	0	0
1	0	1

sau pe biţi

|	0	1
0	0	1
1	1	1

XOR

~	
0	1
1	0

negaţia pe biţi

sau exclusiv sau

^	0	1
0	0	1
1	1	0

Exemplul &	Exemplul |	Exemplul ~	Exemplul ^

int a=10,b=5;
//a=1010_2 b=101_2
cout<<a&b; // 0
int a=10,b=6;
//a=1010_2 b =110_2
cout<<a&b; //2
	int a=10,b=5;
	// a=1010_2 b=0101_2
	// a|b= 1111_2=15
	cout<<a|b; // 15
	int a=10,b=6;
	// a=1010_2 b=0110_2
	// a|b= 1110_2= 14
	cout<<a|b; // 14
		int a=10;
		cout<<~a;
		// 32757
			int a=10,b=5;
			//a=1010_2 b=0101_2
			//a^b= 1111_2=15
			cout<<a^b; // 15
			int a=10,b=6;
			//a=1010_2 b=0110_2
			//a^b= 1100_2= 12
			cout<<a^b; // 12

Operatori relaţionali (<,>,<=,>=) se aplică datelor de tip standard şi returnează un rezultat de tip logic.

Observaţie: Negarea lui > este <=, negarea lui < este >=, negarea lui <= este >, negarea lui >= este <.

Exemplu:

```
int a=5,b=3;
if(a>b)   // a<b  sau  a>=b  sau  a<=b  sau a!=b
   cout<<a;   //5
else
   cout<<b;
```

Operatorul de egalitate şi diferit(==,!=)

Exemplu:

```
int a=5,b=5;
if(a==b)
  cout<<"Da";   // Da
else
  cout<<"Nu";
```

Observaţie: Operatorul '**==**' şi '**!=**' apare doar în expresii logice

2.4 Tipuri standard de date. Conversii implicite şi explicite

1) **Tipul întreg**

Denumirea	Nr. octeţi	Valori	Obs.
int	2 octeţi cu semn	-32768..32767	$0000000000000000_2=0_{10}$ $1111111111111111_2=32767_{10}$
unsigned int	2 octeţi fără semn	0…65535	
long int	4 octeţi cu semn	-2147483648…2147483647	
unsigned long int	4 octeţi fără semn	0…4294967295	

Observaţie: La oricare din tipurile întregi în afară de int cuvântul rezervat int este implicit long int ⇔ long; unsigned int ⇔ unsigned

Exemple:

int a; // declar un număr întreg **a** cărui valoare nu depăşeşte în modul 32767

unsigned b,c; // declar două numere naturale a căror valoare nu depăşeşte 65535

2) Tipul real

Denumirea	Nr. octeţi - în virgulă mobilă	Valori
float	4 octeţi	$[3.4*10^{-38},3.4*10^{38}]U[-3.4*10^{+38},-3.4*10^{-38}]$
double	8 octeţi	$[1.7*10^{-308},1.7*10^{308}]U[-1.7*10^{+308},-1.7*10^{-308}]$
long double	10 octeţi	$[3.4*10^{-4932},1.1*10^{4932}]U[-3.4*10^{+4932},-1.1*10^{-4932}]$

3) Tipul caracter

Denumirea	Nr. octeţi	Valori
char	1 octet cu semn	-128…127
unsigned char	1 octet fără semn	0…255

Observaţie: cuprinde caracterele din codul ASCII
Şirurile de caractere se obţin prin concatenarea (lipirea caracterelor) între ele (nu există tipul şir de caractere între tipurile standard de date C/C++).

Conversiile de tip pot fi de 2 feluri: implicite şi explicite.

Conversii implicite de tip:

Conversiile implicite au loc atunci când este necesar ca operatorii şi argumentele funcţiilor să corespundă cu valorile aşteptate pentru acestea. Acestea pot fi sintetizate prin tabelul:

Tip	Tip la care se converteşte implicit
char	int, short int, long int
int	char (cu trunchiere) short int (cu trunchiere) long int (cu extensia semnului)
short int	ca şi int
long int	ca şi int
float	double, int, short int, long int
double	float, int, short int, long int

Conversii aritmetice.

Când un operator binar se aplică între doi operanzi de tip diferit, are loc o conversie implicită a tipului unuia dintre ei, şi anume, operandul de tip “mai restrâns” este convertit la tipul “mai larg” al celuilalt operand. Astfel în expresia de mai jos:

int i;

float f;

f + i operandul int este convertit în float.

Operatorii aritmetici convertesc automat operanzii la un anumit tip, dacă operanzii sunt de tip diferit. Se aplică următoarele reguli:

- operanzii char şi short int se convertesc în int; operanzii float se convertesc în double.
- dacă unul din operanzi este double restul operanzilor se convertesc în double iar rezultatul este tot double.
- dacă unul din operanzi este long restul operanzilor se convertesc în long , iar rezultatul este tot long.

- dacă unul din operanzi este unsigned restul operanzilor se convertesc în unsigned , iar rezultatul este tot unsigned.
- dacă nu se aplică ultimele 3 reguli, atunci operanzii vor fi de tip int şi rezultatul de asemeni de tip int.

double ←float
↑
long
↑
unsigned
↑
int ←char, short

Astfel n = c - '0' în care c reprezintă un caracter cifră calculează valoarea întreagă a acestui caracter.

Conversii implicite se produc şi în cazul operaţiei de atribuire, în sensul că valoarea din partea dreaptă este convertită la tipul variabilei acceptoare din stânga.

Astfel pentru declaraţiile de ai jos:

```
int i;
float f;
double d;
char c;
```

sunt permise atribuirile:

```
i=f;  // cu trunchierea părţii fracţionare
f=i; d=f; f=d; c=i; i=c;
```

Conversiile de tip explicite (cast).

Conversiile explicite de tip (numite şi cast) pot fi forţate în orice expresie folosind un operator unar (cast) într-o construcţie de forma:

(tip) expresie

în care expresia este convertită la tipul numit.

Operatorul cast are aceeaşi precedenţă cu a unui operator unar.

Exemplu:

Astfel funcţia sqrt() din biblioteca <math.h> cere un argument double, deci va fi apelată cu un cast: sqrt((double) n) pentru a calcula rădăcina pătrată a lui **n**.

2.5 Variabile. Declararea variabilelor

O variabilă este o dată care îşi poate modifica valoarea pe parcursul execuţiei programului.

În limbajul C/C++, înainte de a utiliza o variabilă, trebuie să o declarăm. La declarare, trebuie să specificăm numele variabilei, tipul acesteia şi, eventual, o valoare iniţială pe care dorim să o atribuim variabilei.

Formatul general al unei declaraţii de variabile este:

tip nume_var$_1$ [=expresie$_1$] [, nume_var$_2$ [=expresie$_2$]...];

Observaţii

1. Prin *tip* specificăm tipul variabilelor care se declară.
2. Prin *nume_var*$_1$, *nume_var*$_2$, specificăm numele variabilelor care se declară (acestea sunt identificatori).
3. Se pot declara simultan mai multe variabile de acelaşi tip, separând numele lor prin virgulă.
4. La declarare, putem atribui variabilei o valoare iniţială, specificând după numele variabilei caracterul '=' şi o expresie de iniţializare. Expresia trebuie să fie evaluabilă în momentul declarării.
5. Parantezele [] utilizate în descrierea formatului general au semnificaţia că elementul încadrat între paranteze este opţional (poate să apară sau nu într-o declarație de variabile).

Exemple

```
int a, b=3, c=2+4;
char z;
float x=b*2.5, y;
```

Am declarat trei variabile a, b şi c de tip int, o variabilă z de tip char şi două variabile x şi y de tip float. Variabilei b i-am atribuit valoarea iniţială 3, variabilei c i-am atribuit valoarea 6, iar variabilei x i-am atribuit valoarea 7. 5;

Variabilelor a, y şi z nu le-am atribuit nicio valoare iniţială la declarare.

Declararea unei variabile trebuie să preceadă orice referire la variabila respectivă şi poate fi plasată în interiorul unei funcţii (în cazul nostru, al funcţiei main ( )) sau în exteriorul oricărei funcţii (în cazul nostru, în exteriorul funcţiei main ( )). Dacă declaraţia este plasată în interiorul unei funcţii, variabila se numeşte locală funcţiei, altfel se numeşte globală.

Variabilele globale sunt automat iniţializate cu 0; cele locale nu sunt iniţializate.

La declararea variabilelor nu se admit iniţializări multiple (produc eroare de sintaxă):

Greşit **(deoarece variabilele *b* şi *c* nu sunt încă declarate)**

```
int a=b=c=5;
```

Corect

```
int a, b, c;
a=b=c=5;
```

2.6. Definirea constantelor

Pentru definirea constantelor simbolice se foloseşte (în zona de preprocesare) construcţia:

```
#define nume valoare
```

sau folosind modificatorul const astfel:

```
const tip nume=valoare;
```

Exemplul:

```
#define PI 3.1415;
const float PI=3.1415;
```

2.7. Structura unui program C/C++. Comentariile

Orice program C/C++ este alcătuit dintr-o succesiune de module (numite funcţii),una dintre acestea fiind funcţia principală numită **main()**.

Forma generală a unei surse în C/C++ este:

```
int main()
{
 …..// corpul funcţiei în care se vor scrie declaraţiile şi instrucţiunile care trebuie executate şi care acum e vid
   return 0;
}
```

Când execuţia unui program se termină cu succes, în mod uzual, programul returnează la încheierea execuţiei valoarea **0**.

Programul va conţine şi o serie de fişiere antet (headere) în cazul în care vrem să folosim funcţii standard din fişierele respective.

Zona de preprocesare apare în partea de sus a sursei şi este introdusă de caracterul '**#**'. În această zonă vor fi incluse fişiere antet (headere) din care vor fi folosite funcţii standard (pentru citire/scriere , prelucrări de date, funcţii matematice, etc.)

Exemplu:

```
# include <iostream.h> // am inclus fişierul antet iostream.h care conţine funcţii pentru citire/scriere
```

Observaţie:

Dacă realizăm noi fişiere pe care vrem să le includem în antet acestea vor fi incluse între ghilimele ca de exemplu: **#** include "fisierul_meu.cpp"

În zona de preprocesare se pot defini şi constante simbolice de forma:

```
# define PI 3.1415
```

Comentariile

Reprezintă o succesiune de caractere pe care compilatorul (cel care transcrie codul sursă în limbaj maşină) nu le ia în considerare.

Există în C/C++ două tipuri de comentarii:

- pe mai multe linii cu marcajul /*…*/
- pe o singură linie (de la marcaj până la sf. de linie) cu marcajul //

Sunt utilizate pentru a creşte gradul de lizibilitate a programului şi ajută utilizatorii multiplii la înţelegerea programului.

2.8 Expresii. Instrucţiunea de atribuire

Expresiile sunt formate din operanzi şi operatori. *Operanzii* reprezintă valorile care intră în calculul expresiei iar *operatorii* desemnează operaţiile care se execută în expresie.

În timpul execuţiei unui program, la întâlnirea unei expresii calculatorul evaluează expresia astfel: se înlocuiesc variabilele cu valorile lor şi se obţine valoarea expresiei.

Expresiile pot fi *simple* sau *compuse* (se grupează în paranteze rotunde).

Tipuri (clase) de operatori care pot să apară în expresii:

Operatorii aritmetici : +,-,*,/,%

Operatorul	Tipul operaţiei
+, -	Semne (operatori unari)
*, /,%	De multiplicitate (operatori binari)
+,-	Aditivi (operatori binari)

Observaţii:

Operatorul '%' se poate aplica numai datelor de tip întreg.

Dacă operatorul '/' se aplică cel puţin unui operand de tip real atunci rezultatul este real (idem pentru operatorii +,-,*)

Exemplu:

```
int a=5, b=2;
cout<<-a;          // afişează -5
cout<<a/b;         // 2
cout<<(float)a/b;  // 2.5
cout<<5/2.0;       //2.5
```

Operatorii logici sunt de două tipuri: logici globali şi logici pe biţi (sunt operatori binari toţi în afară de negaţia logică şi pe biţi care sunt unari)

- Operatori logici globali (&&, ||, !)

&&	0	!0=1
0	0	0
!0=1	0	1

||	0	!0=1
0	0	1
!0=1	1	1

!	
0	1
!0=1	0

Observaţie:

Operatorii logici globali se pot aplica datelor de orice tip standard, rezultatul fiind întotdeauna logic.

Exemplul 1:	Exemplul 2:	Exemplul 3:	Exemplul 4:	Exemplul 5:	Exemplul 6:

int a=5,b=3;
if(a>0)
cout<<a; //5
else
cout<<b;
	int a=5,b=3;
	if (a)//⇔ a!=0
	cout<<a; //5
	else
	cout<<b;
		int a=5,b=3;
		if(a&&b)
		cout<<a+b;
		//8
		else
		cout<<a-b;
			int a=5,b=3;
			if(a>0 && b<0) //1&&0==0
			cout<<a+b;
			else
			cout<<a-b; //2
				int a=5,b=3;
				if(a>0 || b<0)
				//1||0==1
				cout<<a+b; //8
				else
				cout<<a-b;
					int a=4;
					cout<<(a<0)<<"
					"<<!a<<" " <<
					(a==4); // 0 0 1

- Operatori logici pe biţi (&, |, ~, ^)

şi pe biţi

&	0	1
0	0	0
1	0	1

sau pe biţi

|	0	1
0	0	1
1	1	1

negația pe biţi

~	
0	1
1	0

sau exclusiv sau XOR

^	0	1
0	0	1
1	1	0

Exemplul &	Exemplul |	Exemplul ~	Exemplul ^

int a=10,b=5;
//a=1010_2 b=101_2
cout<<a&b; // 0
int a=10,b=6;
//a=1010_2 b =110_2
	int a=10,b=5;
	// a=1010_2 b=0101_2
	// a|b= 1111_2=15
	cout<<a|b; // 15
	int a=10,b=6;
		int a=10;
		cout<<~a;
		// 32758
			int a=10,b=5;
			//a=1010_2 b=0101_2
			//a^b= 1111_2=15
			cout<<a^b; // 15
			int a=10,b=6;

cout<<a&b; //2	// a=1010_2 b=0110_2
	// a|b= 1110_2= 14
	cout<<a|b; // 14
			//a=1010_2 b=0110_2
			//a^b= 1100_2= 12
			cout<<a^b; // 12

Operatori relaționali (<,>,<=,>=,!=) se aplică datelor de tip standard şi returnează un rezultat de tip logic.

Observație: Negarea lui > este <=, negarea lui < este >=, negarea lui <= este >, negarea lui >= este <.

Exemplu:

```
int a=5,b=3;
if(a>b)   // a<b  sau  a>=b  sau  a<=b  sau a!=b
   cout<<a;   //5
else
   cout<<b;
```

Operatorul de egalitate (==)

Exemplu:

```
int a=5,b=5;
if(a==b)
  cout<<"Da";   // Da
else
  cout<<"Nu";
```

Observație: Operatorul '**==**' apare doar în expresii logice

Operatori de incrementare/decrementare ++,--

Au două forme: *prefixă* şi *postfixă*

Observaţii:

- Incrementarea (++) / decrementare(--) presupune modificarea cu o unitate a valorii variabilei
- la forma prefixă variabila intră în expresie cu valoarea modificată pe când la forma postfixă variabila se modifică în urma evaluării expresiei
- operatorii de incrementare/decrementare nu pot fi aplicaţi expresiilor
  ++(a+b) // greşit

Exemple:

```
int a=5;
cout<<++a<<" "<<a++;   // a=a+1 =>6   6=>a=a+1=7
int b=3,c=4;
cout<<(a+b)-c++;   // 10-4=6 =>c=5
cout<<(a+b)-(++c); // 10-5=5 =>c=5
int j=5, i=6;
```

```
j=j+(++i);  // j=5+7=12
cout<<j;
j=j+(i++);  // j=12+7=19  şi i=8
cout<<j;
```

Operatori de deplasare pe biţi (<<, >>)

Sunt operatori binari care realizează deplasarea pe biţi la stânga respective la dreapta astfel:

a<<b (reprezintă în baza 2 numărul **a** şi îl deplasează la stânga cu un număr de poziţii binare egal cu valoarea operandului din dreapta adică a lui **b**). Se va completa şirul în dreapta cu un număr de 0-uri egal cu valoarea lui b (b de 0).

Exemplu:

12_{10}<<2=00000000 00001100_2 <<2=00000000 00110000_2=1*2^4+1*2^5=48

12%2=0 ↑
6%2=0 |
3%2=1 |
1%2=1 |
0

Observaţie: Matematic operaţia de deplasare la stânga pe biţi se transcrie: a<<b=a*2^b

a>>b (reprezintă în baza 2 numărul a şi îl deplasează la dreapta cu un număr de poziţii binare egal cu valoarea operandului din dreapta adică a lui b). Se va completa şirul în în stânga cu un număr de 0-uri egal cu valoarea lui b (b de 0).

Exemplu:

12>>3=00000000 00001100_2 >>3=00000000 00000001_2=1*2^0=1

12%2=0 ↑
6%2=0 |
3%2=1 |
1%2=1 |
0

Observaţie: Matematic operaţia de deplasare la dreapta pe biţi se transcrie: a>>b=[a/2^b]

Operatorul condiţional (?:)

expresie1 ? expresie2 : expresie3

Efect: se evaluează *expresie1*, dacă rezultatul logic este adevărat (1) se execută *expresie2* altfel se execută *expresie3*.

Exemplu:

```
int x=-4;
cout<<(x<0 ? -x : x);    // afişează modulul lui x
```

Exemplu:
int a=4,b=2;
cout<<(a>b ? a : b); // afişează maximul dintre valorile a şi b

Operatorul referenţial (&)
Efect: se utilizează pentru a obţine adresa de memorie la care este salvată o variabilă
Exemplu:
int a;
cout<<&a; // rezultatul este un număr în baza 16 reprezentând adresa de memorie a lui a

Operatorul pentru determinarea spaţiului de memorie ocupat (în octeţi)- are două forme: sizeof(tip) şi respectiv sizeof(variabilă)
Exemplu:
int a=20;
cout<<sizeof(int)<<endl; // 2 (reprezentând spaţiul ocupat în memorie)
cout<<sizeof(a); // 2 (reprezentând spaţiul ocupat în memorie)

Operatorul de conversie explicită (tip)
Efect: este utilizat pentru conversia forţată a unei variabile sau expresii la un tip standard "tip"
Exemplu:
float c;
int a=5,b=6;
c=(a+b)/2;
cout<<c; // 5 rezultat de tip întreg
c=(float)(a+b)/2;
cout<<c; // 5.5
c=(a+b)/2.0;
cout<<c; // 5.5

Operatorul , (virgulă)
Sintaxa: $expresie_1$, $expresie_2$,…, $expresie_n$
Rezultatul aplicării operatorului constă în evaluarea expresiilor de la stânga la dreapta, valoarea finală a expresiei fiind valoarea ultimei expresii evaluate.
Exemplu:
int i, a, b;
i=0, b=i+2, a=b*2 // valoarea expresiei este 4

Evaluarea oricărei expresii se realizează astfel: dacă există paranteze rotunde se vor evalua începând cu cea mai din interior (spre exterior). Dacă nu există paranteze rotunde evaluarea se va face în funcție de prioritatea operatorilor iar la prioritate egală în funcție de asociativitate.

Prioritate	Operator	Asociativitate
1	! ~ + - (semne) ++ -- (typecast) sizeof & (referenţial)	dreapta -> stânga
2	*/ %	stânga -> dreapta
3	+ - (op aditivi)	stânga -> dreapta
4	<< >>	stânga -> dreapta

5	< > <= >=	stânga -> dreapta
6	!= ==	stânga -> dreapta
7	& (logic pe biţi)	stânga -> dreapta
8	|	stânga -> dreapta
9	&& (logic global)	stânga -> dreapta
10	||	stânga -> dreapta
11	?:	dreapta -> stânga
12	+=, -=,*=,/=,%=,&=,|=,!=,<<=,>>=	dreapta -> stânga
13	, (virgulă)	stânga -> dreapta

Instrucţiunea de atribuire are sintaxa:

identificator_variabilă = valoare

Unde *identificator_variabilă* este numele variabilei iar *valoare* (care poate fi constantă, variabilă sau expresie) trebuie să aibă tipul compatibil cu cel al variabilei pentru a putea efectua atribuirea.
Efect: Variabilei *identificator_variabilă* i se va atribui valoarea *valoare.*

Exemplu:

```
int a=3,b=5,c=7;
c=(a>=b);   // c=0 şi se numeşte atribuire cu sens logic
c=(a*b-c);  // c= 15
```

În C/C++ se admit atribuiri multiple de forma:

identificator_variabilă$_1$ = identificator_variabilă$_2$ =…=valoare

Într-o astfel de atribuire, atribuirile se realizează de la dreapta la stânga.

Observaţie:
Atribuirile nu se admit în declarare de variabile.

Exemplu:
int a=b=c=5; // greşit deoarece b şi c nu sunt declarate

Exemplu:

```
int a=2, x, y, z;
x=(y=a-1)=z=a;
```

Se va efectua datorită parantezelor atribuirea y=1 iar apoi se vor efectua atribuirile multiple de la stânga la dreapta: x=y=z=2

Instrucţiunea de atribuire poate avea şi formele:

identificator_variabilă *operator* = expresie ⇔

identificator_variabilă = identificator_variabilă *operator* expresie

Unde operatorul poate fi: **+, -, *, /, %, <<, >>**

Exemplu:

```
int a=7;
a+=5 ;  //  a=a+5⇔ a=12
```

2.9 Citirea şi afişarea datelor (operaţii de intrare/ieşire)

Există citire şi scriere cu şi fără format. În continuare ne vom referi la citirea şi scrierea fără format (din C++).

Fluxul de intrare a datelor este “cin”-*console input* (de la tastatură) iar operatorul de extragere care preia datele de intrare şi le salvează în variabile este “>>”.

Sintaxa: cin>>id_variabilă;

Fluxul de ieşire a datelor este “cout”-console output (pe monitor) iar operatorul de inserţie de date în fluxul se ieşire este “<<”.

Sintaxa: cout<<expresie;

Citirea şi scrierea de date este gestionată din fişierul antet iostream.h

Observaţie:

- Citirea datelor de la tastatură se face cu caractere albe (spaţiu, tab) pe care operatorul >> le ignoră.
- Atât citirea cât şi scrierea datelor se poate face înlănţuit după cum se poate observa în exemplul de mai jos:

Exemple:

```
// 3 numere întregi citite/ afişate unul după altul
int a,b,c;
cin>>a>>b>>c;                    // De ex:   7 12 34 \n
cout<<a<<” ”<<b<<” ”<<c;         // se vor afişa valorile 7 12 34 (cu spaţiu între ele)
// 3 numere întregi citite separat, cu spaţiu între ele
int a,b,c;
cout<<”a=”;
cin>>a;              // a= 7 \n
cout<<”b=”;
cin>>b;            // b=12 \n
cout<<”c=”;
cin>>c;            // c=34 \n
cout<<a<<b<<c;   // se vor afişa valorile 71234 (fără spaţiu între ele)
// afișez valoarea din variabila b
cout<<”Valoarea variabilei b este: ”<<b;
// Va afişa pe ecran -> Valoarea variabilei b este: 12
```

2.10 Structuri de control (Structura liniară, Structura alternativă, Structura repetitivă)

Conform teoremei de structură a lui Bohm-Jacoppini există 3 structuri de control

Structura liniară se transcrie în C/C++ prin două instrucţiuni:

1. **Instrucţiunea de atribuire care are sintaxa:**

   identificator_variabilă = valoare

   Unde *identificator_variabilă* este numele variabilei iar *valoare* (care poate fi constantă, variabilă sau expresie) trebuie să aibă tipul compatibil cu cel al variabilei pentru a putea efectua atribuirea.

   Efect: Variabilei *identificator_variabilă* i se va atribui valoarea *valoare.*

   Exemplu:

```
int a=3,b=5,c=7;
c=(a>=b);   // c=0 şi se numește atribuire cu sens logic
c=(a*b-c);  // c= 15
```

În C/C++ se admit atribuiri multiple de forma:
identificator_variabilă$_1$ = identificator_variabilă$_2$ =…=valoare
Într-o astfel de atribuire, atribuirile se realizează de la dreapta la stânga.
Exemplu:

```
int a=2, x, y, z;
x=(y=a-1)=z=a;
```

Se va efectua datorită parantezelor atribuirea y=1 iar apoi se vor efectua atribuirile multiple de la stânga la dreapta: x=y=z=2

Instrucțiunea de atribuire poate avea şi formele:
identificator_variabilă *operator* = expresie ⇔
identificator_variabilă = identificator_variabilă *operator* expresie
Unde operatorul poate fi: **+, -, *, /, %, <<, >>**
Exemplu:

```
int a=7;
a+=5   //  a=a+5⇔ a=12
```

2. **Instrucţiunea compusă care are sintaxa:**

```
{
  instr_1;
  …
  instr_n;
}
```

Observaţie : instrucţiunea compusă se utilizează atunci când într-o instrucţiune de decizie sau repetitivă o ramură **conţine mai mult de o instrucţiune.**

Exemplu:

```
….
int a,b,c;
if(a>b)          // dacă rezultatul evaluării expresiei logice este „adevărat”
{                // trebuie executate două atribuiri (grupate într-o instrucţiune compusă)
 a=a+b;
 c=a+b;
}
else
  cout<<”Nimic”;
```

Structura alternativă se transcrie în C/C++ prin două instrucţiuni:

1. **Instrucţiunea de decizie simplă (**transcrie din pseudocod instrucţiunea **dacă… atunci**)

Sintaxa:

```
if (expresie)
```

```
    instr1;
[ else                opţională
    instr2;]
```

Instrucţiunea evaluează *expresia logică*, dacă este adevărată (1) execută *intr1* iar dacă nu execută *instr2* dacă ramura de altfel este prezentă iar în caz contrar nu execută nimic.

Exemplu 1: Determinarea valorii maxime dintre 2 valori întregi a şi b citite (cu ramură de else)

```
…
int a,b;
cin>>a>>b;
if (a>b)
  cout<<a;
else
  cout<<b;
```

Exemplul 2: Determinarea valorii maxime dintre 2 valori întregi a şi b citite (fără ramură de else)

```
…
int a,b,max;
cin>>a>>b;
max=a;
if (max<b)
  max=b;
cout<<max;
```

Observaţie:

Întotdeauna ramura de else dacă există se va asocia celui mai din interior if.

Exemplu: Evitarea unei asocieri nedorite pentru ramura de *else* se poate realiza în două moduri:

cu ramură de else vidă

```
…
int a,b,c;
cin>>a>>b>>c;
if(a>b)
 if(b>c)
   cout<<a;
 else;   //instr. vidă – nu execută nimic
else
  cout<<"a este mai mic decât b";
```

cu instrucţiune compusă

```
…
int a,b,c;
cin>>a>>b>>c;
if(a>b)
{
  if(b>c)
    cout<<a;
}
else
  cout<<"a este mai mic decât b";
```

2. **Instrucţiunea de decizie multiplă (switch)**

Se utilizează atunci când decizia se ia în urma evaluării mai multor expresii (şi este unică)

Sintaxa

```
switch (expresie)
{
  case  constanta_1: instr_1;break;
  …
  case  constanta_n : instr_n;break;
  [default: instr;]
}
```

Observaţii:

- expresie nu poate avea alt tip decât întreg sau character.
- constantele pot să se găsească în valoarea expresiei sau nu
  - ✓ caz în care se execută instrucţiunea de pe ramura default (dacă există , altfel nu se execută nimic)
  - ✓ dacă instrucţiunea break (ieşire necondiţionată) nu există atunci se vor executa toate instrucţiunile de la prima constantă egală cu valoarea expresiei.

Structura repetitivă se transcrie în C/C++ prin 3 instrucţiuni:

Instrucţiuni repetitive

- cu număr necunoscut de paşi
  - cu test iniţial (while)
  - cu test final (do…while)

Instrucţiunea while

Sintaxa: while(condiţie)

[{]

instrucţiuni;

[}]

Execuţie: Se verifică la fiecare pas condiţia şi dacă dacă rezultatul logic este 1 (TRUE) se vor executa în ordinea în care apar instrucţiunile iar în caz contrar se iese din repetitivă.

Observaţii:

- instrucţiunea while poate să nu se execute niciodată dacă valoarea de adevăr a condiţiei este 0.
- dacă în corpul instrucţiunii while avem mai mult de o instrucţiune, aceste se grupează într-o instrucţiune compusă

Instrucţiunea do…while

Sintaxă: do

{

```
    instrucţiuni;
  } while(condiţie);
```

Execuţie: Se vor executa în ordinea în care apar instrucţiunile apoi se verifică condiţia şi dacă rezultatul logic este 1 (TRUE) se continuă cu execuţia instrucţiunilor iar în caz contrar se iese din repetitivă.

Observaţie: Diferenţa între repetitive cu test iniţial şi cea cu test final este aceea că repetitiva cu test final se execută întotdeauna cel puţin o dată.

Instrucţiune repetitivă cu număr cunoscut de paşi (for)

Sintaxa: for(expresie1;expresie2;expresie3)
[{]
instrucţiuni;
[}]

Efect:

- se va evalua o singură dată la intrarea în repetitivă expresie 1
- expresie2 constituie condiţia de ieşire din repetitivă (este o expresie logică)
- expresie3 modifică valoarea contorului pentru a asigura ieşirea din repetitivă (de obicei)

Observaţii: expresiile din **for** pot lipsi, în schimb caracterul ";" e obligatoriu

Exemplu:

```
for(;;);   // ciclează la infinit
```

Exemplu:

```
for(i=1,j=2;;);  // iniţializează cu 1 pe i şi cu 2 pe j şi ciclează la infinit

for(i=1,j=2;i<n;i=i+2);// i=7 , n=6   1,3,5,7 , j=2

//  i=5  , n=5     1,3,5,  j=2
```

Exemplu

```
…
s=0;                                                    s=0;
for(;x!=0;x=x/10)          // realizează suma cifrelor lui x ⇔   while(x!=0)
  s=s+x%10;                                                        {
                                                                     s=s+x%10;
                                                                     x=x/10;
                                                                   }
```

Exemplu

```
for(s=0;x!=0;s=s+x%10,x=x/10);   // realizează suma cifrelor lui x
```

Aplicaţii:

1) Se considera n număr natural. Calculaţi şi afişaţi numărul de cifre.

Subprograme (Funcţii)

2 tipuri:

- returnează un rezultat
- calculează mai multe valori, nu returnează nimic

Sintaxa:

tip identificator_fc([lista de parametri formali])

{

  Return expresie; // expresie are tipul *tip*

}

Lista de p.f : *tip* id_var,…, *tip* id_var

Apelul funcţiei:

- se face în interiorul unei expresii de forma:
  variabilă=id_fc([lista de parametri efectivi]);
  sau cout<< id_fc([lista de parametri efectivi]);

lista de parametri efectivi: id_var1,…,id_varn

Sintaxa:

Void identificator_fc([lista de parametri formali])

{

}

Lista de p.f : *tip* id_var,…, *tip* id_var

Apelul funcţiei:

Id_fc([listă de parametri efectivi]);

1) int suma(int n)

```
{
 int s=0;
While(n!=0)
    {
     s=s+n%10;
     n=n/10;
    }
    return s;
}
```

```
void suma(int n)
    {
     int s=0;
    While(n!=0)
        {
         s=s+n%10;
         n=n/10;
        }
        cout<<s;
    }
```

apelul în main…

suma(n);

apelul în main…

```
int sum=suma(n);
```

sau

```
cout<<suma(n);
```

Transmiterea parametrilor:

- prin valoare(se poate modifica parametrul în interiorul funcţiei, la ieşire se revine la valoarea iniţială)
- prin referinţă(se modifică valoarea de la adresa stabilită a parametrului, la ieşire valoarea modificată rămâne)

```
void suma(int n, int &s)
   {
   While(n!=0)
     {
      s=s+n%10;
      n=n/10;
     }
    }
```

apelul în main…

```
int s=0;
suma(n,s);
```

```
cout<<setw(4)<<a[i][j];
cout<<setprecision(2)<<a[i];
```

a[i]⇔ *(a+i) a⇔&a[0]

Tipul pointer

Am învăţat încă din clasa a IX-a, că o variabilă este o dată caracterizată prin trei atribute: *tip, valoare şi adresă*. Primele două atribute au fost foarte întâlnite până acum. Atributul "adresă" va face obiectul noţiunilor din acest capitol. Pentru început, considerăm un exemplu foarte simplu, o variabilă de tip întreg care se declară astfel:

```
int x;   // se rezervă în memorie 2 octeţi
```

Toate variabilele unui program sunt memorate în aşa-numita memorie internă RAM. Aceasta este împărţită în nişte "căsuţe", numite locaţii de memorie sau celule de memorie. Fiecare locaţie reprezintă un octet şi se caracterizează printr-o anumită adresă. Adresele locaţiilor de memorie sunt numere exprimate în (baza 16).

În urma declarării unei variabile, se rezervă pentru aceasta un număr de octeţi succesivi în memoria internă RAM (locaţii succesive), număr care depinde de tipul variabilei şi de implementare.

Spaţiul de memorie alocat unei variabile rămâne ocupat pe toată durata programului sau subprogramului (funcţiei) în care a fost declarată variabila. O astfel de variabilă se numeşte "variabilă alocată static", iar procesul de rezervare de memorie este de către compilator şi se numeşte "alocare statică a memoriei".

De ce se alocă memorie pentru variabila x declarată mai sus?

După declarare, variabila x poate primi ca valori numere întregi, prin atribuire sau prin citire. În momentul în care variabila a primit o valoare, acea valoare va fi memorată în zona de memorie alocată variabilei.

O altă posibilă declarare a variabilei x este următoarea: declaram o variabilă x care va conţine adresa de memorie a unui număr întreg (valoarea variabilei x va fi adresa la care se va „depozita” numărul întreg în memoria RAM

Variabila x se numeşte „*pointer către un întreg*” sau „*referinţă către un întreg*”. Fireşte că tipul de date al variabilei x nu va mai fi *int*, ci un tip de date numit tipul pointer către un întreg sau tipul referinţă către un întreg; acesta se declară astfel:

```
int *x;  // x este un pointer către un întreg
```

Caracterul ’*’ reprezintă operatorul de adresare, este un operator unar, adică se aplică unui singur operand.

Observaţie:

Operatorul de adresare este identic ca şi scriere cu cel de înmulţire dar compilatorul nu va face confuzie deoarece operatorul de înmulţire este binar.

Sintaxa de declarare pointerilor este:

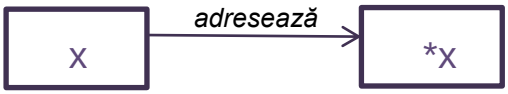
```
tip *identificator1,*identificator2,…* identificatorn;
```

unde tip este tipul de date pe care-l referă pointerii iar $identificator_1$,$identificator_2$,… $identificator_n$ sunt identificatorii variabilelor de tip pointer

Exemple:

```
float *a, *b;  // a şi b sunt pointeri către tipul float (pointeri către numere reale)
char *c ;      // c este pointer către tipul caracter
```

Dacă pointerul x conţine adresa unui număr, numărul se va memora efectiv într-o "variabilă pereche" (*x) pe care o creează automat compilatorul când declarăm pointerul x. "Perechea" ***x** se numeşte **variabilă alocată dinamic**.



adresa numărului întreg (pointer către număr)

numărul întreg (variabila alocată dinamic)

Caracteristicile fundamentale ale unei perechi "pointer - variabila alocată dinamic":

- În secţiunea de declaraţii vom defini un pointer care ulterior va conţine adresa unei variabile alocată dinamic;
- Pentru variabila alocată dinamic vom rezerva memorie prin program,în momentul folosirii efective a variabilei, iar atunci când variabila nu mai este necesară vom elibera zona de memorie ocupată (folosirea eficientă a memoriei).

Observaţie:
În timpul execuţiei unui program, s-ar putea ca un pointer să memoreze adresa unei locaţii de memorie "goală", în care nu se găseşte nicio valoare. În acest caz vom spune că valoarea sa este NULL. Acest NULL reprezintă de fapt valoarea "0 binar" simbolizată printr-o constantă predefinită a limbajului, cu sensul de "nimic" (nul), şi poate fi atribuit numai variabilelor de un tip pointer.

Alocarea şi eliberarea dinamică a memoriei

Alocarea memoriei pentru o variabila referită de un pointer, sau pe scurt alocarea dinamică a memoriei, se realizează în program înainte de utilizarea efectivă a variabilei, iar după ce aceasta nu mai este necesară trebuie să eliberam memoria ocupată.
În C++ alocarea şi eliberarea memoriei se poate face în două moduri:

1. Cu ajutorul funcţiilor **malloc** respectiv **free** (specifice componentei C standard);
2. Cu operatorii identificaţi prin cuvintele cheie **new** respectiv **delete** (specifici extensiei "++" a limbajului).

1. Alocarea şi eliberarea memoriei cu **malloc** şi **free**

Sintaxa: identificator_pointer=(tip*)malloc(dimensiune);
Unde:
identificator_pointer este numele pointerului;
tip este tipul de date spre care trimite pointerul;
dimensiune este dimensiunea spaţiului de memorie (exprimată în număr de octeţi) necesară pentru memorarea variabilei pereche spre care trimite pointerul.

Exemplu:
Fie un pointer x către întreg, declarat astfel:

int *x; // se declară dar nu i se rezervă automat spaţiu în memorie pentru perechea *x
x=(int*) malloc(2); // se alocă 2 octeţi

Pentru a evita orice incertitudini cu privire la mărimea spaţiului de memorie necesar pentru memorarea valorilor de un anumit tip, este recomandabil să folosim operatorul sizeof.

Astfel **alocarea de memo**rie de mai sus devine:

x=(int*) malloc(sizeof(int));

Observaţie:

Chiar dacă în instrucţiunea de alocare apare pointerul x, se alocă memorie pentru variabila ataşată *x, şi nu pentru pointerul x.

Eliberarea spaţiului de memorie ocupat de variabila referită prin pointerul *identificator_pointer* este:

free(identificator_pointer);

2. Alocarea şi eliberarea memoriei cu **new** şi **delete**

Alocarea de memorie se face astfel:

identificator_pointer= new tip;

Eliberarea spaţiului de memorie se face astfel:

delete identificator_pointer;

Exemplu:

int *x; //se declară dar nu i se rezervă automat spaţiu în memorie pentru perechea *x
x= new int; // se alocă 2 octeţi
delete x; // eliberează spaţiul de memorie alocat mai sus cu new

Observaţie:

Variabilele referite prin pointeri pot fi folosite la fel ca şi variabilele alocate static în operaţii de citire, afişare, expresii, etc.

Exemplu:

Calculul ariei unul triunghi folosind pointeri: se consideră doi pointeri b şi h referind valorile bazei şi înălţimii unui triunghi. Calculaţi şi afişaţi aria triunghiului.

```
#include<iostream.h>
void main()
{
 int *b,*h;
 float *A;
 b=new int;
 h=new int;
 A=new float;
 cout << "Dati baza si inaltimea triunghiului: ";
 cin >> *b >> *h;
*A=(*b**h) /2.0;
 cout << "\nAria triunghiului este:  " << *A;
 delete b;
 delete h;
 delete A;
```

```
 return 0;
}
```

Adresa unei variabile alocată static. Operatorul &

O variabilă alocată static x, de tipul int, va putea primi ca valoare un număr întreg. În felul acesta accesăm direct valoarea şi nu adresa variabilei.

Un pointer x către întreg (de tipul int *) va conţine adresa unui întreg (iar întregul se va memora în *x). Astfel, avem acces direct la adresa întregului.

Pentru o variabilă alocată static putem determina adresa sa folosind operatorul ’&’. Astfel, adresa unei variabile x alocată static (adresa de memorie la care se va depozita valoarea variabilei este &x).

Exemplu:

```
int a;
int *x;
a=5;
x=&a;
cout<<*x;  // se va afişa valoarea 5
```

Observaţie:

Nu s-a alocat şi nici eliberat memoria pentru pointerul x pentru că este exact zona de memorie ocupată de variabila a care se eliberează la sfârşitul execuţiei programului.