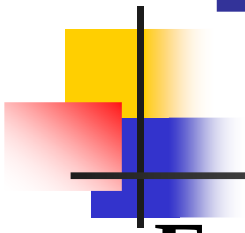
Elemente de programare avansată în limbajul C



1. **Funcții de conversie**
2. **Operatori pe biți**
3. **Tabelul precedenței operatorilor**
4. **Argumente în linia de comandă**
5. **Argumente de tip pointer**
6. **Pointeri la funcții**
7. **Preprocesorul**
8. **Funcții cu număr variabil de argumente**

Funcţii de conversie

Clasificări şi conversii de caractere <ctype.h>

isalpha(c) - returnează *adevărat* (valoare nenulă) dacă c este o literă, respectiv *fals* (zero) în caz contrar.

isupper(c) - returnează *adevărat* dacă c este o literă mare

islower(c) - returnează *adevărat* dacă c este o literă mică

isdigit(c) - returnează *adevărat* dacă c este o cifră

isalnum(c) - returnează *adevărat* dacă c este literă sau cifră

isspace(c) - returnează *adevărat* dacă c este un caracter de spațiere (spațiu, tab, linie nouă).

toupper(c) - transformă caracterul *c* din literă mică în literă mare

tolower(c) - transformă caracterul *c* din literă mare în literă mică

Funcţii de conversie

Conversii de şiruri <stdlib.h>

- Conversia unui şir de caractere la *real*, *întreg*, respectiv *long*:

```
double atof(char *s);
int atoi(char *s);
long atol(char *s);
```

Şirul de caractere dat ca şi parametru - valoare numerică în formatul corespunzător.

Dacă şirul nu poate fi convertit, funcțiile returnează val. **0**.

- Conversia inversă, din format numeric în şir de caractere.

```
char * itoa(int val, char *s, int baza);
char * ltoa(long  val, char *s, int baza);
```

Aceste funcții produc în şirul *s* o reprezentare a *val* în *baza*.

Funcţii de conversie

Conversii de şiruri cu format <stdio.h>

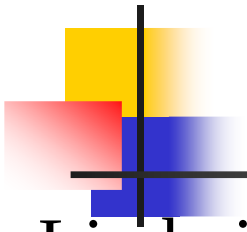
int sscanf(char *s, char *format, adresa1, adresa2, ...);
int sprintf(char *s, char *format, arg1, arg2, ...);

Spre deosebire de **scanf** şi **printf**, transferul de date se face dintr-un şir de caractere şi nu de la intrarea sau ieşirea standard.

Exemplu:

```
char s[80]="23.5  abc  3";
char sir[10];
int n;
float x;
sscanf(s, "%f %s %d", &x, sir, &n);
sprintf(s, "2*%.1f=%.1f ",x, 2*x);
```

În urma execuției secvenței anterioare, variabilele *x*, *sir* şi *n* au valorile: *x=23.5 sir=“abc”* şi *n=3*. Şirul de caractere *s* va fi în final ”2*23.5=47.0 ”



Operatori pe biţi

- Limbajul C furnizează *6 operatori* pentru manipularea biților. Ei nu pot fi aplicați decât operanzilor întregi (de tip char, short, int sau long, cu sau fără semn).

~	complement față de unu(negare bit cu bit)- operator unar
&	ŞI pe biți
^	SAU EXCLUSIV pe biți
|	SAU pe biți
>>	deplasare la dreapta
<<	deplasare la stânga

Operatori pe biţi

- Interpretează operanzii ca şiruri de biți, fiecare bit fiind tratat independent de ceilalți.
- Operanzii pot fi de orice tip întreg.
- **Operatorul de negație** inversează valorile biților operandului: fiecare bit de 1 devine 0 şi fiecare bit de 0 devine 1.

Operator	Denumire	Utilizare
~	negare	~e
&	şi	e1 & e2
^	sau exclusiv	e1 ^ e2
|	sau	e1 | e2
>>	deplasare dreapta	e1 >> n
<<	deplasare stânga	e1 << n

Operatori pe biţi

- Operatorii de **deplasare** >> şi << realizează o deplasare, la dreapta şi respectiv la stânga, a biților expresiei *e1* care este primul operand, cu un număr de poziții egal cu valoarea *n* a operandului al doilea.

Exemplu: unsigned char x=0x97;

x		1	0	0	1	0	1	1	1
~ x		0	1	1	0	1	0	0	0
x >> 2		0	0	1	0	0	1	0	1
x << 1		0	0	1	0	1	1	1	0

Operatori pe biţi

- Operatorii **şi** (&), **sau** (|) şi **sau exclusiv**(^) pe biți prelucrează în sensul precizat de operator, perechile de biți ai celor doi operanzi pe care îi primesc. Au o semnificație diferită de operatorii logici **&&** şi ||, care tratează valorile operanzilor în totalitatea lor ca valori logice.

Exemplu: unsigned char x=0x97;
unsigned char y=0x34;

x	1	0	0	1	0	1	1	1
y	0	0	1	1	0	1	0	0
x & y	0	0	0	1	0	1	0	0
x | y	1	0	1	1	0	1	1	1
x ^ y	1	0	1	0	0	0	1	1

Operatori pe biţi

- Operatorul **şi pe biți**, &, este deseori folosit pentru a seta la zero un anumit număr de biți.

```
n = n & 0177;
```

setează la zero toți biții lui n cu excepția ultimilor 7 biți de ordin inferior.

- ```
  x = x & ~ 077;
  ```

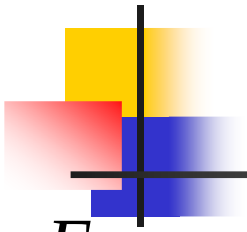
setează la zero ultimii şase biți ai lui x.

- Operatorul **sau pe biți**, |, este folosit pentru a seta anumiți biți la valoarea 1:

```
x = x | SET_ON;
```

setează la unu biții din x corespunzători pozițiilor care în SET_ON au valoarea 1.
```

Operatori pe biţi
Exemple



- *Exemplul 1*: funcția preiabiti(x,n,p) returnează câmpul de n biți (aliniați la dreapta) ai lui x, care începe la poziția p. Presupunem că poziția zero se află la capătul din dreapta.

```
/* preiabiti: preia n biti din x, incepand cu pozitia p*/
unsigned preiabiti(unsigned x, int p, int n)
{
   return (x >> (p+1-n) & ~(~0 << n));
}
```

- *Exemplul 2*: funcția contorbiti numără biții cu valoarea 1 din argumentul său x, de tip întreg.

Operatori pe biţi

Exemple

```
/* contorbiti: numara bitii cu valoarea 1 din x */
int contorbiti (unsigned x) {
   int b;
   for (b = 0; x != 0; x >>=1)
       if (x & 01)
            b++;
   return b;
}
```

- Declararea argumentului x ca fiind de tip *unsigned* asigură completarea cu zerouri a biților rămaşi liberi când acesta este deplasat la dreapta, indiferent de maşina pe care este rulat prog.

Tabelul precedenţei operatorilor

Nivel	Operatori	Asociativitate
1	**( )** (apel funcție) **[ ]** (indexare) **.** **->** (op. de selecție)	stânga
2	+ **-** (op. unari de schimbare de semn) ++ **--** **~** **!** (negație bit cu bit şi respectiv logică) **(tip)** **sizeof** ***** (adresare indirectă) **&** (adresa)	dreapta
3	***** / **%** (op. binari multiplicativi)	stânga
4	+ **-** (op. binari aditivi)	stânga

Tabelul precedenţei operatorilor

5	>> << (op. deplasare)	stânga
6	< > <= >= (op. comparație)	stânga
7	== !=	stânga
8	& (şi pe biți)	stânga
9	^ (sau exclusiv pe biți)	stânga
10	| (sau pe biți)	stânga

Tabelul precedenţei operatorilor

11	**&&** (şi logic)	stânga
12	**||** (sau logic)	stânga
13	**?** **:**	dreapta
14	= **+=** **-=** ***=** **/=** **%=** **&=** **^=** **|=** **>>=** **<<=**	dreapta
15	**,**	stânga

Argumente în linia de comandă

- Declarația generală a unei funcții main este :

```
int main(int argc, char **argv);
```

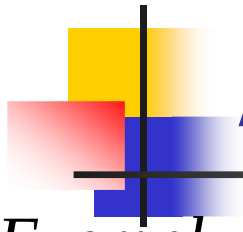
Funcția main returnează un tip (int) şi poate avea argumente. Atât rezultatul funcției main cât şi argumentele ei interesează în condițiile în care **se utilizează programul ca şi o comandă lansată din linia de comandă a sistemului de operare sau când programul este lansat în execuție de un alt program**.

- Valoarea întreagă returnată de main este disponibilă pentru programul care a lansat în execuție programul curent. Se obişnuieşte ca valoarea returnată de un program să fie **0** în caz de terminare corectă şi o valoare nenulă în caz contrar.

Argumente în linia de comandă

- Pentru ieşirea forțată din program în caz de erori se recomandă funcția *exit* cu *parametrul 1*. Spre deosebire de o simplă instrucțiune *return 1;* această funcție realizează în plus şi alte operații, precum închiderea tuturor fişierelor deschise de program.
- Funcția main are două argumente, cu următoarele semnificații: primul argument, *argc(argument count)*, este un întreg având semnificația de număr al argumentelor date în linia de comandă, iar al doilea argument, *argv(argument vector)*, este un vector de şiruri de caractere, reprezentând celelalte argumente din linia de comandă.
- Prin convenție, *argv[0]* este numele prin care s-a apelat programul.

argv[argc] este *NULL* (sfârşitul argumentelor)



Argumente în linia de comandă

*Exemplu:*Program care îşi afişează arg. primite în linia de comandă.

Fie următorul program, compilat în fişierul cu numele prog:

```
#include <stdio.h>
int main(int argc, char **argv) {
  int i;
  printf("numele programului:%s\n",argv[0]);
  if(argc==1) printf("nu are argumente\n");
  else
    for (i=1; i<argc; i++)
      printf("argumentul %d: %s \n", i, argv[ i ]);
  return 0;  /*codul returnat de program*/
}
```

Argumente în linia de comandă

Lansarea în execuție a programului prin linia de comandă:

```
prog   arg1   arg2   arg3
```

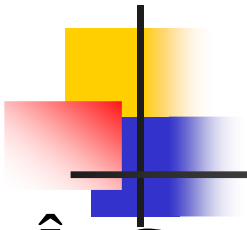
produce ieşirea:

```
numele programului: prog
argumentul 1: arg1
argumentul 2: arg2
argumentul 3: arg3
```

- Observație:

Redirectările de intrare şi ieşire specificate în linia de comandă(vezi cap. **Fişiere**) nu se consideră argumente în linia de comandă.

Argumente de tip pointer
Transmiterea argumentelor prin valoare



- În C, toate argumentele funcțiilor sunt transmise "*prin valoare*". Acest lucru înseamnă că funcția apelată primeşte valorile argumentelor sale prin stocare în variabile temporare create în stivă special în acest scop. În acest fel, ***funcția nu are acces la locațiile de memorie ale variabilelor originale***.
- În C, *funcția apelată nu poate modifica direct variabila corespunzătoare argumentului transmis din funcția apelantă*; ea nu poate modifica decât copia temporară.
- Apelul prin valoare conduce la programe mai compacte, cu mai puține variabile neesențiale, deoarece parametrii pot fi tratați în rutina apelată ca variabile locale, convenabil inițializate ca urmare a apelului.

Argumente de tip pointer

Transmiterea argumentelor prin valoare

- Exemplu: O funcție de interschimbare a două elemente

```
void schimba(int x, int y)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}
```

- Apelul `schimba(a, b);` nu produce efectul dorit. Nu interschimbă argumentele a cu b, ci doar copiile lor x și y.

Argumente de tip pointer
Simularea transmiterii prin adresă

- Când este necesar, se poate face ca o funcție să modifice o variabilă din funcția apelantă. Funcția apelantă trebuie să furnizeze *adresa variabilei* ce va fi accesată (un *pointer* către acea variabilă), iar funcția apelată trebuie să declare *parametrul ca fiind pointer* şi să acceseze variabila *indirect*, prin intermediul acestuia(vezi capitolul despre pointeri).
- O altă excepție este în cazul tablourilor: când numele unui tablou este folosit ca argument,valoarea transmisă funcției este adresa locației primului element al tabloului. Folosind această valoare ca pe o variabilă cu indici, funcția poate accesa şi poate modifica ***direct*** orice element al tabloului.

Argumente de tip pointer
Exemplu

- Reluarea exemplului anterior, utilizând argumente pointeri:

```
void schimba1(int *x, int *y)
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
```

- În acest caz, apelul devine: schimba1(&a, &b); şi interschimbarea se realizează chiar între valorile argumentelor a şi b.

Pointeri la funcţii

Din punct de vedere al conținutului *zonei de memorie* indicate, există următoarele categorii de pointeri:

- **pointeri de date**, care conțin adresa unei variabile;
- **pointeri generici**, pointeri `void *`, pot conține adresa unui obiect oarecare, de orice tip. Se utilizează atunci când nu se cunoaşte precis tipul de date care va fi instanțiat în mod dinamic la execuție;
- **pointeri de funcții**, care conțin adresa codului executabil al unei funcții. În C, o funcție nu este o variabilă, dar este posibil să se definească **pointeri la funcții care se comportă similar unor variabile**: pot fi *atribuiți*, *plasați în tablouri*, *transmişi ca argumente* altor funcții, etc.

Pointeri la funcţii

Exemple de utilizare

- Declarație obişnuită de funcție: tip_rez f(.....);
- Declarație de pointer la funcție: tip_rez (*pf) (.....);
- Atribuiri echivalente: pf = f ; pf = &f ;
- Apeluri echivalente: f(...); pf(....); (*pf)(....);

int *f(void); //o funcție ce returnează un pointer la int

int (*f)(void); //pointer la o funcție ce returnează un int

- Exemplu: un tablou de pointeri de funcții

```
typedef  void  (*pf)(void);
void f1(void); void f2(void);  /*...*/ void f10(void);
pf  ftab[10] = { f1, f2, ….. ,f10 };
….int k; ….. ftab[k] ( ); ……
```

Pointeri la funcţii

Programul 1

```
void   tab  ( double (* f )  (int ) ,  int  j , int  i ) {
for ( ; j < = i ;  j ++)
printf  ( " %d    %f  \ n " , j , ( * f ) ( j )  ) ;  }

double  f1 (int  x ) {
return  2 *  3.14  * x  ;  }

double  fact (int  x ) {
double  f = 1 ;  int  i ;
for ( i = 1 ; i < = x ;  i ++)
f * =  i ;
return  f  ;  }
```

Pointeri la funcţii

Programul 1

```
tab  ( f1,  1,  10 ) ;
tab  ( fact,  0, 10 ) ;
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

- În C, unica manieră de transmitere a unui argument către o funcție, este *prin valoare*. Ea permite însă programatorului să realizeze efectul transmiterii *prin adresă* precum şi al transmiterii altor funcții ca argumente. Ultima posibilitate se realizează transmițând, ca argument, pointerul la funcție.
- În exemplul de mai sus funcția *tab*, tabelează valorile oricărei funcții reale de un argument întreg, între două limite precizate. Funcția, împreună cu limitele se transmit ca argumente.

Pointeri la funcţii

Programul 2

- Se citesc de pe mediul de intrare date(numele şi vârsta) despre un număr oarecare de persoane şi se doreşte generarea listei persoanelor, ordonată întâi în ordine alfabetică şi apoi lista ordonată în ordinea crescătoare a vârstei.
- Se defineşte tipul structurat persoana:

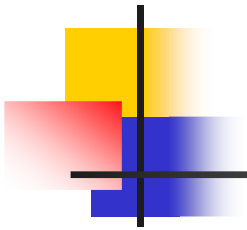
```
typedef struct {
   char *nume;
   int varsta;
 } persoana;
```

- Citirea datelor se face în funcția citeste, declarată astfel:

```
void citeste (int *n, persoana **tab)
```

Pointeri la funcţii

Programul 2



- Funcția citeşte date despre ***n*** persoane şi construieşte un tablou alocat dinamic. Argumentele funcției sunt pointeri pentru a forța transferul lor prin referință (adresă): ***n*** este numărul de persoane şi se citeste în funcție, iar ***tab*** este tabloul în care sunt memorate cele ***n*** persoane.

Tabloul (tabelul) ***tab*** se alocă dinamic în cadrul funcției de citire, deci se modifică însăşi adresa de început a tabloului, de aceea, la apel, argumentul este adresa tabelului.

Pointeri la funcţii
Programul 2

```
void citeste (int *n, persoana **tab) {
    persoana *t;
    char sir[41];
    int i;
     printf("Introduceti numarul de persoane\n");
    scanf("%d", n);
     /* aloca spatiu pentru n structuri de tip persoana */
 if (!(t=(persoana*) malloc ((*n)* sizeof(persoana))))     {
        printf("Eroare alocare dinamica memorie \n");
        exit(1);
    }
```

Pointeri la funcţii

Programul 2

```
    /* atribuie spatiul alocat tabloului de persoane */
    *tab=t;
     /* citeste datele despre fiecare persoana */
    for (i=0; i<*n; i++, t++) {
        printf("\n nume: ");
        scanf("%40s", sir);
        /* aloca memorie pentru nume */
        if (!(t->nume=(char *)malloc (strlen(sir)+1))) {
            printf("Eroare alocare dinamica memorie \n");
          exit(1);
        }
        strcpy(t->nume, sir);
        printf("\n varsta: ");
        scanf("%d", &t->varsta);
    }
}
```

Pointeri la funcţii

Programul 2

- Pentru cele 2 sortări cerute ale listei de persoane, ceea ce diferă este *criteriul de sortare*. O sortare se descompune în 3 tipuri de operații : o comparație care determină ordinea relativă a 2 elemente, o interschimbare a unei perechi de elemente şi un algoritm care efectuează aceste operații până când elementele sunt în ordinea dorită.
- Algoritmul de sortare este independent de operațiile de comparare şi interschimbare. Utilizând funcții diferite de comparare şi interschimbare, se pot realiza sortări dupa diverse criterii (numeric, alfabetic).

- Pentru compararea a 2 persoane se vor utiliza funcții separate ce primesc ca şi parametri 2 persoane *p1* şi *p2*, şi returnează o valoare întreagă pozitivă dacă *p1* > *p2*, o valoare întreagă negativă dacă *p1*< *p2*, respectiv zero dacă între cele 2 persoane nu se poate face o ordonare pe baza criteriului de comparație dat.

Pointeri la funcţii

Programul 2

```
typedef  int (*fct_cmp) (persoana , persoana );
 void sortare (persoana *sir, int n, fct_cmp f) {
    int i,j;
    persoana temp;
    for (i=0; i<n-1; i++)
       for (j=i+1; j<n; j++)
          if ((*f)(sir[ i ], sir[ j ])>0) {
             temp=sir[ i ];
             sir[ i ]=sir[ j ];
             sir[ j ]=temp;
          }
}
int fc1(persoana p1, persoana p2) {
   return strcmp(p1.nume, p2.nume);
}
```

Pointeri la funcţii

Programul 2

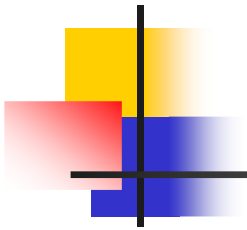
```
int fc2(persoana p1, persoana p2) {
  return p1.varsta-p2.varsta;
}
 void main(void) {
    int n;
    persoana *tabel=NULL;
    citeste(&n, &tabel);
    afiseaza(n, tabel);
    sortare(tabel, n, fc1);
    printf("\n Lista in ordine alfabetica:");
    afiseaza(n, tabel);
    sortare(tabel, n, fc2);
    printf("\n Lista in ordinea varstei:");
    afiseaza(n, tabel);
}
```

Pointeri la funcţii

Programul 2

Se observă că s-a transmis criteriul de comparare ca argument de tip funcție către algoritmul de sortare. În primul rând se declară fct_cmp ca pointer cu tipul ”funcție cu rezultatul de tip întreg şi 2 parametri de tip persoana”.

Funcția sortare are un argument, f, de acest tip fct_cmp. Funcția f este apelată în interiorul corpului funcției sortare, în forma *(*f)(sir[i], sir[j]).* La apelul funcției sortare, locul acestui parametru formal f este luat de funcțiile argumente fc1 şi respectiv fc2.



Preprocesorul

Preprocesarea este o fază care precede compilarea.

Preprocesorul limbajului C este relativ simplu şi în principiu execută substituții de texte. Prin intermediul lui se realizează:

- Includeri de fișiere sursă (antet);
- Macrodefiniții;
- Compilare condiționată.

Directivele de preprocesare au caracterul # la început de linie.

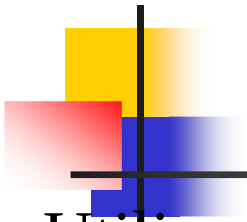
Preprocesorul

Includerea de fişiere antet

- O directivă ***#include*** este înlocuită în timpul preprocesării cu conținutul integral al fişierului specificat, care se inserează în locul ei. Forma directivei include este

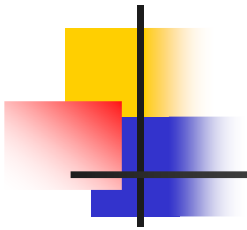
```
#include "nume_de_fisier"     sau
#include <nume_de_fisier>
```

- Diferența între cele două moduri de scriere este următoarea: dacă numele fişierului de inclus este dat între ghilimele, preprocesorul caută fişierul pornind de la directorul curent. Dacă fişierul nu e găsit, sau dacă numele este dat între paranteze unghiulare, căutarea fişierului se face în continuare după reguli definite de implementarea compilatorului de C. Un fişier inclus poate conține la rândul său alte directive #include.



Preprocesorul
Includerea de fişiere antet

- Utilizarea directivei #include e o modalitate de a utiliza aceleaşi declaraţii în cazul programelor care sunt împărţite pe mai multe fişiere.
- Dacă se modifică un fişier inclus, trebuie recompilate toate fişierele care îl includ pe acesta, direct sau indirect.
- Un fişier *antet* poate conține: definiții de tipuri, declarații de funcții, declarații de variabile, definiții de constante, macrodefiniții, alte directive de includere.
- Un fişier antet bine proiectat **nu trebuie să conțină definiții de funcții şi definiții de date!** În principiu un fişier antet poate fi inclus de către mai multe module diferite ale aceleiaşi aplicații. Dacă fişierul antet conține o definiție de funcție sau o definiție de variabilă globală, prin includerea multiplă se realizează definirea multiplă a acestor entități, fapt care constituie o eroare.



Preprocesorul
Macrodefiniţii

- Definirea constantelor simbolice prin directiva ***#define*** este un caz particular de *macrodefiniție*.

- În general, o *macrodefiniție* (un macro) are la bază tot operația de substituție. Un *macro* are o definiție şi poate fi apelat de un număr oarecare de ori. La definirea unui *macro* se specifică de fapt textul care urmează a se substitui la fiecare apel al său. Acest text poate fi variabil, în funcție de anumiți parametri.

Preprocesorul
Macrodefiniţii

- Forma generală a unei macrodefiniții este:

  #define nume(p1, p2, ...,pn) text_de_substituit

  nume = numele macroului

  p1, p2,... , pn = parametrii macroului

  text_de_substituit = textul cu care este înlocuit macroul la fiecare apel

- O definiție poate folosi definiții anterioare.
- *Textul de înlocuit* este restul liniei sursă; dacă se doreşte continuarea definiției pe mai multe linii, la sfârşitul liniei care urmează să fie continuată, se scrie caracterul /.
- În cazul unei macrodefiniții cu parametri, între numele macroului şi paranteza deschisă nu trebuie să existe spații .

Preprocesorul
Macrodefiniţii

- **Exemplu:** Un macro pentru calculul maximului a două numere.

```
#define MAX(x,y)  ((x)>(y) ? (x) : (y))
```

Un exemplu simplu de apel al acestui macro este:

```
float a,b,c;
c=MAX(a,b);
```

La preprocesare, linia `c=MAX(a,b);` se înlocuieste cu

```
c=((a) > (b) ? (a) : (b));
```

Un alt exemplu de apel al macro-ului MAX:

```
int i,j,k;
k=MAX(i + j, i - j);
```

La preprocesare, linia `k=MAX(i + j, i - j);` va fi înlocuită cu

```
k=(i + j) > (i - j) ? (i + j) : (i - j))
```

Preprocesorul
Macrodefiniţii

- Există şi anumite **pericole** legate de utilizarea necorespunzătoare a macrodefinițiilor, care pot avea efecte secundare ascunse.

Exemple:

- MAX(i++, j++) va incrementa de două ori valoarea variabilei care e mai mare, pentru că expandarea macro-ului se face în felul următor:

(i++) > (j++) ? (i++) : (j++)

- Macroul de ridicare a unui număr la pătrat

#define PATRAT(x) x*x dă rezultate eronate în cazul în care este apelat de exemplu pentru x+1: PATRAT(x+1) este expandat în x+1*x+1, ceea ce, având în vedere precedența operatorilor, nu calculează pătratul expresiei (x+1). Pentru a fi corect un macro de ridicare la pătrat ar trebui scris utilizând paranteze:

#define PATRAT(x) (x)*(x)

Preprocesorul
Macrodefiniţii. Exemple

- Transformarea unui caracter din literă mică în literă mare:

```
#define UPPER(c)  ((c)-'a'+'A')
```

- Definirea unui ciclu infinit:

```
#define FOREVER  for(;;)
```

- Interschimbarea a două numere întregi:

```
#define SCHIMBA(X,Y) { int  t;  t=X;  X=Y;  Y=t; }
```

Apelul `SCHIMBA(a,b)` este substituit cu secvența:

```
{ int t;  t=a; a=b; b=t; }
```

Variabila `t` există numai în interiorul instrucțiunii compuse generată de preprocesor!

Preprocesorul
Macrodefiniţii. Exemple

- Macrodefinițiile permit **parametrizarea unei operații cu un nume de tip**, ceea ce se poate folosi ca o facilitate de realizare a unor funcții generice primitive.
- Macro-ul de interschimbare a două elemente poate fi rescris astfel încât să fie parametrizat şi cu tipul elementelor:

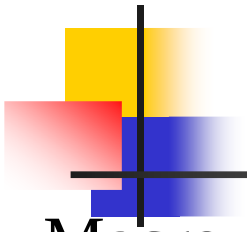
```
#define SWAP(TIP, X, Y) { TIP  t; t=X;  X=Y; Y=t;  }
```

Macrodefiniția SWAP se poate apela cu parametri de orice tip pe care este definită operația =.

```
int a, b;
SWAP(int, a, b);
```

```
float x, y;
SWAP(float, x, y);
```

Preprocesorul
Macrodefiniţii. Exemple



- Macro pentru alocarea dinamică a unui bloc de memorie de *n* elemente de un anumit tip:

```
#define ALOCA(tip, n)  (tip *)malloc(sizeof(tip)*n)
 void main(void) {
    int *ip;
    float *fp;
    ip=ALOCA(int, 10);
    fp=ALOCA(float, 10);  }
```

- Macroexpandarea poate fi oricând suspendată cu directiva:

```
#undef nume
```

Numele respectiv nu mai este expandat în continuarea fişierului.

- Definirea unui macro nu se încheie, de obicei, cu “**;**”.

Preprocesorul
Compilare condiţionată

- Permite să se aleagă dintr-un text general părțile care se compilează împreună.
- Se realizează folosind construcțiile **#if, #ifdef şi #ifndef**

Se notează cu expr o expresie constantă (valoarea poate fi evaluată de preprocesor la întâlnirea ei).

```
#if expr
    text
#endif
```

Dacă expr are valoarea adevărat atunci text se supune preprocesării, altfel se continuă cu ceea ce urmează după #endif.

Preprocesorul
Compilare condiţionată

- #if poate avea şi ramură de else:

```
#if expr
  text1
#else
  text2
#endif
```

- Directivele de preprocesare (compilare condiționată) şi textul care urmează să fie inclus sau nu în programul pentru compilare se dau pe linii de text separate.

Preprocesorul
Compilare condiţionată

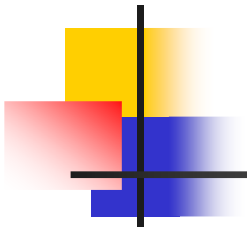
- În interiorul unei **directive #if**, expresia defined(nume) are valoarea **1** dacă nume a fost deja definit de o directivă define, sau **0** în caz contrar.

- Exemplu: includerea o singură dată a conținutului unui fişier antet antet1.h:

```
#if !defined(ANTET1)
   #define ANTET1
   /* continutul fisierului antet1.h */
#endif
```

Preprocesorul
Compilare condiţionată

- Prima includere a fişierului antet1.h defineşte numele ANTET1. În cazul în care ar mai exista includeri ulterioare, chiar şi indirect, prin alte fişiere, preprocesorul vede că numele e deja definit şi sare direct la #endif. Această facilitate este utilă în situații în care un fişier antet ar fi altfel inclus de mai multe ori, direct sau indirect, ca în următoarea situație de exemplu: Se consideră un fişier antet a.h şi un fişier antet b.h care conține o directivă de includere a lui a.h. Dacă un alt fişier main.c conține directive de includere a ambelor fişiere a.h şi b.h, rezultă că fişierul a.h va fi inclus de două ori. Această includere dublă poate conduce la erori datorită redefinirii unor variabile şi constante conținute în a.h.



Preprocesorul
Compilare condiţionată

- Directivele **#ifdef** şi **#ifndef** testează dacă un nume este sau nu este definit.

- Reluarea exemplului anterior:

```
#ifndef ANTET1

   #define ANTET1
   /* continutul fisierului antet1.h */
#endif
```

Preprocesorul
Compilare condiţionată

- **Includerea de fişiere diferite** în funcție de anumite variabile sistem:

```
#if  SYSTEM==SYSV
   #define ANTET "sysv.h"
 #elif  SYSTEM==BSD
   #define ANTET "bsd.h"
 #elif  SYSTEM==MSDOS
   #define ANTET "msdos.h"
 #else
   #define ANTET "default.h"
 #endif
 #include ANTET
```

Preprocesorul
Compilare condiţionată

- *Exemplu* - Definirea de tipuri mutual recursive: se consideră două tipuri structurate T1 şi T2, fiecare dintre acestea conține un câmp de tip pointer la celălalt tip.

```
typedef struct {
    T2 a;
    int b;} *T1;

typedef struct {
  T1 a;
  int b;} *T2;
```

- Se pune problema ordinii corecte în care trebuie să fie definite aceste tipuri. Dacă T1 este definit înaintea lui T2, compilatorul nu îl cunoaşte pe T2 când întâlneşte definiția lui T1, care are un câmp (a) de acest tip.

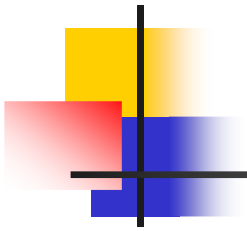
Preprocesorul
Compilare condiţionată

- Problema există şi invers: dacă T2 ar fi definit înaintea lui T1. Pentru a rezolva problema referirilor reciproce, se utilizează declarații incomplete de structuri, astfel:

```
typedef struct t_T1 * T1;
typedef struct t_T2 * T2;

struct t_T1 {
  T2 a;
  int b;
  };

struct t_T2 {
  T1 a;
  int b;
  };
```



Preprocesorul
Compilare condiţionată

- Cazul: T1 şi T2, sunt definite în fişiere separate t1.h şi t2.h:

```
/* fisierul t1.h */

#if !defined(tip_T1)
   #define tip_T1
   typedef struct t_T1 *T1;
   #include "t2.h"
   struct t_T1 {
     T2 a;
     int b;
     };
#endif
```

Preprocesorul

Compilare condiţionată

```
 /* fisierul t2.h */
#if !defined(tip_T2)
   #define tip_T2
   typedef struct t_T2 *T2;
   #include "t1.h"
   struct t_T2 {
     T1 a;
     int b;
     };
#endif
 /* fisierul main.c */
 #include "t1.h"
 #include "t2.h"
 void main() {
  ----------          }
```

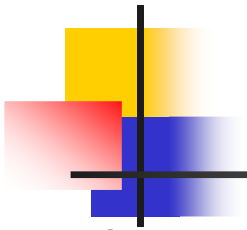
Funcţii cu număr variabil de argumente

- Este posibilă definirea unor funcții care să se apeleze cu un număr variabil de argumente. Funcțiile de bibliotecă *printf* şi *scanf* sunt din această categorie:

```
int scanf(char *format, ...);
int printf(char *format, ...);
```

În acest caz, la definirea funcției, lista de parametri conține doar enumerarea primilor parametri, cei care sunt ficşi, iar ceilalți parametri sunt specificați prin puncte de suspensie: **...** . Funcțiile de acest gen folosesc şi interpretează de obicei informația transmisă prin argumentele obligatorii pentru a şti câte argumente există efectiv în apel şi care sunt tipurile acestora, în maniera în care *printf* şi *scanf* interpretează *formatul*.

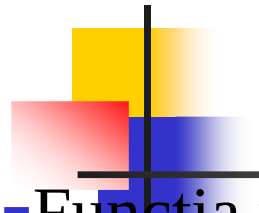
Funcţii cu număr variabil de argumente



- Referirea valorilor argumentelor variabile nu se poate face în mod direct, deoarece aceste argumente nu au nişte nume corespondente în lista parametrilor. Problema numărului variabil de argumente şi a corespondenței tipurilor acestora cu tipurile parametrilor, trebuie să fie tratată de către programator; compilatorul nu dispune de nici o informație pentru a verifica apelurile. Există un set de funcții (de fapt macrodefiniții) declarate în *stdarg.h* relative la listele cu un număr variabil de argumente. În acest sens avem tipul ***va_list*** şi funcțiile ***va_start***, ***va_arg*** şi ***va_end***.

- Tipul ***va_list*** descrie un pointer către lista de argumente. În funcția definită de programator se va declara o variabilă locală de acest tip, necesară pentru adresarea argumentelor.

Funcţii cu număr variabil de argumente



- Funcția ***va_start*** primeşte ca argumente variabila de tip va_list şi numele ultimului parametru fix al funcției. Ea realizează inițializarea variabilei de tip va_list cu adresa primului argument variabil.

```
va_start(va_list ap, ultim_fix);
```

- Funcția ***va_arg*** întoarce valoarea argumentului următor, dintre cei variabili:

```
tip_par va_arg(va_list ap, tip_par);
```

La primul apel, va_arg returnează primul argument dintre cei variabili, apoi la fiecare nou apel returnează următorul argument. Tipul argumentului se specifică prin tip_par. Se aplică conversii implicite pentru transferul valorilor din partea variabilă, valorile transferate fiind întotdeauna extinse la tipurile int şi double.

- Funcția ***va_end*** realizează operațiile necesare încheierii funcției şi trebuie apelată înainte de revenirea din funcție.

Funcţii cu număr variabil de argumente - Exemplu

- Să se scrie o funcție care poate primi un număr oarecare de argumente de tipuri diferite, ale căror valori le afişează. Primul parametru al funcției este un parametru fix, de tip şir de caractere, cu rolul de a descrie tipurile parametrilor care urmează, aprox. În maniera *formatului* din printf.

Fiecare caracter din acest şir specifică tipul unui parametru. Caracterele şi tipurile permise sunt:

d = întreg, r = real, c = caracter, s= şir.

De exemplu, dacă şirul este *”ddrcd”*, înseamna că funcția va avea 5 parametri, dintre care primii doi de tip întreg, următorul real, următorul caracter şi ultimul întreg.

Funcția func din secvența următoare realizează acest lucru:

Funcţii cu număr variabil de argumente - Exemplu

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


  void func(char *msg, ...) {
     char c;
     int va_d;
     char va_c;
     double va_r;
     char * va_s;
      int count=0;
      va_list ap;
     va_start(ap, msg);
```

Funcţii cu număr variabil de argumente - Exemplu

```
while((c=*msg++)!='\0')
      switch(c) {
        case 'd': va_d=va_arg(ap, int);
                  count++;
                  printf("Parametrul nr %d este de tip intreg"
                         " si are valoarea %d \n", count, va_d);
                   break;
        case 'r': va_r=va_arg(ap, double);
                  count++;
                  printf("Parametrul nr %d este de tip real"
                         " si are valoarea %lf \n", count, va_r);
                  break;
```

Funcţii cu număr variabil de argumente - Exemplu

```
    case 'c': va_c=va_arg(ap, char);
                count++;
                printf("Parametrul nr %d este de tip caracter"
                      " si are valoarea %c \n", count, va_c);
                break;
      case 's': va_s=va_arg(ap, char *);
                count++;
                printf("Parametrul nr %d este sir de caractere"
                      " si are valoarea %s \n", count, va_s);
                break;
      default:
                printf("Tipul specificat este eronat !\n");
                break;
      }
    va_end(ap);
}
```

Funcţii cu număr variabil de argumente - Exemplu

Funcția *func* definită anterior poate fi apelată ca în secvența următoare:

```
int main(void) {
    int n=7;
    double x=5.5;
    func("ddrd", 5, n, x, 5+1);  /* corect */
    func("cds", 'x',34,"abc");   /* corect */
    func("cds","abc",5.5, 33);   /* rezultate eronate ! */
   return 0;   }
```

- Primele două forme de apel sunt corecte.
- Al treilea apel dă rezultate eronate, pentru că argumentele **nu** sunt de tipurile aşteptate.