



UNIVERSITATEA
BABEŞ-BOLYAI

100
O SUTĂ DE ANI DE
UNIVERSITATE
ROMÂNEASCĂ
1919-2019
UNIVERSITATEA BABEŞ-BOLYAI

UBBFSEGA
Universitatea Babeş-Bolyai | Facultatea de Ştiinţă Economică şi Gestură Afacerilor

FSEGA
CLUJ-NAPOCĂ



CENTRUL DE FORMARE CONTINUĂ,
ÎNVĂȚĂMÂNT LA DISTANȚĂ ȘI CU
FRECUENȚĂ REDUSĂ

Specializarea: Informatică Economică

SUPORT DE CURS **INTRODUCERE ÎN PROGRAMAREA CALCULATORELOR**

ANUL I Semestrul 2



CUPRINS

1. Informații generale despre curs, seminar, lucrare practică sau laborator	4
1.1. Informații despre curs	4
1.2. Condiționări și cunoștințe prerezchizite	4
1.3. Descrierea cursului	4
1.4. Organizarea temelor în cadrul cursului	4
1.5. Formatul și tipul activităților implicate de curs	4
1.6. Materiale bibliografice obligatorii:	5
1.7. Materiale și instrumente necesare pentru curs	5
1.8. Calendar al cursului	5
1.9. Politica de evaluare și notare	6
1.10. Elemente de deontologie academică	6
1.11. Studenți cu dizabilități:	6
1.12. Strategii de studiu recomandate:	7
2. Suportul de curs	
2.1. Modul 1: NOȚIUNI INTRODUCTIVE DESPRE ALGORITMI	8
2.1.1. Scopul și obiectivele modulului	8
2.1.2. Conținutul informațional detaliat	8
2.1.2.1. Noțiunea de algoritm; caracteristici	8
2.1.2.2. Reprezentarea algoritmilor	13
2.1.2.3. Definirea limbajelor de programare	18
2.1.3. Sumar	20
2.1.4. Întrebări recapitulative	20
Întrebări pentru testul scris	
Intrebări grilă:	
2.1.5. Bibliografie modul	22
2.2. Modul 2: LIMBAJUL DE PROGRAMARE C	23
2.2.1. Scopul și obiectivele modulului	23
2.2.2. Scurtă recapitulare a conceptelor prezentate anterior	23
2.2.3. Conținutul informațional detaliat	23
2.2.3.1. Scurt istoric al limbajului C	23
2.2.3.2. Structura unui program și a unei funcții C	24
2.2.3.3. Etapele realizării unui program	27
2.2.3.4. Funcții de intrare/ieșire	29
2.2.3.5. Funcții de citire/scriere caractere	33
2.2.3.6. Compilare și Precompilare	34
2.2.3.7. Construcții de bază în C	38
2.2.3.8. Instrucțiuni	44
2.2.3.9. Expresii	55
2.2.3.10. Regula conversiilor implicate	63
2.2.3.11. Variabile	64
2.2.3.12. Comentarii	67
2.2.4. Probleme propuse	68
2.2.5. Sumar	71

2.2.6. Întrebări recapitulative	71
Întrebări pentru testul scris	
Intrebări grilă	
2.2.7. Bibliografie modul	77
 2.3. Modulul 3: NOȚIUNI AVANSATE DE LIMBAJ C	78
2.3.1. Scopul și obiectivele modului	78
2.3.2. Scurtă recapitulare a conceptelor prezentate anterior	78
2.3.3. Conținutul informațional detaliat	78
2.3.3.1. Structuri	78
2.3.3.2. Asignari de nume pentru tipuri de date	81
2.3.3.3. Uniuni	82
2.3.3.4. Câmpuri de biți	83
2.3.3.5. Enumerări	84
2.3.3.6. Pointeri și adrese	86
2.3.3.7. Aritmetică adreselor	89
2.3.3.8. Transmiterea argumentelor la apelurile de funcții	91
2.3.3.9. Pointeri și tablouri	93
2.3.3.10. Pointeri pe caractere și functii	95
 2.3.3.11. Pointeri pe funcții	99
2.3.3.12. Argumentele funcției main	100
2.3.3.13. Operații cu fișiere	101
2.3.3.14. Lucrul cu fișiere la nivel de siruri de caractere	102
 2.3.3.15. Prelucrarea fișierelor binare	
106	
2.3.4. Sumar	109
2.3.5. Întrebări recapitulative	109
Întrebări pentru testul scris	
Intrebări grilă	
2.3.6. Bibliografie modul	113
 Bibliografia completă a cursului	113

1. Informații generale despre curs, seminar, lucrare practică sau laborator

1.1. Informații despre curs:

Date de contact ale titularului de curs:

Nume: Lector dr. Cristian Bologa
Birou: 431 sediul Fac. de Științe Economice și Gestiunea Afacerilor, str. Teodor Mihali 58-60
Telefon: 0264-418652
Fax: 0264-412570
E-mail: cristian.bologa@econ.ubbcluj.ro
Consultații: Conform cu orarul afișat la sala 431

Date de identificare curs și contact tutori:

Numele cursului: Introducere în programarea calculatoarelor
Codul cursului: ELR0078
Anul, Semestrul: anul 1, sem.2
Tipul cursului(oblig., optional, facult.): obligatoriu
Număr credite: 4
Pagina web a cursului:
<https://cursuri.elearning.ubbcluj.ro/course/view.php?id=246>
Tutori : Lector dr. Cristian Bologa
cristian.bologa@econ.ubbcluj.ro
Asist. drd. Paula Zalhan
paula.zalhan@econ.ubbcluj.ro

1.2. Condiționări și cunoștințe prerechizite

Înscrierea la acest curs nu este condiționată de parcurgerea altor discipline.

1.3. Descrierea cursului

Cursul vizează următoarele aspecte:

- a) Insușirea gândirii algoritmice;
- b) Descrierea algoritmilor folosind scheme logice sau pseudocod
- c) Codificarea utilizând limbajul C

1.4. Organizarea temelor în cadrul cursului

Disciplina este structurată în 3 module de învățare:

1. Noțiuni introductory despre algoritmi
2. Noțiuni introductory în limbajul C
3. Noțiuni avansate în limbajul C

1.5. Formatul și tipul activităților implicate de curs

Parcurgerea acestei discipline va presupune atât întâlniri față în față, cât și muncă individuală. Astfel, metodele utilizate pe parcursul predării cursului sunt: expunerea teoretică, prin mijloace auditive și vizuale; explicația abordărilor conceptuale;

rezolvări de probleme; răspunsuri directe la întrebările studenților. În ceea ce privește activitatea cursanților, se va încuraja participarea activă a studenților prin problematizarea informațiilor prezentate, implicarea în rezolvarea problemelor propuse; găsirea de soluții alternative la problemele propuse.

Studentul are libertatea de a-și gestiona singur, fără constrângeri, modalitatea și timpul de parcurgere a cursului. Este însă recomandată parcurgerea succesivă a modulelor prezentate în cadrul suportului de curs, în ordinea indicată și rezolvarea sarcinilor sugerate la finalul fiecărui modul.

1.6. Materiale bibliografice obligatorii:

Cristian Bologa –Algoritmi și structuri de date, Editura Risoprint, 2006.

Liviu Negrescu, Limbajele C și C++ pentru începători , Vol. I (p.1 si 2) - limbajul C (editia XI), Editura Albastră, Cluj-Napoca, 2005

D. Knuth - Arta programării calculatoarelor, vol, 1, 2, 3, ed. Teora, 1999 (traducere)

1.7. Materiale și instrumente necesare pentru curs

În vederea participării la un nivel optim la activitățile cursului, este recomandat ca studenții să aibă acces la următoarele resurse:

- calculator conectat la internet (pentru a putea accesa conținutul cursului și pentru a putea participa interactiv pe parcursul derulării acestuia);
- un mediu de programare ce utilizează un compilator C sau C++
- imprimantă
- acces la resursele bibliografice (abonament la Biblioteca Centrală Universitară);
- acces la echipamente de fotocopiere.

1.8. Calendar al cursului

Pe parcursul semestrului sunt programate 4 întâlniri față în față cu toți studenții.

În vederea eficientizării acestor întâlniri față în față, pentru fiecare din acestea, **se recomandă parcurgerea de către student a suportului de curs** pus la dispoziție încă de la începutul semestrului, iar ulterior întâlnirii, este indicată rezolvarea sarcinilor și exercițiilor aferente fiecărui modul parcurs. De asemenea, anterior întâlnirilor programate, studenților li se recomandă să parcurgă capitolele corespunzătoare temelor abordate la fiecare întâlnire din cel puțin una din sursele bibliografice indicate. În acest mod, se va facilita orientarea cursului asupra

aspectelor de finețe din conținutul disciplinei și se va permite concentrarea pe modalitățile de aplicare la nivel practic a informațiilor deja parcurse.

1.9. Politica de evaluare și notare

Evaluarea studenților se va efectua conform detalierei de mai jos:

Nota pe partea teoretică: examen scris din sesiunea de examene, care va consta dintr-un set de întrebări din materia predată la curs (40%)

Nota aferentă părții practice: nota de la examenul practic sustinut în sesiunea de examene (40%)

Nota pe activitate: activitatea atât la întâlnirile față în față cât și la întâlnirile la distanță vor constitui 20% din nota finală.

Promovarea examenului este condiționată de obținerea notei 5 la fiecare dintre primele 2 componente.

1.10. Elemente de deontologie academică

Se vor avea în vedere următoarele detalii de natură organizatorică:

- Orice tentativă de fraudă sau fraudă depistată va fi sancționată prin acordarea notei minime sau, în anumite condiții prin exmatriculare.
- În condițiile în care un student a promovat acest examen la o altă facultate, în măsura în care programa analitică coincide, cadrul didactic titular al disciplinei poate decide recunoașterea notei obținute.
- Rezultatele examenelor vor fi comunicate în maxim 5 zile, la avizierul catedrei, pe pagina web și pe adresa de email de grup.
- Contestațiile se vor soluționa în maxim 24 de ore de la comunicarea rezultatelor.

1.11. Studenți cu dizabilități:

Titularul cursului și tutorii își afirmă disponibilitatea, în limita posibilităților, de a adapta la cerere, conținutul și metodelor de transmitere a informațiilor, precum și modalitățile de evaluare (examen oral, examen practic) în funcție de tipul dizabilității cursantului. Vom urmări facilitarea accesului egal al tuturor cursanților la activitățile didactice.

1.12. Strategii de studiu recomandate:

Se recomandă parcurgerea sistematică a modulelor cuprinse în cadrul cursului, punându-se accent pe pregatirea individuală continuă a studenților și pe evaluările formative pe parcursul semestrului. Se recomandă cursanților alocarea unui număr de cel puțin 6 ore săptămânal pentru parcurgerea și însușirea cunoștințelor necesare promovării cu succes a acestei discipline.

2. Suportul de curs

2.1. Modul 1: NOȚIUNI INTRODUCTIVE DESPRE ALGORITMI

2.1.1. Scopul și obiectivele modulului

După parcurgerea acestui modul, cursanții vor cunoaște noțiunile introductive din domeniul algoritmilor.

2.1.2. Conținutul informațional detaliat

2.1.2.1. Noțiunea de algoritm; caracteristici



Algoritm: un instrument de rezolvare a *problemelor*. O problemă se consideră a fi constituită din *date* de intrare și un enunț care specifică relația existentă între datele de intrare și *soluția* problemei.

În cadrul algoritmului sunt descrise prelucrările necesare pentru a obține soluția problemei pornind de la datele de intrare.



Un algoritm este o succesiune bine precizată de prelucrări care aplicate asupra datelor de intrare ale unei probleme permit obținerea în timp a soluției acesteia.

Exemplu:

- rezolvarea ecuației de gradul doi,
- ciurul lui Eratostene (pentru generarea numerelor prime mai mici decât o anumită valoare),
- schema lui Horner (pentru determinarea câtului și restului împărțirii unui polinom la un binom) etc.

Soluția problemei se obține prin *execuția* algoritmului. Algoritmul poate fi executat pe o mașină formală (în faza de proiectare și analiză) sau pe o mașină fizică (calculator) după ce a fost implementat într-un limbaj de programare.

Spre deosebire de un program, care depinde de un limbaj de programare, un algoritm este o entitate matematică care este independentă de mașina pe care va fi executat.

Exemplu: Algoritmul de aflare a sumei dintre două numere

Pas 0 : START

Pas 1 : Citim numerele a,b

Pas 2 : S := a + b

Pas 3 : Afisam rezultatul : S

Pas 4 : STOP



Soluția unei probleme, din punct de vedere informatic, este dată printr-o mulțime de comenzi (instrucțiuni) explicite și neambigue, exprimate într-un limbaj de programare.

*Această mulțime de instrucțiuni prezentată conform anumitor reguli sintactice formează un **program**.*

Algoritmul descrie soluția problemei independent de limbajul de programare în care este redactat programul.

Un algoritm este compus dintr-o mulțime finită de pași, fiecare necesitând una sau mai multe operații.

Un algoritm nu operează numai cu numere. Pe lângă algoritmii numeric există și algoritmii algebrici și algoritmii logici.

Sintetic, un algoritm și un program pot fi definiți astfel :

*Program= exprimarea într-un limbaj de programare a unui **algoritm***

*Algoritm= exprimarea într-un limbaj de reprezentare a unui **raționament***



Un **program** este o descriere precisă și concisă a unui algoritm într-un limbaj de programare.

De aceea, noțiunile de « algoritm » și « program » se folosesc uneori greșit ca sinonime.



*Algoritmizarea este o cerință fundamentală în rezolvarea oricărei probleme cu ajutorul calculatorului. Experiența a demonstrat că nu orice problemă poate fi rezolvată prin *algoritmizarea rezolvării*, adică prin descrierea unui algoritm de rezolvare.*

Problemele se pot împărti în 2 clase:

- *clasa problemelor decidabile* (o problemă este decidabilă dacă există un algoritm pentru rezolvarea ei)
- *clasa problemelor nedecidabile* (o problemă este nedecidabilă dacă nu există un algoritm pentru rezolvarea ei).

Există probleme care au apărut ulterior apariției calculatoarelor. Astfel a apărut “*problema celor 4 culori*” conform căreia orice hartă poate fi colorată folosind patru culori, astfel încât oricare două țări cu frontiere comune să fie colorate diferit. Enunțarea problemei a fost făcută în anul 1852; problema a fost rezolvată în anul 1977 doar prin utilizarea calculatorului și prin utilizarea unei metode noi (*metoda Backtracking*).

Există algoritmi cu caracter general, pentru clase largi de probleme și algoritmi specifici unor probleme particulare.

Principalele categorii de algoritmi cu caracter general sunt:

- algoritmi de împărțire în subprobleme (“Divide et Impera”),
- algoritmi de căutare cu revenire (“Backtracking”),
- algoritmi de optim local (“Greedy”),
- algoritmi de programare dinamică etc.



Rezolvarea unei probleme dintr-un anumit domeniu reprezintă o *activitate ce presupune existența unor procese*:

-**proces demonstrativ** (*demonstrația*) caresă arate existența unei soluții sau a mai multor soluții și să determine efectiv *soluțiile exacte*;

-**proces computațional** (*algoritm*) care să codifice un proces demonstrativ, o metodă sau o tehnică de rezolvare în scopul *determinării aproximative* a soluțiilor exacte.



Principalele **caracteristici** ale unui algoritm sunt:

-generalitatea – algoritmul trebuie să fie cât mai general astfel ca să rezolve o clasă cât mai largă de probleme și nu o problemă particulară sau punctuală. El trebuie să poată fi aplicat oricărui set de date inițiale ale problemei pentru care a fost întocmit.

Exemplu : algoritmul de rezolvare a ecuației de gradul II $ax^2+bx+c=0$ trebuie să rezolve toate cazurile pentru o mulțime infinită de date de intrare (a, b și c aparținând numerelor reale).

-claritatea – acțiunile algoritmului trebuie să fie clare, simple și rigurose specificate.
Un algoritm trebuie să descrie cu precizie ordinea operațiilor care se vor efectua.

Fiecare programator trebuie să respecte anumite reguli de editare a programelor.

Literatura de specialitate prevede mai multe reguli de indentare a unui program:

- instrucțiunile unei secvențe se vor scrie aliniate, începând de la aceeași coloană;
- instrucțiunile unei instrucțiuni compuse se vor scrie începând toate din aceeași coloană, aflată cu 2-4 caractere la dreapta față de începutul instrucțiunii compuse;
- pe o linie pot fi scrise mai multe instrucțiuni, cu condiția ca ele să aibă ceva comun. Astfel, pentru a obține un program mai compact, 2 până la 4 instrucțiuni scurte de atribuire pot fi scrise pe același rând.
- denumirile variabilelor să fie astfel alese încât să reflecte semnificația acestor variabile.
- programul trebuie amplu comentat, prin inserarea comentariilor în text.

Se consideră că nu trebuie să existe reguli stricte de editare a unui program. În schimb, fiecare programator trebuie să aibă propriile lui reguli de scriere, care să fie unitare pe tot parcursul programului și care să ofere claritate programului.

-finitudinea – un algoritm trebuie să admită o descriere finită și să conducă la soluția problemei după un număr finit de operații.

-corectitudinea – un algoritm trebuie să poată fi aplicat și să producă un rezultat corect pentru orice set de date de intrare valide.

Corectitudinea este de 2 tipuri:

corectitudine totală (faptul că pentru orice date de intrare algoritmul determină valori corecte de ieșire)

parțială (finititudinea algoritmului pentru orice set de date de intrare).

Verificarea corectitudinii unui algoritm se poate face folosind:

- varianta experimentală prin testarea algoritmului
- varianta analitică se bazează pe demonstrarea funcționării corecte a algoritmului pentru orice date de intrare, garantând astfel corectitudinea.

- performanța – algoritmul trebuie să fie eficient privind resursele utilizate și anume să utilizeze memorie minimă și să se termine într-un timp minim.

- robustețea – reprezintă abilitatea algoritmului de a recunoaște situațiile în care problema ce se rezolvă nu are sens și de a se comporta în consecință (de exemplu, prin mesaje de eroare corespunzătoare).

Un algoritm robust nu trebuie să fie afectat de datele de intrare eronate.

-extensibilitatea - posibilitatea adaptării programului la unele schimbări în specificațiile problemei.

-reutilizabilitatea -este posibilitatea reutilizării întregului program sau a unor părți din el în alte aplicații.

-compatibilitatea -presupune ușurința de combinare cu alte produse program.

-portabilitate -este posibilitatea de folosire a produsului program pe alte sisteme de calcul, diferite de cel pe care a fost conceput.

-eficiență -reprazintă măsura în care sunt folosite eficient resursele sistemului de calcul.

Analiza mai presupune utilizarea unor tehnici de demonstrare specifice matematicii.

-testarea programului –conține două faze: depanare (debugging) și trasare (profiling). Depanarea este procesul rulării unui program pe diverse seturi de date de test și corectarea eventualelor erori depistate. Trasarea este procesul de rulare pas cu pas a execuției unui program, pe diverse seturi de date de test, pentru a urmări evoluția valorii unor variabile și a putea depista astfel eventuale erori logice ale programului.

Structura unui algoritm este constituită din următoarele *elemente de bază*:

- *Date - variabile și tipuri de date* utilizate pentru accesul la memorie și generarea de valori conform calculelor implementate în procesul de calcul prin intermediul instrucțiunilor;
- *Expresii* - este alcătuită din unul sau mai mulți operanzi, legați între ei prin operatori. Operanzii pot fi constante sau variabile;
- *Instrucțiuni* - *instrucțiuni sau comenzi executabile* pentru operații Input/Output și operații de prelucrare a datelor din memorie conform procesului de calcul;
- *Proceduri și funcții* - *subprocese de calcul* cu o structură asemănătoare unui algoritm ce pot fi executate prin așa-numitele *instrucțiuni de apelare*.



Practica dezvoltării aplicațiilor software arată următoarele faze:

1. **specificarea problemelor** – descrierea clară și precisă a problemelor indiferent din ce domeniu provin acestea. Se impune deci definirea unui enunț precis al problemei.
2. **projecțarea soluțiilor** – includerea problemelor în clasa de probleme corespunzătoare și alegerea modului de reprezentare a problemelor prin formularea *etapelor* și *procedeeelor* corespunzătoare pentru procesele de rezolvare;
3. **implementarea soluțiilor** – *elaborarea algoritmilor* și *codificarea* acestora într-un limbaj de programare modern. După codificare, are loc obținerea formei executabile a programului.
4. **analiza soluțiilor** – *eficiența soluțiilor* raportată la resursele utilizate: *memorie, timp, utilizarea dispozitivelor I/O, etc.*;
5. **testarea și depanarea** – *verificarea execuției programului* cu diverse seturi de date de intrare pentru a putea răspunde rezolvării oricărei probleme pentru care aplicația a fost elaborată. Această etapă presupune și adaptarea soluțiilor implementate pentru *eliminarea erorilor* în rezolvarea unei anumite probleme și *compatibilitatea* cu sistemul de calcul și sistemul de operare folosite.
6. **documentarea** – în majoritatea cazurilor, programele sunt utilizate de alte persoane decât cele care le-au elaborat. Pentru o utilizare corectă a lor, este necesară întocmirea unei documentații a programului care conține în general descrierea problemei, schema de sistem, schemele logice, programul sursă, instrucțiuni de utilizare. Deseori utilizarea programului este ilustrată figurativ, folosind exemple concrete.

7. **exploatarea, actualizarea și întreținerea** – exploatarea presupune utilizarea curentă a programului în rezolvarea cazurilor concrete din clasa de probleme pentru care a fost proiectat. Actualizarea și întreținerea au și un aspect corectiv, de a elimina eventualele erori descoperite pe parcursul exploatarii programului.

Fazele 1 și 2 formează etapa de **analiză și proiectare** iar fazele 3-7 cea de **programare-execuție**.

2.1.2.2. Reprezentarea algoritmilor

Algoritmii pot fi specificați:

- În limbaj natural
- Pseudocod
- Scheme logice
- Diagrame arborescente
- Tabele de decizie

Pseudocod



Limbajul pseudocod are o sintaxă și semantică asemănătoare limbajelor de programare moderne, având o anumită flexibilitate în ceea ce privește sintaxa, în ideea că prin codificarea unui algoritm într-un limbaj de programare, operația să fie cât mai comodă. Semantică pseudocodului este apropiată de limbajele de programare utilizând însă cuvinte și expresii uzuale din limbajul natural.

În dicționarul de informatică pseudocodul este definit ca « limbaj utilizat în proiectarea și documentarea programelor obținut prin grefarea unor reguli sintactice pe limbajul natural. Structurile de control sunt reprezentate prin folosirea unor cuvinte cheie (dacă ... atunci ... altfel ... execută ... până când etc) și printr-o anumită aliniere în pagină a liniilor ».

Limbajul pseudocod are două tipuri de propoziții: propoziții standard (corespond structurilor de control) și propoziții nestandard (texte ce conțin părți ale algoritmului încă incomplet elaborate, nefinisate). Comentariile în pseudocod se includ între acolade. Propozițiile standard încep cu cuvinte cheie și fie se scriu cu litere mari, fie se subliniază.

Scheme logice



Prin *schemă logică* se înțelege o reprezentare grafică a unui algoritm în care fiecărui pas i se atașează un simbol denumit *bloc*. Ordinea de parcurgere a blocurilor în schemele logice se precizează cu ajutorul săgeților.

Exprimarea unui algoritm cu scheme logice sau cu pseudocod este echivalentă.

Într-o schemă logică se utilizează următoarele tipuri de blocuri:

Blocul de inceput/sfarsit - orice schema logica incepe cu un bloc de inceput si se termina cu blocul de stop		PROCEDURE ENDPROCEDURE
Blocul de citire (doua variante) - se citesc de la dispozitivul de intrare valorile variabilelor specificate in lista_variabile		READ lista_variabile
Blocul de scriere (doua variante) - se scriu la dispozitivul de iesire valorile obtinute in urma evaluarii expresiilor din lista		WRITE lista_expresii
Blocul de atribuire - se evaluateaza expresia, iar valoarea obtinuta este memorata in variabila, vechea valoare a variabilei pierzandu-se		Variabila = expresie
Blocul de decizie - se evaluateaza conditia: daca e adevarata se continua cu prelucrarea indicata de ramura da , altfel cu ramura nu ;		IF conditie THEN Secvență instructiuni ELSE Secvența instructiuni ENDIF
Blocul conector are forma de cerc. El se foloseste pentru a conecta diferite secvențe ale unei scheme logice. Sunt utile atunci când se continua descrierea algoritmului pe mai multe pagini.		



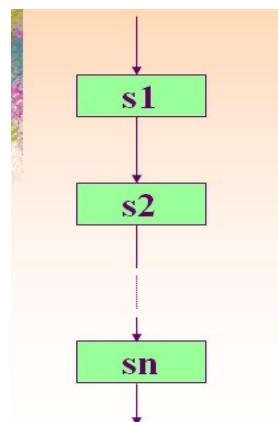
Orice schemă logică care are un singur punct de intrare și un singur punct de ieșire poate fi reprezentată cu ajutorul a trei structuri de bază: **structura secvențială (liniară)**, **structura alternativă**, **structura repetitivă**

O schemă logică construită numai cu structuri de acest tip este numită *schemă logică structurată*. Un algoritm structurat conține doar cele 3 structuri enumerate mai sus.

Structura secvențială

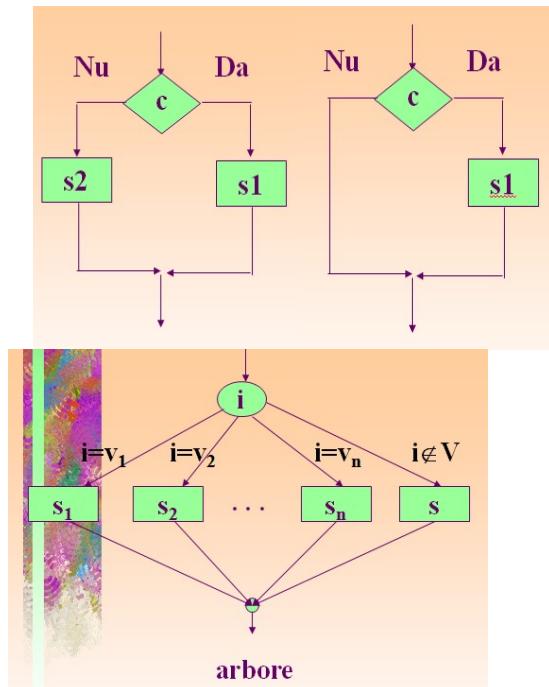
Reprezintă un proces de calcul format dintr-o operație elementară, cum este citirea unei valori sau o operație de atribuire, dar poate fi și o combinație de alte structuri. Structura secvențială indică execuția succesivă a operațiilor de bază și a structurilor de control în ordinea în care apar în schema logică; în general, orice schemă logică cuprinde secvențele:

- citirea datelor
- inițializarea variabilelor
- prelucrări
- tipărire rezultatelor



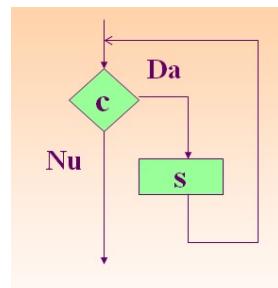
Structura alternativă

În funcție de valoarea de adevăr a condiției, se execută una din secvențe, după care se trece la prelucrarea următoare; cele două ramuri se exclud mutual; este posibil ca una din ramuri să fie vidă.



Structura repetitivă cu condiționare anterioară (WHILE-DO)

Secvența se execută ciclic, cât timp condiția este adevarată; dacă la prima evaluare a condiției, aceasta este falsă, corpul nu se execută niciodată. În cadrul secvenței A este necesar să se modifice valoarea unei variabile care să afecteze valoarea de adevăr a expresiei c. În caz contrar, vom ajunge la un ciclu infinit.



In pseudocod avem:

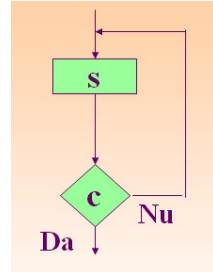
```
WHILE condiție
    Secvență instrucțiuni
ENDWHILE
```

Cele 3 structuri prezentate (secvențială, alternativă și repetitivă cu condiționare anterioară) sunt denumite structuri de bază. Orice algoritm poate fi reprezentat folosind cele 3 structuri de bază.

În continuare prezentăm încă 2 tipuri de structuri, care sunt folosite în practică, dar care au la bază și deci pot fi înlocuite de structurile de bază.

Structura repetitivă cu condiționare posterioară (REPEAT-UNTIL)

Condiția se evaluează după o primă execuție a secvenței (deci secvența se execută cel puțin o dată); se revine la execuția secvenței, dacă nu este adevărată. Dacă condiția este adevărată se încheie această secvență.



In pseudocod avem:

```

REPEAT
    Secvență instructiuni
UNTIL condiție

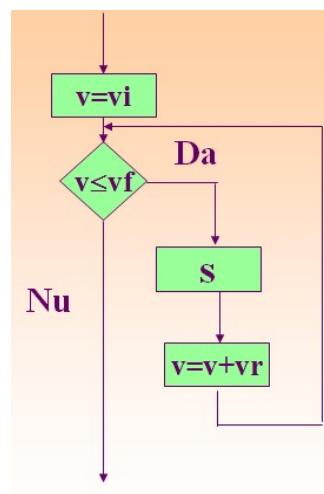
```

Structura repetitivă cu un număr cunoscut de pași (FOR)

O variabilă numită generic contor, controlează ciclarea; contorul se inițializează cu o valoare inițială (valvă), iar ciclarea se realizează cat timp contor $\leq vf$, o valoare finală.

La sfârșitul corpului ciclului, variabila contor este actualizată, fiind mărită; pentru a nu se cicla la infinit, trebuie ca pasul de mărire (vr) să fie pozitiv.

Structura se poate folosi și cu o valoare finală mai mică decât cea inițială, caz în care avem pasul negativ. Această structură este o particularizare a structurii WHILE-DO.



```

FOR limite de iterare
    Secvență instructiuni
ENDFOR

```

2.1.2.3. Definirea limbajelor de programare

Comunicarea interumană este indispensabilă în viața de zi cu zi. Pentru realizarea acesteia, oamenii folosesc diverse modalități, limbajele reprezentând poate cea mai importantă facilitate de comunicare. Una din primele aptitudini importante pe care un copil o dobândește este să vorbească. El învăță limba maternă. De obicei, aceasta îi este suficientă pentru comunicarea de zi cu zi cu alți indivizi din propria societate. Dacă însă, un individ călătorește, el are nevoie să cunoască și alte limbi pentru a se putea descurca, pentru a putea comunica cu alți oameni din alte societăți ale lumii. Cu toate că, în principiu, oamenii atribuie înțelesuri similare pentru lucruri similare, ei vorbesc (exprimă) aceste înțelesuri în limbi diferite. Diferența se manifestă atât la nivelul sunetelor și înlănțuirii acestora pentru a forma cuvintele cât și la nivelul simbolurilor grafice folosite. A învăța să vorbim într-o altă limbă presupune a învăța să rostим cuvintele din limba respectivă, să le înlănțuim de o manieră potrivită conform regulilor gramaticale a limbii învățate și să folosim simbolurile specifice acestei limbi pentru a descrie vizual sintagmele de comunicare. Dacă nu mânuim corespunzător aceste elemente, atunci când vom fi puși în situația de a comunica cu cineva care înțelege doar limba respectivă, vom eșua în încercarea noastră.

Am realizat această scurtă introducere pentru a putea realiza o comparație între actele de comunicare inter-umană și comunicarea dintre om și calculator. În această comunicare, putem vedea calculatorul ca pe un partener care e dispus să ne rezolve problemele. Pentru aceasta, trebuie să-i specificăm modul în care să rezolve aceste probleme. Putem să realizăm acest lucru utilizând algoritmii. Dar algoritmii trebuie descriși într-un limbaj inteligibil pentru calculator. Putem vedea această comunicare ca un caz special al unei solicitări. Dacă cerem ceva cuiva, atunci va trebui să folosim cuvinte, expresii pe care acesta să le înțeleagă.



Conform Dicționarului Explicativ al Limbii Române, prin *limbă* se înțelege un sistem de comunicare alcătuit din sunete articulate, specifice omului, prin care acesta își exprimă gândurile sau dorințele. Astfel, un limbaj este un mijloc de comunicare a ideilor prin sunete și culoare, reprezentând un mijloc de transmitere a informației între indivizii unei categorii.

Orice limbaj are la bază simboluri care formează limbajul respectiv. Astfel, limba vorbită are la bază sunete, limbajele scrise au la bază literele, limbajele vizuale au la bază simbolurile grafice. Semiotica este ramura științei care se ocupă cu studiul simbolurilor.

Revenind la comunicarea dintre om și calculator, trebuie să definim noțiunea de limbaj de programare.



Astfel, prin *limbaj de programare* înțelegem o notație sistematică prin care este descris un proces de calcul.

Un proces de calcul este constituit dintr-o mulțime de pași pe care o mașină îi poate executa pentru a rezolva o anumită problemă.

Astfel, un limbaj de programare este un intermedian între realitatea reprezentărilor utilizatorului asupra problemei de rezolvat și

realitatea calculatorului cu care lucrează [Şerbănaşti 1987]. La rezolvarea unei probleme cu calculatorul, un programator trebuie să privească fiecare element al limbajului din 2 puncte de vedere:

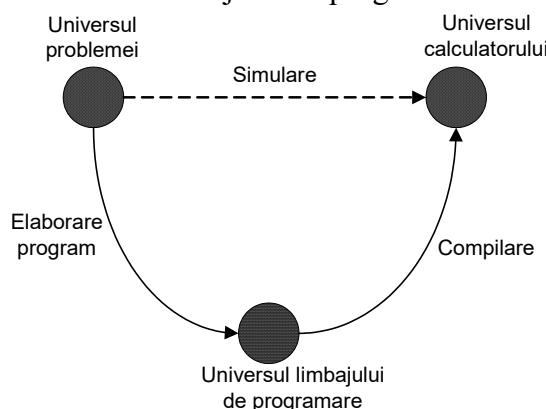
- unul logic, al problemei; astfel programatorul trebuie să ştie ce înțelege să reprezinte din problemă cu ajutorul elementului de limbaj
- unul fizic, al implementării, care se referă la ceea ce realizează calculatorul la execuția elementului de limbaj considerat.

Programul este un compromis între cele 2 puncte de vedere. Astfel, programatorul trebuie să-şi reprezinte şi să înțeleagă următoarele universuri:



- universul problemei
- universul limbajului de programare
- universul calculatorului

Acste trei universuri sunt în general diferite. Pentru a realiza corespondența între acestea, există definite diverse activități ale informaticii. Figura 1 surprinde corespondența între universurile limbajelor de programare.



Universurile unui limbaj de programare [Şerbănaşti 1987].

Orice limbaj are 3 aspecte caracteristice:



- aspectul sintactic
- aspectul semantic
- aspectul pragmatic

Sintaxa unui limbaj conține ansamblul regulilor prin care pornind de la simbolurile de bază care alcătuiesc alfabetul limbajului, se construiesc structuri compuse.

Mulțimea regulilor sintactice care descriu ansamblul propozițiilor sau a formulelor corecte din cadrul limbajului formează *gramatica*.

Deci, sintaxa și gramatica ne ajută să identificăm modul în care putem combina simbolurile de bază ale limbajului pentru a produce elemente acceptate de limbaj. Ele reprezintă imperative riguroase, care pot fi formalizate matematic. Sintaxa se descrie teoretic cu ajutorul sistemelor formale. Următoarele elemente se utilizează pentru descrierea sintaxei unui limbaj:

- diagramele sintactice

- arbori de analiză
- metalimbaje (BNF – Backus Naur Form, EBNF – Extended Backus Naur Form, Asn.1)

Prin ***semantică*** se înțelege sensul construcțiilor sintactice. Ea reprezintă un set de reguli ce determină semnificația propozițiilor dintr-un limbaj. Este vorba de reguli de evaluare a acestor propoziții în termenii unor mulțimi de valori cunoscute de limbajul respectiv. Astfel, semantica reprezintă înțelesul fiecărei formule corecte admise de gramatică.

Pragmatica se referă la capacitatea de a utiliza construcțiile sintactice și semantice. Referitor la înțelegerea aspectului pragmatic al limbajelor, putem să se închipuim următorul exemplu: o persoană poate cunoaște foarte bine aspectele sintactice și semantice ale unui limbaj, dar nu are capacitatea de a utiliza corect aceste reguli. De asemenea, există persoane care vorbesc (folosesc) o limbă fără a cunoaște aspectele sintactice și semantice ale acesteia. Pragmatica nu se poate formaliza.

In consecință, fiecare problemă o vom aborda în felul următor:

1. vom descrie algoritmul care rezolvă problema fie în pseudocod, fie cu scheme logice
2. vom transpune punctual algoritmul în limbajul de programare ales
3. ne vom asigura că rezolvarea furnizată se executează în mod corect în limbajul de programare ales

2.1.3. Sumar

Acest modul face o introducere în problematica algoritmilor.

2.1.4. Întrebări recapitulative

Întrebări pentru testul scris:

1. Ce este un algoritm?
2. Ce caracteristici are un algoritm?
3. Ce caracteristici suplimentare are un program?
4. Care sunt etapele dezvoltării unei aplicații software?
5. Ce este pseudocodul?
6. Ce este o schemă logică?
7. Care sunt blocurile utilizate într-o schemă logică?
8. Ce este structura liniară?
9. Ce este structura alternativă?
10. Ce este structura repetitivă cu condiționare anterioară?
11. Ce este structura repetitivă cu condiționare posterioară?
12. Care sunt etapele elaborării unui program?
13. Ce este un limbaj de programare?
14. Ce este sintaxa?
15. Ce este semantica?
16. Ce este pragmatica?



Intrebări grilă:

1. Ce este un algoritm?

- a) O rețetă de rezolvare a oricărei probleme
- b) O succesiune bine precizată de prelucrări care aplicate asupra datelor de intrare ale unei probleme permit obținerea soluției acesteia
- c) Un program scris într-un anumit limbaj de programare care rezolvă o anumită problemă

2. Care este rezolvarea firească a unei probleme:

- a) se scrie algoritmul apoi se elaborează programul
- b) se scrie programul apoi se elaborează algoritmul
- c) oricare din cele două de mai sus

3. Un algoritm poate fi:

- a) finit
- b) infinit
- c) poate fi și finit și infinit

4. Un algoritm este independent de mașina pe care va fi executat:

- a) adevarat
- b) fals
- c) nu se poate preciza

5. O problemă este decidabilă dacă:

- a) putem decide soluția ei
- b) putem găsi un algoritm pentru rezolvarea ei
- c) putem găsi o singură soluție la rezolvarea problemei

6. Editarea unui program (indentarea) este obligatorie oricare ar fi limbajul de programare folosit:

- a) adevărat
- b) fals
- c) nu se poate preciza

7. Corectitudinea unui program înseamnă:

- a) obținerea soluției corecte pentru setul de date de intrare folosit
- b) obținerea soluției corecte pentru orice set de date de intrare
- c) folosirea unui set de date de intrare corect

8. Portabilitatea unui program înseamnă:

- a) posibilitatea de a rula fără erori programul pe orice mașină de calcul
- b) posibilitatea de a muta programul de pe o mașină pe alta
- c) posibilitatea de a exporta rezultatele de pe o mașină pe alta

9. Care din următoarele propoziții este falsă:

- a) Trasarea este procesul de rulare pas cu pas a execuției unui program, pe diverse seturi de date de test, pentru a urmări evoluția valorii unor variabile și a putea depista astfel eventuale erori logice ale programului
- b) Reutilizabilitatea unui program este posibilitatea de a refolosi datele de intrare la o nouă rulare a programului
- c) Performanța unui program presupune să utilizeze memorie minimă și să se termine într-un timp minim

10. Care din următoarele propoziții este adevărată:

- a) Orice algoritm poate fi reprezentat folosind cele 3 structuri de bază (secvențială, alternativă și repetitivă)
- b) În cazul structurii repetitive cu condiționare anterioară, secvența cuprinsă în ciclul repetitiv se va executa cel puțin o dată
- c) Sintaxa înseamnă sensul construcțiilor sintactice

11. Care din următoarele propoziții este adevărată:

- a) structura repetitivă cu număr cunoscut de pași este un caz particular al unei structuri repetitive condiționată anterior
- b) în cazul structurii repetitive cu număr cunoscut de pași contorul trebuie să fie obligatoriu număr pozitiv
- c) în cazul unei structuri repetitive cu număr cunoscut de pași nu vom putea ajunge la un ciclu infinit

12. Care din următoarele propoziții este adevărată:

- a) testarea și depanarea unui program permite depistarea erorilor sintactice ale programului
- b) un program poate fi scris în pseudocod sau într-o schemă logică
- c) în totdeauna este de preferat să lucrăm structurat într-un program, adică să folosim doar structurile de bază din programare

2.1.5. Bibliografie modul

Cristian Bologa –Algoritmi și structuri de date, Editura Risoprint, 2006 –capitolul 1.

2.2. Modul 2: LIMBAJUL DE PROGRAMARE C

2.2.1. Scopul și obiectivele modulului

Scopul acestui modul este deprinderea conceptelor de bază din programare utilizând limbajul C.

2.2.2. Scurtă recapitulare a conceptelor prezentate anterior

Primul modul urmărește introducerea în problematica algoritmilor.

2.2.3. Conținutul informațional detaliat

2.2.3.1. Scurt istoric al limbajului C

- dezvoltat și implementat în 1972 la AT&T Bell Laboratories de Dennis Ritchie <http://cm.bell-labs.com/cm/cs/who/dmr/chist.html>
- limbaj de **programare structurat**(blocuri, cicluri, funcții)(concept apărut în ALGOL 60, apoi ALGOL 68, PASCAL, ...)
- necesitatea unui limbaj pentru **programe de sistem**(legătura strânsă cu **sistemul de operare UNIX** dezvoltat la Bell Labs)
- C dezvoltat inițial sub UNIX; în 1973, UNIX rescris în totalitate în C
- cartea de referință: Brian Kernighan, Dennis Ritchie:**The C Programming Language** (1978)
- în 1988 limbajul a fost standardizat de ANSI(American National Standards Institute)
- limbaj de nivel **mediu**: oferă tipuri, operații, instrucțiuni simple fără facilitățile complexe ale limbajelor de nivel (foarte) înalt (nu: tipuri multime, concatenare de siruri, etc.)
- limbaj de programare **structurat** (funcții, blocuri)
- permite programarea **la nivel scăzut**, apropiat de hardware, acces la reprezentarea binară datelor
- mare libertate în lucrul cu memoria foarte folosit în programarea de sistem, interfață cu hardware
- produce un cod **eficient** (compact în dimensiune, rapid la rulare) apropiat de eficiența limbajului de asamblare datorită caracteristicilor limbajului, și maturității compilatoarelor
- **slab tipizat!** -necessită mare atenție în programare, conversii implicate și explicate între tipuri, char e tip întreg, etc.

2.2.3.2. Structura unui program și a unei funcții C



O funcție C este un modul care grupează în interiorul unei perechi de acolade un set de operații codificate sub forma unor instrucțiuni.

Observații:

- Fiecare funcție poate accepta parametri de intrare la apel și poate returna o valoare la revenire.
- Fiecare funcție are un nume precedat de un cuvânt cheie care desemnează tipul funcției (tipul valorii returnate de funcție).
- Numele funcției este urmat de opareze rotunde între care se specifică tipul și numele parametrilor funcției. Parantezele sunt necesare chiar dacă nu există parametri.
- În limbajul C nu este permisă definirea unei funcții în interiorul altrei funcții (lucru care este permis în limbajul Pascal). În limbajul C, toate funcțiile trebuie definite în mod independent.



Un program C se compune din una sau mai multe funcții, dintre care una este funcția principală. Fiecare funcție are un nume propriu, cu excepția funcției principale care se numește *main*. Orice program trebuie să aibă o funcție main iar execuția programului începe cu prima instrucțiune din această funcție.

Structura generală a unei funcții Ceste următoarea:

```
tipreturn numefunctie(lista_parametri)
{
    instrucțiuni specifice funcției: declarare variabile locale,instrucțiuni executabile
    return expresie;
}
```

Dacă tipul returnat de funcție nu este specificat, acesta este implicit considerat *int*. Dacă lista parametrilor nu este vidă, fiecare parametru poate fi eventual specificat doar prin numele său, tipurile parametrilor fiind precizate prin declarații care preced corpul funcției.

Prima linie este headerul (antetul) funcției iar ceea ce este inclus între acolade se numește corpul funcției.

Corpul funcției este un bloc care poate conține definiții, declarații de variabile locale și instrucțiuni necesare realizării scopului funcției. De altfel, corpul funcției poate fi considerat o instrucțiune compusă care descrie prelucrările necesare pentru a ajunge la valoarea funcției pornind de la valorile parametrilor.

Rezultatul acțiunii funcției este dat de expresia conținută de instrucțiunea (sau instrucțiunile) return aflată în corpul funcției.

Există funcții care nu au parametri și/sau care nu returnează nici un rezultat. Specificarea acestui aspect se face utilizând cuvântul cheie *void*. Utilizarea lui *void* pentru a indica o listă de parametri vidă este redundantă în C++.



Afișarea unui mesaj pe ecranul utilizator necesită prezența unei instrucțiuni *printf* (*Exemplul 1*).

Exemplu 1:
#include<stdio.h>
int main()
{
 printf("Salut anul 2IE!\n");
 return 0;
}

Observații:

- prima linie: obligatorie pentru orice program care citește sau scrie, este o directivă de preprocesare, include fișierul *stdio.h* care conține declarațiile funcțiilor standard de intrare/ieșire – adică informațiile (nume, parametri) necesare compilatorului pentru a le folosi corect
- *printf* (“*print formatted*”): o funcție standard implementată într-o bibliotecă care e inclusă (linkeditată) la compilare; e apelată aici cu un parametru șir de caractere
- șirurile de caractere: incluse între ghilimele duble “ ”
- \n este notația pentru caracterul de linie nouă.



Exemplul 2: Citește de la tastură numele unei persoane, afișează pe ecran o urare de bun venit în lumea C, folosind acest nume.

```
#include <stdio.h>
int main()
{
    char nume[10];
    printf("Numele dumneavoastră:");
    scanf("%s", nume);
    printf("Bine ai venit în lumea C! %s\n", nume);
    return 0;
}
```

Sfărşitul unei instrucţiuni se delimitează folosind caracterul ;.

Apelul unei funcţii transferă controlul de la funcţia apelantă la funcţia apelată. Apelul se face utilizând numele funcţiei, având între paranteze lista parametrilor actuali. Apelul, fiind o instrucţiune, se termină cu caracterul ‘;’ (punct şi virgulă).



Exemplu: Media a doua numere

```
#include <stdio.h>
/* calculeaza si afiseaza media a doua numere */
int main () {
    int a,b;
    float c;          /* declaratii de variabile */
    printf("a="); scanf("%d", &a); /* citire date initiale */
    printf("b="); scanf("%d", &b); /* citire date initiale */
    c= (a+b) / 2.0;      /* instructiune de calcul */
    printf ("%f\n", c);      /* afisare rezultat */
    return 0;
}
```



Exemplu: Pentru două variabile de tip întreg , interschimbați conținutul lor.

```
#include <stdio.h>
int main() {
    int a,b,aux;
    printf("Introduceti cele doua variabile (a,b):");
    scanf("%d,%d",&a,&b);
    aux=a; /* se face interschimbarea */
    a=b;
    b=aux;
    printf("Dupa interschimbare: a=%d, b=%d\n",a,b);
    return 0;
}
```



Exemplu: Pentru două variabile de tip intreg , interschimbați conținutul lor. Nu se folosește o variabilă suplimentara.

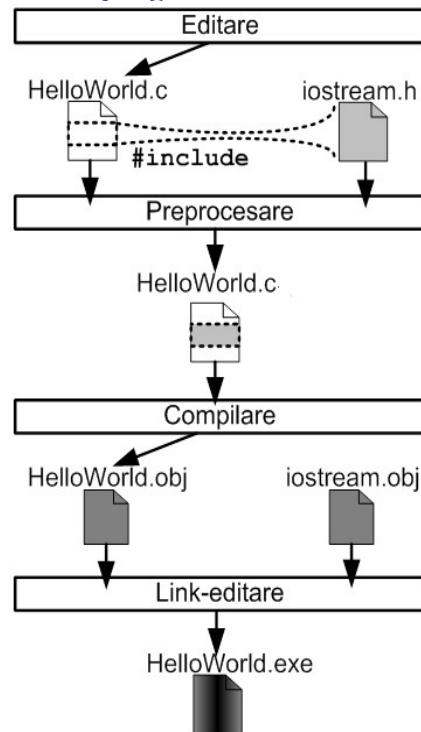
```
#include <stdio.h>
int main() {
```

```

int a,b;
printf("Introduceti cele doua variabile (a,b):");
scanf("%d,%d",&a,&b);
a=a-b;
b=a+b;
a=b-a;
printf("După interschimbare: a=%d, b=%d\n",a,b);
return 0;
}

```

2.2.3.3. Etapele realizării unui program



Realizarea unui program presupune implicarea mai multor etape. Aceste etape sunt independente de limbajul de programare utilizat și implică existență câtorva restricții cu privire la computerul/limbajul utilizat. Etapele realizării unui program sunt:

1. Studierea detaliată a cerințelor aplicației. Este foarte important ca cerințele impuse de aplicație să fie foarte bine explicitate. Adică înainte de a trece la realizarea unui program pentru o anumită aplicație trebuie ca cea aplicație să fie foarte bine analizată și cerințele pe care aceasta le impune trebuie să fie complete și consistente. De exemplu o cerință de genul "scrie un program care să rezolve ecuațiile" este evident că este incompletă și se impun întrebări de genul "ce tip de ecuații", "câte ecuații", "care este precizia", etc.
2. Analiza problemei și determinarea rezolvării acesteia. În această etapă se decide asupra unei metode de rezolvare a problemei (*algoritm*).

3. Traducerea algoritmului realizat la etapa anterioară într-un limbaj de programare evoluat corespunzător. Forma scrisă a acestui program este denumită *program sursă* (*PS, source program sau source code*). În această etapă programul trebuie citit și verificat pentru a-i se stabili corectitudinea. Aceasta se face prin introducerea unui set de valori și verificarea dacă programul furnizează valorile corespunzătoare corecte. Odată verificat programul este scris într-un anumit limbaj prin intermediul unui Editor.
4. *Compilarea* programului în limbaj mașină. Astfel programul obținut în limbaj mașină se numește cod obiect (*object code*). În această etapă compilatorul poate determina erori de sintaxă ale programului. O eroare de sintaxă este o greșală în gramatica limbajului. De exemplu în C trebuie ca fiecare linie să se termine cu ; Dacă se uită plasarea ; atunci compilatorul va semnala eroarea de sintaxă. Astfel se repetă compilarea până la eliminarea tuturor erorilor de sintaxă.
5. Programul obținut în urma compilării, *object code*, este apoi corelat (*linked*) cu o serie de biblioteci de funcții (*function libraries*) care sunt furnizate de sistem. Toate acestea se petrec cu ajutorul unui program numit *link-editor* (*linker*) iar apoi programul *linked object code* este încărcat în memoria computerului de către un program numit *loader*.
6. Rularea programului compilat, link-editat și încărcat cu un set de date pentru testare. Astfel se vor pune în evidență erorile de logică ale programului. Erorile de logică sunt erori care sunt produse de metoda de rezolvare a problemei. Astfel deși programul este scris corect din punct de vedere al sintaxei acesta poate executa ceva ce este incorrect în contextul aplicației. Poate fi ceva simplu, de exemplu realizarea unei operații de scădere în loc de adunare. O formă particulară a erorilor de logică este apariția erorilor de rulare (*run-time error*). O eroare de rulare va produce o oprire a programului în timpul execuției pentru că nu anumite instrucțiuni nu pot fi realizate. De exemplu o împărțire la zero sau încercarea de accesare a datelor dintr-un fișier inexistent.

Astfel se impune ca în această etapă programul să fie reverificat și apoi erorile să fie recopcate prin intermediul editorului ceea ce impune ca etapele 3,4, și 5 să fie repetate până la obținerea rezultatelor satisfăcătoare.

7. Programul poate fi pus în execuție pentru rezolvarea problemei pentru care a fost conceput. Este posibil ca pe parcursul execuției sale să se mai depisteze anumite erori de logică. Astfel se impune reformularea algoritmului și reluarea etapelor de realizare a programului.

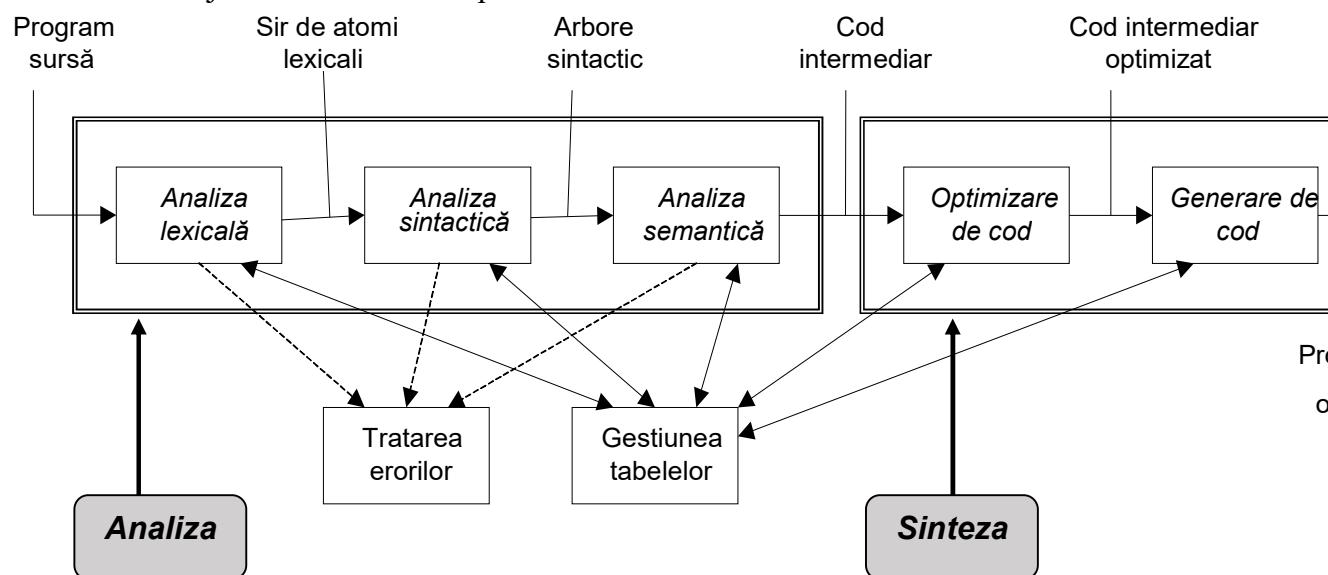
Programul poate fi rulat în mod debug – adică pas-cu-pas pentru a putea verifica execuția punctuală a fiecărei variabile. În acest sens, în mod debug, putem insera breakpoints, iar programul va rula până la întâlnirea acestora, sau putem executa programul până la linia unde se află cursorul.

Pentru execuția pas cu pas, avem 2 opțiuni:

- Step-in – care permite intrarea și execuția pas-cu-pas în funcțiile din linia curentă
- Step-over – care permite execuția într-un singur pas (totul o singură dată) a funcțiilor din linia curentă de cod.

La execuția în mod debug avem activată fereastra Watch în care putem solicita afișarea valorilor pentru variabilele din program.

Prezentăm mai jos schema unui compilator.



În procesul de comunicare om-calculator intervine un program intermediar, translatorul, care asigură traducerea programelor scrise de utilizator din cod sursă într-un alt limbaj mai apropiat de calculator. Dacă limbajul ţintă este codul mașină, translatorul se numește compilator.



Astfel, execuția unui program sursă se realizează, în cadrul limbajelor compilative, se face în 2 faze:

- compilare, care traduce codul sursă în program obiect
- execuție, care rulează codul obiect pe calculator, folosind datele inițiale ale programului și produce rezultate

Conceptual, compilatorul realizează două mari clase de operații: *analiza textului sursă și sinteza codului obiect*.

Compilatorul unui limbaj de asamblare se numește asamblor.

În practică, pe lângă compilatoarele obișnuite, există și alte tipuri de compilatoare.

Astfel, *preprocesoarele* sunt translatoare care traduc dintr-un limbaj de nivel înalt în alt limbaj de nivel înalt.

2.2.3.4. Funcții de intrare/ieșire

Funcția printf

convertește, formatează și tipărește argumentele sale la ieșirea standard sub controlul șirului de control. Este o funcție de bibliotecă care realizează ieșiri cu format (în stdio.h).

Formatul general al acestei instrucțiuni este:

`printf(control,param1, param2, ..., paramn);`

control este un sir de caractere ce conține 2 tipuri de obiecte:

- 1)caractere ordinare care sunt simplu copiate în sirul de ieșire
 - 2)semnificații de conversie care cauzează conversia și tipărirea următoarelor argumente successive ale lui printf.
- param1, param2, ..., paramn sunt expresii ale căror valori se scriu conform specificatorilor de format prezenti in parametrul de control.

Un specificator de format începe prin caracterul % și se încheie prin caracterul de conversie. Între caracterul % și caracterul de conversie pot apărea:

- *semnul minus* care specifică cadrarea la stânga în câmp a argumentului convertit
- un *sir de cifre zecimale optional*, care specifică dimensiunea minimă a câmpului în care se editează data. Dacă data necesită mai puține poziții decât câmpul descris de acest sir atunci el va fi completat la stânga cu caractere nesemnificate (sau la dreapta dacă s-a folosit semnul minus în specificatorul de format respectiv). Caracterele nesemnificate implicate sunt spațiile. Dacă sirul de cifre începe cu un zero nesemnificativ, caracterele nesemnificate vor fi zerouri.
- *Un punct optional* separă sirul de cifre ce definește dimensiunea minimă a câmpului de sirul următor de cifre ce semnifică precizia
- un *sir de cifre* ce definește precizia care specifică numărul maxim de caractere acceptate dintr-un sir de caractere sau numărul de zecimale care se vor scrie în cazul numerelor reale.

Deci semnificațiile de conversie au structura:

`%[-][sir_de_cifre][.][sir_de_cifre]c`

Caracterul de conversie c poate avea următoarele valori:

Car	Semnificație
d	Data se convertește din int în întreg zecimal cu semn
i	Întreg zecimal cu semn
o	Data se convertește din int în întreg octal fără semn
x	Data se convertește din int în întreg hexagesimal fără semn (cu litere mici)
X	Data se convertește din int în întreg hexagesimal fără semn (cu litere mari)
u	Data de convertește din unsigned în întreg zecimal fără semn
c	caracter
s	sir de caractere
e	număr în virgulă flotantă sau flotant dublă precizie și e convertit în zecimal sub forma: [-] m.nnnnnn e [±]xx. Implicit numărul de zecimale este 6 iar dacă se specifică precizia, numărul de zecimale va fi indicat de aceasta.
E	analog dar se folosește E
f	număr în virgulă mobilă simplă sau dublă precizie iar argumentul va fi convertit în notație zecimală

g	în acest caz se va folosi forma cea mai scurtă dintre %e și respectiv %f
G	forma cea mai scurtă dintre %E și respectiv %f
p	afisează un pointer (o adresă)
n	plasează în argumentul asociat numărul de caractere afișat până în acel moment de funcția printf

Exemplu:

“%5f” arată ca numărul are o lungime de cel puțin 5 caractere
 “%05d” determină umplerea cu zerouri a unui număr mai mic de 5 cifre, astfel încât lungimea totală să fie 5
 “%.2f” solicită două poziții după punctul zecimal, dar lungimea lui nu este supusă restricțiilor
 “%-5.2f” determină afișarea pe cel puțin 5 poziții din care exact 2 zecimale aliniat la stânga (datorită prezentei semnului ‘-’)
 “%.0f” suprimă tipărirea părții fracționare
 “%5.10s” determină afișarea unui sir pe cel puțin 5 poziții dar nu mai mult de 10 caractere
 “%” determină afișarea caracterului %
 printf(“Folosim%*n* caracterul %%*n*”,&p); //determină încărcarea în variabila //pointată de pointerul p a valorii 7 (‘Folosim’ are 7 litere)

Există posibilitatea de a utiliza doi modelatori de format pentru a afișa întregi de tip short și long. Acești doi modelatori sunt h și l. Se folosește %hd pentru a tipări un short int și %ld pentru a tipări un long int.

Modelatorul L se aplică numerelor în virgulă mobilă permitând afișarea unui long double. Astfel, se folosește L în fața caracterului de conversie e, f sau g.

Prezența modelatorului # în fața caracterului de conversie g, G, f, e sau E determină afișarea unui punct zecimal chiar dacă nu există cifre zecimale.

Funcția scanf

Citește date de la intrarea standard pe care le convertește în format intern conform specificatorilor. *Prototipul funcției scanf* se găsește în headerul stdio.h și are forma:

scanf(control,param1, param2, ..., param*n*);

Funcția scanf returnează numărul câmpurilor citite sau EOF dacă se întâlnește prematur marca de sfârșit de fișier (detalii despre EOF la capitolul despre fisiere).

Sirul de control poate conține 3 tipuri de obiecte: specificatori de format, caractere albe sau alte caractere.

Specificatorul de format este precedat de caracterul % și specifică tipul valorii ce urmează a fi citite. Aceștia sunt identici cu cei prezentați la funcția printf.

Un caracter alb în sirul de control determină neglijarea unuia sau mai multor caractere albe din intrare. Un caracter care nu este alb determină ca scanf să citească și să negligeze caracterul respectiv din intrare.

Exemplu:

scanf("%d,%d",&a,&b); //determină citirea unui întreg, apoi caracterul virgulă care se negligează și în final citirea unui alt întreg

Dacă caracterul specificat nu se găsește în intrare, se termină execuția funcției scanf. Toate variabilele a căror valori se citesc cu funcția scanf trebuie transmise prin adresă. Astfel, pentru variabilele simple se folosește operatorul &.

Pentru un sir de caractere citit cu %s se poate scrie doar numele sirului. Un element de tablou precum a[i] se poate citi folosind &a[i] sau simplu folosind a+i.

Datele din intrare trebuie separate prin caractere albe. Caracterele de punctuație punct, virgulă și punct și virgulă nu sunt considerate separatori. Caracterele albe sunt folosite ca și separatori dar pot fi citite și ca orice caracter folosind codul de format %c.

Caracterul * plasat după % și înainte de litera de format determină citirea datei de tipul respectiv dar nu se asignează nici unei variabile.

Exemplu:

scanf("%d%*c%d",&x,&y);

-dacă în intrare vom avea 10/20 va determina citirea în variabila x a valorii 10, citirea caracterului / care nu se alocă nici unei variabile după care variabilei y i se atribuie valoarea 20.

În specificatorul de format poate fi indicată lungimea maximă a câmpului care poate fi citit.

Exemplu:

scanf("%2d%3d",&x,&y); //dacă în intrare avem 121234 atunci x=12, y=123

după care următoarea instrucțiune scanf:

scanf("%d",&z); //va conduce spre z=4.

Un număr se consideră citit complet când s-a întâlnit un caracter care nu poate participa la scrierea numărului respectiv sau când s-a luat din zonă numărul de caracter specificat în format.

În cazul sirurilor de caractere, scanf citește până la întâlnirea primului spațiu alb. În cazul în care se dorește citirea unui sir de caractere până la întâlnirea caracterului sfârșit de rând, se va folosi funcția gets().

Specificatorul %n determină ca funcția scanf să atruiie variabilei spre care indică argumentul corespunzător numărul de caractere citit din stream-ul de intrare până în momentul întâlnirii specificatorului %n.

O facilitate a funcției scanf este acesta-numitul *scanset*. Scanset definește o listă de caractere ce vor fi selectate de scanf și memorate într-o variabilă de tip tablou de caractere. Când se întâlnește un caracter ce nu apare în scanset, scanf pună caracterul null la sfârșitul sirului de caractere corespunzător și trece la următorul câmp.

Scanset se indică între paranteze drepte. Pentru specificarea unui interval în scanset se poate utiliza caracterul deci [0123456789] este echivalent cu [0-9]. Pentru litere mici și mari se poate folosi [a-zA-Z], iar [^0-9] înseamnă orice caracter în afara cifrelor.

Exemplu:

```
#include<stdio.h>
int main()
{ char s1[80], s2[80];
  scanf("%[0123456789]%s",s1,s2);
}
```

Efectul acestui program este citirea unui sir de cifre în sirul de caractere s1 iar la întâlnirea primului caracter care nu este cifră se trece la citirea sirului de caractere în s2 până la întâlnirea primului caracter alb.

2.2.3.5. Funcții de citire/scriere caractere

```
int getchar(void)
```

În stdio.h

Efect: Funcția așteaptă apăsarea unei taste citind de la intrarea standard codul ASCII al caracterului din poziția curentă și returnând codul ASCII al acestuia (tipul valorii returnate este int) sau constanta simbolică EOF dacă s-a întâlnit marca de sfârșit de fișier (perechea CTRL/Z, detalii în cap. destinat fisierelor). Tasta apăsată are automat ecou pe ecran.

Exemplu: Se citește o literă mare din fisierul de intrare și se scrie ca literă mică.

```
#include <stdio.h>
int main()
{
  putchar(getchar()-'A'+'a');
}
```

Exemplu: Se testează dacă s-a citit o literă mare și numai în acest caz se aplică transformarea în literă mică. În cazul în care la intrare nu se află o literă mare, se scrie caracterul respectiv.

```
#include <stdio.h>
int main()
{
  int c;
  putchar(((c=getchar())>='A'&&c<='Z')?c-'A'+'a':c);
}
```

```
int putchar(int c)
```

În stdio.h

Efect: afișează caracterul al cărui cod ASCII este egal cu valoarea expresiei dintre paranteze; returnează codul ASCII al caracterului scris la ieșirea standard sau EOF la detectarea unei erori. Se folosește pentru a scrie un singur caracter în fișierul standard de ieșire în poziția curentă a cursorului.

Exemplu:

```
putchar('a'); //trimite la ieșirea standard caracterul 'a'  
putchar('\n'); //trimite la ieșirea standard codul '\n' care are ca efect trecerea  
//cursorului pe linia următoare în prima coloană  
putchar('A'+10); //scrie caracterul de cod A + 10 = 65+10=75 adica litera K
```



Urmăriți fișierul video de la următoarea adresă:

<https://drive.google.com/open?id=0B0E2G7UxqRojQUdmb19Bb0ZwZ3M>

Acesta prezintă aspectele teoretice și practice prezentate până la acest moment.

2.2.3.6. Compilare si Precompilare

Fisierul este unitatea de compilare. Un program poate fi compus din mai multe fisiere care se compileaza, iar fiecare fisier care se compileaza trebuie să aiba extensia c. Compilatorul tratează diferit fisierele care au extensia c de fisierele care au extensia cpp. De aceea este absolut necesar ca pentru a se invoca compilatorul corect fisierele sa aiba extensia C.

Fiecare fisier al programului este compus din funcții. Fiecare funcție se compilează separat și pentru acestea se generează cod obiect. La compilare se generează cod obiect pentru toate funcțiile din program.

Dupa compilare umeaza linkeditarea care colectează toate codurile compilate ale functiilor utilizate in program (inclusiv a celor referite prin include-uri de funcții system) si apoi se creaza prima imagine a zonei de memorie de date si cod a programului.

Compilatorul C conține un preprocesor capabil să facă substituții lexicale, macroinstructiuni, compilări condiționate și includeri de fisiere.



Preprocesorul execută operații anterioare compilării propriu-zise, operații ce sunt specificate prin directive preprocesor. Preprocesorul este apelat automat înainte de a începe compilarea.

Preprocesorul limbajului C este relativ simplu și el, în principiu, execută substituții de texte.

Prin intermediul lui se pot realiza:

- Includeri de texte;
- Definiții și apeluri de macrouri simple;
- Compilare condiționata.

Acestea au întotdeauna ca și prim caracter simbolul # și permit efectuarea de manipulări în textul programului sursă înaintea compilării sale.

Liniile care încep cu simbolul # au o sintaxă independentă de restul limbajului, pot apărea oriunde în cadrul programului sursă (dar se recomandă plasarea la începutul lui) și au efecte cu remanență până la sfârșitul execuției programului sau până când o nouă directivă anulează acest lucru. Fiecare din directivele preprocesor trebuie să se găsească pe o linie separată.

Substituția lexicală se face prin directiva:

```
#define identificator [text]
```

Dacă identificatorul este recunoscut ca element lexical în analiza programului sursă, preprocesorul va înlocui toate aparițiile ulterioare ale identificatorului cu secvența text precizată. Dacă textul trebuie să se extindă pe mai multe linii, liniile intermediare vor fi precedate de un caracter '\'. Textul poate fi și vid ceea ce va avea ca și efect suprimarea din textul sursă a tuturor aparițiilor identificatorului. Pentru a distinge mai ușor identificatorul substituit, se recomandă folosirea majusculelor.

Exemplu:

```
#define void          //suprimarea din textul sursă a tuturor aparițiilor
identificatorului void
#define then          //suprimarea tututor aparițiilor identificatorului then
#define begin {      //înlocuirea identificatorului begin cu {
#define end }        //înlocuirea identificatorului end cu }
#define N 100 //înlocuiește identificatorul N cu valoarea 100
```

O dată făcută o substituție lexicală, ea poate fi folosită ca parte a unei alte substituții lexice. Astfel, având ultima substituție lexicală din exemplul precedent, este corect să scriem:

```
#define NPATRAT N*N
```

Un identificator care apare deja într-o linie #define poate fi ulterior redefinit ulterior în program cu o altă linie #define. Preprocesorul va substitui în continuare identificatorul cu noua definiție a lui.

Substituția nu se face în cazul în care identificatorul apare între ghilimele, în acest caz el nejucând practic rolul unui identificator.

Exemplu:

```
#define PI 3.14
...
printf("valoarea lui PI are un numar infinit de zecimale!\n");
...
```

În acest caz, nu se face expandarea.

Macroinstructiuni

Sunt substituții cu parametri formali și se realizează cu ajutorul directivei:

```
#define identificator(lista_param_formali) text
```

Parametri formali sunt înlocuitori de cei din lista parametrilor reali. Nu trebuie să existe nici un spațiu între identificator și lista_param_formali.

Exemplu:

```
#define max(a,b) (a)>(b)?(a):(b)  
#define abs(a) (a)<0?-(a):(a)
```

Se obține o scriere asemănătoare unui apel de funcție dar care este tratată mai direct prin înlocuire în textul sursă și nu printr-un apel de funcție ceea ce este mai avantajos din punct de vedere al performanțelor de viteză. Macroinstructiunile se execută mai repede decât funcțiile lor echivalente deoarece compilatorul nu necesită depunerea pe stivă a parametrilor și apoi returnarea valorilor. Având în vedere că macroinstructiunile măresc dimensiunea codului, este indicat să ne limităm doar la calcule simple pentru a nu crește talia programului.

Se remarcă utilizarea parantezelor rotunde la încadrarea parametrilor formali. Dacă acestea sunt omise, evaluarea se poate face eronat dacă parametri formali sunt substituitori de expresii.



Exemplu:

```
#include<stdio.h>  
#define PAR(a) a%2==0?1:0//încercăm folosirea macroinstructiunii fără paranteze  
rotunde  
int main()  
{ if PAR(9+1)) printf("este par\n");  
else printf("este impar\n");  
}
```

Efectul utilizării în acest mod a macroinstructiunii este eronat deoarece:

$9+1\%2=0$ va conduce la $9+0 = 0$ adică fals ceea ce va genera mesajul "este impar" în loc să obținem mesajul "este par".

Cauza este lipsa parantezelor la definirea macroinstructiunii. Utilizarea sub forma:

`#define PAR(a) (a)%2==0?1:0` va conduce spre un rezultat corect deoarece:

$(9+1)\%2=0$ conduce spre $0 == 0$ ceea ce este adevărat iar mesajul generat va fi "este par".

Includerea fișierelor se face prin una din directivele:

```
#include <nume_fisier>
#include "nume_fisier"
```

Această directivă are rolul de a solicita compilatorului ca, în faza de preprocesare, acesta să includă în textul sursă în care este folosită această directivă, întregul conținut al fișierului desemnat, cu scopul de a fi compilate împreună. Este validă prezența unei directive include într-un fișier care este apelat la rândul lui printr-o altă directivă include. Standardul ANSI C prevede existența unui număr minim de opt niveluri de includere imbricate.

Fișierul este căutat după cum urmează:

- a)dacă fișierul se delimitizează între paranteze unghiulare se va căuta fișierul în directorul specificat de mediul de programare Dacă fișierul nu este găsit, se continuă căutarea în directoarele standard ale mediului.
- b) dacă fișierul se delimitizează între ghilimele fișierul este căutat:
 - b1)Dacă se specifică calea ca și parte componentă a numelui fișierului doar în acel director se va face căutarea.
 - b2)Dacă nu se specifică calea, fișierul va fi căutat în directorul curent (cel în care am fost în momentul în care am lansat mediul de programare).

În loc ca programatorul să folosească propriile sale declarații pentru funcțiile de bibliotecă, se recomandă includerea în cadrul programului sursă a unor fișiere antet (header), care conțin declarațiile funcțiilor de bibliotecă apelate. Aceste fișiere au în general extensia .h și se găsesc în directorul INCLUDE.

Fisierele *include* vor contine doar DECLARATII și nu definitii de funcții.
Declaratiile și definitiile pot fi de functii sau variabile globale. Daca un fisier include contine definitii, este dincolo de controlul programatorului ca aceste fisiere sa apară incluse de 2 ori intr-un proiect, ceea ce genereaza eroare de linkeditare in sensul ca o definitie este gasita (la linkeditare) ca si duplicata.,

Includerea fișierelor se folosește pentru gestionarea corectă a unui proiect, în general de mari dimensiuni, care se compune din mai multe module. Pentru încorporarea la începutul fiecărui modul de fișiere de descriere se pot crea:

- fișiere conținând definiții generale ale proiectului: declarații de tip și directive ale preprocesorului, în particular directive define
- fișiere conținând declarații de variabile externe în cazul în care definirea lor a fost regrupată într-un modul special.

Programele C mari pot fi structurate pe mai multe fișiere, care pot fi compilate distinct. Aceasta prezintă mai multe avantaje:

-în cazul erorilor sintactice și semantice, doar fișierul eronat trebuie recompilat, după modificările sau corecțiile aduse. Programatorul poate profita de faptul că unitatea de

compilare este fișierul, repartizând funcțiile programului său în mai multe fișiere. În acest fel, modificarea unei funcții va fi urmată de recompilarea doar a fișierului care o conține și a fișierelor care depind de el, nu a întregului program, obținându-se astfel un câștig de timp însemnat.

-structurarea programelor pe mai multe fișiere permite simularea mecanismului de încapsulare a datelor. Variabilele și funcțiile având clasa de memorare static pot fi accesate doar în cadrul fișierului în care au fost definite. Variabilele și funcțiile declarate cu clasa de memorare extern sunt definite în alt fișier decât cel curent, dar ele pot fi accesate în fișierul curent.

Observație: Dacă se modifică un fișier care este inclus printr-o directivă #include, atunci toate fișierele care depind de el trebuie recompilate.

2.2.3.7. Construcții de bază în C

Caracterele

La scrierea programelor C se folosește un subset al setului de caractere al codului ASCII. Caracterele din acest set se codifică în intervalul [0,127]. Un astfel de întreg poate fi păstrat în binar pe un octet (8 biți).

Se împart în trei grupe:

- caractere grafice în intervalul (32,127), aici intrând literele mari având codul ASCII în intervalul [65,96], literele mici [97,122], cifrele [48,57] și caracterele speciale
- caractere negrafice (de control); au coduri în intervalul [0,32) cu excepția caracterului Del având codul ASCII 127.
- caracterul spațiu având codul ASCII 32



Codul ASCII de valoare 0 definește caracterul nul; este un caracter impropriu, nu poate fi generat de la tastatură și nu are efect în ieșire. Este utilizat pentru a determina un sir arbitrar de caractere deoarece nici un caracter de la intrare nu poate coincide cu el.

Codul ASCII 10 deplasează cursorul în coloana 1 din linia următoare. În C referirea la acest cod se face cu combinația '\n'.

Nume (identificatori)



Un *identificator* este o succesiune de litere, cifre și caracterul de subliniere (_) care începe cu o literă (mică, mare) sau cu caracterul de subliniere. Sunt semnificative primele 32 de caractere. Literele mici se consideră distințe de cele mari.

Exemplu: x, a5b9, p_i_c, _cifra

Contraex: 9a\$, a+b, a%w, a_salut!

Identificatorii sunt folosiți pentru a identifica sau denumi funcții și variabile.

Cuvinte cheie

Sunt identificatori rezervați ai limbajului, având o semnificație bine determinată. Acestea nu pot fi utilizate decât conform sintaxei limbajului. Au fost definite 32 de cuvinte cheie:

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

Unele compilatoare de C mai adaugă la aceste liste încă câteva cuvinte cheie.

Tipurile de bază din limbajul C



Un tip de date reprezintă mulțimea valorilor pe care le pot lua datele de tipul respectiv, modul de reprezentare a acestora în memorie precum și operațiile care se pot efectua cu datele respective. Tipurile de bază definite în limbajul C sunt: întreg, real și caracter. Unele dintre aceste tipuri diferă de la o implementare la alta a limbajului.



Un tip de date reprezintă:

- modul de reprezentare a datei respective în memoria calculatorului. Astfel, se precizează cum se convertesc biți din memoria calculatorului pentru a produce date reprezentată de valoare cu tipul respectiv
- dimensiunea pe care o ocupă în memorie o dată de tipul respectiv
- multimea de valori admisibile pentru datele de tipul respectiv.

În tabelul următor sunt prezentate tipurile fundamentale, memoria necesară stocării valorilor de acel tip și limita valorilor ce pot fi memorate într-o variabilă de acel tip.

Tipul	Dimensiune memorie	Limita valorilor
char	1 byte (octet)	$-2^7 \dots 2^7 - 1$
int	deinde implementare (uzual 4 bytes)	

short int	2 bytes	
unsigned char	1 byte	0..255
unsigned int	depinde de implementare	
long int	4 bytes	
unsigned long int	4 bytes	
float	4 bytes	
double	8 bytes	
long double	8 bytes	

Primul tip este tipul caracter. O variabilă de acest tip va avea ca valoare codul ASCII asociat caracterului respectiv.

De exemplu, dacă unei variabile i se atribuie caracterul 'a', variabila va conține valoarea 97 (numărul de ordine al caracterului 'a' în codul ASCII).

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0 000	NUL	(null)	32	20	040	 	Space	64	40	100	@	Ø	96	60	140	`	
1	1 001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	
2	2 002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	
3	3 003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	
4	4 004	EOT	(end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	
5	5 005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	
6	6 006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	
7	7 007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	
8	8 010	BS	(backspace)	40	28	050	({	72	48	110	H	H	104	68	150	h	
9	9 011	TAB	(horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	
10	A 012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	
11	B 013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	
12	C 014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	
13	D 015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	
14	E 016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	
15	F 017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	
16	10 020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	
17	11 021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	
18	12 022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	
19	13 023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	
20	14 024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	
21	15 025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	
22	16 026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	
23	17 027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	
24	18 030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	
25	19 031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	
26	1A 032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	
27	1B 033	ESC	(escape)	59	3B	073	;	:	91	5B	133	[[123	7B	173	{	
28	1C 034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D 035	GS	(group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	
30	1E 036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	
31	1F 037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		

Tipul întreg cel mai des utilizat în programe este **int**, dar numărul de bytes pe care se memorează valorile de acest tip diferă de la o implementare la alta. Numărul de bytes

al unei valori de tip int reprezintă cuvântul calculatorului respectiv. Astfel, tipul int este echivalent cu **short int** sau **long int**, în funcție de implementare.

Celelalte tipuri întregi sunt obținute prin folosirea prefixului **signed** sau **unsigned**, indicând dacă valorile respective sunt cu semn (conțin și valori negative), respectiv fără semn (valori naturale).

Ultimele trei tipuri din tabel sunt tipuri reale, diferența dintre ele constând în cantitatea de memorie utilizată, intervalul de valori și precizia memorării valorilor (numarul de zecimale retinute).

Valorile reale se reprezintă conform notației din standardul IEEE (folosind semnul, mantisa și exponentul). De exemplu, pentru tipul **float** se utilizează reprezentarea în simplă precizie, folosind un bit de semn, 7 biti pentru exponent și 24 de biti pentru mantisa. Aceasta reprezentare are un exponent în limita a 10^{-37} și 10^{38} cu pîna la 7 zecimale precizie. Valoarea maximă a unui float este de 1.701411 E38.

Din tabel se observă că nu există un tip logic, astăzi cum este el definit în alte limbi de programare. În limbajul C nu s-a definit acest tip dar se folosește următoarea convenție: o expresie este considerată adevărată dacă valoarea obținută la evaluarea ei este nenula și falsă în caz contrar.

Variabilele care se vor folosi ca variabile logice vor fi declarate de tip întreg

O altă caracteristică aparte o constituie introducerea tipului **void**, pentru a desemna "nimic". Se mai folosește la declararea explicită a funcțiilor care nu au parametrii.

Pentru a putea utiliza o variabilă într-un program C, este obligatorie declararea acesteia, astfel:

TIP_DE_DATE *lista_variabile*;
unde:

- TIP_DE_DATE este orice tip predefinit sau derivat;
- *lista_variabile* conține lista variabilelor de acel tip, despărțite prin virgulă;

Exemplu:

```
int a,b,c;
char ch;
float x,y;
long int z;
int g=7;
```

Observatie: La declararea variabilei întregi g s-a realizat și initializarea acesteia (=stabilirea unei valori initiale)



Variabilele care sunt declarate în interiorul corpului unei funcții se numesc variabile locale, iar cele declarate în afara oricărei funcții din program se numesc variabile globale. Variabilele globale vor

putea fi utilizate în toate funcțiile ce compun programul, iar variabilele locale doar în cadrul funcției în care au fost definite.

În cadrul *tipurilor derivate* avem:

- *tipuri structurate* –tablouri, structuri, uniuni, câmpuri de biți, enumerări
- *tipuri funcție* –caracterizate atât prin tipul rezultatului furnizat cât și prin numărul și tipul argumentelor necesare pentru obținerea rezultatului.
- *tipuri pointer* –permit adresări indirecte la entități de diverse tipuri

Un tip specific limbajului C este *tipul void*. Acesta se folosește la declararea explicită a funcțiilor care nu returnează nici o valoare sau a funcțiilor care nu au parametri.

Tipul de date int

Valorile acestui tip sunt numere întregi, cuprinse în intervalul [-32767,32767] reprezentate în memorie pe 2 octeți, în cod complementar. Tipul de date int suportă modificatorii de tip unsigned (datele sunt numere normale) și long (modifică dimensiunea reprezentării). Se obțin astfel următoarele tipuri de date întregi:

Tip	Reprezentare	Valori
int	2 octeti cu semn	[-32768, 32767]
unsigned int	2 octeti fara semn	[0, 65.535]
long int	4 octeti cu semn	[-2.147.483.647, 2.147.483.647]
unsigned long int	4 octeti fara semn	[0, 4.294.967.295]

Atunci când un tip de date nu este precizat implicit este considerat int. Prin urmare putem specifica numai modificatorul long sau unsigned sau unsigned long, tipul fiind implicit int.

Constantele întregi sunt numere întregi din intervalul corespunzător tipului. Ele pot fi precizate în baza 10 folosind notația uzuală, în baza 8 constanta fiind precedată de un 0 nesemnificativ și în baza 16 constanta având prefixul 0x sau 0X.

Exemple:

123
-12345678
01234
0x1a0
0xFFFFF

Explicitarea tipului de constantă numerică se poate face utilizând un sufix. Pentru tipul întreg, utilizarea sufivelui U determină alocarea tipului unsigned. Analog, folosim sufivelul L pentru tratarea unui întreg ca long. Dacă utilizăm sufivelul F pentru tipul virgulă mobilă, numărul va fi tratat ca float iar dacă folosim sufivelul L numărul va fi de tip long double.

Exemplu:

100U	unsigned
-1234L	long int
123.45F	float
123456.789	long double

Tipul de date char

Tipul caracter în C se reprezintă pe un octet și are ca valoare codul ASCII al caracterului respectiv. Constanta caracter grafic se scrie incluzând caracterul respectiv între caracterul apostrof.

Exemplu: ‘A’

→ intern vom avea o reprezentare pe un octet ce conține valoarea 65 (codul ASCII al literei A).

Tip	Reprezentare	Valori
char	1 octet cu semn	[-128, 127]
unsigned char	1 octet fara semn	[0, 255]

Constantele de tip char (unsigned char) pot fi numere întregi din intervalul specificat care au codurile ASCII în intervalul specificat. Valorile de tip char par să aibă o natură duală, caractere ASCII și numere întregi. Caracterul A și codul 65 au aceeași semnificație.

Caracterele grafice (coduri ASCII de la 32 la 127) se pot specifica încadrând caracterul respectiv între apostrofuri.

Exemplu:

‘a’, ‘9’, ‘*’

Caracterele negrafice se pot specifica încadrând între apostrofuri o secvență de evitare (secvență escape). Secvențele escape sunt formate din caracterul \ (backslash) urmat de codul ASCII al caracterului exprimat în baza 8 sau în baza 16 precedat de un x.

Exemplu:

Secvență escape	Caracter
‘\65’	‘5’
‘\x35’	‘5’ (exprimat în baza 16)
‘\5’	Caracterul ♣
‘\356’	Caracterul €

Unele caractere negrafice au asociate secvențe escape speciale:

Secvență	Semnificația
\a	allert (bell)
\b	backspace

\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab

Constantele sir de caractere sunt constituite dintr-o succesiune de caractere încadrată între ghilimele.

Exemplu:

”Acesta este un sir”

”Prima linie \n A doua linie”

Tipuri reale

Tipurile reale sunt float și double. Tipul double acceptă și modifierul long.

Tipul	Reprezentare
float	4 octeti în virgulă mobilă
double	8 octeti în virgulă mobilă
long double	10 octeti în virgulă mobilă

Un număr flotant este un număr rațional care se compune din semn (dacă e cazul), o parte întreagă (care poate fi vidă), o parte fracționară (care și ea poate fi vidă) litera e sau E și un exponent (care și ele pot lipsi). Nu pot fi văzute toate componente. Partea întreagă este o succesiune de cifre hexagesimale. Partea fracționară se compune dintr-un punct urmat de o succesiune de cifre zecimale, succesiune care poate fi vidă. Se reprezintă în forma uzuală sau științifică ca exponent a lui 10, urmat de e.

Exemplu:

-1
5e-5
2.e-4
12.4
-2.67e-10
-2.67E+8

2.2.3.8. Instructiuni

Instructiunile C implementează structurile de bază ale programării structurate, prezentate în cursul 1 și identificabile în schemele logice și pseudocod astfel:

- structura secvențială* (instructiunea compusă) ;
- structura alternativă* (instructiunea if) ;
- structura repetitivă* condiționată anterior (instructiunea while) și condiționată posterior (instructiunea do while)



Toate instrucțiunile limbajului C se termină cu ‘;’ exceptie făcând instrucțiunile care se termină cu } (instrucțiunea compusă și instrucțiunea switch) după care nu se pune ‘;’.

Instrucțiunea vidă

Instrucțiunea vidă are formatul:

;

și este folosită pentru a înlocui o instrucțiune care practic nu este, dar contextul (sintaxa) o cere.

Una din situații o constituie instrucțiunile de ciclarewhile și for a căror sintaxă impune corp al instrucțiunii. Dacă acesta lipsește, atunci instrucțiunea vidă va ține loc de corp al instrucțiunii.

Exemplu 1

```
.....
for (i=0;(i+1)*(i+1)<n;i++);
.....
```

Această instrucțiune de ciclare determină cel mai mare număr al cărui pătrat nu este mai mare decât numărul natural n adică \sqrt{n} (partea întreagă din radical din n). Se observă că tot ceea ce este de făcut se face în linia instrucțiuniifor, dar sintaxa lui for impune corp al instrucțiunii și în locul acestuia se pune instrucțiunea vidă ;.

Instrucțiunea compusă

Forma generală:

```
{  
declaratii_si_definitii;  
  
instructiune1;  
instructiune2;  
...  
instructiune n;  
}
```

Se folosește în situațiile în care sintaxa impune o singură instrucțiune, dar codificarea impune prezența unei secvențe de instrucțiuni. Blocul de instrucțiuni conținează ca o singură instrucțiune.

Observații:

1. După acoladă închisă nu se pune ;.
 2. Corpul unei funcții are aceeași structură ca și instrucțiunea compusă.
3. O instrucțiune compusă permite folosirea mai multor instrucțiuni acolo unde sintaxa cere o instrucțiune, aceasta fiind echivalentă sintactic cu o singură instrucțiune.

4. instrucțiunea compusă reprezintă un domeniu de vizibilitate. Astfel, declarațiile și definițiile din interiorul unei instrucțiuni compuse au valabilitate doar în interiorul acesteia.

Instrucțiunea if

În limbajul C, instrucțiunea condițională de bază este instrucțiunea if.

Forma generală:

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

Se evaluează expresia; dacă este diferită de 0 se execută instrucțiune1 altfel instrucțiune2

O formă simplificată are instrucțiune2 vidă:

```
if (expresie)
instructiune;
```

În problemele de clasificare se întâlnesc decizii de forma:

```
if (expr1)
    instr1;
else if (expr2)
    instr2;
...
else
    instrn;
```

Exemplu: dorim să contorizăm caracterele citite pe categorii: litere mari, litere mici, cifre, linii și altele:

```
if (c == '\n')
linii++;
else if (c>='a' && c<='z')
lmici++;
else if (c>='A' && c<='Z')
lmari++;
else if (c>='0' && c<='9')
cifre++;
else
altele++;
```

Observații: 1. Ramura else poate lipsi.

2. Întotdeauna una și numai una dintre instrucțiuni se execută adică instrucțiunea de pe ramura if sau cea de pe else.

3. Este posibilă utilizarea mai multor instrucțiuni if imbicate.

4. Deoarece există posibilitatea prezenței instrucțiunii if fără ramura else, rezultă că este necesară următoarea regulă: un else se pune în corespondență cu primul if care se află înaintea lui în textul sursă și care nu are asociat un alt else.

Exemplu:

```
if(x)
if(y) printf("1");
else printf("2");
```

În acest caz, ramura else care tipărește valoarea 2 este legată de instrucțiunea if(y).

```
if(x)
{if (y) printf("1");
else printf("2");}
```

În acest caz, datorită instrucțiunii compuse, ramura else care tipărește valoarea 2 aparține instrucțiunii if(x).

Instrucțiunea while

Instrucțiunea while implementează structura repetitivă condiționată anterior. Forma instrucțiunii while este:

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

Atât timp cât *expresie* este diferită de 0 se execută *instructiune*. Instrucțiunea *while* mai este cunoscută și sub numele de ciclare cu test inițial. Bucla *while* este folosită atunci când numărul de iterații nu este cunoscut apriori.

Observații.

1. Instrucțiunea se repetă cât timp valoarea expresiei nu este nulă. Pentru ca ciclul să nu fie infinit, este obligatoriu ca instrucțiunea care se execută să modifice cel puțin una din variabilele care intervin în expresie, astfel încât aceasta să poată lua valoarea 0, sau să conțină o operație de ieșire necondiționată din ciclu (de exemplu break).

2. Dacă de la început expresia are valoarea 0, instrucțiunea nu se execută niciodată.

3. Sintaxa permite executarea în while a unei singure instrucțiuni, prin urmare, atunci când este necesară efectuarea mai multor operații, acestea se grupează într-o singură instrucțiune compusă.



Exemplu: Programul care numără caracterele unui text.
Introducerea se încheie cu *enter*.

```
#include <stdio.h>
```

```

int main ()
{
    int i=0;
    printf("Scrieti un text : \n");
    while (getch()!='r') /*se citeste textul fara afisare in consola */
        i++;
    printf("Numarul de caractere este %d\n",i);
    return 0;
}

```



Exemplu: Program care calculează factorialul:

```

#include <stdio.h>
int main()
{
    long n, fact;
    printf("n="); scanf("%ld", &n);
    fact=1;
    while (n>1)
    {
        fact = fact * n;
        n--;
    }
    printf("Factorialul este : %ld\n", fact);
}

```

Instrucțiunea do while

Instrucțiunea do while implementează structura repetitivă condiționată posterior.
Forma instructiunii do while este:

```

do
    instructiune;
    while (expresie);

```

Instrucțiunea este asemănătoare cu instructiunea *while* cu deosebirea că, aici, testul se face la sfârșitul buclei. Bucla se va executa cel puțin o dată, chiar dacă testul nu este îndeplinit de la început. Instrucțiunea se execută cât timp expresia este diferită de 0.

Observație: Spre deosebire de instrucțiunea *while*, în cazul acestei instrucțiuni, corpul ei se execută cel puțin o dată, indiferent de valoarea inițială de adevăr a expresiei.

Preferăm să utilizăm instructiunea *while* când este necesar să testăm o condiție înainte de efectuarea unor prelucrări.

Preferăm să utilizăm do – while când condiția depinde de la început de prelucrările din ciclu, prin urmare este necesar să o testăm după executarea instrucțiunii.



Exemplu: Să se numere cifrele numărului natural memorat în variabila n.

```
#include<stdio.h>
int main()
{
    int n, nr;
    printf("introduceti un numar intreg: "); scanf("%d", &n);
    nr=0;
    do
    {
        n/=10;
        nr++;
    } while(n);
    printf("numarul de cifre este %d\n", nr);
    return 0;
}
```

Instrucțiunea for

În cazul în care se cunoaște numărul de iterații ale unui ciclu repetitiv, se poate ușor utiliza instrucțiunea for. Formatul general al acestei instrucțiuni este următorul:

```
for(exp1;exp2;exp3) instrucțiune;
```

Efectul acestei instrucțiuni este următorul: se evaluează exp1 după care se testează valoarea de adevăr a exp2. Dacă exp2 are valoarea 0, se termină execuția instrucțiunii for, și se continuă cu următoarea instrucțiune de după instrucțiunea for. Dacă exp2 are valoarea diferită de 0, se execută instrucțiune după care se evaluează exp3. Se revine la evaluarea lui exp2 și se continuă ciclic până când exp2 va deveni, caz în care se continuă cu următoarea instrucțiune de după instrucțiunea for.

Instrucțiunea for este echivalentă cu următoarea secvență de instrucțiuni:

```
exp1;
while(exp2)
{
    instrucțiune;
    exp3;
}
```

Observație: Oricare din cele 3 expresii pot să lipsească. De asemenea instrucțiunea poate lipsi. Dacă lipsesc toate cele 3 expresii se obține un ciclu infinit:

for(;;) instrucțiune;

Exemplu :



```
#include <stdio.h>
int main()
{
    int i;
    for( i = 1; i <= 10; i++ )
        printf("%d ", i );
    printf("\n");
    return 0;
}
```

Exemplu: Să se scrie programul C care calculează $n!$ unde știm că $n!=n*(n-1)!$ iar $0!=1$.



```
#include<stdio.h>
int main()
{
    int n=6, fact=1, i;
    for(i=1;i<=n;i++) fact*=i;
    printf("%d! = %d\n", n, fact);
    return 0;
}
```



Urmăriți fișierul video de la următoarea adresă:

<https://drive.google.com/open?id=0B0E2G7UxqRojb2V1bUFqdkFWWmM>

Acesta prezintă aspectele teoretice și practice prezentate până la acest moment.

Instrucțiunea return

Instrucțiunea return este folosită pentru a returna rezultatul dintr-o funcție.

Este considerată drept instrucțiune de salt întrucât determină executarea programului să revină (să execute un salt înapoi) la punctul în care s-a făcut apelarea funcției.

Dacă instrucțiunii return îi este asociată o valoare, acea valoare devine valoarea calculată prin acea funcție. Dacă nu este specificată nici o valoare calculată a funcției, se presupune returnarea unei valori inutile (unele compilatoare Cvor returnă valoarea zero dacă nu este indicată nici o valoare).

Forma generală a instrucțiunii return permite 2 formate:

```
return; // este necesară dacă tipul de return al funcției este void  
return expresie;
```

Primul format se utilizează când funcția nu returnează o valoare iar cel de-al doilea se utilizează atunci când funcția returnează o valoare (funcția returnează valoarea expresiei specificate).

În cadrul unei funcții putem folosi instrucțiuni return la discreție. Cu toate acestea, o funcție va fi stopată din execuție la întâlnirea primei instrucțiuni return. Acolada de la sfârșitul unei funcții determină de asemenea revenirea din aceasta. Situația este analogă cu un return căruia nu i s-a precizat nici o valoare.

O funcție declarată de tip void nu poate conține o instrucțiune return care specifică o valoare. Din moment ce o funcție de tip void nu are valoare calculată, este de bun simț faptul că o instrucțiune return interioară acesteia nu poate returna o valoare.

Tipul valorii returnate va fi cel specificat la tipul funcției care returnează valoarea (tipul int în cazul în care nu s-a specificat un tip pentru această funcție). Funcția apelantă poate ignora valoarea returnată.

O funcție care nu returnează nimic se poate încheia și fără prezența instrucțiunii return. Astfel, ea se încheie după execuția ultimei instrucțiuni și întâlnirea acoladei de sfârșit a funcției ‘}’.



Exemplu: Să se scrie o funcție care returnează maximul a 2 numere primite ca parametru.

```
int max(int x, int y)  
{  
if(x>y) return x;  
else return y;  
}
```

Instrucțiunea switch

Permite realizarea unei structuri selective. Ea este o generalizare a instrucțiunii if care poate fi realizată prin structuri alternative imbricate.

Această instrucțiune determină transferul controlului unei instrucțiuni sau unui bloc de instrucțiuni în funcție de valoarea unei expresii și are următorul *format general*:

```
switch (expresie) {  
    case c1: instr1;  
    [case c2: instr2;]  
    ...  
    [default: instr_default;]  
}
```

Efectul acestei instrucțiuni este următorul: se evaluează expresia dintre paranteze. Dacă valoarea expresiei este egală cu c_i atunci se execută instrucțiunea corespunzătoare constantei c_i . Dacă valoarea expresiei este diferită de orice c_i indicat, atunci se execută instrucțiunea corespunzătoare clauzei default, dacă există această cluză după care se trece la următoarea instrucțiune de după instrucțiunea switch.

După ce se execută instrucțiunea corespunzătoare unei constante indicate, dacă nu există instrucțiunea break atunci necondiționat se execută toate instrucțiunile de mai jos celei corespunzătoare constantei c_i , eventual până la prima instrucțiune break întâlnită.

Instrucțiunea switch este deseori folosită pentru a prelucra anumite comenzi de la tastatură, cum ar fi selecția unei opțiuni dintr-un meniu.

Observații:

1. Valoarea expresiei dintre paranteze trebuie să fie de un tip compatibil cu int (char, enum sau orice variantă de int). Nu se pot utiliza numere reale, siruri, pointeri sau structuri dar se pot folosi elemente de tip compatibil cu int din cadrul sirurilor sau structurilor. Este interzisă apariția a două constante pentru case în aceeași instrucțiune switch cu aceeași valoare.
2. Standardul ANSI C prevede că instrucțiunea switch poate să aibă cel puțin 257 de instrucțiuni tip case. Există posibilitatea de a imbrica mai multe instrucțiuni switch una în alta, fără să apară conflicte chiar dacă unele constante case dintr-un switch interior și unul exterior conțin valori comune.



Exemplu:

```
...  
switch (luna) {  
    // februarie  
    case 2: zile=28; break;  
    // aprilie, iunie,..., noiembrie  
    case 4: case 6: case 9: case 11: zile =30; break;  
    // ianuarie, martie, mai,..decembrie  
    default: zile=31; break;
```

```
}
```

```
...
```

Instrucțiunea break

Instrucțiunea break este utilizată pentru ieșirea forțată dintr-o buclă, înainte de sfârșitul acesteia. În general, aceste instrucțiuni de ieșire forțată se aplică unor algoritmi nestructurați. Se recomandă evitarea acestora, știind fiind faptul că orice algoritm nestructurat poate fi transformat într-un algoritm structurat.

Dacă este necesar, corpul unei instrucțiuni de ciclare poate să conțină mai multe instrucțiuni break, corespunzătoare mai multor condiții de ieșire forțată. Bineînțeles, pot exista și ieșiri normale din aceste bucle, chiar dacă au fost prevăzute mai multe instrucțiuni break.

Instrucțiunea permite ieșirea dintr-un singur ciclu, nu și din eventualele cicluri care ar conține instrucțiunea repetitivă în care s-a executat instrucțiunea break.

Exemplu:



```
#include <stdio.h>
#define PROMPT ':'

int main()
{
    float a, b, result;
    char oper, eroare;
    while (putchar(PROMPT),(scanf("%f%c%f",&a, &oper, &b)!=EOF)){
        eroare=0;
        switch(oper){
            case '+': result=a+b; break;
            case '-': result=a-b; break;
            case '*': result=a*b; break;
            case '/': if(b)result=a/b;
            else
                {
                    puts("**** Impartire la 0 ****");
                    eroare=1;
                }
            break;
            default : printf("**** Operator ilegal %c ***\n", oper);
            eroare=1;
        }//switch
        if(!eroare)
            printf("rezultatul e %f\n", result);
    }
}
```

```
 } //while  
 } //main
```



Exemplu:

```
char c; int a, b, r;  
printf("Screti o operatie intre doi intregi: ");  
if (scanf("%d %c %d", &a, &c, &b) == 3) { /* toate 3 corect */  
    switch (c) {  
        case '+': r = a + b; break; //iese din corpul switch  
        case '-': r = a - b; break; // idem  
        default: c = '\0'; break; // fanion caracter eronat  
        case 'x': c = '*'; // 'x' e tot ?nmultire, continua  
        case '*': r = a * b; break; //ca si pt. apoi iese  
        case '/': r = a / b; //la sfarsit nu trebuie break  
    }  
    if (c)  
        printf("Rezultatul: %d %c %d = %d\n", a, c, b, r);  
    else  
        printf("Operatie necunoscuta\n");  
    }  
else printf("Format eronat\n");
```

Instrucțiunea continue

Se utilizează în interiorul ciclurilor și permite saltul la începutul secvenței de instrucțiuni care formează corpul ciclului respectiv continuând cu următoarea iterație a ciclului, deci nu se părăsește bucla. Ea se asemănă cu instrucțiunea break, dar în loc să forțeze încheierea buclării, instrucțiunea continue forțează trecerea la următoarea iterație a buclei.

Pentru bucla for, instrucțiunea continue determină execuția secvenței de incrementare și a testului de condiționare, iar pentru buclele while și do-while controlul programului este trecut testului de condiționare.



Exemplu:

```
while (1)  
{  
    scanf("%lf", &x);
```

```

if (x < 0.0)
break; //iesim din while cand x este negativ
printf("%lf\n", sqrt(x));
}
while (contor < n)
{
scanf("%lf", &x);
if (x > -0.01 && x <=0.01)
continue; //valorile mici nu se iau in considerare
++contor;
suma += x;
}

```

Instrucțiunea goto

Instrucțiunea de salt goto permite saltul la o anumită instrucțiune din cadrul funcției, instrucțiune precedată de o etichetă. Formal, instrucțiunea goto nu este necesară niciodată, în cazul programării structurate. Nu se recomanda utilizarea acestei instrucțiuni; programarea care utilizează salturi neconditionate se numește programare în stil “spaghetti”.

Formatul instrucțiunii goto este:

goto etichetă;

unde etichetă este un nume care identifică, sub forma:

etichetă: instrucțiune;

instrucțiunea cu care se continuă execuția programului.

Aceeași etichetă poate fi referită de mai multe instrucțiuni goto, dar o etichetă nu poate identifica decât o singură instrucțiune. Unicul scop pentru care se definește o etichetă este de a fi referită de o instrucțiune goto.

2.2.3.9. Expresii

Un *operator* este un simbol care indică compilatorului necesitatea execuției unei operații matematice sau logice.

O expresie este compusă dintr-un operator și unul sau doi operanzi. Expresiile se evaluatează în funcție de regula de evaluare a operatorului implicat în expresie. Rezultatul evaluării unei expresii este o valoare (rezultat al expresiei). În plus, evaluarea expresiei poate să aibă efect și asupra operanzilor, în sensul modificării valorii acestora.



In funcție de tipul operatorilor, aceștia sunt

- Operatori unari – se aplică asupra unui singur operand

- Operatori binari – se aplică asupra a 2 operanzi.

Expresiile se evaluatează fie de la dreapta la stânga, fie de la stânga la dreapta, în funcție de operatorul implicat în expresie.

Expresiile pot fi compuse, în sensul în care rezultatul evaluării unei expresii reprezintă un operand într-o expresie nouă. În acest sens, operatorii sunt evaluați în funcție de tabela de precedență a operatorilor, descrisă mai jos. Expresiile pot conține () care schimbă prioritatea de evaluare în cadrul expresiilor compuse, adică expresia dintre () se evaluatează prioritar.

Precedență	Operator	Descriere	Asociativitate
1	++ --	Incrementare / decrementare formă postfixată	De la stânga la dreapta
	0	Apel de funcție	
	[]	Referirea unui element dintr-un sir	
	.	Acces la membrii unei structuri sau uniuni	
	->	Acces la membrii unei structure sau uniuni prin pointer	
2	++ --	Incrementare / decrementare prefixată	De la dreapta la stânga
	+ -	Plus / minus unar	
	! ~	NOT logic și NOT pe biți	
	(type)	Conversie de tip	
	*	Dereferețierea unui pointer	
	&	Adresa (unei variabile)	
	sizeof	Mărimea (in octeți a unei variabile / valori)	
3	* / %	Inmulțire, împărțire, rest	De la stânga la dreapta
4	+ -	Adunare și scădere (operatori binari)	
5	<< >>	Deplasare la stânga/dreapta pe biți	
6	< <=	Operatori relaționali de comparație	
	> >=		
7	== !=	Operatori relaționali pentru egalitate și diferit	
8	&	AND pe biți	
9	^	XOR pe biți (or exclusiv)	
10		OR pe biți	
11	&&	AND logic	
12		OR logic	
13	?:	Operatorul ternar condițional (singurul operator cu 3 operanzi)	De la dreapta la stânga
14	=	Operatorul de atribuire simplu	
	+= -=	Atribuire cu sumă și diferență	
	*= /= %=	Atribuire cu inmulțire, împărțire și rest	

	<code><<= >>=</code> <code>&= ^= =</code>	Atribuire cu operații de deplasare pe biți Atribuire cu operații pe biți AND , OR sau XOR	
15	,	Operatorul virgulă	De la stânga la dreapta

Descriem mai jos operatorii uzuali utilizati in programele noastre.

Operatori aritmetici:

Operator	Semnificație
-	Minus unar (semn)
* / %	Înmulțire, împărțire, restul împărțirii întregi
+ -	Adunare, scădere



Prioritatea operatorilor scade de sus în jos

Operatorul '%' nu se poate aplica asupra variabilelor de tip float și double.

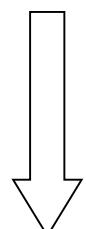
Operatorul '/' dacă se aplică la 2 variabile sau constante întregi se va face o împărțire întreagă fără a se lua în considerare restul!

Exemplu:

```
float a=5;
a=a + 1/2;    //variabila float a va conține în continuare valoarea 5.0
a=a + 1./2;   //în acest caz valoarea variabilei a va fi 5.5
```

Operatori relaționali și logici

Operator	Semnificație
!	negare logică
<code>>>= <<=</code>	mai mare, mai mare sau egal, mai mic, mai mic sau egal
<code>== !=</code>	egal, diferit
<code>&&</code>	și logic
<code> </code>	sau logic



Prioritatea operatorilor scade de sus în jos (deci ! este cel mai prioritări iar || cel mai puțin prioritări).

P	q	<code>!p</code>	<code>p&&q</code>	<code>p q</code>
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

Observații:



- Valoarea de fals în C este reprezentată de valoarea 0. Orice valoare diferită de 0 este interpretată ca valoare de adevărat!
- În urma evaluării expresiilor logice în C rezultatul va fi 0 pentru fals și 1 pentru adevărat.
- Operatorii din această clasă au prioritate mai mică decât cei aritmici.

Exemplu: Să se evaluateze valoarea de adevăr a expresiei:

$$10>1+12 \rightarrow \text{fals (0)}$$

$10>8\&\&!(15<10)\|6<=12$; se evaluatează $15<10 \rightarrow 0$ apoi $!0 \rightarrow 1$ $10>8$ are valoarea de adevăr 1 iar $1\&\&1$ conduce spre valoarea de adevăr 1. Evaluarea se oprește aici deoarece $1\|x=1$ oricare ar fi x . Deci valoarea de adevăr a întregii expresii este adevărat (1).

Operatori logici pe biți

Operator	Semnificație
&	și
	sau
^	xoR (sau exclusiv)
~	complement față de 1
>>	decalare la dreapta
<<	decalare la stânga

Operatorii logici pe biți lucrează la nivel de biți și se aplică pe toți octetii variabilelor implicate.

Exemplu:

Să se găsească rezultatul expresiei $7\&8$.

7	are reprezentarea	0000000000000000111
8	are reprezentarea	00000000000000001000
Aplicând operatorul și pe biți:		00000000000000000000

Deci $7\&8=0$.

Dacă folosim și logic $7\&\&8 \rightarrow 1$ (adevărat și adevărat \rightarrow adevărat).

Observație: Operatorii relaționali și logici generează întotdeauna un rezultat de adevărat sau fals (1 sau 0) în timp ce operatorii pe biți generează o valoare ce poate fi diferită de 0 sau 1 în funcție de operațiile prevăzute.

Operatorii de decalare mută toți biții unei variabile spre stânga sau spre dreapta cu un număr de poziții specificat.

var>>nr_pozitii	decalare la dreapta cu numărul de poziții specificat
var<<nr_pozitii	decalare la stânga cu numărul de poziții specificat

La fiecare decalare spre un capăt, la celălalt capăt se adaugă zerouri.

char x	x după fiecare execuție	Valoarea lui x
x=7	00000111	7
x=x<<1	00001110	14
x=x<<3	01110000	112
x=x<<2	11000000	192
x=x>>1	01100000	96
x=x>>2	00011000	24

Fiecare decalare la stânga cu o poziție este echivalentă cu o înmulțire cu 2, pierzându-se bitul cel mai semnificativ. Fiecare decalare la dreapta cu o poziție este echivalentă cu o împărțire cu 2 și se pierde bitul cel mai puțin semnificativ.



Exemplu:

Se consideră 2 numere întregi n,p unde $n \in (0,65.535)$ și $p \in (0,15)$. Să se seteze pe 1 bitul p din reprezentarea internă a lui n și să se afișeze noua valoare a lui n.

```
#include<stdio.h>
{ unsigned int n=5,p=1;
  n|=1<<p;           //operatorul sau se folosește pentru setarea pe 1 a unor biți
                      //având în vedere că x|0=x și x|1=1 oricare ar fi x
  printf("%u\n",n);
}
```

Să se seteze pe 0 bitul p din reprezentarea internă a lui n și să se afișeze noua valoare a lui n;

```
#include<stdio.h>
{
  unsigned int n=7,p=1;
  n&=~(1<<p);       //operatorul și se folosește pentru setarea pe 0 a unor biți
                      //având în vedere că x&0=0 și x|1=x oricare ar fi x
  printf("%u\n",n);
}
```

Să se înlocuiască primii p biți semnificativi din reprezentarea internă a lui n cu complementul lor față de 1.

```
#include<stdio.h>
{
  unsigned int n=7,p=3;
```

```

n^= ~0<<(8*sizeof(n)-p); //vom crea o mască formată din p biți 1 urmați de
//biți 0; în total vom avea 8*sizeof(n) biți
//iar x^0=x și x^1!=x, oricare ar fi x
printf("%u\n",n);
}

```

Operatorul de atribuire -Operatorul de atribuire se utilizează în construcții de forma:

nume_variabila=expresie;

Acesta are prioritatea cea mai mică. Construcția de mai sus se numește *expresie de atribuire*, fiind un caz particular de expresie. Tipul ei coincide cu tipul lui nume_variabila iar valoarea ei este valoarea atribuită lui nume_variabila. Dacă expresia din dreapta semnului egal are un tip diferit de cel al variabilei nume_variabila atunci întâi valoarea ei se convertește spre tipul acestei variabile și pe urmă se realizează atribuirea.

Mesajele de eroare ale compilatorului C precum și literatura de specialitate folosește 2 termeni legați de expresia de atribuire: *lvalue* și *rvalue*. *lvalue* este orice variabilă care poate să apară în partea stângă a expresiei de atribuire. Practic lvalue reprezintă variabila din expresia de atribuire. *rvalue* se referă la expresia din membrul drept al expresiei de atribuire. Practic rvalue conține valoarea expresiei.

Este permisă utilizarea unor expresii de atribuire multiplă de forma:

a = b = ... = x =expresie;

Pentru operația de atribuire, în afara semnului egal, se mai poate folosi și construcția op= unde op este un operator care face parte din mulțimea { %, /, *, -, +, &, |, ^, <<, >> }. Folosind această construcție putem obține programe C mai scurte, prin comprimarea instrucțiunilor de atribuire. Astfel expresia de atribuire:

v=v op expresie este echivalentă cu v op = expresie.

Construcție cu operator =	Exemplu	Forma echivalentă
=	v=10	v=10;
+=	v+=2	v=v+2
-=	v-=a	v=v-a
=	v=5	v=v*5
/=	v/=3	v=v/3
%=	v%=2	v=v%2
&=	v&=b	v=v&b
=	v =b	v=v b
^=	v^=a	v=v^a
<<=	v<<=poz	v=v<<poz
>>=	v>>=poz	v=v>>poz

Operatorii de incrementare / decrementare ++ / --

Operatorii de incrementare/decrementare pot să apară în formă prefixată sau postfixată. În forma prefixată putem avea $++v$ sau $--v$ iar în forma postfixată putem avea $v++$ sau $v--$.

Operatorul postfixat ($v++$) presupune ca expresia să fie evaluată la valoarea (lui v) dinainte de incrementare iar apoi să se realizeze incrementarea operandului.

Operatorul prefixat (de exemplu $++v$) presupune ca mai întâi să se realizeze incrementarea lui v iar apoi valoarea incrementată se returnează ca și valoare a expresiei.

Astfel, instrucțiunea $a=++v$ este echivalentă cu secvența:

$v=v+1$ (mai intai se face incrementarea)

$a=v$

Instrucțiunea $a=v++$ este echivalentă cu secvența:

$a=v$ (prima data se face evaluarea expresiei si abia apoi incrementarea)

$v=v+1$.

Exemplu:

```
int a,v=0;  
a=++v; //după executarea instrucțiunii v=1 și a=1  
a=v++; //a=1 și v=2  
a=v--; //a=2 și v=1  
a=--v; //a=1 și v=0;
```

Codul obiect produs de majoritatea compilatoarelor C este mai eficient în cazul utilizării operatorilor de incrementare/decrementare decât cel obținut prin utilizarea instrucțiunii de atribuire echivalente.

Operatorul de conversie explicită (cast)

Operatorul de conversie explicită este un operator prefixat care se utilizează în construcții de forma:

(tip) operand

De multe ori, este necesară forțarea tipului unui operand sau al unei expresii. Prin aceasta, tipul operandului (și implicit valoarea lui) se convertește spre tipul indicat între paranteze. Conversiile explicite sunt utile în cazurile în care se dorește ca rezultatul unei operații să fie de alt tip decât cel determinat implicit, pe baza tipurilor operanzilor.

Exemplu: Dacă se dorește obținerea rezultatului real, netrunchiat, al împărțirii a 2 numere întregi, cel puțin unul dintre operanzi trebuie convertit explicit la tipul double.

```
int a=10, b=4;  
double c;  
c=a/b;           //deși c este de tip double, el va conține valoarea 2  
c=(float)a/b    //în acest caz c va conține valoarea 2.5
```

Operatorul dimensiune (sizeof)

Operatorul dimensiune returnează numărul de octeți ai reprezentării interne a unei date. Acest operator prelucrează tipuri, spre deosebire de ceilalți operatori care prelucrează valori. Operatorul dimensiune se utilizează în construcții de forma:

```
sizeof(data)
```

unde data poate să fie variabilă simplă, nume de tablou, tip, element de tablou sau element de structură.

Deoarece tipul operanzzilor este determinat încă din faza de compilare, sizeof este un operator cu efect la compilare, adică operandul asupra căruia se aplică sizeof nu este evaluat, chiar dacă este reprezentat de o expresie.

Operatorul condițional

Operatorul condițional se utilizează în expresii de forma: exp1?exp2:exp3;

Această construcție are următorul *efect*: Se evaluatează valoarea expresiei exp1. Dacă ea are valoarea adevărat, atunci se evaluatează exp2, valoarea acesteia fiind și valoarea întregii expresii. Dacă exp1 are valoarea fals, atunci se evaluatează exp3 iar valoarea lui exp3 va fi și valoarea întregii expresii.

Exemplu:

```
int a=5, b=3,c;  
c=a>b?b:a;
```

Se evaluatează $a > b$, adevărat deci $c = 3$.

Construcția de mai sus este echivalentă cu a scrie:
if($a > b$) $c = b$;
else $c = a$;



Exemplu:

Să se scrie programul C care comparând valorile a 2 variabile o afișează pe cea mai mare.

```
#include<stdio.h>
int main()
{ int a=7,b=3;
  printf("maxim=%d\n",a>b?a:b);
}
```

Să se scrie programul C care comparând valorile a 3 variabile o afișează pe cea mai mare.

```
#include<stdio.h>
int main()
{
  int a=7,b=3,c=9;
  printf("maxim=%d\n",a>b?(a>c?a:c):(b>c?b:c));
}
```

Operatorul virgulă

Se utilizează când se dorește evaluarea mai multor expresii, acestea fiind evaluate de la stânga la dreapta, întreaga expresie luând valoarea ultimei evaluări.

Operatorul virgulă se folosește în expresii de forma:

expr1, expr2, ... exprn

Observație: Putem utiliza operatorul virgulă pentru a putea permite o serie de instrucțiuni de atribuire, scăpând astfel de necesitatea unei instrucțiuni compuse acolo unde instrucțiunile solicită acest lucru. Întreaga expresie care cuprinde mai multe instrucțiuni de atribuire este considerată o singură instrucțiune.

2.2.3.10. Regula conversiilor implicate

O expresie poate conține operanzi de tipuri diferite. În aceste cazuri, C aplică un sistem de conversii denumit promovarea tipului. Astfel se aplică următoarele reguli:

- Oricare operand de tipul char sau short va fi convertit în int. De asemenea fiecare float este extins spre double prin introducere de zeroruri în partea sa fracțională. Când un double este convertit spre float, de exemplu printr-o asignare, double-ul este rotunjit înainte de trunciere pe lungimea unui float.

- Dacă unul dintre operanzi este de tipul double, atunci și celălalt se convertește spre tipul double și rezultatul va fi de tipul double.
- Dacă un operand este long, celălalt este convertit în long și acesta va fi tipul rezultatului. Dacă un operand este unsigned, celălalt este convertit în unsigned și acesta va fi tipul rezultatului.

2.2.3.11. Variabile

Pentru tratarea corectă a entităților identificabile prin nume simbolice (variabile, funcții, constante simbolice), tipul lor trebuie precizat anterior primei utilizări printre declarație sau definiție corespunzătoare. În C toate variabilele trebuie declarate înainte de a fi folosite.

Exemplu:

```
char a,b,c;
int d;
double d;
float f;
```

Variabilele se pot defini în 3 locuri:

- în interiorul funcțiilor (locale) ;
- în cadrul definiției parametrilor funcțiilor (parametri formali);
- în afara oricărei funcții (globale).

În consecință, vorbim de *variabile locale, parametri formali și variabile globale*.

Variabilele locale (denumite și automatice) sunt accesibile doar instrucțiunilor care se găsesc în interiorul blocului în care au fost declarate.

În C, obligatoriu toate variabilele locale trebuie definite la începutul blocului în care sunt definite, înainte de orice instrucțiune a programului.

Exemplu:

```
void functie()
{
    int a;
    a=1;
    int b;      //incorrect în C
    b=2;
}
```

Dacă o funcție urmează să folosească argumente, ea trebuie să declare variabilele pe care le acceptă ca valori ale argumentelor. Aceste variabile sunt denumite *parametri formali* ai funcției. Parametri formali pot fi utilizați ca variabilele locale obișnuite. La ieșirea din funcție, acestea sunt distruse la rândul lor.



Exemplu: Să se scrie un program C care calculează $3!$ și $n!$, unde n se citește de la tastatură.

```

#include<stdio.h>
int factorial(int n) //n este parametru formal și este declarat
de tip întreg
{
    int i,fact=1;
    for(i=2;i<=n;i++) fact=fact*i; //în variabila locală fact calculăm n!
    return(fact); //funcția factorial returnează valoarea lui n!
}
int main()
{
    int v;
    printf("3!=%d\n",factorial(3)); //în funcția printf apelăm funcția factorial având
ca parametru real valoarea 3
    printf("Introd o valoare:");
    scanf("%d",&v);
    printf("%d!=%d\n",v,factorial(v)); //fcț factorial este apelată având ca param. real
val. citită în variabila v
}

```

Variabilele globale se definesc înfără oricarei funcții. Declarația unei variabile globale trebuie plasată înainte de prima utilizare. Este bine ca aceste declarații să se facă la inceputul unui fișier cod sursă. Ele sunt utile atunci când mai multe funcții utilizează aceleași variabile.



Exemplu: Să se rescrie programul de mai sus utilizând o variabilă globală.

```

#include<stdio.h>
int fact=1; //se definește variabila globală fact de tip întreg și se da val 1
void factorial(int n)
{
    int i;
    fact=1;
    for(i=2;i<=n;i++) fact=fact*i; //variabila globală fact va conține valoarea lui n!
}
int main()
{
    int v;
    factorial(3); //apelăm funcția factorial cu un parametru
    //real având valoarea 3
    printf("3!=%d\n",fact); //afișăm valoarea lui 3!; se observă că valoarea
    //variabilei globale fact este cunoscută atât în
    //funcția main cât și în funcția factorial
    printf("Introd o valoare:");
    scanf("%d",&v);
    factorial(v); //apelăm din nou funcția factorial pentru valoarea v
    printf("%d!=%d\n",v,fact); //afișăm valoarea lui v!; din nou folosim
    //variabila globală fact
}

```

Astfel putem defini *variabile globale* care pot fi accesate de toate funcțiile care sunt definite după instrucțiunea de definire a acesteia. Spre deosebire de variabilele locale, care se distrug în momentul părăsirii funcției, variabilele globale rămân disponibile pentru utilizare pe toată durata de execuție a programului. Definițiile pot să apară oriunde în interiorul unui program, dar funcțiile definite înaintea lor nu le vor recunoaște.

Practica tradițională în C utilizează literele mici pentru nume de variabile și literele mari pentru constante simbolice.

Variabile de tip tablou

Pentru tablouri se folosesc construcții de forma:

tip lista_de_elemente;

unde element are forma nume[lim1][lim2]... [lim n]. lim1, lim2 ... lim n reprezintă numere întregi sau expresii constante (expresii a căror valoare poate fi evaluată în faza de compilare).



Observație importantă: Indicii în C pornesc de la 0!

Exemplu:

```
int a[10];           //se declară un tablou de 10 numere întregi
float b[5][4];      //se declară un tablou de numere reale având 5 linii și 4 coloane
char c[4];          //se declară un sir de caractere ce poate conține maxim 3
                    caractere
```

Definiții / declaratii de variabile

Pentru o variabilă, putem avea o singură definiție și oricate declarații. Pentru variabilele locale, definițiile de variabile sunt în același timp și declarații și implică alocarea variabilelor respective în zona de stivă a memoriei.

Pentru variabilele globale, definiția implică alocare de memorie pentru variabilă în zona globală de date a programului. Pentru ca o variabilă globală să poată fi utilizată (de exemplu într-un alt fisier decât în cel unde a fost definită) se utilizează cuvântul cheie **extern** înainte de definiție.

Definiția unei variabile poate fi completată prin specificarea unei valori inițiale și atunci se spune că am inițializat variabila respectivă.

Momentul efectuării inițializării precum și eventualele inițializări implicate sunt condiționate de clasa de memorare în care este inclusă variabila respectivă.

- ❖ Variabilele alocate în zona globală de date a programului se inițializează o singură dată, înaintea începerii execuției programului, cu valoarea declarată, sau, în absența acesteia cu valoarea implicită zero.
- ❖ Variabilele din clasa *automatic* (locale funcțiilor) sunt inițializate numai dacă s-a cerut explicit, inițializarea fiind reluată la fiecare apel al funcției în care au fost definite.

Variabilele automatice sunt locale fiecărei apelări a unui bloc sau unei funcții și sunt declasate (își pierd declarația) la ieșirea din blocul respectiv.

Dacă nu sunt inițializate, aceste variabile conțin valori reziduale. Nici o funcție nu are acces la variabilele din altă funcție. În funcții diferite, pot exista variabile locale fără ca acestea să aibă vreo legătură între ele.

În absența inițializării explicite, variabilele globale sunt inițializate implicit cu valoarea 0 în timp ce variabilele locale au valori inițiale nedefinite (reziduale).

O caracteristică a limbajului C este că o declarație se poate referi la tipuri diferite, derivate din același tip de bază. Astfel, declarația:

```
char c, *p, sir[10];
```

se referă la trei variabile de tipuri diferite: caracterul c, pointerul la caracter p și sirul de caractere sir pentru care se rezervă 10 octeți.

2.2.3.12.Comentarii

Un comentariu începe cu succesiunea de caractere /* și se termină */ în cadrul comentariului neavând voie să apară aceste construcții (deci comentariile nu pot fi imbricate). Comentariile pot fi plasate oriunde în program, atât timp cât ele nu apar în mijlocul unui cuvânt cheie sau identificator.

Înse poate introduce un comentariu linie folosind succesiunea de caractere //. Datorită simplității celei de a doua forme, în exemplele prezentate am folosit comentarii linie, cu mențiunea că acestea sunt specifice limbajului C++.

Exemplu:

```
/* Aceasta este  
un comentariu în C*/  
int tablou[10][10]; //acesta e un comentariu linie C++; se declară un tablou  
//bidimensional
```

Urmăriți fișierul video de la următoarea adresă:



[https://drive.google.com/open?id=0B0E2G7UxqRojd3lBdjV4eG5n
NGM](https://drive.google.com/open?id=0B0E2G7UxqRojd3lBdjV4eG5nNGM)

Acesta prezintă aspectele teoretice și practice prezentate până la acest moment.

2.2.4. Probleme propuse

Se citește un număr întreg reprezentând o temperatură în grade Fahrenheit. Să se afișeze această temperatură exprimată în grade Celsius, utilizând formula: $C = (5 / 9) * (F - 32)$.

Să se scrie un program care determină dacă un an este bisect sau nu. Anii bisecți sunt divizibili cu 4, excepție făcând cei care sunt divizibili cu 100, care trebuie să fie divizibili și cu 400.

Se citește un număr natural n. Să se verifice dacă n este prim sau nu.

Se citește un număr natural n. Să se genereze sirul primelor n numere prime.

Se citește un număr natural n. Să se descompună în factori primi.

Se citește un număr natural n. Să se numere câte cifre conține acest număr.

Se citește un număr natural n. Să se numere câte cifre distincte conține acest număr.

Se citește un număr natural n și o cifră c. Să se indice numărul de apariții ale cifrei c în cadrul numărului n.

Se citește un număr întreg. Să se convertească numărul în baza 16.

Se citește un sir de caractere reprezentând un număr în baza 16. Să se afișeze numărul convertit în baza 10.

Se citește un număr întreg n. Să se verifice dacă numărul este divizibil cu diferența dintre cifra maximă și cea minimă.

Se citește un număr întreg n. Să se calculeze numărul maxim care se poate forma din cifrele acestui număr.

Se citește un număr natural n. Să se verifice dacă numărul n este palindrom sau nu. Un număr este palindrom dacă este egal cu inversul lui.

Exemplu: 121 este palindrom $121=121$

123 nu este palindrom $123 \neq 321$

Se citește un număr exprimat în baza p ($p \leq 100$). Să se convertească acest număr în baza q ($q \leq 10$).

Să se verifice, până la un număr dat n, conjectura lui Goldbach. Aceasta afirmă că orice număr par se poate scrie ca sumă de două numere prime. Pentru toate

numerele mai mici decât n, să se afișeze o descompunere în sumă de două numere prime.

Se citește un număr natural n. Să se genereze elementele sirului lui Fibonacci mai mici decât n, fără a folosi un sir.

$$\text{fibo}(0)=\text{fibo}(1)=1; \text{fibo}(n)=\text{fibo}(n-1)+\text{fibo}(n-2)$$

Se citește un număr natural n. Să se genereze un sir ce va conține primele n numere Fibonacci. Să se afișeze acest sir.

Se citește un număr natural n. Să se genereze toate numere perfecte mai mici decât n. Un număr este perfect dacă este egal cu suma divizorilor săi.

Exemplu: $6=1+2+3$; $28=1+2+4+7+14$

Se citesc două numere naturale m și n. Să se calculeze cel mai mic multiplu comun al celor două numere.

Se citește un sir de n numere întregi. Să se eliminate duplicatele din acest sir, astfel încât în sir să rămână doar o singură dată fiecare element.

Se citesc 2 siruri de m respectiv n numere întregi direct ordonate crescător. Să se interclaseze cele 2 siruri astfel încât să se obțină un al 3-lea sir direct ordonat crescător format din elementele celor 2 siruri.

Se citește un număr întreg n. Să se genereze sirul 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, ... n, n, ... n unde n apare de n ori.

Se citește un număr întreg n, $n > 16$. Să se calculeze 2^n . Având în vedere că acest număr, pentru un n mare, nu poate fi reprezentat exact, se va simula înmulțirea cu 2 folosind un sir compus din cifrele numărului.

Se citesc m și n lungimile a 2 numere întregi foarte lungi. Se citesc apoi cele m cifre reprezentând numărul a și n cifre reprezentând numărul b. Fiecare număr se va păstra în câte un sir ce va conține cifrele sale. Să se calculeze într-un al 3-lea sir suma celor două numere foarte mari.

Se citesc m și n lungimile a 2 numere întregi foarte lungi. Se citesc apoi cele m cifre reprezentând numărul a și n cifre reprezentând numărul b. Fiecare număr se va păstra în câte un sir ce va conține cifrele sale. Să se calculeze într-un al 3-lea sir produsul celor două numere foarte mari.

Să se scrie un program care calculează valoarea seriei:

$$S = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots + \frac{(-1)^n}{n}$$

Să se scrie un program care calculează valoarea numărului PI, știind că acesta este determinat de seria:

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots\right)$$

Se citesc 2 numere întregi, n și x . n copii stau în cerc și numără de la 1 la x . Cel care primește numărul x este eliminat și numărătoarea reîncepe de la următorul copil. Jocul se termină când rămâne un singur copil. Să se afișeze, la fiecare numărătoare, numărul inițial al copilului eliminat și numerele inițiale ale copiilor rămași în joc.

Se citesc coeficienții unui polinom de gradul n și o valoare reală v . Să se calculeze valoarea polinomului în punctul v .

Se citesc coeficienții unui polinom de gradul n . Se citește un număr natural k , $k < n$. Să se afișeze toate derivatele de ordinul k ale acestui polinom.

Se citesc coeficienții unui polinom de gradul n . Se citește un număr natural k , $k < n$ și o valoare v reală. Să se calculeze valoarea derivatei de ordinul k a polinomului în punctul v .

Problema determinării datei Paștelui: Algoritmul de mai jos stabilește data în care cade Paștele într-un anumit an. Algoritmul a fost elaborat de astronomul napolitan Aloysius Lilius și de matematicianul german Christopher Clavius, la sfârșitul secolului XVI. Este folosit de cele mai multe biserici apusene pentru a determina data duminică Paștelui pentru orice an de după 1582.

Fie Y anul pentru care se stabilește data Paștelui. Algoritmul este următorul:

1. calculează $G = (Y \bmod 19) + 1$
2. calculează $C = (Y/100) + 1$ (când Y nu este multiplu de 100, C este numărul secolului; 2005 este în secolul 21)
3. calculează $X = (3C/4) - 12$ și $Z = ((8C+5)/25) - 5$ An bisect este anul care este divizibil cu 4, cu excepția anilor care au doi de zero la sfârșit -1900, 2000, 2200- caz în care sunt bisecți doar anii divizibili cu 400 -2000, 2400. Z este o corecție specială, menită să sincronizeze Paștele cu orbita Lunii.
4. calculează $D = (5Y/4) - X - 10$ (găsește duminica)
5. calculează $E = (11G + 20 + Z - X) \bmod 30$. Dacă $E = 25$ iar numărul G este mai mare decât 11 sau dacă $E = 24$, atunci E se mărește cu 1 (E este aşa-numitul epact, care specifică apariția lunii pline)
6. calculează $N = 44 - E$. Dacă $N < 21$, atunci stabilește $N = N + 30$ (Pastele trebuie să fie în prima duminică ce urmează după prima lună plină, care apare în sau după 21 martie. Perturbațiile Lunii fac ca această zi să nu fie exactă, însă contează mai mult calendarul lunar, decât Luna. Ziua a N -a a lui martie, este luna plină calendaristică)
7. calculează $N = N + 7 - ((D + 7) \bmod 7)$
8. (se găsește luna) dacă $N > 31$, data este $(N - 31)$ aprilie, altfel, data este N martie

Observație: mod este restul împărțirii; toate împărțirile, se rotunjesc în jos, deci se ia doar partea întreagă. O eroare a acestui algoritm, este ca la punctul 5, E să iasă

negativ, iar restul pozitiv. De exemplu în anul 14.250, G=1, X=95, Z=40, iar E ar fi fost -24 și s-ar fi obținut ca data a Paștelui, data de 37 aprilie.

Se citește de la tastatură un sir de caractere reprezentând un număr întreg (eventual semn, urmat de un sir de cifre). Să se convertească acest număr într-un întreg (explicitarea funcției atoi() cu prototipul în math.h sau stdlib.h). Conversia se face până la primul caracter ce nu poate face parte din numărul întreg.

Se citește de la tastatură un sir de caractere terminat cu EOF. Să se numere caracterele albe (spațiu, return de car, tabulator) din acest sir de caractere.

Se citește de la tastatură un sir de caractere terminat cu EOF. Un cuvânt este orice succesiune de unul sau mai multe caractere diferite de blanc, tab sau linie nouă. Să se numere cuvintele distincte din acest sir de caractere.

Se consideră un fișier C care conține textul sursă al unui program C. Să se genereze un alt fișier, care să eliminate din textul sursă comentariile. Atenție la tratarea simbolurilor corespunzătoare începutului și sfârșitului de comentariu. Numele fișierului de intrare (codul sursă cu comentarii), respectiv numele fișierului sursă de ieșire se citesc din linia de comandă a programului, ca argumente ale funcției main.

Să se scrie un program care determină și afișează procentele de apariție a literelor din cadrul unui fișier text. Caracterele diferite de litere sunt ignorate iar literele mari sunt tratate la fel ca și literele mici.

2.2.5. Sumar

În cadrul acestui modul au fost prezentate concepțele de bază din programarea utilizând limbajul C.

2.2.6. Întrebări recapitulative

Întrebări pentru testul scris:

1. Ce este o funcție și care este structura unei funcții?
2. Care sunt etapele realizării unui program?
3. Care sunt tipurile de bază în limbajul C?
4. Care sunt operatorii aritmetici?
5. Care sunt operatorii relaționali?
6. Care sunt operatorii logici?
7. Care sunt operatorii logici pe biți?
8. Cum funcționează operatorii de incrementare/decrementare?
9. Ce este o variabilă?
10. Ce este o declarație și ce este o definiție?



Intrebări grilă:

1) Care va fi valoarea lui x după execuția secvenței:

```
int x=5,y=2;
char op='+';
switch(op) {
    case '+':x+=y;
    case '-':x-=y;
    default : x+=1;
}
```

a) eroare
b) x=5
c) x=6
d) x=7
e) x=10

2) int x,z=10;

```
for( x=0; x < 5; x++ ) { z--;
    if( x < 3 ) continue;
    if( x > 3 ) break;
    z-- }
```

printf("z=%d\n", z);

Ce valoare va fi afisata:

a z=4 b z=5 c z= 6
d z=7 e z=10

3) Care din urmatorii identificatori C nu este valid?

a. ___ b. S___ c. ___ident d. 1___
e. ___1

4) int x = 6,a=2,b=6,c=10;

```
if (x == b) x = a;
else x = b;
if (x != b) c+= b;
else c+=a;
printf ("c = %d\n", c);
```

Ce se va afisa la tiparire:
a c = 10 b c = 12
c c = 14 d c = 16 e c = 18

5) int x = 10;

do { ++x; } while (x > 50);

```
printf("x=%d\n", x);
```

Ce se va afisa dupa executie:

- a. x=10 b. x=11 c. x=50 d. x=51

e Ciclu infinit, nu va ajunge niciodata la printf

6) int i,j;

```
int ctr = 10;
```

```
int myArray[2][3];
```

```
for (i=0; i<3; i++)
```

```
    for (j=0; j<2; j++)
```

```
        { myArray[j][i] = ctr;
```

```
        --ctr; }
```

Care este valoarea elementului myArray[1][2]:

- a. 7 b. 6 c. 5 d. 4 e. 2

7)

```
int z = 0,y;
```

```
for( y=1; ++y < 8; )
```

```
    z += y;
```

```
printf("z=%d\n", z);
```

Ce se va afisa la executia codului de mai sus?

- a. z=8 b. z=9 c. z=28 d. z=27 e. z=35

8) Pe ce lungime se reprezinta in memorie sirul "ABCDE"?

- a. 5 b. 6 c. 10 d. 11 e. 12

9) Cand e necesara declararea unui sir?

a. Cand e necesar sa se pastreze constante

b. Cand e necesar sa se pastreze date de acelasi tip

c. Cand e necesar sa se obtina o eliberare automata a memoriei

d. Cand e necesar sa se pastreze date de tipuri diferite

e. Cand e necesara alocarea dinamica a memoriei

10) #include <stdio.h>

```
void func()
```

```
{ int x = 10;
```

```
    int y = 10;
```

```
    x++; y++;
```

```
    printf( "%d -- %d ", x, y );
```

```
}
```

```
void main()
```

```
{ func();
```

```
    func(); }
```

Care va fi rezultatul executiei:

- a. 10 -- 10 10 -- 10

- b. 11 -- 11 11 -- 11

- c. 11 -- 11 12 -- 12

- d. 10 -- 10 11 -- 11

e. 12 -- 12 12 -- 12

11) int x = 3;

if(x = 2); x = 0;

if(x = 3) x++; else x += 2;

Ce valoare va contine x dupa executie:

a Eroare la compilare

b 1 c 2 d 3 e 4

12) int m = 14; int n = 6; int o;

o = m % ++n;

n += m++ - o;

Care vor fi valorile lui m,n,o:

a m=15 n=21 o=0

b m=15 n=20 o=0

c m=15 n=20 o=2

d m=15 n=19 o=2

e m=15 n=7 o=2

13)double x = 4.5e-2;

Ce valoare va contine x dupa executia codului de mai sus?

a. 4500 b. 450 c. 4.5 d. 0.045 e. 0.0045

14) Daca avem date 6 variabile intregi a=1; b=2; c=3; d=4; e=5; f=6; ce se va tipari dupa executia codului de mai jos?

a= a > b ? a : c > d ? e : f;

printf("%d",a);

a. 3 b. 4 c. 5 d. 6 e. niciuna de mai inainte

15) int z; int x = 5; int y=-10; int a=4; int b=2;

z = x++ + ++y * b / a;

Ce numar va contine z dupa executia exemplului de mai sus?

a. 2 b. 1 c. 0 d. -2 e. -3

16) Care din urmatoarele enunturi nu este adevarat:

a O functie poate fi utilizata chiar daca ea nu primeste si nu returneaza nici o valoare

b Revenirea in programul principal dintr-o functie apelata se poate realiza utilizand o instructiune return

c Utilizand o instructiune return functia apelata poate returna programului apelant doua sau mai multe valori

d O functie poate sa nu primeasca nici un parametru de apel

e O functie apelata poate apela la randul ei o alta functie

17) Daca dorim ca intr-o instructiune for in prima expresie ce se executa la inceputul instructiunii for o singura data sa introducem mai multe expresii de initializare, le vom desparti prin:

a Caracterul : b Caracterul , c Caracterul ;

d Caracterul spatiu

e Oricare din caracterele de mai sus

18) Daca un vector a fost declarat prin

float num[MAX]

care din urmatoarele exemple permite citirea de la tastatura a elementelor vectorului:

- a for(j=1;j<=MAX;j++) scanf("%f",num[j]);
- b for(j=0;j<=MAX;j++) scanf("%f",num[j]);
- c for(j=0;j<MAX;j++) printf("%f",num[j]);
- d for(j=0;j<MAX;j++) scanf("%f",&num[j]);
- e for(j=1;j<=MAX;j++) scanf("%d",&num[j]);

19) Una din urmatoarele propozitii este adevarata: Daca o functie este apelata prin instructiunea gama(&alfa, &beta)

a Functia apelata primeste ca parametri 2 adrese iar valorile acestor variabile pot fi modificate in cadrul functiei

b Functia apelata primeste ca parametri 2 adrese dar variabilele respective nu pot fi modificate

c Functia apelata primeste ca parametri 2 adrese iar la revenire valorile variabilelor raman nemodificate

d Apelul este identic cu unul de forma gama(alfa,beta)

e Eroare la compilare

20) double z;

z = (double) (5 / 10); printf("z = %.2f\n", z);

Care va fi rezultatul executiei:

- a z = -0.50
- b z = -1.0
- c z = 0.00
- d z = 0.50
- e z = 1.00

21) Care din urmatoarele citeste un caracter si il retine in variabila c:

- a c = gets();
- b getchar(&c);
- c c = getchar();
- d getc(&c);
- e c = getc();

22) char c1;

c1 = 'A' + 4; printf("c1 = %c\n", c1);

Care din urmatoarele descriu acest cod:

- a Programul va da eroare la compilare
- b Va afisa un singur caracter valid
- c Programul se va compila dar se va bloca la rulare
- d Va afisa un caracter invalid la iesire
- e Programul va afisa c1=A4.

23) int i,j;

```
int ctr = 0;
int myArray[2][3];
for (i=0; i<3; i++)
    for (j=0; j<2; j++)
        { myArray[j][i] = ctr;
          ++ctr; }
```

Care este valoarea elementului myArray[1][2]:

- a 1
- b 2
- c 3
- d 4
- e 5

24) #include <stdio.h>
void func()
{ int x = 0;
 int y = 0;
 x++; y++;
 printf("%d -- %d ", x, y);
}
int main()
{ func();
 func();
 return 0; }

Care va fi rezultatul executiei:

- a 0 -- 0 1 -- 1
- b 0 -- 0 0 -- 0
- c 1 -- 1 2 -- 2
- d 2 -- 2 2 -- 2
- e 1 -- 1 1 -- 1

25) Ce cuvant indica ca o functie nu returneaza nici o valoare:

- a undefined
- b nothing
- c void
- d null
- e empty

26) Care din urmatorii identificatori nu este valid:

- a pace_in_cosmos
- b nrMare
- c g42277
- d __ident
- e char

27) Directiva "include" permite ca:

- a Un fisier sursa sa fie inclus in alt fisier sursa
- b Un fisier obiect sa fie inclus intr-un program

- c Un fisier sursa sa fie inclus intr-o biblioteca
- d Un fisier obiect sa fie inclus intr-o biblioteca
- e Un fisier obiect sa fie inclus in alt fisier obiect

28) O bucla WHILE este preferabila unei bucle FOR cand:

- a Numarul de treceri prin bucla este cunoscut inainte ca bucla sa fie executata
- b Cele 2 instructiuni sunt identice
- c Este intotdeauna preferabila o instructiune for
- d Cand avem nevoie de o iesire fortata
- e Cand conditia de iesire din bucla apare incidental

29) int a, b=1, c=2, d=3, e=4, f=5;

a = b = c = d = e = f;

Care va fi valoarea lui a:

- a 1
- b 2
- c 3
- d 4
- e 5

30) int x = 1/2;

```
if (x) printf ("x=%d\n", x);
else printf ("x=%d\n", x*2);
```

Care va fi rezultatul executiei:

- a x=0
- b x=0.5
- c x=1
- d x=2
- e Eroare la compilare

31) int myArray[] = { 1, 2, 3, 10, 20, 30 };

Câți octeți va ocupa sirul myArray:

- a Nu este o initializare corecta si va produce eroare la compilare.
- b 6
- c 7
- d 12
- e 14

După parcurgerea modulului, cursanții vor fi capabili să programeze utilizând limbajul C. Exemple de probleme propuse și probleme rezolvate se găsesc în modulul 3.

2.2.7. Bibliografie modul

Cristian Bologa –Algoritmi și structuri de date, Editura Risoprint, 2006 –capitolul 2
 Liviu Negrescu, Limbajele C și C++ pentru incepatori , Vol. I (p.1 si 2) - limbajul C (editia XI), Editura Albastra, Cluj-Napoca, 2005

2.3. Modulul 3: NOȚIUNI AVANSATE DE LIMBAJ C

2.3.1. Scopul și obiectivele modului

Acest modul continuă noțiunile de limbaj C începute în modulul anterior și prezintă partea de tipuri structurate, de pointeri și de lucrul cu fișiere în limbajul C.

2.3.2. Scurtă recapitulare a conceptelor prezentate anterior

In cadrul primului modul au fost prezentate noțiunile introductive legate de algoritmi și de limbaj. Modulul doi inițiază cursanții în primele noțiuni legate de limbajul C.

2.3.3. Conținutul informațional detaliat

2.3.3.1. Structuri

Frecvent, în practică, se dorește prelucrarea unor informații structurate, compuse din mai multe valori primitive.



O *structură* este o colecție de valori eterogene stocată într-o zonă de memorie compactă. O structură este compusă din una sau mai multe elemente, numite câmpuri, în care fiecare câmp are propriul său nume și tip. Memoria alocată unei structuri este o secvență continuă de locații, câte o locație pentru fiecare câmp. Câmpurile sunt numite și membri ai structurii sau elemente ale structurii. Ordinea de memorare a câmpurilor corespunde cu ordinea de descriere a acestora în cadrul structurii.

Structurile permit organizarea datelor complexe, permitând ca un grup de variabile legate să fie tratate ca o singură entitate. O structură are următorul *format general*:

```
struct nume_structura {  
    tip1 camp1;  
    tip2 camp2;  
    ...  
    tipN campN;  
} variabila1, ..., variabilaN;
```

Observații:

1. O declarație de structură se termină cu ‘;’, deoarece reprezintă definiția unor variabile.
2. O declarație de structură neurmătă de o listă de variabile definește doar un tip structură, un şablon pe baza căruia se pot declara variabile de tipul structurii astfel definite.



Exemplu:

```
struct punct {  
    int x;  
    int y;  
} p;  
  
int main ()  
{  
    struct punct alt_punct;  
    printf ("Introduceticeledouacoordonate: ");  
    scanf ("%d %d", &p.x, &p.y);  
    p.x = p.x + 21;  
    p.y = p.y + 10;  
    //între structuri se pot face atribuiriri, se copiaza TOATE câmpurile  
    alt_punct = p ;  
    printf ("Coordonatelesunt: (%d %d) \n", alt_punct.x, alt_punct.y);  
    return 0;  
}
```

Exemplul anterior declară o structură numită punct, alcătuită din două *câmpuri*: x și y, precum și variabila p de acest tip. Orice variabilă de tip *struct tpunct* va putea memoră coordonatele plane ale unui punct (exemplu variabila alt_punct). Membrii unei variabile structurate se accesează cu operatorul .(punct).

```
p.x = p.x + 21;  
p.y = p.y + 10;
```

Exemplu:

```
structdata_calendaristica { //definim structura cu numele data_calendaristica  
    int zi;  
    int luna;  
    int an;  
    int secol;  
    int mileniu;  
} d; //variabila d va fi de tip structura  
//data_calendaristica  
  
struct data_calendaristica data_angajarii, data_nasterii;  
//definim variabilele data_angajarii, data_nasterii de tip  
// structura data_calendaristica
```

Numele complet al unui câmp se obține din numele structurii urmat de caracterul “.” (denumit operatorul punct) și numele câmpului referit. Astfel pentru a referi câmpurile din structură (numite și elemente sau membrii ai structurii) vom folosi

d.zi, d.lunași d.an. Câmpul selectat se comportă ca o variabilă și i se pot aplica toate operațiile care se pot aplica variabilelor de acel tip.

Un membru al unei structuri poate avea același nume cu o structură sau cu o variabilă oarecare, nemembru, deoarece nu se vor genera conflicte datorită faptului că numele de structuri se află într-un spațiu de memorie separat de cel al numelor de variabile iar referirea unui membru al unei structuri respectiv referirea unei variabile se face diferit în cadrul unui program.

Exemplu tipic de structură este înregistrarea unui salariat. Aceasta poate conține atribută precum marca salariatului, numele, prenumele, adresa, salarul brut, data nașterii, locul nașterii, data angajării etc. Se observă că structurile pot fi imbricate, adică membrii unei structuri pot fi la rândul lor structuri (este cazul datei angajării sau al datei nașterii, care sunt la rândul lor structuri):

```
struct angajat {  
    int id;  
    char nume[25];  
    char prenume[25];  
    struct data_calendaristica data_nasterii;  
    struct data_calendaristica data_angajarii;  
    ...  
} ang;
```

Referirea în acest caz se face sub forma `ang.data_nasterii.zi` sau pentru un alt membru `ang.data_angajarii.an`.

Standardul ANSI C specifică faptul că structurile trebuie să permită imbricarea pe cel mult 15 niveluri.

Atunci când o structură este folosită ca argument al unei funcții, întreaga structură este transmisă folosind metoda standard de apelare prin valoare. Aceasta înseamnă că orice modificare a conținutului structurii în interiorul funcției căreia îi se transmite ca parametru nu afectează structura utilizată ca argument. Dacă se dorește adresa unui membru individual al unei structuri, se folosește operatorul ‘&’ înaintea numelui structurii. Operatorul ‘&’ precede numele structurii, nu pe cel al membrului individual.

Exemplu:

```
functie(&ang.id, ang.nume); //în cazul numelui nu este necesară utilizarea  
                           //operatorului '&' având în vedere că nume este deja o adresă
```

Se poate ca informația conținută într-o structură să fie atribuită unei alte structuri de același tip folosind o singură instrucțiune de atribuire (deci nu trebuie atribuită valoarea fiecărui membru separat).

Exemplu:

```
struct angajat ang1, ang2; //definim variabilele ang1 și ang2 de tipul struct angajat
```

```
ang2=ang1;           //atribuim valorile câmpurilor variabilei ang1 câmpurilor
                    //corespondente din variabila ang2
```

Se pot defini masive de structuri.

```
struct angajat salariati[300]; //definim un masiv de 300 elemente, fiecare element
                                fiind de tip structură
```

Referirea unui element din acest masiv se face astfel:

```
int ln;
ln=salariati[5].data_nasterii.luna; //variabila ln va conține câmpul
                                         //data_nasterii.luna din al 6-lea element al structurii
```

2.3.3.2. Asignari de nume pentru tipuri de date

Tipurile de baza ale limbajului C, numite și tipuri predefinite se identifică printr-un cuvânt cheie (int, char, float). Tipurile structurate se definesc printr-o declaratie struct nume {};

Programatorul poate să atribuie un nume unui tip, indiferent de faptul că acesta este predefinit sau definit de utilizator. Aceasta se realizează prin intermediul constructiei typedef care are urmatorul format

```
typedef definitie_tip identifier;
```

După ce s-a atribuit un nume unui tip, numele respectiv poate fi utilizat pentru a declara date de acel tip, exact la fel cum se utilizează în declarații cuvintele cheie int, char, float, etc.

Exemplu:

1. Definiția: `typedef unsigned short USHORT;`

atribuie numele USHORT definitiei de tip unsigned short.

Deasemenea, dacă dorim să lucrăm cu siruri de lungime 40 sub numele str40 folosim definiția:

```
typedef char str40[41];
```

In contextul acestei definiții declarația
str40 a,b;

defineste două stringuri de lungime 41 de caractere. Astfel instrucțiunea
`printf("\nsizeof(a) = %d\n", sizeof(a));`

va tipări următoarele
`sizeof(a) = 41`

2. Declarațiile
`typedef int INTREG;`
`typedef float REAL;`

pot fi folosite ca si cuvintele cheie int si float. Astfel declaratia:
INTREG x,y,a[100];

este identica cu int x,y,a[100];

```
typedef struct data_calend { int zi;char luna[11];int an;  
} DC;
```

Prin aceasta declaratie se atribuie denumirea DC tipului structurat data_calend. In continuare putem declara date de tip DC:

```
DC data_angajarii, data_nasterii;  
DC data_crt={20,"septembrie",1991};
```

2.3.3.3.Uniuni



O *uniune* este o locație de memorie care este partajată de două sau mai multe variabile, în general, de tipuri diferite. Sintaxa declaratiei unei uniuni este similară cu cea a structurii:

```
union nume_generic {  
    tip nume_variabilă;  
    tip nume_variabilă;  
    ...  
} variabile_uniune;
```

unde identificatorii declarați ca membrii reprezintă nume cu care sunt referite obiectele de tipuri diferite care utilizează în comun zona de memorie.

Spațiul de memorie alocat corespunde tipului de dimensiune maximă.

Tipurile uniune oferă posibilitatea unor conversii interesante, deoarece aceeași zona de memorie poate conține informații organizate în moduri diferite, corespunzătoare tipurilor membrilor.

Exemplu:

```
union exemplu_uniune {  
    int i;  
    char ch;  
};  
union exemplu_uniune eu; //declarăm variabila eu de tipul uniune definit mai sus
```

Când este declarată o variabilă de tip uniune, compilatorul alocă automat memorie suficientă pentru a păstra cel mai mare membru al acesteia. Evidența dimensiunilor și a aliniamentului o va ține compilatorul.

În variabila eu atât întregul i cât și caracterul ch împart aceeași locație de memorie, în care i ocupă 2 octeți iar ch doar unul. Ne putem referi la datele stocate în variabila uniune definită atât ca la un caracter, cât și ca la un întreg, din orice parte a programului. Sintactic, membri unei uniuni sunt accesibili prin construcții de forma:

nume_uniune.membru sau pointer_la_uniune->membru.

Exemplu:

```
eu.i=10;  
eu.ch='a';
```

Mărimea unei structuri sau a unei uniuni poate fi egală sau mai mare decât suma mărimilor membrilor săi. De aceea, când este necesară cunoașterea mărimii unei structuri sau uniuni, se va folosi operatorul sizeof.

Uniunile pot apărea în structuri și masive și invers. Sintaxa pentru a apela un membru al unor astfel de structuri imbricate este aceeași (deci se folosește operatorul “.” sau “->” în cazul în care referirea se face folosind un pointer).

Observație: Este ilegal a declara o structură sau o uniune care face apel la ea însăși, dar o structură sau uniune poate conține un pointer la un apel spre ea însăși.

2.3.3.4.Câmpuri de biți

Limbajul C ofera posibilitatea de a structura datele la nivel de bit. Astfel, unor membri de structuri sau uniuni, li se pot aloca, dintr-un octet, biți individuali sau grupuri de biți. În felul acesta, se definesc câmpuri de biți care pot fi accesate fiecare, separate de restul octetului, pentru evaluare și/sau modificare.

Forma generală de definire a unei structuri cu membri de tip câmp de biți este:

```
struct nume_generic {  
    tip nume1:lungime;  
    tip nume2: lungime;  
    ...  
    tip numen: lungime;  
} lista_variabile;
```

Pentru campurile de biți există următoarele **restrictii**:

- tipul poate fi *int*, *signed* sau *unsigned*;
- *lungime* este o constantă întreagă cu valori între 0 și 15;
- nu se poate evalua adresa unui camp de biți;
- nu se pot organiza tablouri de campuri de biți;
- nu se poate ști cum sunt rulate câmpurile (de la dreapta la stânga sau invers, funcție de echipament).

Câmpurile de biti pot fi utile atunci cand informatia furnizata de anumite echipamente este transmisa prin octet sau atunci cand se doreste accesul la bitii unui octet sau atunci cand memoria este limitata si anumite informatii pot fi stocate intr-un singur octet.

Câmpurile de biți permit accesul la nivel de bit pentru elementele acestui tip de structură. Deoarece în C nu există tipul boolean, astfel se pot stoca mai multe variabile booleene (adevărat sau fals, 1 sau 0) utilizând un singur octet.

De fapt, un câmp de biți este un tip special de structură, care precizează cât de lung trebuie să fie fiecare câmp, lungime exprimată în biți.

Exemplu:

```
struct angajat {  
    struct date_pers datep;  
    float salar;  
    unsigned activ:1;  
    unsigned orar:1;  
} ang;
```

Se observă că este corect să amestecăm membri obișnuiți ai unei structuri cu câmpuri de biți. Astfel, vom defini o înregistrare care în loc să utilizeze 2 octeți pentru a păstra informațiile activ (angajat activ sau inactiv) și orar (retribuție lunară sau orară) va folosi doar 2 biți pentru reținerea celor 2 informații. Rezultă de aici o economie de memorie de 14 biți (pentru acest caz) pentru fiecare element de structură de acest tip.

Nu este obligatorie numirea fiecărui element dintr-o structură de tip câmp de biți. Dacă dorim utilizarea doar a ultimilor 2 biți dintr-un octet, putem defini câmpul de biți astfel:

```
struct cb {  
    unsigned: 6;  
    unsigned bit7: 1;  
    unsigned bit8: 1;  
}
```

Nu se poate obține adresa unui câmp de biți, deoarece nu se știe dacă aceste câmpuri vor fi rulate de la dreapta spre stânga sau invers, depinzând de compilator.

2.3.3.5.Enumerări



O *enumerare* este un set de constante care specifică toate valorile permise pe care le poate avea o variabilă de acel tip. Permite utilizatorului să folosească în program nume sugestive în locul unor valori numerice. De exemplu în locul numărului unei luni calendaristice se poate folosi denumirea ei sau în loc de 0 și 1 se poate folosi ADEVĂRAT sau FALS.

Formatul general de declarare a unei enumerări este:

```
enum nume_generic {lista enumerărilor} lista_variabile_enumerare;
```

– constantele pot avea specificate valori (și o valoare se poate repeta)
enumluni_curs {ian=1, feb, mar, apr, mai, iun, oct=10, nov, dec};

- implicit, sirul valorilor e crescător cu pasul 1, iar prima valoare e 0
- un nume de constantă nu poate fi folosit în mai multe enumerări
- tipurile enumerare sunt tipuri întregi, variabilele enumerare se pot folosi la fel ca variabilele întregi
- cod mai lizibil decât prin declararea separată de constante

```
enum {D, L, Ma, Mc, J, V, S} zi; // tip anonim; declară doar variabilă zi  
                                // tipul nu are nume, nu mai putem declara altundeva variabilele
```

```
intnr_ore_lucru[7]; //număr de ore pe zi  
for (zi = L; zi <= V; ++zi) nr_ore_lucru[zi] = 8;
```

```
enum curcubeu {rosu, orange, galben, verde, albastru, indigo, violet};  
enum curcubeu ec;  
printf("%d %d\n", rosu, verde); //va afișa 0 3.
```

Asupra unei variabile enumerare se pot face atribuiri:
ec=albastru;

sau valoarea lor poate fi comparată cu una din valorile din enumerare:

```
if (ec==albastru) printf("Albastru!\n");
```

Pentru a specifica valoarea unuia sau mai multor simboluri putem folosi inițializări. Simbolurilor care apar după inițializare li se atrbuie automat valori ce urmează după acea valoare.

```
enum curcubeu {rosu, orange, galben=100, verde, albastru, indigo, violet} ec;  
printf("%d %d\n", verde,violet); //va afișa 101 104.
```

Exemplu:



```
typedef enum {verde, galben,rosu} culoare;  
culoare semafor;  
semafor=galben;  
.....  
switch(semafor){  
case verde:libera_trecere();break;
```

```

case galben: asteapta();break;
case rosu: oprire_motor();break;
default: semafor_inactiv()
}

```

Exemplu:



```

#include <stdio.h>
enum culori {ROSU, GALBEN, VERDE, ALBASTRU, VIOLET,
NUMAR_CULORI};
typedef enum culori TCuloare;

int main ()
{
    TCuloare cer, padure;
    printf("In enum sunt %d culori\n",NUMAR_CULORI);
    cer=ALBASTRU;
    padure=VERDE;
    printf("cer=%d\n",(int)cer);
    printf("padure=%d\n",(int)padure);
    return 0;
}

```

In mod natural, diferitele tipuri definite de utilizator pot fi combinate, astfel încât putem avea de exemplu tablouri de structuri sau invers, tablouri ale caror elemente sunt enumerări. Pentru a evita declarații de genul struct node x,y,z sau enum boolean s,b; limbajul C furnizează mecanismul de definire a numelui de tipuri cu ajutorul lui typedef. Aceasta creează tipuri sinonime cu cele denumite și care pot fi utilizate în declarații de variabile, mărind claritatea programului. Este sugestivă definirea în programul anterior a tipului TCuloare și utilizarea lui în programul principal în locul tipului enum culori.

Urmăriți fișierul video de la următoarea adresă:



<https://drive.google.com/open?id=0B0E2G7UxqRojaEdsVU5EaXVRWDA>

Acesta prezintă aspectele teoretice și practice prezentate până la acest moment.

2.3.3.6. Pointeri și adrese



Un pointer este o variabilă care conține o adresă de memorie.

Pointerii sunt foarte mult utilizăți în C pe de o parte pentru că uneori sunt singura cale de rezolvare a unei anumite probleme, iar pe de altă parte pentru că folosirea lor duce la alcătuirea unui cod mai compact și mai eficient.

Ca metodă, pointerii se utilizează pentru un plus de simplitate.

Un pointer fiind o variabilă ce conține o adresă de memorie, în particular, ea poate referi adresa unei variabile din program.

Deci este posibilă adresarea acestei variabile "indirect" prin intermediul pointerului.

Variabilele de tip pointer se declară prin construcții de forma:

```
tip *nume;
```

Fie x o variabilă, de tip int și px este un pointer la acea variabilă.

Operatorul & dă adresa unei variabile, astfel încât instrucțiunea :

```
px=&x
```

dă variabilei px adresa lui x, px înseamnă "pointează pe x". Operatorul & poate fi aplicat numai variabilelor și elementelor unui tablou.

Invers, dacă avem un pointer px, prin *px se face referire la valoarea care se găsește memorată la adresa pointată de px.

Exemplu.

```
int *p, n=5, m;      //declarăm p ca pointer la întreg și n și m întregi
p=&n;              //p va conține adresa variabilei n deci p va pointa (indica) spre n
m=*p;              //lui m i se dă valoarea care se găsește la adresa indicată de p
                   //deci m va conține valoarea lui n; în consecință m=5
m=*p+1;            //m va conține valoarea 6
```

Construcții ca &(x+1) și &3 sunt interzise. Este deosebita interzisă păstrarea adresei unei variabile registru.

Operatorulunar * tratează operandul său ca o adresă, accesează această adresă și obține conținutul.

Astfel, dacă y este tot un int

```
y = *px  
asignează lui y, ori de câte ori este cazul, conținutul locației unde pointează px.
```

Astfel secvența

```
px = &x;  
y = *px;  
asignează lui y aceeași valoare ca și  
y = x
```

Totodată este necesară declararea variabilelor care apar în secvență:

```
int x, y;  
int *px;
```

Declararea lui x și y este deja cunoscută. *px este un int, adică în momentul în care px apare în context sub forma *px, este echivalentă cu a întâlni o variabilă de tip int. De fapt, sintaxa declarării unei variabile imită sintaxa expresiilor în care ar putea să apară respectiva variabilă. Acest raționament este util în toate cazurile care implică declarații complicate.

Exemplu:

double atof(), *dp; //atof() și *dp au valoare de tip double.

De notat declarația implicită, ceea ce vrea să însemne că un pointer este constrâns să pointeze o anumită categorie de obiecte (funcție de tipul obiectului pointat).

Pointerii pot apare în expresii.

De exemplu, dacă px pointează pe întregul x atunci *px poate apare în orice context în care ar putea apărea x.

y = *px + 1

dă lui y o valoare egală cu x plus 1.

printf("%d\n", *px)

imprimă o valoare curentă a lui x și d = sqrt((double) *px) face ca d = radical din x, care este forțat de tipul double înainte de a fi transmis lui sqrt.

În expresii ca

y = *px + 1

operatorii unari * și & au prioritate mai mare decât cei aritmici, astfel această expresie, ori de câte ori pointerul px avansează, adună 1 și asignează valoarea lui y.

Referiri prin pointer pot apărea și în partea stângă a asignărilor. Dacă px pointează pe x atunci

*px = 0

îl pune pe x pe zero și

*px += 1

îl incrementează pe x, ca și

(*px)++

In acest ultim exemplu parantezele sunt necesare; fără ele, expresia va incrementa pe px în loc să incrementeze ceea ce pointează px deoarece operatorii unari * și + sunt evaluati de la dreapta la stanga.

Dacă pointerii sunt variabile, ei pot fi manipulați ca orice altă variabilă. Dacă py este un alt pointer pe int, atunci

py = px

copiază conținutul lui px în py făcând astfel ca py să se modifice odată cu px.

O mare atenție trebuie acordată tipului variabilei spre care pointează un pointer. Următorul exemplu este sugestiv în acest sens:

```
int *p;  
double x=1.23, y;  
p=&x;  
y=*p;      //valoarea lui y va fi total eronată datorită tipului pointerului p  
           //p este pointer spre întreg; *p va lua din x doar primii 4 octeti și apoi ii va  
           //converti la int  
           // valoarea *p va fi convertita la double si salvata in y  
           //atentie : nu rezulta eroare de compilare, se da doar un warning
```

2.3.3.7. Aritmetică adreselor

Pentru salvarea datelor, un program scris în C poate folosi 3 tipuri de memorie – care partajază o zonă comună și anume zona de date a programului. Astfel, zona de date se imparte în următoarele: memoria statică, stiva și memoria dinamică (heap).

Codul aferent funcțiilor care sunt necesare pentru execuția programului (funcția main precum și toate celelalte funcții care sunt apelate în program) este salvat într-o altă zonă de memorie denumită zona de cod.

Zona de date împreună cu zona de cod reprezintă spațiul de memorie alocat de sistemul de operare pentru execuția unui program.

În cele ce urmează, ne vom referi la zona de date. Pointerii obisnuiți din program referă adrese din această zonă.

1. **Memoria globală sau statică** găzduiește variabile definite globale sau variabilele definite cu **static** (în fișiere, funcții...). Caracteristica esențială a acestei zone este că ea este alocată și inițializată de compilator înainte ca execuția programului să intre în funcția main, și există până după ce această funcție principală se încheie. Mărimea zonei globale este determinată la compilare și este dată de mărimea variabilelor globale și statice din program. Mărimea acestei zone este fixă, pe totă durata execuției programului. Dacă nu se specific altfel, toate variabilele din zona globală sunt initializate cu valoarea 0.

2. **Stiva** este o zonă de memorie unde compilatorul alocă spații conform principiului stivei (LIFO). Aceste spații sunt legate strict de funcții, de variabilele definite în cadrul funcțiilor. În această zonă se definesc de către compilator variabilele locale (funcțiilor sau domeniilor de vizibilitate). Aici intră parametrii cu care funcția este apelată și variabilele locale obișnuite - NU și cele precedate de static. Deci, funcția main are zona de stiva A, la intrare în funcție. Dacă în main există un apel la funcția f(), la apelul acesteia, în stivă se adaugă zona B care conține variabilele definite în f. O zonă pe stivă există cât timp execuția se află în respectiva funcție sau în funcții apelate mai departe din aceasta. Odată ce funcția returnează, zona aferentă de stivă este dezalocată, și zona de stivă curentă va fi cea aferentă a funcției apelante.

3. În **memoria dinamică** sau **heap** alocăm variabilele dinamic, adică în timpul rulării programului (cu funcția malloc). Această zonă de memorie stă la dispoziția programului, care face alocări în funcție de cele petrecute în timpul execuției.

Pe parcursul execuției programului, zona de stivă respectiv heap crește și descrește, după cum programul intră sau ieșe din funcții, sau după cum programatorul aloca / dezalocă variabile dinamice.

Pentru primele 2 tipuri de memorie (zona globală și zona de stivă), programatorul știe la momentul scrierii programului cantitatea de memorie necesară la un anume punct în execuția programului, și anume variabilele disponibile într-un punct de execuție, însă pe heap se aloca variabile în funcție de necesarul la rulare. Spre exemplu, un editor text va aloca sirul de caractere pe heap, câtă vreme nu știe dacă utilizatorul va introduce 10, 100 sau mai multe caractere.

Dacă p este un pointer, atunci $p++$ incrementează pe p în aşa fel încât acesta să pointeze pe elementul următor indiferent de tipul variabilei pointate, iar $p+=i$ incrementează pe p pentru a pointa peste i elemente din locul unde p pointează curent.

Dacă p și q sunt pointeri, relații ca $<$, $>$, $==$, $!=$, funcționează.

$p < q$

este adevarata, de ex, în cazul în care adresa lui p este mai mică (logic) decât adresa lui q . dacă p și q pointează către 2 elemente ale aceluiași tablou, atunci $p < p$ înseamnă faptul că p arată către un element cu indice mai mic decât q .

Dacă se testează cu \leftrightarrow pointeri care sunt adrese efective, rezultatul (din punct de vedere al variabilelor din program) nu are sens. Însă acest lucru este permis de către compilator.

Relațiile $==$ și $!=$ sunt și ele permise, între 2 pointeri. De asemenea, orice pointer poate fi testat cu `NULL`. Dacă un pointer este $== \text{NULL}$ înseamnă că adresa de memorie spre care pointează este adresa 0, ceea ce e un non-sens.

Instructiunea

$p + n$

deseamneaza al n -lea obiect din memorie după cel pointat curent de p . Acest lucru este adevarat indiferent de tipul obiectelor pe care p a fost declarat ca pointer. Compilatorul atunci cind il intilneste pe n , il decaleaza în funcție de lungimea obiectelor pe care pointeaza p , lungime determinata prin declaratia lui p (`sizeof(*p)`).

Este validă și scăderea pointerilor: dacă p și q pointează pe elementele aceluiași tablou, $p-q$ este numărul de elemente dintre p și q . Acest fapt poate fi utilizat pentru a scrie o nouă versiune a lui `strlen`.

```
int strlen(char *s)//returnează lungimea sirului
{
    char *p = s;
    while (*p != '\0')
        p++;
    return(p-s);
```

}

Prin declarare, p este initializat pe s, adică să pointeze pe primul caracter din s. În cadrul buclei while este examinat fiecare caracter până se întâlnește \0 care semnifică sfîrșitul sirului de caractere iar apoi se scad cele 2 adrese.

Este posibila omiterea testului explicit iar astfel de bucle sunt scrise adesea

```
while (*p)
    p++;
```

Deoarece p pointeaza pe caractere, p++ face ca p sa avanseze de fiecare data pe caracterul urmator, iar p-v da numarul de caractere parcuse, adica lungimea sirului. Aritmetica pointerilor este consistenta: daca am fi lucrat cu float care ocupa mai multa memorie decit char, și daca p ar fi un pointer pe float, p++ ar avansa pe urmatorul float.

Toate manipularile de pointeri iau automat în considerare lungimea obiectului pointat în asa fel încât trebuie să nu fie alterat.

Operații permise : adunarea sau scaderea unui pointer cu un intreg, scaderea sau compararea a doi pointeri.

Nu este permisa adunarea, impartirea, deplasarea logica, sau adunarea unui float sau double la pointer.

2.3.3.8. Transmiterea argumentelor la apelurile de funcții

La apelul unei funcții argumentele se transmit prin stivă, în ordinea în care apar în lista de argumente a funcțiilor. Aceasta înseamnă că pentru fiecare argument al funcției, pe stivă se crează o variabilă locală (funcției) cu numele argumentului, iar valoarea transmisă este utilizată la inițializarea acestei variabile locale. Evident, la sfârșitul execuției funcției, aceste variabile locale fiind salvate pe stivă se pierd, și valorile din ele nu mai sunt disponibile după terminarea apelului funcției. Spunem că parametrii s-au transmis **funcției prin valoare**.

Practic, la transmiterea prin valoare, se creează copii temporare ale parametrilor care se transmit. Funcția apelată lucrează cu aceste copii iar la revenire valorile variabilelor parametru vor fi nemodificate. Deci este imposibil ca funcția apelată să modifice unul din argumentele reale din apelant.

Exemplu:



```
#include<stdio.h>
void schimbare(int x, int y) //se vor crea copii ale variabilelor x și y
```

```

{ int tmp;
tmp=x;
x=y;
y=tmp;           //în cadrul copiilor variabilelor se face inversarea
}               //la revenire copiile variabilelor x și y se distrug

int main()
{ int x=5, y=7;
schimbare(x,y);      //transmitere prin valoare
printf("%d %d\n",x,y); //valorile rămân nemodificate adică se va afișa 5 7
}

```

In acest fel, se transmit doar argumentele de tipul input pentru funcții.

Dacă dorim ca funcția să modifice valorile variabilelor primite ca și argument iar valorile să se regăsească modificate în funcția apelantă după finalizarea apelului funcției apelate, trebuie să facem disponibile funcției apelate adresele variabilelor transmise ca și argumente. În acest caz, spunem că facem transmitere prin **adresă sau referință**.

La transmiterea prin adresă, se transmite adresa variabilelor parametru. Toate operațiile în cadrul funcției apelate se fac asupra zonei originale. Dacă se dorește alterarea efectivă a unui argument al funcției apelante, trebuie furnizată adresa variabilei ce se dorește a se modifica (un pointer la această variabilă). Funcția apelată trebuie să declare argumentul corespunzător ca fiind un pointer.



Exemplu:

```

#include<stdio.h>
void schimbare(int *x, int *y)
{
    int tmp;
    tmp=*x;
    *x=*y;
    *y=tmp;           //se face inversarea asupra zonei originale
}

int main()
{ int x=5, y=7;
schimbare(&x,&y);      //transmitere prin adresă
printf("%d %d\n",x,y); //valorile sunt inversate adică se va afișa 7 5
}

```

Pentru o funcție, argumentele care sunt de tipul input-output sau doar output trebuie transmise (obligatoriu) prin adresă.

În cazul în care apar masive, având în vedere că numele tabloului este un pointer constant spre primul element al lui, se folosește numele masivului ca adresă de început. Elementele masivului nu vor fi copiate iar operațiile se vor face astfel asupra zonei originale.

O utilizare comună a argumentelor de tip pointer (transmitere prin adresă) se întâlnește în cadrul funcțiilor care trebuie să returneze mai mult decât o singură valoare. De exemplu, constatăm faptul că funcția scanf utilizează transmiterea prin adresă, datorită faptului că, în argumentele sale, trebuie să regăsim valorile care se citesc.

2.3.3.9. Pointeri și tablouri

In C, există o relație strânsă între pointeri și tablouri, încât pointerii și tablourile pot fi tratate simultan. Orice operație care poate fi rezolvată prin indici în tablouri, poate fi rezolvată și cu ajutorul pointerilor. Versiunea cu pointeri va fi în general, mai rapidă.

Declarația

```
int a[10]
```

definește un tablou de dimensiunea 10, care este un bloc de 10 variabile de tip int salvate la adrese consecutive numite a[0], a[1], ..., a[9] notația a[i] desemnează elementul deci pozițiile în tablou, numărate de la începutul acestuia.

Dacă pa este un pointer pe un întreg, declarat ca

```
int *pa
```

atunci asignarea

```
pa = &a[0]
```

face ca pa să pointeze pe al "zero-ulea" (primul) element al tabloului a; aceasta înseamnă ca pa conține adresa lui a[0].

Așadar asignarea $x = *pa$ copia conținutul lui a[0] în x.

Dacă pa pointează pe un element oarecare al lui a atunci prin definiție $pa + 1$ pointează pe elementul urmator și în general $pa + i$ pointează cu i elemente înaintea elementului pointat de pa iar $pa + i + 1$ pointează cu i elemente după elementul pointat de pa.

Astfel, dacă pa pointează pe a[0]

$*(pa + 1)$

referă conținutul lui $a[1]$, $pa + i$ este adresa lui $a[i]$ și $*(pa+i)$ este conținutul lui $a[i]$.

Acstea observații sunt adevărate indiferent de tipul variabilelor din tabloul a . Definiția "adunării unității la un pointer" și prin extensie, toată aritmetica pointerelor este de fapt calcularea prin lungimea în memorie a variabilei pointat. Astfel, în $pa+i$, i este înmulțit cu lungimea variabilei pe care pointează pa ($\text{sizeof}(*pa)$) înainte de a fi adunate la pa .

Corespondența între indexare și aritmetica pointerelor este evident foarte strânsă. De fapt, numele unui tablou este convertit de către compilator într-un pointer constant pe începutul zonei de memorie unde se află tabloul.
Efectul este că numele unui tablou este o expresie pointer.

Aceasta are câteva implicații utile. Din moment ce numele unui tablou este sinonim cu locația elementului său zero, asignarea

$pa = \&a[0]$

poate fi scrisă și $pa = a$

O referință la $a[i]$ poate fi scrisă și ca $*(a+i)$. Evaluând pe $a[i]$, C îl convertește în $*(a+i)$; cele două forme sunt echivalente. Aplicând operatorul & ambilor termeni ai acestei echivalențe, rezultă că $\&a[i]$ este identic cu $a+i$: $a+i$ adresa elementului al i -lea în tabloul a .

Reciproc, dacă pa este un pointer el poate fi utilizat în expresii cu un indice; $pa[i]$ este identic cu $*(pa+i)$.

Pe scurt orice tablou și exprimare de indice pot fi scrise ca un pointer și deplasament și orice adresă chiar în aceeași instrucțiune. Trebuie ținut seama de o diferență ce există între numele tablou și un pointer. Un pointer este o variabilă, astfel ca $pa=a$ și $pa++$ sunt operații.

Dar, un nume de tablou este o constantă, de aceea construcții ca $a=pa$ sau $a++$ sunt interzise.

Atunci când se transmite un nume de tablou unei funcții, ceea ce se transmite este locația de început a tabloului. În cadrul funcției apelate acest fapt argument este o variabilă ca oricare alta astfel încât un argument nume de tablou este un veritabil pointer, adică o variabilă continind o adresa.

Ne vom putea folosi de aceasta pentru a scrie o nouă versiune a lui `strlen`, care calculează lungimea unui sir.

```
int strlen(char *s) // returnează lungimea sirului s
{
    int n;
```

```

char *ps;
for (n = 0, ps = s; *ps != '0'; ps++)
    n++;
return n;
}

```

Ca parametri formali în definirea unei functii

char s[]

și

char *s;

sunt echivalenți; alegerea celui care trebuie scris este determinată în mare parte de expresiile ce vor fi scrise în cadrul funcției. Atunci cind un nume de tablou este transmis unei funcții, aceasta poate, după necesități să-o interpreteze ca tablou sau ca pointer și să-l manipuleze în consecință. Funcția poate efectua chiar ambele tipuri de operații dacă î se pare potrivit și corect.

Este posibilă și transmiterea către o funcție doar a unei părți dintr-un tablou prin transmiterea unui pointer pe începutul subtabloului. De exemplu, dacă a este un tablou;

f(&a[2])

și

f(a + 2)

ambele transmit funcției f adresa elementului a[2] deoarece &a[2] și a+2 sunt expresii pointer care referă la treilea element al lui a. În cadrul lui f, declarea argumentului poate fi

```

f(int arr[])
{
    ...
}
sau
f(int *arr)
{
    ...
}

```

Așa cum a fost concepută funcția f, apătul că argumentul referă de fapt o parte a unui tablou mai mare, nu are consecințe.

2.3.3.10. Pointeri pe caractere și funcții

Un sir constant scris astfel " ... " este un tablou de caractere (de tipul char[]). În reprezentare internă, compilatorul termină un tablou cu caracterul \0 în aşa fel încât programele să poată detecta sfârșitul.

Lungimea în memorie pentru un sir de caractere ("...") este astfel mai mare cu 1 decât numărul de caractere cuprinse între ghilimele.

```
char message[40] = "now is the time";
```

defineste un sir de 40 de caractere și îl asignează cu sirul de caractere "now is the time".

Exemplu :strcpy(char s[], char t[]) copiază sirul t în sirul s, versiunea cu tablouri :

```
int mystrcpy(char s[], char t[]) //copiază t în s
{
    int i;
    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
    return i;
}
```

Versiunea lui strcpy cu pointeri

```
strcpy(char *s, char *t) //copiază t în s, versiunea pointeri
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Deoarece argumentele sunt transmise prin valoare (atentie: avem pointeri transmisi prin valoare), mystrcpy poate utiliza s și t în orice fel se dorește. Aici ei sunt convențional utilizati ca pointeri, care parcurg tablourile pînă în momentul în care s-a copiat \0 sfîrșitul lui t, în s.

In practică, mystrcpy nu va fi scris asa cum s-a aratat mai sus. O a doua posibilitate ar fi

```
int mystrcpy(char s[], char t[]) //copiază t în s
{
    int i = 0;
    while ((*s++ = *t++) != '\0') i++;
    return i;
}
```

In aceasta ultimă versiune se imită incrementarea lui s și t în partea de test. Valoarea lui *t++ este caracterul pe care a pointat înainte ca t să fi fost incrementat; prefixul ++ nu-l schimba pe t înainte ca acest caracter să fi fost adus. În același fel, caracterul este stocat în vede a pozitie s înainte ca s să fie incrementat. Acest caracter este

deasemenea valoarea care se grupeaza cu \0 pentru simbolul buclei. Efectul net este ca, caracterele sunt copiate din t în s, inclusiv sfîrșitul lui \0.

Ca o ultima abreviere vom observa că și gruparea cu \0 este redundantă, astfel că bucla while poate fi scrisă:

```
while (*s++ = *t++)
```

Desi aceasta versiune poate părea complicată la prima vedere, aranjamentul ca notatie este considerat suveran daca nu există alte rațiuni de a schimba.

Rutina este mystrcmp(s, t), compara sirurile de caractere s și t și returneaza negativ, zero sau pozitiv în functie de relația dintre s și t; care poate fi: s<t, s=t sau s>t.
Valoarea returnată este obținută prin scăderea caracterului de pe prima poziție unde s diferă de t.

```
int mystrcmp(char s[], char t[]) // returneaza <0 daca s<t, 0 daca s==t, >0 daca s>t
{
    int i;
    i = 0;
    while (s[i] == t[i])
        if (s[i++] == '\0')
            return(0);
    return(s[i] - t[i]);
}
```

Versiunea cu pointeri a lui strcmp:

```
int mystrcmp(char s[], char t[]) // returneaza <0 daca s<t, 0 daca s==t, >0 daca s>t
{
    for ( ; *s == *t; s++, t++)
        if (!*s)
            return(0);
    return(*s - *t);
}
```

Dacă ++ și -- sunt folosiți altfel decât operatori prefix sau postfix pot apărea alte combinații de * și ++ și --, deși mai puțin frecvente.

Exemplu: *++p incrementează pe p înainte de a aduce caracterul pe care pointează p.
*--p decrementează pe p în același condiții.

La lucrul cu siruri de caractere, se preferă iterarea folosind pointeri de tipul char*, și nu indecsă pe sir.

Se solicită atenție sporită la initializarea sirurilor de caractere.

Astfel, dacă în program scriem un sir de caractere utilizând semnul “”, programul va aloca sirul ca și o constantă și il va păstra la o zonă de memorie care este dincolo de controlul nostru în program.

Astfel, sirul “how are you?” este tratat de către program ca și o constantă de tipul char[], iar pentru această constantă se alocă 13 octeti (12 octeți pentru caracterele sirului și un octet pentru \0).

Următoarea initializare este validă și corectă:

```
char sir[30] = "how are you?";
```

astfel, compilatorul alocă o zonă de memorie pentru variabila sir , de mărime 30 octeți, iar pe această zonă copiază sirul “how are you?”.

Dacă scriem:

```
char *psir = "how are you?";
```

atunci compilatorul alocă o constantă de tipul char* cu valoarea “how are you?” și face ca pointerul psir să arate către zona de memorie unde este alocată constanta de tipul char*. Aceasta înseamnă că dacă vom dori să modificăm continutul zonei de memorie pointate de psir (care este constant) va genera o eroare de compilare.

Concluzia este că dacă dorim să utilizăm variabile (neconstante) de tip sir de caractere în program, avem 2 alternative:

1. Definim variabila ca și sir de caractere folosind sintaxa de sir : char* sir[dimensiune]. În acest caz, se alocă (în zona globală sau zona de stivă) variabila sir de dimensiunea specificată, iar mai apoi vom putea manipula această zonă de memorie după cum dorim
2. Definim variabila ca și pointer de tipul char*. : char *psir;. Apoi va trebui să alocăm dinamic acest sir folosind funcția de alocare malloc. După aceea vom putea manipula continutul acestei zone de memorie. La finalul utilizării, programatorul trebuie să să dezaloce zona de memorie folosind funcția free. Alocarea de memorie este realizată dinamic, în zona de heap.

În ambele cazuri, programatorul va manipula conținutul zonei de memorie folosind funcțiile de lucru pe șiruri de caractere din string.h: strcpy, strcmp, strcat etc.

În cazul 2, programatorul trebuie să aloce o zonă de memorie cel puțin egală cu lungimea sirului care se dorește a fi salvat + 1, și anume strlen(sir) + 1

Pe parcursul programelor pe care le scrie, vom evita să realizăm atribuirile de tipul:

```
pchar = "text";
```

pentru ca acestea nu au sens: ele fac variabila pointer de tipul char* să pointeze către o zonă de memorie dincolo de controlul programului nostru.

2.3.3.11. Pointeri pe funcții

Deși o funcție nu este o variabilă, ea are o adresă în memorie care poate fi atribuită unui pointer. *Adresa unei funcții* este punctul de intrare în funcție. De aceea, un pointer către funcție poate fi utilizat pentru a apela o funcție. În momentul compilării, fiecarei funcții i se stabilește un punct de intrare care se găsește la o adresă de memorie. Un pointer care conține adresa punctului de intrare, poate fi folosit pentru a apela acea funcție.



Adresa unei funcții se obține utilizând numele funcției fără paranteze sau argumente (există o similaritate cu modul de a obține adresa unui tablou, ca fiind numele tabloului). Următorul exemplu de program sperăm că este edificator.



```
#include<stdio.h>
int suma(int a, int b)
{ return(a+b);
}
int produs(int a, int b)
{ return(a*b);
}
void main(void)
{ int a,b;
  int (*p)(int,int);           //p este pointer spre o funcție cu 2 parametri
  intregi
  scanf("%d %d",&a,&b);      //citim valorile a 2 întregi
  p=suma;                      //pointerul p primește adresa funcției suma
  printf("suma este %d\n",p(a,b)); //folosind un pointer la funcții vom obține
                                   //suma celor 2 numere citite
  p=produs;                    // pointerul p primește adresa funcției produs
  printf("produsul este %d\n",p(a,b)); //folosind un pointer la funcții vom
  obține                           //produsul celor 2 numere citite
}
```

Pointerii la funcții pot fi folosiți și ca parametri ai unei funcții. Datorită dificultăților de a înțelege un program ce utilizează pointeri la funcții, în general aceștia se evită. Totuși, în anumite cazuri ale unor probleme complexe, se impune folosirea pointerilor la funcții.

Constatăm faptul că se poate declara un pointer la o funcție în felul următor :

*tipreturn (*pfunctie)(lista argumente)*

Astfel se declară pfunctie ca și un pointer către o funcție care returnează o valoare de tipreturn și primește la intrare o lista de argumente precum cea specificată.

Datorită faptului că numele unei funcții din program reprezintă un pointer către zona de cod care conține instrucțiunile functiei respective în program, se va putea face atribuirea :

pfunctie = numefunctie;

astfel, apelul funcției numefunctie cu o lista de parametrii de apel se poate face fie numefunctie(listaparamdeapel)

sau prin intermediul pointerului la funcție

*(*pfunctie)(listaparamdeapel)*

In acest sens, prin intermediul unor pointeri la funcție se poate realiza o funcționalitate dinamică a unui program.

Se poate defini un array de pointeri la funcție:

*tipreturn (*sir_pointeri_functie[DIM])(lista_argumente)*

Astfel, sir_pointeri_functie este un sir de DIM pointeri la functii. De exemplu, prin utilizarea sirurilor de pointeri la funcții se pot implementa funcționalități similare meniurilor fără ca să fie nevoie utilizarea construcțiilor gen switch.



Urmăriți fișierul video de la următoarea adresă:

<https://drive.google.com/open?id=0B0E2G7UxqRojYldpc2I3ZDFmWnM>

Acesta prezintă aspectele teoretice și practice prezentate până la acest moment.

2.3.3.12. Argumentele functiei main

Sintaxa functiei main este

`int main(int argc, char *argv[])`

cu urmatoarea semnificatie:

- Argc reprezintă numărul argumentelor transmise în cadrul liniei de comandă. Dacă linia de comandă conține doar numele executabilului, atunci argc este 1 iar argv va fi un sir de lungime 1 de siruri de caractere
- Argv reprezintă un sir de caractere care va conține argumentele transmise în linia de comandă executabilului. argv[0] reprezintă întotdeauna un sir care conține numele executabilului care se execută.

Sirul argv va avea exact atatea elemente câte sunt indicate în variabila argc.

Astfel, prin argumentele liniei de comanda (sau argumentele functiei main) programul poate citi argumente care sunt transmise din sistemul de operare către program.

2.3.3.13.Operații cu fișiere

Un fișier este o colecție de date de același tip, numite înregistrări, memorate pe suport extern (hard – disc,CD,...). Avantajele utilizării fișierelor sunt o consecință a proprietăților memoriilor externe. Fișierele permit memorarea unui volum mare de date persistente. Un fișier are o înregistrare care marchează sfârșitul de fișier. În cazul fișierelor de intrare de la tastatură sfârșitul de fișier se generează prin CTRL+Z. El poate fi pus în evidență folosind constanta simbolică EOF definită în stdio.h.

Prelucrarea fișierelor implică un număr de operații specifice acestora. Orice fișier, înainte de a fi prelucrat, trebuie să fie deschis. De asemenea la terminarea prelucrării unui fișier acesta trebuie închis.

Operațiile de deschidere – închidere se pot realiza prin intermediul unor funcții speciale de bibliotecă.

Alte operații frecvente:

- *Crearea unui fișier;*
- *Actualizarea unui fișier;*
- *Pozitionarea într-un fișier;*
- *Consultarea unui fișier;*
- *Adăugarea de înregistrări într-un fișier;*
- *Stergerea unui fișier.*

Sistemul de I/O din C separă printr-o anumită abstractizare programatorul de echipament. Forma abstractă se numește stream iar instrumentul efectiv care se poate utiliza este *fișierul*.

In C, conform standardului, se poate lucra cu fișiere fie la nivel binar, fie la nivel de caracter. Diferența constă în modul în care sunt tratate înregistrările din fișier. Astfel, dacă acestea sunt tratate la nivel de caracter, atunci operațiile de IO se vor realiza prin utilizarea unor structuri de tipul sir de caractere, referite fie sub formă de variabile de tip sir de caractere (char[]) fie pointeri spre caractere (char*).

Dacă accesul la fișier se face în mod binar, atunci datele se vor referi prin pointeri de tipul `void*`. Prin intermediul acestora, practic se poate face referire la orice tip de date din program. Astfel, fișierele tratate la nivel binar pot salva date de orice tip. Evident, în comparație cu tratarea fișierelor la nivel sir de caractere, nu avem un delimitator de înregistrare (precum caracterul `\0`), astfel încât, la fiecare operație de scriere / citire trebuie să se specifice lungimea înregistrării, adică numărul de octeți scriși / citiți.

Trebuie să remarcăm faptul că limbajul de programare C unifică modul de lucru cu fișierele. Astfel, intrările / ieșirile sunt tratate unitar, fie că vorbim de fișiere de pe disc fie că vorbim despre dispozitivele standard de intrare / ieșire precum `stdin`, `stdout` sau `stderr`. Astfel, funcțiile de citire / scriere din dispozitivele standard, din fișiere sau siruri de caractere precum `scanf` / `fscanf` / `sscanf` respectiv `printf` / `fprintf` / `sprintf` sunt realizate în mod unitar și au utilizare similară.

2.3.3.14. Lucrul cu fișiere la nivel de siruri de caractere

Deschiderea unui fișier

La acest nivel se utilizează funcția `fopen` pentru deschiderea unui fișier. Ea returnează un pointer spre tipul `FILE` (tipul fișier), tip definit în fișierul `stdio.h`. În caz de eroare, funcția `fopen` returnează pointerul `NULL`. Funcția `fopen` realizează o alocare de memorie și returnează zona de memorie alocată în cazul în care sistemul de operare reușește să deschidă fișierul specificat în modul de deschidere solicitat de funcție.

Prototipul funcției `fopen` este:

`FILE *fopen(const char *cale, const char *mod);`

unde:

`cale` reprezintă calea pe disc a fișierului care se deschide.

`mod` este un pointer spre un sir de caractere care definește modul de prelucrare al fișierului după deschidere. Acest sir se definește în felul următor:

”r” deschidere în citire (read); fișierul trebuie să existe.

”w” deschidere în scriere (write); se deschide un fișier gol. Dacă fișierul deja există pe disc, conținutul acestuia este șters și se crează un fișier nou

”a” deschidere pentru adăugare (append); datele vor fi scrise la finalul fișierului. Dacă fișierul nu există, atunci acesta este creat.

”r+” deschidere pentru modificare (citire sau scriere); fișierul trebuie să existe

”w+” crează un fișier gol pentru scriere / citire;

”a+” deschide un fișier pentru citire și adăugare.

Dacă se deschide un fișier inexistent cu modul ”w” sau ”a”, atunci el este deschis în creare. Dacă se deschide un fișier existent cu modul ”w”, atunci conținutul vechi al fișierului se pierde și se va crea unul nou cu același nume.

Menționăm că `stdin`, `stderr`, `stdaux` și `stdprn` sunt pointeri spre tipul `FILE` și permit ca funcțiile de nivel superior de prelucrare a fișierelor să poată trata intrarea standard și ieșirile standard pe terminal și imprimantă la fel ca și fișierele pe celelelate suporturi. Singura deosebire constă că aceste fișiere nu se deschid și nici se închid de către

programator. Ele sunt deschise automat la lansarea în execuție a programului și se închid la apelul funcției *exit*. Variabilele *stdin*, *stderr*, *stdaux* și *stderr* există în program ca și variabile globale, iar declarațiile lor sunt introduse prin *stdio.h*.

Citire / scriere de caractere

Fișierele pot fi scrise și citite caracter cu caracter, folosind două funcții simple și anume *putc* pentru scriere și *getc* pentru citire. Funcția *putc* are prototipul:

`int putc(int c, FILE *pf);`

unde:

- *c* este codul ASCII al caracterului care se scrie în fișier;
- *pf* este pointerul spre tipul FILE a cărui valoare a fost returnată de funcția *fopen* la deschiderea fișierului în care se scrie. În particular, *pf* poate fi unul din pointerii:
stdout (ieșire standard);
stderr (ieșire pe terminal în caz de eroare);
stdaux (comunicație serială);
stdprn (ieșire paralelă la imprimantă).

Funcția *putc* returnează valoarea lui *c* respectiv -1 în caz de eroare.

Funcția *getc* are prototipul

`int getc(FILE *pf);`

unde *pf* este pointerul spre tipul FILE a cărui valoare a fost returnată de funcția *fopen* la deschiderea fișierului. Ea returnează codul ASCII al caracterului citit sau EOF la sfârșit de fișier sau eroare. În particular, *pf* poate fi pointerul *stdin* (intrare de la tastatură).

Similar cu acestea, sunt funcțiile *fputc* și *fgetc*.

Pentru lucrul cu intrarea / ieșirea standard, se pot folosi funcțiile *getchar* sau *putchar*, cu sintaxă similară cu *getc* și *putc*, fără să fie necesară transmiterea pointerului *stdin* sau *stdout*.

Inchiderea unui fișier

După terminarea prelucrării unui fișier, acesta urmează să fie închis. În acest caz se utilizează funcția *fclose*. Ea are prototipul

`int fclose(FILE *pf);`

unde *pf* este pointerul spre tipul FILE a cărui valoare a fost definită la deschiderea fișierului prin intermediul funcției *fopen*. Funcția *fclose* returnează:

- 0 la închiderea normală a fișierului
- EOF: în caz de eroare.

Pentru toate fișierele deschise prin *fopen* trebuie să existe un apel *fclose*. *fclose* are 2 roluri:

- eliberează resursele alocate la nivelul sistemului de operare
- eliberează memoria alocată pentru pointerul de tip FILE* prin intermediul funcției *fopen*.



Exemplu. Programul copiază intrarea standard la ieșirea standard stdout, folosind funcțiile *getc* și *putc*.

```
#include <stdio.h>
int main() /* copiază intrarea stdin la iesirea stdout */
{ int c;
  while((c = getc(stdin)) != EOF) putc(c,stdout);
}
```



Exemplu. Programul scrie la ieșirea stdout caracterele unui fișier a cărui cale este argumentul din linia de comandă (pentru detalii în legătură cu parametrii transmiși prin linia de comandă). Dacă nu există un argument în linia de comandă, atunci se citește de la intrarea standard.

```
#include <stdio.h>

int main(int argc,char *argv[]) /* listeaza la ieșirea stdout un fisier a carui cale este argumentul din linia de comanda */
{
    FILE *pf;
    int c;
    if(argc == 1) /* nu exista argument in linia de comanda */
        pf = stdin;
    else // se deschide fisierul a carui cale se afla in argv[ 1 ] //
        if((pf = fopen( argv[1], "r"))==NULL) {
            printf("nu se poate deschide fisierul %s/n",*argv);
            return -1;
        }
    while((c = getc(pf)) != EOF )
        putchar(c);
    fclose(pf);
    return 0;
}
```

Operațiile de intrare/ieșire cu format

Biblioteca standard a limbajului C conține funcții care permit realizarea operațiilor de intrare/ieșire cu format. Se pot utiliza funcțiile *fscanf* și *sprintf*, prima pentru citire cu format dintr-un fișier, iar a doua pentru scriere cu format într-un fișier.

Funcția *fscanf* este asemănătoare cu funcția *scanf*. Ea are un parametru în plus față de *scanf*, un pointer spre tipul FILE care definește fișierul din care se face citirea. Acest pointer este primul parametru al funcției *fscanf*. Ceilalți parametri au aceeași semnificație ca și în cazul funcției *scanf*.

Rezultă că funcția `fscanf` poate fi apelată printr-o expresie de atribuire de forma
`nr=fscanf(pf,control,par1, par2...parn);`

unde:

- `pf` este un pointer spre tipul `FILE` și valoarea lui a fost definită prin apelul funcției `fopen` (aceasta definește fișierul din care se face citirea) iar ceilalți parametri sunt identici cu cei utilizati la apelul funcției `scanf`.

Funcția `fscanf`, ca și `scanf`, returnează numărul câmpurilor citite din fișier.

La întâlnirea sfârșitului de fișier se returnează valoarea EOF definită în fișierul `stdio.h`. Pentru `ps=stdin`, funcția `fscanf` este identică cu `scanf`.

Funcția `fprintf` este analoagă cu funcția `printf`. Primul ei parametru este un pointer spre tipul `FILE` și el definește fișierul în care se scriu datele. Ceilalți parametri sunt identici cu cei ai funcției `printf`.

Funcția `fprintf`, ca și funcția `printf`, returnează numărul caracterelor scrise în fișier sau -1 în caz de eroare. Pentru `sf=stdout`, funcția `fprintf` este identică cu `printf`. De asemenea, se pot utiliza pointerii standard obișnuiți:

- | | |
|---------------------|--|
| <code>stderr</code> | - afișarea mesajelor de eroare pe terminal |
| <code>stdaux</code> | - comunicație serială |
| <code>stdprn</code> | - ieșirea paralelă la imprimantă. |

Menționăm că la comunicația serială se poate conecta o imprimantă serială.

Intrări/ieșiri de siruri de caractere

Biblioteca standard a limbajului C conține funcțiile `fgets` și `fputs` care permit citirea respectiv scrierea într-un fișier ale cărui înregistrări sunt siruri de caractere.

Funcția `fgets` are prototipul

`char *fgets(char *s, int n, FILE *pf);`

unde:

- `s` este pointerul spre zona în care se face citirea caracterelor
- `n-1` este numărul maxim de caractere care se citesc iar `pf` este pointerul spre tipul `FILE` care definește fișierul din care se face citirea.

Citirea se încheie fie dacă se citesc `n-1` caractere, fie dacă se întâlnește delimitatorul de sfârșit de linie.

De obicei s este numele unui tablou de tip `char` de dimensiune cel puțin `n`. S trebuie să fie alocat înainte de începerea execuției funcției `fgets` și trebuie să aibă capacitatea de minim `n` octeți.

Sirul se termină cu '\0' (caracterul NULL) care este inserat de funcția `fgets` la capătul sirului de caractere citit. La întâlnirea caracterului '\n', citirea se oprește. În acest caz, în zona receptoare se transferă caracterul '\n' și apoi caracterul NULL. În mod normal, funcția returnează valoarea pointerului `s`. La întâlnirea sfârșitului de fișier se returnează valoarea NULL.

Funcția `fputs` scrie într-un sir fișier un sir de caractere care se termină prin '\n'. Ea are prototipul

`int fputs(const char *s,FILE *pf);` unde:

- s este pointerul spre zona care conține sirul de caractere care se scrie;
- pf este pointerul spre tipul FILE care definește fișierul în care se scrie.
Funcția *fput* returnează codul ASCII al ultimului caracter scris sau -1 în caz de eroare.

Aceste funcții sunt realizate folosind funcția *getc* pentru *fgets* și *putc* pentru *fputs*.

2.3.3.15. Prelucrarea fișierelor binare

Fișierele organizate ca date binare (octetii nu sunt considerați ca fiind coduri de caractere) pot fi prelucrate la acest nivel folosind funcțiile *fread* și *fwrite*.

In acest caz, se consideră că înregistrarea este o colecție de date structurate numite articole. La o citire, se transferă într-o zonă specială, numită zonă tampon, un număr de articole care se presupune că au o lungime fixă. In mod analog, la scriere se transferă din zona tampon un număr de articole de lungime fixă. Cele două funcții au prototipurile de mai jos:

```
size_t fread(void *ptr, size_t dim, size_t nrart, FILE *pf);
size_t fwrite(const void *ptr, size_t dim, size_t nrart, FILE *pf);
```

unde:

- ptr este pointerul spre zona tampon ce conține articolele citite / scrise (înregistrarea citită / scrisă);
- dim este un întreg ce reprezintă lungimea unui articol;
- nrart este un întreg ce reprezintă numărul articolelor care se transferă;
- pf este un pointer spre tipul FILE care definește fișierul din care se face citirea.

De obicei, dim este dimensiunea (sizeof) a tipului către care pointează pointerul ptr.

Ambele funcții returnează numărul articolelor transferate sau -1 în caz de eroare.



Exemplu. Programul citește de la intrarea *stdin* datele ale căror formate sunt definite mai jos și le scrie în fișierul *misc.dat* din directorul curent.

Formatul datelor de intrare este următorul:

Tip	denumire	val	um	cod	pret	cantitate
I	REZISTENTA	010	KG	12345678	1.5	10000.0

```
#include <stdio.h>
#define MAX 50
```

```
typedef struct {
    char tip[2];
    char den[MAX + 1];
```

```

int val;
char unit[3];
long cod;
float pret;
float cant;
} TIP_ARTICOL;

int cit(TIP_ARTICOL *str) //citeste datele de la intrarea standard si le scrie in
structura spre care pointeaza str
{
    int c,nr;
    float x,y;

    while((nr = scanf("%1s %50s %3d %2s %1d", str->tip, str->den, &str->val,
str->unit, &str->cod))!= 5 ||
        scanf("%f %f", &x, &y) != 2)
    {
        if(nr == EOF ) return EOF;
        printf("rand eronat; se reia \n");
        // avans pana la newline sau EOF
        while((c = getchar()) != '\n' && c != EOF);
        if(c == EOF) return EOF;
    } //sfarsit while
    str->pret = x; str->cant = y;
    return nr;
} // sfarsit cit

int main() // creeaza fisierul misc.dat cu datele citite de la stdin
{
    FILE *pf;
    TIP_ARTICOL a[6];
    int i, n;

    if(pf = fopen("misc.dat","w+")) == NULL)
    {
        printf(" nu se poate deschide in creare fisierul misc.dat\n");
        return -1;
    }
    for(i = 0; i < 6; i++) {
        n = cit(&a[i]);
    }
    if( 6 != fwrite( &a, sizeof(TIP_ARTICOL), 6, pf) ) {
        // eroare la scrierea in fisier
        printf("eroare la scrierea in fisier\n");
        return -1;
    }
    fclose(pf);
}

```

```

// sfarsit main
    return 0;
}

```

Exemplu. Programul listează articolele fișierului misc.dat, creat prin programul anterior, pentru care tip=I.

```

#include <stdio.h>
#define MAX 50

typedef struct {
    char tip[2];
    char den[MAX + 1];
    int val;
    char unit[3];
    long cod;
    float pret;
    float cant;
} TIP_ARTICOL;

main() /* citeste fisierul misc.dat */
{
    FILE *pf;
    int i,n;
    TIP_ARTICOL a[6];

    if((pf = fopen("misc.dat","r")) == NULL) {
        printf("nu se poate deschide fisierul misc.dat in citire\n");
        return -1;
    }
    printf("%60s\n\n\n", "INTRARI\n\n");
    // se citeste o inregistrare
    while((n = fread(a, sizeof(TIP_ARTICOL), 6, pf)) > 0)
        // se listeaza articolele de tip I dintr-o inregistrare
        for (i = 0; i < n; i++) {
            if ( a[i].tip[0] == 'I')
                printf("%s %3d %s %d %.2f %.2f\n", a[i].den,
a[i].val, a[i].unit, a[i].cod, a[i].pret, a[i].cant);
        }
    fclose(pf);
    return 0;
}

```

Pozitionarea într-un fișier

Biblioteca standard a limbajului C conține funcția fseek, cu ajutorul căreia se poate deplasa capul de citire/scriere al discului în vederea prelucrării înregistrărilor

fișierului într-o ordine oarecare, diferită de cea secvențială (aceeași aleator). Ea are prototipul

*int fseek(FILE *pf, long deplasament, int origine);*

unde pf este pointerul spre tipul care definește fișierul în care se face poziționarea capului de citire/scriere iar deplasament și origine au aceeași semnificație ca și în cazul funcției lseek.

Funcția *fseek* returnează valoarea zero la poziționare corectă și o valoare diferită de zero în caz de eroare.

O altă funcție utilă în cazul accesului aleator este funcția *ftell*, care indică poziția capului de citire în fișier. Ea are prototipul

*long ftell(FILE *pf);* unde:

- pf este pointerul spre tipul FILE care definește fișierul în cauză.

Funcția returnează o valoare de tip *long* care definește poziția curentă a capului de citire/scriere, și anume reprezintă deplasamentul în octeți a poziției capului față de începutul fișierului.

Ștergerea unui fișier

Un fișier poate fi șters apelând funcția *unlink*. Aceasta are prototipul

*int unlink(const char *cale);*

unde cale este un pointer spre un sir de caractere identic cu cel utilizat la crearea fișierului în funcția de *creat* sau *fopen*.

Funcția *unlink* returnează valoarea zero la o ștergere reușită, respectiv -1 în caz de eroare.

Urmăriți fișierul video de la următoarea adresă:



<https://drive.google.com/open?id=0B0E2G7UxqRojTXNpRXhxbFI2eEE>

Acesta prezintă aspectele teoretice și practice prezentate până la acest moment.

2.3.4. Sumar

Modulul a oferit studenților noțiunile de C necesare pentru elaborarea unor programe de dificultate medie în limbajul C.

2.3.5. Întrebări recapitulative

Întrebări pentru testul scris:

1. Ce este o structură?
2. Ce este o uniune?
3. Ce este un câmp de biți?
4. Ce este o enumerare?
5. Ce este un pointer?

6. Care sunt operațiile care se pot face cu pointerii?
7. Care sunt argumentele funcției main?



Intrebări grilă:

- 1) Una din următoarele propoziții nu este adevărată:
 a Funcția fopen() returnează un pointer de fișier
 b O enumerare e un set de constante care specifică toate valorile permise pe care le poate avea o variabilă de acest tip
 c Caracterul \n este terminator la un sir de caractere
 d În cazul transmiterii prin adresă, operațiile asupra variabilelor se fac în zona original
 e se pot scădea doi pointeri de același tip

- 2) Una din următoarele propoziții este adevărată:
 a Un identificator nu poate să înceapă cu caracterul de subliniere
 b Constanta caracter se delimitizează folosind ghilimele
 c Orice valoare diferită de zero reprezintă fals
 d Operatorul de atribuire are prioritatea cea mai mare
 e O variabilă de tip pointer ocupă întotdeauna doi octeți

3) int a[5] = {1, 2, 3, 4, 5};
 int *aPtr;
 aPtr = a;
 printf("element=%d\n", *(aPtr + 2));
 Ce se va afisa după executia codului de mai sus?
 a. element=1 b. element=2 c. element=3
 d. element=4 e. eroare

4) char y = 'A';
 char *ptr_y;
 In exemplul de mai sus, cum vom face astfel incat ptr_y sa pointeze catre y ?
 a. ptr_y = (char *)y;
 b. *ptr_y = y;
 c. (char *) *ptr_y = y;
 d. *ptr_y = &y;
 e. ptr_y = &y;

5) int x[] = {1, 2, 3, 4, 5};
 int *ptr = x; x++;
 printf("%d ", *(ptr + 2));

```
printf( "%d\n", * ptr + 1 );
```

Ce se va afisa dupa executia seventei:

- a 3 2 b 4 2 c 4 3 d 3 1
- e eroare

6)

```
int x[] = {1, 2, 3, 4, 5};  
int u; int *ptr = x;  
????  
for( u = 0; u < 5; u++ ) {printf("%d-", x[u]); }  
printf( "\n" );
```

In exemplul de mai sus, ce instructiune trebuie sa inlocuiasca ???? astfel incat sa se afiseze 1-2-3-10-5- la executia codului?

- a. *(ptr + 3) = 10;
- b. *ptr[3] = 10;
- c. *ptr + 3 = 10;
- d. (*ptr)[3] = 10;
- e. *(ptr[3]) = 10;

7) Care din urmatoarele functii va citi un numar specificat de elemente dintr-un fisier?

- a. fileread()
- b. readfile()
- c. fread()
- d. getline()
- e. gets()

8) Care din urmatoarele propozitii este adevarata:

- a Un identificator nu poate depasi 32 de caractere
- b Constanta sir de caractere nu poate sa contina un singur caracter
- c Se pot aduna doi pointeri de același tip
- d Operatorul % se poate aplica asupra unei variabile de tip double
- e Operatorii relationali au prioritate mai mica decat cei aritmetici

9)Care din urmatoarele propozitii este falsa:

- a Exceptand tipul void, toate tipurile pot fi precedate de identificatori
- b Constanta caracter se delimita folosind caracterul apostrof
- c In cazul transmiterii prin valoare, functia apelata lucreaza cu copii ale parametrilor care se transmit
- d Un identificator poate sa inceapa cu o cifra
- e Se poate incrementa/decremenata un pointer

10) In cazul in care exista urmatoarea declaratie: int t[3] poate fi utilizata expresia "t++":

- a. Da si va indica urmatorul element al tabloului
- b. Nu, numele tabloului este un vector constant si nu poate fi modificat
- c. Nu, in general unui pointer nu i se poate aplica operatia t++
- d. Da, dar rezultatul este imprevizibil
- e. Da si va indica primul element al tabloului

11) In expresia float *fptr tipul de date float se refera la:

- a. Variabila fptr
- b. Adresa variabilei fptr
- c. Tipul variabilei pointate de fptr
- d. Este identica cu declaratia int *fptr
- e. Eroare la compilare, nu exista acest tip de declaratie

12) Ce instructiune trebuie adaugata secentei de mai jos pentru ca in final variabila pointer ptrj sa indice spre valoarea 4:

```
void main(void)
{ int j=3, *ptrj;
*ptrj++; }
a ptrj=j b ptrj=*j c ptrj=4 d ptrj=&j
e.Nici una din ele
```

13) char *ptr;
char myString[] = "abcdefg";
ptr = myString; ptr += (ptr + 5);

Ce sir va contine ptr dupa aceasta secenta:

- a abcdefgabcdefg5
- b acbdefg5
- c efg
- d cdefg
- e Necunoscuta; cod incorect

14) int* ptr; int y[10]; int i;
for (i=0; i < 10; i++) { y[i] = i; }
ptr = y; ptr += 8;
printf("ptr=%d\n", *ptr);

Ce se va afisa dupa executie:

- a ptr=0
- b ptr=7
- c ptr=8
- d ptr=9
- e cod incorect

15) f = fopen(fileName, "r");
if(????)
{ fprintf(stderr, "Could not open file!");
exit(-1); }

Cu ce trebuie inlocuita secenta ??? pentru a determina daca fisierul nu a putut fi deschis?

- a f == NULL
- b f == EOF
- c f == -1
- d fclose(f)
- e f != 0;

16) typedef struct customer_record

```

{ long cust_id;
  char custName[50];
} CUSTOMER_REC;
CUSTOMER_REC customer[50];
int i;
/* instructiuni diverse */
for (i = 0; i < 50; i++) { printf("%s\n", ?????); }
Cu ce trebuie inlocuita sevenita ????? pentru a fi afisat numele fiecarui client in exemplul dat:
a  customer_record[i].custName;
b  customer[i].custName;
c  customer_record.custName[i];
d  CUSTOMER_REC[i].custName;
e  customer.custName[i];

```

După parcurgerea modului, studenții vor fi capabili să rezolve probleme de dificultate medie în limbajul C.

2.3.6. Bibliografie modul

Cristian Bologa –Algoritmi și structuri de date, Editura Risoprint, 2006, capitolul 2
 Liviu Negrescu, Limbajele C și C++ pentru incepatori , Vol. I (p.1 si 2) - limbajul C (editia XI), Editura Albastra, Cluj-Napoca, 2005

Bibliografia completă a cursului

Cristian Bologa –Algoritmi și structuri de date, Editura Risoprint, 2006.
 Liviu Negrescu, Limbajele C și C++ pentru incepatori , Vol. I (p.1 si 2) - limbajul C (editia XI), Editura Albastra, Cluj-Napoca, 2005
 D. Knuth - Arta programării calculatoarelor, vol, 1, 2, 3, ed. Teora, 1999 (traducere)

ŞEF DEPARTAMENT

TITULAR DE DISCIPLINĂ