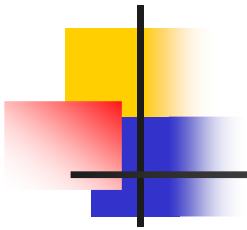
Funcţii definite în program

1. Bazele definirii şi utilizării funcțiilor
2. Funcții ce returnează altfel de valori decât întregi
3. Variabile externe
4. Reguli pentru domeniile de vizibilitate
5. Variabile statice
6. Variabile registru
7. Structura de bloc
8. Inițializarea variabilelor



Funcţii definite în program

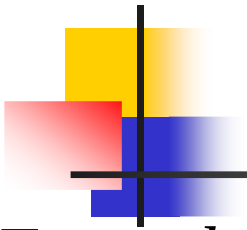
Bazele definirii şi utilizării funcţiilor

- Există două mari *avantaje* ale utilizării funcțiilor definite în program:

1) împărțirea unor sarcini de calcul ample în altele mai mici, ceea ce conduce la scrierea mai simplă a programului în ansamblu;

2) reutilizarea unui cod deja scris sub formă de funcții pentru programe anterioare (ex. funcțiile de bibliotecă).

- **Limbajul C a fost special conceput să permită utilizarea simplă şi eficientă a funcțiilor; programele C sunt, în general, formate din multe funcții mici şi nu din câteva funcții mari.**



Funcţii definite în program

Bazele definirii şi utilizării funcţiilor

Exemplu: Să se scrie un program care tipăreşte liniile din fişierul de intrare ce conţin un anumit "şablon" (şir de caractere prestabilit).

- Algoritmul problemei se împarte în trei părți:

```
while (mai exista o linie)
    if (linia contine sablonul)
        tipareste-o
```

- Deşi ar fi posibil să plasăm întregul cod în funcția main, o soluție mai bună este să facem din fiecare parte a algoritmului o funcție separată (funcțiile preialinie, indicesir şi printf – funcție de bibliotecă).

Funcţii definite în program

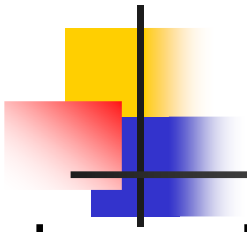
Bazele definirii şi utilizării funcţiilor

- Funcția indicesir(s, t) returnează *poziția* sau indicele din şirul s unde începe şirul t sau -1 dacă s nu îl conține pe t. Deoarece în C tablourile încep de la poziția 0, indicii vor fi *zero* sau strict pozitivi şi deci o valoare negativă precum -1 este convenabilă pentru semnalarea eşecului.
- În program va apare şi funcția preialinie care citeşte, linie cu linie, textul furnizat în fişierul de intrare.

```
#include <stdio.h>
#define MAXLINIE 1000 /* lung. max. a liniei de intrare */
int preialinie(char linie[ ], int max);
int indicesir(char sursa[ ], char decautat[ ]);
```

Funcţii definite în program

Bazele definirii şi utilizării funcţiilor



```
char sablon[ ] = “ea”;   /* sablonul cautat */
/* gaseste toate liniile in care apare sablonul */
int main()  {
   char linie[MAXLINIE];
   int gasite = 0;
   while (preialinie(linie, MAXLINIE) > 0)
      if (indicesir(linie, sablon) >= 0)  {
         printf(“%s”, linie);
         gasite++;   }
   return gasite;
}
```

Funcţii definite în program

Bazele definirii şi utilizării funcţiilor

```
/* preialinie: preia linia, o copiaza in s, returneaza lung.*/
int preialinie(char s[ ], int lim)  {
   int c, i;
   i = 0;
   while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
       s[ i++ ] = c;
   if (c == '\n')
       s[ i++ ] = c;
   s[ i ] = '\0';
   return i;
}
```

Funcţii definite în program

Bazele definirii şi utilizării funcţiilor

```
/* indicesir: returneaza indicele lui t din s sau -1 daca t
        nu exista in s*/
int indicesir(char s[ ], char t[ ])  {
   int i, j, k;
   for (i = 0; s[ i ] != '\0'; i++)  {
      for (j=i, k=0; t[ k ]!='\0' && s[ j ]==t[ k ]; j++, k++) ;
      if (k > 0 && t[ k ] == '\0')
         return i;
   }
   return -1;
}
```

Funcţii definite în program
Bazele definirii şi utilizării funcţiilor

- Fiecare **definiție de funcție** are forma

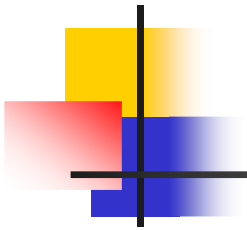
tip-rezultat nume-functie(declaratii de parametri)
{
declaratii si instructiuni
}

- Anumite părți pot lipsi; o funcție minimală este:

nimic() { }

care nu face nimic şi nu returnează nimic. Este uneori utilă ca variantă provizorie pe parcursul dezvoltării programului.

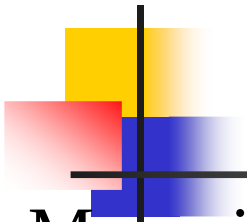
- Dacă tipul returnat este omis se presupune că este int.



Funcţii definite în program

Bazele definirii şi utilizării funcţiilor

- Un program este o colecție de definiții de variabile şi de funcții.
- *Comunicarea între funcții* se efectuează prin:
  - parametri (argumente);
  - valori returnate;
  - variabile externe.
- Funcțiile pot apărea în orice ordine în fişierul sursă, iar programul sursă poate fi împărțit în mai multe fişiere, atât timp cât nici o funcție nu este divizată.



Funcţii definite în program

Bazele definirii şi utilizării funcţiilor

- Mecanismul pentru *returnarea unei valori* din funcția apelată către apelanta sa (rezultatul funcției) este instrucțiunea return .
- După return poate urma orice expresie :

  return *expresie*;

- Dacă este necesar, e*xpresie* va fi convertită la tipul returnat de funcție.
- Pentru claritate, *expresie* poate fi inclusă, opțional, între paranteze.
- Dacă nu are nevoie de ea, funcția apelantă poate să ignore valoarea returnată.
- Dacă instrucțiunea return lipseşte sau dacă este prezentă dar nu este urmată de e*xpresie*, nu se returnează nici o valoare apelantei.

Funcţii definite în program

Bazele definirii şi utilizării funcţiilor

- Dacă se ajunge la acolada închisă } care indică terminarea funcției, controlul revine de asemenea la apelantă fără să transmită nici o valoare. Nu este ilegal, dar poate fi un avertisment pentru o posibilă greşeală, dacă o funcție returnează o valoare într-un caz şi în alt caz nu returnează valoare.

- Pentru a fi compilată în mod unitar, o funcție trebuie redactată în întregime în acelaşi fişier sursă. Procedeul pentru compilarea şi încărcarea unui program C care se află în *mai multe fişiere sursă* variază de la un sistem la altul. El va fi prezentat într-un capitol ulterior.

- *Exemplu*: Să se determine maximul dintre trei numere întregi, utilizând o funcție care determină maximul dintre două numere întregi.

Funcţii definite în program

Bazele definirii şi utilizării funcţiilor

```
#include <stdio.h>
int max (int a, int b)  {
   if (a > b)
      return a;
   else
      return b;   }
main( )  {
   int x, y, z;
   printf("Introduceti 3 numere intregi\n");
   scanf("%d%d%d", &x, &y, &z);
   printf("Maximul este %d\n", max( x, max( y, z)));
}
```

Funcţii definite în program

Funcţii ce returnează altfel de valori decât întregi

- Exemplele de funcții de până acum, fie nu au returnat nici o valoare (void), fie au returnat un int.
- Ce se întâmplă dacă o funcție trebuie să returneze un rezultat de alt tip?

1. *Funcția trebuie, obligatoriu, să declare tipul valorii pe care o returnează*, din moment ce aceasta nu este int. Numele tipului precede numele funcției.

2. *În rutina apelantă sau înainte de codul apelantului se declară explicit funcția apelată* întrucât rutina apelantă trebuie să ştie că funcția apelată returnează o valoare ce nu aparține tipului int.

Funcţii definite în program

Funcţii ce returnează altfel de valori decât întregi

- *Exemplul 1*: Să se scrie o funcție putere care calculează baza^{exponent}, unde exponent este un număr natural.

```
#include <stdio.h>
float putere (float baza, int exponent)
{
   float p;
   int i;
   if (exponent == 0) return 1.0;
   for (i = 1, p = 1.0; i <= exponent; i++)
      p *= baza;
   return p;
}
```

Funcţii definite în program

Funcţii ce returnează altfel de valori decât întregi

```
main()  {
   int e;
   float b, p;
   printf("Introduceti baza si exponentul \n");
   scanf("%f%d", &b, &e);
   p = putere(b, e);
   printf("%f la puterea %d = %f\n", b, e, p);
   printf("5 la puterea 3 = %f\n", putere(5, 3));
   printf("(2 la 3) la 2 = %f\n", putere(putere(2, 3), 2));
}
```

Funcţii definite în program

Funcţii ce returnează altfel de valori decât întregi

- *Exemplul 2*: Scrierea şi utilizarea funcției atof care converteşte şirul de caractere s în numărul echivalent în reprezentare virgulă mobilă în dublă precizie.

- Biblioteca standard include o funcție atof; este declarată în fişierul antet <stdlib.h>.

Funcţii definite în program

Funcţii ce returnează altfel de valori decât întregi

```
#include <ctype.h>
/* atof: converteste sirul de caractere s la tipul double */
double atof(char s[ ])
{
   double val, putere;
   int i, semn;
   for (i=0; isspace(s[i]); i++) /* treci peste spatiile albe */
      ;
   semn = (s[i] == '-') ? -1 : 1;
   if (s[i] == '+'|| s[i] == '-')
      i++;
```

Funcţii definite în program

Funcţii ce returnează altfel de valori decât întregi

```
  for (val = 0.0; isdigit(s[i]); i++)
     val = 10.0 * val + (s[i] –’0’);
  if (s[i] == ‘.’)
     i++;
for (putere = 1.0; isdigit(s[i]); i++) {
     val = 10.0 * val + (s[i] –’0’);
     putere *= 10.0;
  }
  return semn * val / putere;
}
```

Funcţii definite în program

Funcţii ce returnează altfel de valori decât întregi

- *Rutina apelantă* este un calculator rudimentar care citeşte numerele câte unul pe linie, precedate opțional de un semn, le adună şi tipăreşte suma curentă după fiecare număr introdus.

```
#include <stdio.h>
#define MAXLINIE 100
/* calculator rudimentar */
main()
{
   double suma, atof(char [ ]);
   char linie[MAXLINIE];
```

Funcţii definite în program

Funcţii ce returnează altfel de valori decât întregi

```
int preialinie(char linie[ ], int max);
  suma = 0.0;
  while (preialinie(linie, MAXLINIE) > 0)
      printf("\t%g\n", suma += atof(linie));
  return 0;
}
```

- Declarația

```
double suma, atof(char [ ]);
```

precizează că suma este o variabilă de tip double şi că atof este o funcție care primeşte un argument de tip char [ ] şi returnează un double.

Funcţii definite în program

Funcţii ce returnează altfel de valori decât întregi

- Funcția atof trebuie *declarată şi definită în mod concordant*.
- Dacă funcția atof şi apelul său din main au, în acelaşi fişier, tipuri care nu concordă, eroarea va fi detectată de compilator.
- Dacă funcția atof ar fi compilată separat (ceea ce este mai probabil) neconcordanța nu ar fi detectată. Funcția atof ar returna un double pe care funcția main l-ar trata, de exemplu, ca pe un int şi ar apărea rezultate lipsite de sens.
- Dacă o funcție nu are prototip, aceasta este *declarată implicit* de prima sa apariție într-o expresie, cum ar fi:

```
suma += atof(linie)
```

Funcţii definite în program

Funcţii ce returnează altfel de valori decât întregi

- Dacă un nume care nu fost declarat în prealabil apare într-o expresie, urmat de o paranteză deschisă, acesta este declarat prin context ca fiind nume de funcție, despre funcție se presupune că este de tip int, iar despre argumentele acesteia nu se presupune nimic.
- Dacă o declarație de funcție nu include argumente, ca de exemplu:

```
double atof();
```

acest lucru este de asemenea interpretat ca însemnând că nu se poate deduce nimic despre argumentele funcției atof şi, în consecință, nu se efectuează nici o verificare asupra parametrilor.

Funcţii definite în program

Funcţii ce returnează altfel de valori decât întregi

- Se recomandă ca, într-o declarație de funcție, dacă funcția acceptă argumente, să declarăm parametrii corespunzători sau să folosim tipul void dacă funcția nu acceptă argumente.

```
/* atoi: converteste sirul s la un intreg folosind atof */
int atoi(char s[ ])
{
   double atof(char s[ ]);
   return (int) atof(s);
}
```

- Valoarea returnată este convertită la tipul funcției înainte de a se face returnarea. Operatorul *cast* precizează că operația este intenționată şi elimină orice avertisment din partea compilatorului.

Funcţii definite în program
Variabile externe

- Un program C este format dintr-o colecție de obiecte externe care sunt variabile sau funcții. Adjectivul "*extern*" este folosit ca antonim pentru "intern" care descrie argumentele şi variabilele definite în interiorul funcțiilor.
- Variabilele externe sunt *definite în afara oricărei funcții* şi sunt astfel *disponibile mai multor funcții*.
- Funcțiile în sine sunt întotdeauna externe, deoarece *limbajul C nu permite definirea funcțiilor în interiorul altor funcții*.
- Variabilele şi funcțiile externe au proprietatea că toate trimiterile către ele prin acelaşi nume, chiar şi din funcții compilate separat, constituie trimiteri către acelaşi obiect (standardul numeşte această proprietatea "*legare externă*").

Funcţii definite în program

Variabile externe

- Se pot defini variabile şi funcţii externe care să fie vizibile doar dintr-un singur fişier sursă.
- Deoarece variabilele externe sunt *global accesibile* ele pot constitui o alternativă la argumentele funcţiilor şi la valorile returnate de acestea pentru *transmiterea datelor între funcţii*.
- Orice funcţie poate accesa o variabilă externă făcând trimitere la aceasta prin nume, dacă numele a fost declarat în vreun fel.
- Dacă mai multe funcţii urmează să folosească în comun un număr mare de variabile, variabilele externe sunt mai convenabile pentru a reprezenta aceste variabile, decât listele de argumente lungi. Acest lucru trebuie folosit cu precauţie pentru că poate avea efect dăunător asupra structurii programului fiind prea multe conexiuni de date între funcţii.

Funcţii definite în program

Variabile externe

- Variabilele externe sunt utile şi datorită *domeniului de vizibilitate şi duratei de existență mai mari*.
- Variabilele externe au *caracter permanent*. Ele păstrează valorile de la o invocare de funcție la următoarea, spre deosebire de variabilele *automatice* care sunt interne unei funcții (sunt create când se intră în funcție şi dispar când se iese din aceasta).
- Astfel, *dacă două funcții trebuie să folosească în comun anumite date, dar nici una dintre ele nu o apelează pe cealaltă, este adesea convenabil ca datele comune să fie păstrate în variabile externe* în loc să fie transmise şi preluate prin intermediul argumentelor.

Funcţii definite în program

Reguli pentru domeniile de vizibilitate

- **Domeniul de vizibilitate** al unui nume este partea programului în care acesta poate fi folosit (este vizibil).
- **Durata de viață (existență)** a unei variabile este timpul scurs din momentul alocării ei în memorie şi până la desființarea locației respective.
- Pentru *o variabilă automatică* declarată la începutul unei funcții, domeniul de vizibilitate este limitat la *funcția în care este declarat numele*. Variabilele locale cu acelaşi nume din diferite funcții nu au legătură unele cu altele. Acelaşi lucru este valabil şi pentru parametrii funcției, care sunt de fapt variabile locale.
- Domeniul de vizibilitate al unei *variabile sau funcții externe începe din locul în care aceasta este declarată şi ține până la sfârşitul fişierului în curs de compilare*.

Funcţii definite în program

Reguli pentru domeniile de vizibilitate

```
main()  { … }

int ps = 0;
double val[MAXVAL];

void push(double f)  { … }

double pop(void) { … }
```

- Variabilele ps şi val pot fi folosite în funcțiile push şi pop prin numirea lor, fără să mai fie nevoie de alte declarații. Ele nu sunt vizibile în main; la fel, nu sunt vizibile în main nici funcțiile push sau pop.

Funcţii definite în program

Reguli pentru domeniile de vizibilitate

- Dacă trebuie să se facă referire la o variabilă externă înainte ca aceasta să fie definită, sau dacă este definită într-un fişier sursă diferit de cel în care este folosită, atunci este obligatorie o declaraţie extern.
- Care este diferenţa între *declaraţia* unei variabile şi *definiţia* sa?
- O **declaraţie** anunţă *proprietăţile* unei variabile (în primul rând tipul său).
- O **definiţie** anunţă *proprietăţile* unei variabile şi produce *alocarea de spaţiu* de stocare pentru valoarea variabilei.

Funcţii definite în program

Reguli pentru domeniile de vizibilitate

- *Exemple*:

1. Liniile

```
int ps = 0;
double val[MAXVAL]
```

din exemplul anterior, apar în afara oricărei funcții: sunt *definiții* de variabile externe (ps şi val).

2. În schimb, liniile

```
extern int ps;
extern double val[ ];
```

declară pentru restul fişierului sursă că ps este un int şi că val este un tablou de tip double, dar nu crează variabilele şi nu alocă spațiu de stocare pentru acestea.

Funcţii definite în program

Reguli pentru domeniile de vizibilitate

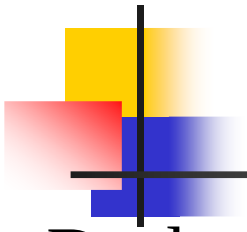
- Trebuie să existe o *singură definiție* a unei variabile externe, doar într-unul din fişierele care compun programul sursă. Pentru a accesa o astfel de variabilă în alte fişiere, aceste fişiere trebuie să conțină declarații extern corespunzătoare.

- Dimensiunile tablourilor trebuie specificate la definire. Ele sunt opționale în cazul declarațiilor extern.

- *Inițializarea* unei variabile externe se poate face *doar la definire*, adică acolo unde ea nu este declarată extern.

Funcţii definite în program

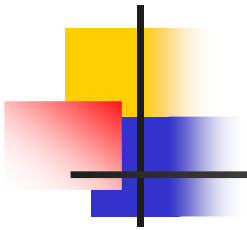
Variabile statice



- Declarația (clasa de memorare) static aplicată unui *obiect extern* (variabilă sau funcție), limitează domeniul de vizibilitate al acelui obiect la restul fişierului în curs de compilare.
- Declarația static face ca o variabilă externă să nu poată fi accesată de funcții din afara fişierului în curs de compilare.
- Declarația static externă este utilizată cel mai adesea pentru variabile, dar poate fi aplicată şi funcțiilor.
- De obicei, numele de funcții sunt globale (sunt vizibile pentru orice parte a programului). Dacă o funcție este declarată ca fiind static, numele acesteia nu este vizibil în afara fişierului în care este declarată.

Funcţii definite în program

Variabile statice



- Declaraţia static poate fi aplicată, de asemenea, şi *variabilelor interne*. Din punct de vedere al vizibilităţii, variabilele interne de tip static sunt locale unei anumite funcţii, la fel ca şi variabilele *automatice*, dar ele există în permanenţă. Nu sunt create şi nici nu sunt distruse de fiecare dată când este activată funcţia.

- Variabilele interne de tip static au spaţiu de stocare permanent în cadrul unei funcţii şi, în consecinţă, pot fi folosite pentru *transmiterea de valori de la un apel la altul al aceleiaşi funcţii*.

Funcţii definite în program
Variabile statice-exemple

- *Clasa de memorie* **static** este precizată prefixând declarația normală prin cuvântul static.
- Dacă într-un fişier ce conține funcțiile getch şi ungetch declarăm static variabilele buf şi bufp

```
static char buf[DIMBUF];
static int bufp = 0;
int getch(void)  { … }
int ungetch(int c)  { … }
```

atunci variabilele buf şi bufp nu pot fi accesate de nici o altă funcție din alte fişiere sursă, doar de getch şi ungetch.

Funcţii definite în program

Statutul variabilelor - sinteză

- Variabilele **interne** unei funcții: sunt vizibile doar în interiorul funcției respective. Din punctul de vedere al duratei de viață pot fi:

  - *automatice* - sunt create când se intră în funcție şi dispar când se revine din aceasta;

  - *statice* - există pe toată durata programului; nu sunt create şi nici nu sunt distruse la fiecare activare funcției; în consecință, pot fi folosite pentru transmiterea de valori de la un apel la altul *al aceleiaşi funcții*.

În mod *implicit*, variabilele interne sunt *automatice*. Pentru a deveni *statice*, ele trebuie declarate static.

Funcţii definite în program

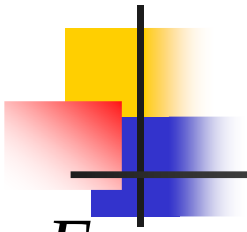
Statutul variabilelor - sinteză

- Variabilele **externe** tuturor funcțiilor: sunt vizibile cel puțin până la sfârşitul fişierului sursă respectiv. Variabilele externe au *caracter permanent*: locațiile lor de memorie există pe toată durata execuției programului => ele pot fi folosite pentru transmiterea de date între apelurile de funcții diferite.
Variabilele externe pot fi declarate:
  - static – domeniul de vizibilitate se limitează la fişierul în curs de compilare;
  - extern – *declară* proprietățile unei variabile *definită* într-un alt loc; unica locație de memorie pentru variabilă va fi în fişierul în care ea este definită (în care nu este extern).
- Declarația static se poate aplica şi funcțiilor, cu acelaşi efect privind vizibilitatea.

Funcţii definite în program
Variabile registru

- O declarație register avertizează compilatorul că variabila în discuție va fi intens folosită.
- Variabilele de tip register se păstrează în registrele maşinii (care se găsesc chiar în interiorul unității centrale de prelucrare) şi fac programul mai performant.
- Compilatorul poate să acceseze o valoare mult mai repede atunci când ea este localizată într-un registru.
- Pentru că numărul de registre este limitat, compilatorul nu poate să asocieze permanent o variabilă unui registru, dar poate să încerce să țină variabila cât mai mult în registru.
- Cuvântul register este ignorat în cazul declarațiilor în exces.

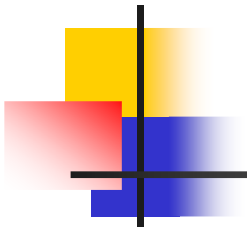
Funcţii definite în program
Variabile registru



- *Exemple*: register int x;
  register char c;
- Declarația register nu poate fi aplicată decât entităților locale: variabile automatice şi parametrii unei funcții.

```
f (register unsigned m, register long n)
{
    register int i;
    ....
}
```

- Restricțiile referitoare la numărul şi tipurile posibile pentru *variabile registru* diferă de la maşină la maşină.



Funcţii definite în program

Structura de bloc

- C-ul nu este un limbaj structurat pe blocuri ca şi Pascal-ul, deoarece funcțiile nu pot fi definite în interiorul altor funcții.

- În schimb, variabilele pot fi definite în maniera unei structuri în blocuri în interiorul unei funcții.

- *Declarațiile de variabile pot fi plasate după acolada deschisă* care introduce *orice* instrucțiune compusă, nu numai după acolada care marchează începutul unei funcții. Domeniul de vizibilitate al variabilei este cuprins între cele două acolade ({ … }).

Funcţii definite în program

Iniţializarea variabilelor

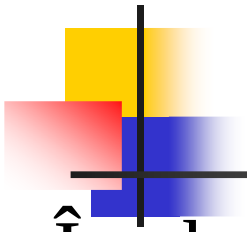
- O variabilă *automatică* declarată şi inițializată într-un bloc este inițializată de fiecare dată când se intră în bloc.

```
if (n > 0)  {
   int i=0;  /* declara şi initializeaza pe i la
                fiecare parcurgere*/
             ...
}
```

- O variabilă declarată static este inițializată numai prima dată când se intră în bloc.

Funcţii definite în program

Iniţializarea variabilelor



- În absența inițializării explicite este garantată inițializarea cu zero doar a variabilelor *externe* şi *statice*.

- Variabilele *automatice* şi *registru* au valori inițiale nedefinite.

- *Pentru variabilele externe şi statice*, valoarea de inițializare explicită trebuie să fie o expresie constantă; *inițializarea se efectuează o singură dată*, în principiu înainte ca programul să înceapă execuția.

Funcţii definite în program

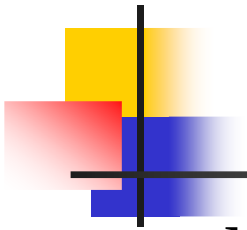
Iniţializarea variabilelor

- *În cazul variabilelor automatice şi registru, iniţializarea se efectuează de fiecare dată când se intră în funcţie sau în bloc*. Valoarea de iniţializare poate fi şi o expresie care să implice valori definite anterior, chiar şi apeluri de funcţii.

```
int cautbin(int x, int v[ ], int n)  {
        int prim = 0;
        int ultim = n – 1;
        int mijl;
        ...
    }
```

Funcţii definite în program

Iniţializarea variabilelor



- Un tablou se poate iniţializa punând după declaraţia sa o listă de valori de iniţializare:

```
int zile[ ] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

- Când este omisă dimensiunea, compilatorul va calcula lungimea numărând valorile de iniţializare (12 în acest caz).
- Dacă sunt mai puţine valori decât dimensiunea specificată, elementele lipsă vor fi zero atât pentru variabilele externe sau statice cât şi pentru cele automatice.
- Situaţia inversă: prea multe valori de iniţializare, constituie o eroare.

Funcţii definite în program

Iniţializarea variabilelor

- Tablourile de caractere se pot iniţializa cu un şir de caractere, astfel:

```
char luna[ ] = “iunie”;
```

în loc de

```
char luna[ ] = { ‘i’, ‘u’, ‘n’, ‘i’, ‘e’, ‘\0’};
```

- Dimensiunea tabloului în acest caz este 6 (5 caractere plus terminatorul ‘\0’).

Funcţii definite în program

Exemple de programe

/*__Calculul valorii unei functii__, intr-un punct

f(x) = |x^2, x <= -1

|x+3, -1 < x <= 1

|arctg(x), x > 1 */

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
float f(float x)  {
   if (x <= -1) return x*x;
   else if (x <= 1) return x+3;
        else return atan(x);   }
```

Funcţii definite în program

Exemple de programe

```
int main(void) {
   float v;
   printf("Calculul valorii unei functii pentru ");
   printf("un argument citit de la tastatura  \n");
   printf("Valoarea argumentului este:");
   scanf("%f",&v);
   printf("Valoarea functiei in punctul %.2f este%.2f \n",
                        v, f(v));
   system(“pause”);
   return 0;
}
```

Funcţii definite în program

Exemple de programe

Algoritmul lui Euclid de aflare a celui mai mare divizor comun a două nr. întregi

Formularea în cuvinte a algoritmului este următoarea:

Dacă unul dintre numere este zero, c.m.m.d.c. al lor este celălalt număr.

Dacă nici unul dintre numere nu este zero, atunci c.m.m.d.c. nu se modifică dacă se înlocuieşte unul dintre numere cu restul împărţirii sale cu celălalt.

Funcţii definite în program

Exemple de programe

```
/*determinarea cmmdc si cmmmc pentru doua numere intregi*/
#include <stdio.h>
int cmmdc(int m, int n) {
    int r;
    r=m % n;
    while (r != 0) {
       m=n;
       n=r;
       r=m % n;
    }
    return n;
}
```

Funcţii definite în program

Exemple de programe

```
int main(void) {
   int n1,n2,divizor;
   printf("Calculul cmmdc si cmmmc cu alg. lui Euclid\n");
   printf("Numerele sunt:\n");
   printf(“n1 = ");  scanf("%d",&n1);
   printf(“n2 = ");  scanf("%d",&n2);
   divizor=cmmdc(n1,n2);
   printf("CMMDC = %d\n",divizor);
   printf("CMMMC = %d\n",(n1*n2)/ divizor);
   return 0;
}
```