**1. Metode de proiectarea algoritmilor - Proiectarea modulara**

**1.1 Subprograme.**

**1.1.1.Noțiunea de subprogram**

Un subprogram este definit ca un set de instrucțiuni care pot fi reutilizate în mai multe locuri dintr-un program atunci când este convenabil. Această reutilizare are ca rezultat mai multe tipuri de economii, de la spațiul de memorie până la timpul de codare.

Subprogramul reprezintă descrierea unui proces de calcul cu ajutorul variabilelor virtuale (parametri formali). Apelul unui subprogram este execuția procesului de calcul pentru cazuri concrete, cu ajutorul parametrilor reali.

O declarație de subprograme cuprinde - un antet de supbrogram care precizează interfața subprogramului cu mediul său și blocul subprogramului care descrie funcționarea lui internă.

Utilizarea subprogramelor are avantaje:

a) reutilizarea codului – după ce am scris un subprogram îl putem apela de oricâte ori este nevoie;

b)modularizarea programelor – subprogramele ne permit să împărțim problema dată în mai multe subprobleme, mai simple;

c)reducerea numărului de erori care pot să apară în scrierea unui program;

d)depistarea cu ușurință a erorilor – fiecare subprogram va fi verificat la crearea sa, apoi verificăm modul în care apelăm subprogramele.

**1.1.2.Tipuri de subprograme**

Subprogramele pot fi de două tipuri:

* **funcții** – subprograme care determină un anumit rezultat, o anumită valoare, pornind de la anumite date de intrare. Spunem că valoarea este returnată de către funcție, iar aceasta va fi apelată ca operand într-o expresie, valoarea operandului în expresie fiind de fapt valoarea rezultatului funcției.
* **proceduri** – subprograme care se folosesc într-o instrucțiune de sine stătătoare, nu într-o expresie. Ele îndeplinesc o sarcină, au un efect și nu returnează un rezultat. De exemplu, citirea unor variabile, afișarea unor valori, transformarea unor date, etc.

**1.1.3.Structura subprogramelor**

Toate problemele se pot descompune în subprobleme, care devin module subordonate, unele repetându-se de un număr de ori, altele executându-se numai în funcție de anumite condiții.

Subprogramele se clasifică în funcție de originea lor în: subprograme standard (predefinite în biblioteci ale limbajului de programare) și nestandard sau create de utilizator care trebuie declarate și definite de utilizator.

Subprogramele sunt părţi ale unui program, identificabile prin nume, care se pot activa la cerere prin intermediul acestor nume.

Utilizarea subprogramelor într-un program presupune abordarea a două noţiuni:

-definirea unui subprogram şi apelul unui subprogram.

Definirea unui subprogram reprezintă de fapt descrierea unui proces de calcul cu ajutorul variabilelor virtuale(parametri formali) iar apelul unui subprogram este execuţia procesului de calcul pentru

-cazuri concrete (cu ajutorul parametrilor reali).

Un subprogram (funcţie) are o **definiţie** şi atâtea **apeluri**câte sunt necesare.

Definirea unui subprogram presupune declararea antetului și a corpului, operație care are următoarea structură:

tip\_returnat nume\_funcţie (lista parametrilor formali) //antet funcție

{

instrucţiune; // corpul funcţiei

}

unde:

tip\_returnat=>Reprezintă tipul rezultatului calculat şi returnat de funcţie şi poate fi: int, char, char\*, long, float, etc.

În cazul în care tipul rezultatului este diferit de void, corpul funcţiei trebuie să conţină cel puţin o instrucţiune return. Instrucţiunea return va specifica valoarea calculată şi returnată de funcţie care trebuie să fie de acelaşi tip ca şi tip\_returnat.

nume\_funcție Numele dat subprogramului de către cel ce îl va defini și apela. (Este un identificator)

**1.1.4.Transmiterea parametrilor: parametri (actuali, formali, corespondenţa), transmitere prin valoare, transmitere prin referinţă/adresă.**

Parametrii din antetul unei funcții se numesc **parametri formali**, iar cei din apelul funcției se numesc **parametri efectivi**.

Tipuri de parametri:

**A.Parametri formali**: aceștia sunt substituenții definiți în antetul funcției sau procedurii și servesc ca variabile locale în domeniul de aplicare al funcției.

**B.Parametri actuali**: Acestea sunt valorile sau expresiile reale transmise în funcție sau procedură atunci când este apelată.

În funcție de modalitatea de definire și de apel, subprogramele se clasifică în:

-subprograme care nu returnează o valoare (proceduri);

-cele care returnează o valoare(funcții).

Apelul unei funcţii care nu returnează o valoare are forma generală:

nume\_funcţie (lista parametrilor efectivi);

unde:

parametru efectiv = parametru actual = parametru real = parametru de apel

lista parametrilor efectivi = vidă sau o expresie sau mai multe despărţite prin virgulă

Apelul unei funcţii care returnează o valoare trebuie să fie apelată într-o expresie al cărei operand va fi. (valoarea funcției fiind utilizată în expresie).

Apelul unui subprogram provoacă o serie de operații secundare, pe care programatorul nu le vede direct, dar de care trebuie să țină seama. Reguli de lucru cu variabile de lucru:

- identificatorii declarați în cadrul unui subprogram sunt vizibili numai în interiorul acelui subprogram și se numesc elemente locale sau automate,fiind automat create când subprogramul este apelat și depuse pe stiva sistemului;

- identificatorii care desemnează parametrii unui subprogram au statut de variabile locale; -elementele declarate în afara modulelor au statut de elemente globale și sunt cunoscute în tot fișierul text al programului.

Transferul parametrilor la apel

Atunci când se execută o funcţie, ea foloseşte parametrii actuali care i-au fost transmişi prin apelul său, ţinând, însă, cont de natura parametrilor formali.

Scheme de transmisie a parametrilor

Schemele de transmisie a parametrilor determină modul în care parametrii actuali sunt transferați parametrilor formali în timpul apelurilor de funcții sau proceduri. Cele mai comune scheme de transmisie a parametrilor sunt:

A.Trecere după valoare: în această schemă, o copie a valorii parametrului actual este transmisă parametrului formal. Orice modificare a parametrului formal din cadrul funcției nu afectează valoarea inițială a parametrului actual.

Exemplu:

int f1(int a, float b);

int f2(char c);

Toți cei 3 parametri sunt transmiși prin valoare. Orice modificare pe parcursul executării subprogramelor nu va afecta valoarea originală, doar copia.

B.Transmitere prin referință: În această schemă, adresa de memorie a parametrului actual este transmisă parametrului formal. Orice modificare a parametrului formal afectează direct valoarea inițială a parametrului actual.

Transmiterea prin adresă este asemănătoare cu cea prin referință, doar că prima se adresează în special pointerilor.

int f(int&x); // referință

int g(int\*x); // adresă

***Parametrii actuali:***

Spre deosebire de parametrii formali, la apel nu vom adăuga nici tipul de dată al variabilelor, nici dimensiunile tablourilor (tipul trebuie să corespundă totuși). În apel vom introduce variabile în ordinea în care dorim să corespundă cu parametrii formali, pentru a fi inițializați corespunzător.

int B[101],C[101],X[101],b,c,x;

CitireVector(B,b);

CitireVector(C,c);

CitireVector(X,x);

C.Trecere prin indicator: Această schemă este similară cu trecerea prin referință, în care adresa de memorie a parametrului actual este transmisă parametrului formal.

**1.1.5.Apelul subprogramelor: utilizarea apelului în cadrul expresiilor, apel de tip instrucţiune, autoapelare (recursivitate).**

A apela un subprogram, înseamnă a-l lansa în executare. Pentru a putea fi apelat din cadrul unui bloc, un subprogram trebuie declarat la nivelul blocului respectiv.

**Declararea și apelul funcțiilor**

O funcție e un subprogram care calculează și returnează programului apelant o singură valoare. Aceasta valoare este asociată numelui funcției. Iar tipul poate fi simplu, string sau reper. Valoarea returnata de funcție nu poate avea alt tip structurat decât string.

FUNCTION nume\_functie(lista parametrii formali): identificator de tip;-nume\_functie reprezintă numele funcției, al cărei tip este identificator de tip

-identificator de tip = nume de tip simplu: STRING

Blocul funcției trebuie să conțină obligatoriu o instrucțiune de atribuire prin care identificatorul funcției primește valoarea unei expresii.

Identificatorul funcției nu are voie să apară în partea dreaptă a unor atribuiri decât dacă funcția este recursivă.

Apelul unei funcții decurge astfel:

-se întrerupe calculul expresiei în care a apărut apelul funcției ;

-se transmit parametrii, dacă există, exact ca la proceduri ;

-se execută funcția;

Prin recursivitate se înțelege faptul că un subprogram se apeleaza pe el însuși, apelul apărând atunci când subprogramul este încă activ. Există două tipuri de recursivitate:

-recursivitate directă - când un subprogram se autoapelează în corpul său ;

recursivitate indirecta – când avem două subprograme (x și y), iar x face apel la y și invers ;

Se folosesc algoritmi recursivi atunci când calculele aferente sunt descrise în forma recursivă.

Recursivitatea este frecvent folosită în prelucrarea structurilor de date definite recursiv. Un subprogram recursiv trebuie scris astfel încât să respecte regulile :

a) Subprogramul trebuie să poată fi executat cel puțin o dată fără a se autoapela ;

b)Subprogramul recursiv se va autoapela într-un mod în care se tinde spre ajungerea în situația de execuție fără autoapel.

Pentru implementarea recursivității se folosește o zonă de memorie în care se poate face salvarea temporară a unor valori. La fiecare apel recursiv al unui subprogram se salvează în această zonă de memorie starea curentă a execuției sale.

Deși variabilele locale ale subprogramului apelant au aceleași nume cu cele ale subprogramului apelat, orice referire la acești identificatori se asociază ultimului set de valori alocate în zona de memorie.

Zona de memorie rămâne alocată pe tot parcursul execuției subprogramului apelat și se dealocă în momentul revenirii în programul apelat. Zona de memorie nu este gestionată explicit de programator ci de către limbaj.

La terminarea execuției subprogramului apelat recursiv, se reface contextul programului din care s-a făcut apelul. Datorită faptului că la fiecare autoapel se ocupă o zona de memorie, recursivitatea este eficientă numai dacă numărul de autoapelări nu este prea mare pentru a nu se ajunge la umplerea zonei de memorie alocată.

Recursivitatea oferă avantajul unor solutii mai clare pentru probleme și a unei lungimi mai mici a programului. Ea prezintă însă dezavantajul unui timp mai mare de execuție și a unui spațiu de memorie alocată mai mare. Este de preferat ca atunci când programul recursiv poate fi transformat într-unul iterativ să se facă apel la cel din urmă.

**1.2 Modularizarea programelor**

**1.2.1.Variabile globale şi locale**

Modularizarea. În cadrul unor probleme complexe este necesară descompunerea acestora în subprobleme mai simple, pentru care se pot scrie module de program mai simple. Fiecare modul este independent de celelalte, în final ele interacţionând printr-un program principal, un fel de interfaţă. Modulele pot fi implementate de programatori diferiţi din cadrul unei echipe. Ele pot fi testate, modificate, depanate în mod independent, neafectând şi celelalte module. Modulele au aceeaşi structură ca şi programele principale, putând declara variabile atât în module (numite variabile **locale**) cât şi în programul principal (numite variabile **globale**). Modulele create sunt proceduri şi funcţii.

Așadar, **variabilele globale** sunt create în exteriorul funcției, în timp ce **variabilele locale** sunt create în interiorul unei funcții.

**A. Utilizarea variabilelor globale**

Deci variabilele globale sunt cele declarate în afara oricărei funcții.  
Totalitatea locurilor unde este accesibilă o variabilă se numește "domeniu de existență al variabilei". Variabilele globale nu pot fi accesibile din interiorul corpului unei funcții; cu alte cuvinte, domeniul de existență al unei variabile globale nu include corpurile funcțiilor. Dacă dorim să obținem accesul la o variabilă globală în cadrul unei funcții putem extinde domeniul de existență al variabilei prin specificarea în funcție a numelui variabilei în cadrul unei instrucțiuni **GLOBAL**.  
Instrucțiunea GLOBAL are următoarea formă:

**GLOBAL variabila1, variabila2, variabila3;**

Exemplu:

<?php  
$var1 = 135;  
$var2 = 250;  
function Suma() {  
        GLOBAL $var1, $var2;  
        return $var1 + $var2;  
}  
echo "Suma este ". Suma();  
?>

**Se va afișa: Suma este 385**

În exemplul de mai sus declarația: "GLOBAL $var1, $var2" face ca variabilele $var1 si $var2 să fie recunoscute și în interiorul funcției.

### B. Utilizarea variabilelor locale

Variabilele locale sunt create în interiorul funcției și sunt distruse când se încheie apelul la funcția respectivă. În consecință, variabilele locale sunt disponibile numai pe durata execuției funcției asociate.  
Argumentele funcțiilor constituie un tip important de variabilă locală. Cu toate acestea, putem crea o variabilă locală prin simpla atribuire a unei valori unei variabile din interiorul unei funcții.  
Pentru a ilustra deosebirea dintre variabilele locale și cele globale, iată un script care definește o variabilă locală denumită "$x" și o variabilă globală cu același nume:

<?php  
function v\_local() {  
        $x = 5;  
        echo "<br />In corpul functiei x = $x";  
}  
$x = 2;  
echo "<br />In corpul scriptului x = $x";  
v\_local();  
echo "<br />In corpul scriptului x = $x";  
?>

**Se va afișa:**

In corpul scriptului x = 2  
In corpul functiei x = 5  
In corpul scriptului x = 2

**1.2.2.Domeniul de vizibilitate**

Domeniul de vizibilitate se folosește atunci când într-un program este declarată o variabilă care poate fi folosită în interiorul altor funcții din interiorul programelor.

S-au creat variabile locale și globale pentru a exista o modulare cat mai perfectă, deoarece o variabilă care are aceeași denumire dintr-o funcție să nu se amestece cu altă variabilă cu același nume din altă funcție, care trebuie să fie diferită. Astfel s-au creat variabilele locale și globale și a fost creat acest domeniu de viziblitate.

**1.2.3.Integrarea subprogramelor în cadrul programului**

După ce au fost create aceste subprograme ele trebuie integrate în interiorul unui program pentru a avea mai multă vizibilitate și a ajunge la un rezultat cât mai bun.

**2. Metode de proiectarea algoritmilor**

Programare Programare procedural procedurală

Paradigmă în programare, stil în programare prin care un program apelează subprograme (funcții și proceduri, pentru a rezolva subprobleme) prin transmiterea corespunzătoare a parametrilor (actuali, aflați în corespondență cu parametrii formali descriși în subprograme).

**Programare structurată**

Disciplină în programare prin folosirea unor structuri bine precizate (fără Goto). Orice problemă se poate descrie doar prin utilizarea următoarelor trei structuri:

• secvențială,

• alternativă,

• repetitivă.

**Programare Bottom-Up** –Metoda ascendentă

Utilizând propoziţiile limbajului şi subalgoritmi existenţi, îi asamblează în alţi subalgoritmi pentru a ajunge în final la algoritmul dorit. Deci, va fi scris mai întâi subalgoritmul apelat şi apoi cel care apelează. Ca rezultat al proiectării ascendente se ajunge la o mulţime de subalgoritmi care se apelează între ei. Este important să se cunoască care subalgoritm apelează pe care, lucru redat printr-o diagramă de structură, ca şi în cazul programării descendente. Această metodă are marele dezavantaj că erorile de integrare vor fi detectate târziu, abia în faza de integrare. Se poate ajunge abia acum la concluzia că unii subalgoritmi, deşi corecţi, nu sunt utili.

De cele mai multe ori nu se practică o proiectare ascendentă sau descendentă pură ci o combinare a lor, o proiectare mixtă: Top\_Down & Top\_Down Bottom.

Prin **scheme de algoritm** înțelegem tipare comune pe care le putem aplica în rezolvarea unor probleme similare. O gamă largă de probleme se pot rezolva folosind un numar relativ de scheme.

În Computer Science divide et impera se referă la o clasă de algoritmi care au ca principale carcateristici faptul că împart problema în subprobleme similare cu problema inițială dar de dimensiuni mici, rezolvă problemele recursiv și apoi combină soluțiile pentru a crea soluție pentru problema originală.

Problema se împarte în 3 etape:

– divide – problema dată într-un număr de subprobleme ;

– impera – subproblemele sunt rezolvate recursiv;

– combină – soluțiile subproblemelor sunt combinate pentru a obține soluția problemei inițiale.

Folosind această metodă sunt câteva avantaje și dezavantaje:

Avantaje:

– produce algoritmi eficienți;

– descompunerea problemei în subprobleme;

– faciliteaza paralelizarea algoritmului în vederea execuției sale pe mai multe procese;

– sunt independenți, creștere mare în viteză.

Dezavantaje:

– se adaugă un overhead datorat recursivității (reținerea pe stivă a apelurilor funcției).

**Scheme de algoritm**

Algoritmul Merge sort este un exemplu clasic de rezolvare divide Impera.

Divide: Divide cele n elemente ce trebuie sortate în 2 secvențe de lungime n/2;

Impera: Sortează secvențele recursive folosind merge sort;

Combină: Secvențele sortate sunt asamblate pentru a obține vectorul sortat.

Observație:

Recurența se oprește când secvența ce trebuie sortată are lungimea 1 (un vector cu un singur element este întotdeauna sortat).

Operația cheie este asamblarea soluțiilor parțiale.

Exemplu:

A) Merge Sort(A,p,r)

1. Dacă p>r

2. Atunci q<-[(p+r)/2] //divide

3. Merge\_Sort(A,p,q) //impera;

4. Merge\_Sort(A,q+1,r)

5. Merge(A,p,q,r); // combina

Merge(A,p,q,r) // p și r sunt capetele intervalului, q este mijlocul

n1<-q-p+1 // număr de elemente din partea stângă

n2<-r-q// număr de elemente din partea dreaptă

crează vectori S[1<-n1+1]

D[1<-n2+1]

Pentru I de la 1 la n1

S[i]<-A[p+i-1]//se copiază partea stângă în S

Pentru j de la 1 la n2

D[j]<-A[q+j] // se copiază partea dreaptă în D

S[n1+1]->infinit

D[n2+1]->infinit

Pentru k de la p la r // se copiază înapoi vectorul

Dacă S[i] <=D[j]

Atunci A[k]<-S[i]

Altfel A[k]<-D[j]

Se folosesc 3 apeluri recursive

B) Greedy

Este o metodă de rezolvare eficientă a unor probleme de optimizare. Soluțiile trebuie să satisfacă un criteriu de optim global. Se aleg soluțiile parțiale ce sunt îmbunătățite repetat pe baza criteriului de optim local până ce se obțin soluțiile finale.

Soluțiile parțiale ce nu pot fi îmbunătățite sunt abandonate → proces de rezolvare irevocabil.

Schema generală de rezolvare a unei probleme:

Rezolvare\_lacoma(Crit\_optim,Problema)

1. sol\_partiale = sol\_initiale(Problema)

2. sol\_fin = multime vida;

3. Cat timp (sol\_partial diferit de multimea vida)

4. Pentru fiecare (s in sol\_partiale)

5. Daca (s este o solutie a problemei {//daca este solutie

6. sol\_fin = sol\_fin U {s}; // finala se salveaza

7. sol\_partiale = sol\_partiale/{s};

8. }

9. ASTFEL// SE POATE OPTIMIZA

Daca (optimizarea\_posibila(s,Crit\_Optom,Problema))

10. sol-partiale=sol-partiale/{s} U//da

optimizare(s,Crit\_Optom,Problema)

Astfel sol-partiale = sol\_partiale/{s};

11. Intoarce sol\_fin;

Comparatie Divide Impera si Greedy

D&I:top-down;

Greedy: Top-down;

Criteriu de optim:

D&I:Nu

Greedy: Da

Exemplu:

Arbori Huffman:

– metoda de codificare folosită la compresia fișierelor;

– construcția unui astfel de arbore se realizează printr-un algoritm Greedy.

Exemplu:

Codificarea propoziției Ana are mere

1. K = multimea de simboluri ce pot fi codificate(a,n,” “,r,e,m)

2. Arbore de codificare a cheilor k este un arbore binar ordonat cu proprietatile:

-doar frunzele arborelui conțin cheile din K.

-nu există mai mult de o cheie în frunză;

– toate nodurile interne cu exact 2 copii;

– arcele sunt codificate cu 0 si 1(arcul din stânga unui nod codificat cu 0, iar cel din dreapta cu 1)

– k reprezinta codul unei chei –este șirul etichetelor de pe calea de la rădăcina arborelui la frunza care conține cheia k (k este din K)

– p(k) –frecvența de apariție a cheii k în textul ce trebuie comprimat;

Exemplu pentru propoziția Ana are mere:

p(a) = p(e) = 0.25;

p(n)=p(m)=0.083;

p(r)=p( ) = 0.166.

A - arborele de codificare a cheilor

lg\_cod(k) – lungimea codului chei k conform A;

nivel(k,A) – nivelul pe care apare in A, frunza ce conține cheia k.

Costul unui arbore de codificare A, al unor chei K relativ la o frecvență p este:

Cost(A) = Ʃlg-cod(k)\*p(k) = Ʃnivel(k,A)\*p(k)

k = Kk=K

Observație:

Un arbore de codificare cu cost minim al unor chei k relativ la o frecvență p este un arbore Huffman, iar codurile cheilor sunt coduri Huffman.

Algoritm de construcție:

1. Pentru fiecare k din K se construiește un arbore cu un singur nod care conține cheia k și este caracterizat pe ponderea w = p(k). Subarborii construiți formează o mulțime numită Arb.

2. Se aleg 2 subarbori a și b din Arb astfel încât a și b au o pondere minimă;

3. Se construiește un arbore binar cu o rădăcina r care conține nici o cheie si cu descedenți a si b. Ponderea arborelui este definită cu w(r) = w(a)+w(b).

4. Arborii a și b sunt eliminați din Arb;

5. Se repetă procesul de construcție descris la pașii 2-4 până când mulțimea Arb conține un singur arbore –Arborele Huffman pt cheia k

Programarea dinamica

**2.1.Programarea structurata. Principii.**

Datele se pot grupa în structuri de date. Prelucrările asupra datelor se pot grupa şi ele în structuri despre care vorbim în teorema Bohm-Jacopini. Teorema Bohm-Jacopini afirmă ca orice algoritm se poate reprezenta folosind 3 tipuri de structuri fundamentale: structură liniară, structură alternativă şi structură repetitivă.

**Structura liniară**

Structura liniară (numită şi secvențială) este alcătuită din următoarele instrucţiuni:

• comentarii

• declararea variabilelor

• instrucţiunea de citire

• instrucţiunea de scriere

• instrucţiunea de atribuire

• instrucţiunea compusă (sau blocul de instrucţiuni).

**Operatorii de atribuire** sunt:

Principalul operator de atribuire este = și înseamnă asocierea operatorului din stânga cu valoarea expresiei din dreapta (și nu trebuie confundat cu egalitatea). Valoarea unei expresii de atribuire este valoarea atribuită, adică valoarea lui $a=5 este 5.

**Interschimbarea conținutului a două variabile**

Cea mai cunoscută metodă este **interschimbarea cu variabilă auxiliară.**

Exemplu de aplicare a interschimbabilității: dacă avem 2 pahare de suc (un pahar A-cu limonadă și un pahar B-cu suc de portocale) și vrem să le interschimbăm conținutul ne vom folosi de un pahar gol (C). Procesul următor se rezumă la 3 operații simple:

1)Turnăm limonada în paharul gol. Acum paharul A este gol, paharul B este plin cu suc de portocale, iar paharul C conține limonadă.

2)Turnăm sucul de portocale în paharul gol. Acum paharul A conține suc de portocale, paharul B este gol, iar paharul C este în continuare plin cu limonadă.

3)La final turnăm limonada în paharul gol. Acum paharul A este plin cu suc de portocale, paharul B conține limonadă, iar paharul C este gol, exact ca la început.

**2.2.Instructiunile simple ale limbajului.**

**2.3.** **Structuri de decizie/alternative: cu o singură ramură, cu două ramuri, selecţie multiplă (case / switch)**

**Ramificarea condiţionată - if**

Prima şi totodată instrucţiunea de ramificare de bază este *if*. Această instrucţiune este cunoscută în aproape toate limbajele chiar aşa cum se numeşte: „dacă”.

Pe lângă acest cuvânt mai trebuie introdusă o anumită condiţie de care va depinde rezultatul acestei

instrucţiuni.

Programul se execută până ce ajunge la partea care provoacă ramificarea (if), unde verifică condiţia. Dacă condiţia este îndeplinită (true), se execută codul definit (if code). Dacă condiţia nu este îndeplinită (false), atunci va fi omis if.

code (nu se va executa), ci se continuă cu încărcarea paginii.

if-else se poate deplasa de la condiţia de bază, cea minimă:

if (conditia)

comanda;

care reprezintă doar o condiţie care trebuie îndeplinită pentru executarea unei anumite instrucţiuni. Prin posibilităţile soluţiei alternative:

if (conditia)

{ comanda }

else

{ comanda2 };

deseori se va întâmpla să fie nevoie să se ia de dinainte decizia dacă o anumită acţiune se va executa, însă este mai important să se determine ce grup, din mai multe acţiuni oferite, va trebui să se execute exact la momentul dat. Când este vorba de o astfel de problemă, folosim instrucţiunea **else** în continuarea instrucţiunii condiţionate **if**. Instrucţiunea **else** se execută dacă condiţia nu este îndeplinită.

**Bucla *for***

Bucla *for* se utilizează atunci când ştim în prealabil cu exactitate de câte ori trebuie executat blocul de instrucţiuni. Această buclă are următoarea sintaxă:

*for (iniţializare; condiţie; incrementare) {*

*bloc de instrucţiuni;*

*}*

Primul parametru se foloseşte pentru crearea şi pentru setarea valorii iniţiale a numărătorului/counter. Al doilea parametru conţine condiţia pentru counter, iar al treilea determină cum anume este incrementat counter-ul.

Exemplu cu For în care sunt afișate primele n numere impare:

 <?php

    for($i=0;$i<=15;$i++){

        if($i % 2 == 0){

            continue;

        }

        echo $i.', ';

    }

    ?>

Ca rezultat se vor afișa numerele: 1, 3, 5, 7, 9, 11, 13, 15,

Exemplu cu For în care sunt afișate primele n numere naturale:

for ($i = 0; $i <= 10; $i++) {

           if($i == 5){

               // daca $i este 5, continua si sari peste 5\*

               continue;

           }

echo "$i";}

Ca rezultat se vor afișa numerele: 012345678910

Alt exemplu de afișare a numerelor pare de la 1 la 20, folosind for:

for($i = 1; $i<=20; $i++){

                   if($i % 2 == 0){

                       echo 'Numerele pare sunt : ' . $i . '<br>';

                   }

               }

Se va afișa:

Numerele pare sunt : 2  
Numerele pare sunt : 4  
Numerele pare sunt : 6  
Numerele pare sunt : 8  
Numerele pare sunt : 10  
Numerele pare sunt : 12  
Numerele pare sunt : 14  
Numerele pare sunt : 16  
Numerele pare sunt : 18  
Numerele pare sunt : 20

**Bucla *while***

În PHP, bucla *while* reprezintă cea mai simplă buclă. La fel ca şi instrucţiunea *if* şi bucla *while* depinde de condiţii. Trebuie să facem o paralelă între *while* şi *if*. Pentru a şti care instrucţiune este mai corectă pentru a fi utilizată, începem de la instrucţiunea care trebuie să se execute după îndeplinirea condiţiilor. Principala deosebire este tocmai faptul că instrucţiunea if *execută doar o singură dată* blocul de instrucţiuni care urmează după îndeplinirea condiţiilor, în timp ce bucla while *execută instrucţiunile atâta timp cât condiţia este îndeplinită.*

Bucla while intră în categoria în care nu trebuie cunoscut în prealabil numărul de iteraţii. Pur şi simplu, această buclă se execută de atâtea ori cât îi permite condiţia care se execută în ea. Bucla while necesită o anumită condiţie, care se introduce ca şi parametru în timpul iniţializării buclei şi se execută până ce condiţia este îndeplinită. Pentru asigurarea executării codului în interiorul buclei while, condiţia definită la începutul instrucţiunii while trebuie să fie îndeplinită. Condiţia pe care o definim în timpul creării buclei while nu se deosebeşte deloc de condiţia pe care am utilizat-o în timpul definirii instrucţiunilor if.

Condiţia are un rol foarte important, deoarece uneori trebuie să întrerupem executarea buclei ca să nu ne găsim în **bucla moartă**.

Sintaxa:

while (*conditie*) {

*//Blocul de cod care se va executa daca conditia este indeplinita*

}