

Tehnici de programare

preprocesorul C; definiții și macrouri; aplicații ale macrourilor

Programele C sunt de fapt parcurse în două faze distincte: **preprocesarea** (denumită astfel fiindcă are loc înainte) și **procesarea** propriu-zisă. În faza de preprocesare este folosit un subset al limbajului C care conține instrucțiuni specifice acestei faze. Aceste instrucțiuni mai sunt denumite și **directive de preprocesare** și ele se recunosc ușor, deoarece liniile pe care sunt amplasate încep cu caracterul **#** (diez). Partea din limbajul C care se ocupă de preprocesare se numește **preprocesorul C**. Limbajul preprocesorului și limbajul C din faza de procesare sunt atât de diferite între ele, încât uneori sunt considerate ca fiind două limbaje distincte. Preprocesorul este un limbaj foarte rudimentar, cu doar câteva instrucțiuni. Preprocesorul se ocupă în special cu substituții textuale, el acționând mai mult ca un procesor de text, pentru a transforma codul C propriu-zis într-o formă finală.

Până acum am folosit o singură directivă de preprocesare, **#include**:

```
#include <nume_fisier>
```

#include încarcă fișierul specificat și inserează în cod conținutul fișierului, înlocuind directiva. Este ca și cum am face copy/paste la conținutul fișierului și l-am pune în locul lui **#include**. Semnele **<...>** în jurul fișierului înseamnă că acesta se va căuta prima oară în directoarele standard unde este instalat compilatorul de C (ex: gcc). În loc de **<...>** se pot folosi și ghilimele (**"..."**) și atunci fișierul specificat se va căuta prima oară chiar în directorul fișierului care îl include. Fișierele nu trebuie neapărat să aibă extensia **.h** (header) ci pot fi orice fel de fișiere text.

Exemplul 1: Dacă avem pe disc fișierul **pt.h** cu următorul conținut:

```
// pt.h
typedef struct{
    float x,y;
}Pt;
```

și fișierul **1.c**, care conține:

```
// 1.c
#include "pt.h"

int main()
{
    return 0;
}
```

atunci, după faza de preprocesare a lui **1.c**, fază în care directiva **#include "pt.h"** va fi substituită cu conținutul fișierului **pt.h**, fișierul **1.c** va avea următorul conținut:

```
// 1.c
// pt.h
typedef struct{
    float x,y;
}Pt;
```

```
int main()
{
    return 0;
}
```

În acest fel putem să organizăm codul proiectelor mai mari în mai multe fișiere. Mai multe despre aceasta într-un laborator viitor, când vom vorbi despre proiecte C organizate în mai multe fișiere.

Definiții și macrouri

Definițiile și macrourele sunt alte facilități puse la dispoziție de preprocesorul C.

Definițiile se implementează folosind directiva **#define**. Ea are rolul de a atribui unui simbol un anumit text. De fiecare dată când simbolul definit va fi întâlnit în codul C, el va fi înlocuit cu textul atribuit lui. **#define** are următoarea sintaxă:

```
#define NUME    text_cu_care_se_inlocuieste_numele
```

În general, prin convenție, simbolurile definite cu **#define** sunt compuse din litere mari.

Exemplul 2: În următorul program, **#define** este folosit atât pentru a se stabili un număr maxim de elemente ale vectorului, cât și pentru definirea unei constante numerice:

```
#include <stdio.h>

#define N    100
#define PI   3.14159265358979323846

int main()
{
    int n=0,i,schimbat;
    float raze[N],r,tmp;
    printf("nr cercuri:");scanf("%d",&n);
    for(i=0;i<n;i++){
        printf("raze[%d]=",i);scanf("%g",&raze[i]);
    }
    do{
        schimbat=0;
        for(i=1;i<n;i++){
            if(raze[i-1]>raze[i]){
                tmp=raze[i-1];raze[i-1]=raze[i];raze[i]=tmp;
                schimbat=1;
            }
        }
    }while(schimbat);
    for(i=0;i<n;i++){
        r=raze[i];
        printf("pentru raza %g, aria este %g si perimetrul %g\n", r, PI*r*r, 2*PI*r);
    }
    return 0;
}
```

Constantele definite cu **#define** (*N* și *PI*) vor fi înlocuite cu corpul lor peste tot unde se vor întâlni în programul C. De exemplu, *raze[N]* va deveni *raze[100]*. Se constată că, deoarece limbajul C este *case-sensitive* (face diferența

între literele mari și mici), se pot folosi simultan atât N cât și n . N este o definiție de preprocesor, iar n este o variabilă de tip *int*.

Avantajul major când folosim definiții este că dacă dorim să modificăm o anumită valoare, atunci vom avea de modificat programul doar într-un singur loc (în corpul definiției) și schimbarea va fi vizibilă automat peste tot. Altfel, dacă am fi folosit direct valori în program, am fi putut avea următoarea situație: dacă acea valoare apare în 10 locuri, este posibil să o modificăm doar în 8 locuri cu noua valoare, iar în 2 locuri să o lăsăm cu vechea valoare. Astfel vor fi introduse în program buguri destul de greu de depistat, mai ales că este posibil ca ele să apară doar în anumite situații particulare.

În general, într-un program este bine să apară sub formă numerică doar valorile cu semnificație excepțională (ex: -1, 0, 1). Orice alte valori trebuie introduse prin definiții, constante sau enumerări.

Macrouri

Macrourile sunt definiții care au asociată o listă de argumente. Ele se implementează tot cu *#define*, conform următoarei sintaxe:

#define *NUME*(arg₁,...,arg_n) text_care_conține_numele_argumentelor

Argumentele sunt doar o serie de nume, fără niciun tip. Peste tot în textul specificat unde apar numele argumentelor, acestea se vor înlocui cu valorile date în cod (ca un fel de search/replace între un nume și conținutul său curent).

Atenție: lista argumentelor trebuie să urmeze imediat după *NUME*, fără niciun spațiu între ele. Dacă există un spațiu între *NUME* și paranteza deschisă, atunci *#define* se va considera că definește *NUME* fără argumente, iar argumentele vor fi considerate ca fiind corpul definiției.

Exemplul 3: În următorul program se definește MIN ca un macro de 2 argumente:

```
#include <stdio.h>

#define MIN(a,b) ((a)<(b)?(a):(b))

int main()
{
    printf("%d\n",MIN(7,5));
    printf("%g\n",MIN(3.8,21));
    return 0;
}
```

Peste tot unde se întâlnește în program macroul MIN, acesta va fi înlocuit cu corpul său, în care numele argumentelor vor fi și ele înlocuite cu valorile lor efective, de la folosire. De exemplu, *MIN(3.8,21)* se va înlocui cu *((3.8)<(21)?(3.8):(21))*. Parantezele sunt necesare pentru a păstra neschimbată ordinea operațiilor, atât în cazul în care *a* și *b* pot fi subexpresii (parantezele din jurul lui *a* și *b* fac ca acestea să fie tratate indivizibil, chiar dacă sunt subexpresii), cât și în cazul în care *MIN* se folosește în cadrul unei expresii mai mari (parantezele din jurul lui *MIN*).

Pentru a înțelege mai bine rolul parantezelor, vă rugăm să rezolvați următoarea aplicație: se consideră programul de mai jos. Fără a executa programul, determinați ce afișează fiecare *printf*. Ulterior executați programul și verificați dacă ați avut dreptate. Explicați de ce.

```
#include <stdio.h>

#define M1(a,b)  a*b+1           // nicio paranteza
#define M2(a,b)  (a)*(b)+1       // paranteze doar in jurul argumentelor
```

```

#define M3(a,b)  (a*b+1)           // paranteze doar in jurul macroului
#define M4(a,b)  ((a)*(b)+1)       // paranteze in jurul argumentelor si a macroului

int main()
{
    printf("%d\n",-M1(2+3,4+5));
    printf("%d\n",-M2(2+3,4+5));
    printf("%d\n",-M3(2+3,4+5));
    printf("%d\n",-M4(2+3,4+5));
    return 0;
}

```

Un alt pericol la folosirea macrourilor este dat de argumentele care au efecte colaterale (en: side-effects - orice alte efecte produse, în afară de valoarea returnată (afișare, incrementare, etc)). Dacă argumentul apare de mai multe ori în corpul macroului, cum este cazul lui *a* sau *b* din *MIN*, atunci efectele colaterale vor apărea de mai multe ori, nu doar o singură dată, cum ne-am fi așteptat. Această situație apare în programul următor:

```

#include <stdio.h>

#define MIN(a,b)  ((a)<(b)?(a):(b))

int minFn(int a,int b)
{
    return a<b?a:b;
}

int main()
{
    int i,j;
    i=0;j=2;
    printf("%d\n",minFn(++i,++j));    // => 1
    i=0;j=2;
    printf("%d\n",MIN(++i,++j));     // => 2
    return 0;
}

```

În acest program, minimul a fost implementat atât ca funcție, cât și ca macro. Dacă se apelează funcția *minFn(++i,++j)*, efectele colaterale ale apelării ei și anume incrementarea lui *i* și *j* se vor executa o singură dată, așa cum ne așteptăm. În schimb, *MIN(++i,++j)* se înlocuiește în program prin *((++i)<(++j)?(++i):(++j))*. În această situație, *++i* se va executa de 2 ori, ceea ce duce la rezultatul neașteptat.

Ținând cont de toate aceste aspecte, se preferă pe cât posibil reducerea folosirii macrourilor și înlocuirea lor prin funcții obișnuite. În general, macrourele au două avantaje față de funcțiile obișnuite:

- sunt generice din punct de vedere al tipurilor - *MIN* funcționează la fel pentru orice tipuri de numere, pe când *minFn* este specifică pentru numere întregi. Acest avantaj a fost exemplificat în exemplul 3, în care *MIN* a fost folosit atât pentru numere întregi, cât și pentru numere reale. În limbajul C++ se pot implementa și funcții generice, folosind mecanismul *template*.
- se pot folosi la generarea unor secvențe de cod, ceea ce poate ușura dezvoltarea programelor în care aceste secvențe de cod sunt repetitive.

Macrourele mai sunt folosite și ca mecanism de optimizare, fiindcă permit introducerea de cod direct acolo unde este folosit. De exemplu, dacă se folosește funcția *minFn*, apelul ei are nevoie un timp suplimentar necesitat de transferul parametrilor, transferul execuției în codul funcției și revenirea din funcție. Acest timp suplimentar nu

există la folosirea lui *MIN*, deoarece macroul se înlocuiește direct prin codul util. Compilatoarele mai noi detectează aceste cazuri particulare (funcții relativ reduse ca dimensiune, pentru care este mai eficient să se scrie direct codul lor în locul apelului) și realizează automat această optimizare (en: function inlining), deci din punct de vedere al eficienței nu este nicio diferență între folosirea lui *minFn* sau a lui *MIN*, dacă programul este compilat cu optimizare (ex: opțiunea **-O2** pentru gcc).

Aplicații ale macrourilor

Folosirea macrourilor poate ajuta mult la ușurarea unor sarcini în programare, cum ar fi: simplificarea codului repetitiv, implementarea funcțiilor sau a structurilor de date generice (care acceptă diverse tipuri de date), implementarea tabelor cu valori precalculate, etc.

Simplificarea codului repetitiv

Un exemplu de folosire a macrourilor pentru simplificarea codului repetitiv este următorul: avem o structură de date *Persoana*, cu câmpurile {*nume*, *email*, *adresa*, *observatii*}, toate de tip text și alocate dinamic, pentru a ocupa un spațiu minim de memorie. Fiecare dintre câmpuri are maxim 1000 de caractere. Să se scrie o funcție pentru introducerea unei persoane de la tastatură, alocată dinamic. Funcția va returna persoana citită. O primă variantă a acestei funcții este:

```
// varianta 1: implementare directa a citirii ficarui camp

typedef struct{
    char *nume;
    char *email;
    char *adresa;
    char *observatii;
}Persoana;

Persoana *introducere()
{
    Persoana *p;
    char linie[1000];

    if((p=(Persoana*)malloc(sizeof(Persoana)))==NULL){
        printf("memorie insuficienta");
        exit(EXIT_FAILURE);
    }

    printf("Nume: ");
    fgets(linie,1000,stdin);
    linie[strcspn(linie,"\r\n")]='\0';
    if((p->nume=(char*)malloc((strlen(linie)+1)*sizeof(char)))==NULL){
        free(p);
        printf("memorie insuficienta");
        exit(EXIT_FAILURE);
    }
    strcpy(p->nume,linie);

    printf("E-mail: ");
    fgets(linie,1000,stdin);
    linie[strcspn(linie,"\r\n")]='\0';
    if((p->email=(char*)malloc((strlen(linie)+1)*sizeof(char)))==NULL){
        free(p->nume);    // in plus fata de p->nume
```

```

    free(p);
    printf("memorie insuficienta");
    exit(EXIT_FAILURE);
}
strcpy(p->email,linie);

// analogic pentru p->adresa, p->observatii

return p;
}

```

Se constată pentru introducerea fiecărui câmp este destul de mult cod, care în marea majoritate este repetitiv. Singurele diferențe la citirea emailului față de nume sunt schimbarea textului din *printf* și eliberarea numelui în caz de eroare. Pentru a reduce aspectul repetitiv, care nu este de dorit într-un program, am putea implementa o funcție auxiliară pentru citirea unei linii:

```

// varianta 2: cu o functie auxiliara pentru citirea unei linii

char *introLinie(char *numeCamp)
{
    char linie[1000];
    char *dst;

    printf("%s: ",numeCamp);
    fgets(linie,1000,stdin);
    linie[strcspn(linie,"\r\n")]='\0';
    if((dst=(char*)malloc((strlen(linie)+1)*sizeof(char)))==NULL)return NULL;
    strcpy(dst,linie);
    return dst;
}

Persoana *introducere()
{
    Persoana *p;

    if((p=(Persoana*)malloc(sizeof(Persoana)))==NULL){
        printf("memorie insuficienta");
        exit(EXIT_FAILURE);
    }

    if((p->nume=introLinie("Nume"))==NULL){
        free(p);
        printf("memorie insuficienta");
        exit(EXIT_FAILURE);
    }

    if((p->email=introLinie("E-mail"))==NULL){
        free(p->nume);
        free(p);
        printf("memorie insuficienta");
        exit(EXIT_FAILURE);
    }

    // analogic pentru p->adresa, p->observatii

```

```
    return p;
}
```

În funcția *introLinie* am putut să factorizăm citirea și alocarea dinamică a unei linii de text. Nu am putut să mutăm acolo și mesajul de eroare și ieșirea din program, deoarece dacă *introLinie* ar ieși din program în caz de eroare, nu am putea elibera memoria alocată anterior. Chiar și așa, codul funcției *introducere* s-a simplificat. În continuare, pentru a simplifica și mai mult introducerea unui câmp, vom folosi macroui:

```
// varianta 3: cu macroui si functie auxiliara pentru citirea unei linii

#define INTRO_B(CAMP,AFIS)    if((p->CAMP=introLinie(#AFIS))==NULL){

#define INTRO_E                free(p);                \
                                printf("memorie insuficienta"); \
                                exit(EXIT_FAILURE);          \
                                }

Persoana *introducere()
{
    Persoana *p;

    if((p=(Persoana*)malloc(sizeof(Persoana)))==NULL){
        printf("memorie insuficienta");
        exit(EXIT_FAILURE);
    }

    INTRO_B(nume,Nume)
    INTRO_E

    INTRO_B(email,E-mail)
        free(p->nume);
    INTRO_E

    INTRO_B(adresa,Adresa)
        free(p->email);
        free(p->nume);
    INTRO_E

    INTRO_B(observatii,Observatii)
        free(p->adresa);
        free(p->email);
        free(p->nume);
    INTRO_E

    return p;
}
```

Deoarece între începutul și sfârșitul introducerii unui câmp trebuie eliberate câmpurile introduse anterior, ceea ce diferă de la un câmp la altul, s-au folosit două macroui, *INTRO_B* (begin) și *INTRO_E* (end). *INTRO_B* are două argumente, câmpul în care se introduce și textul care trebuie afișat de *printf*. Acestea au fost notate cu litere mari, pentru a fi mai ușor de urmărit în codul macroului.

În *INTRO_B*, parametrul *AFIS* a fost dat ca un identificator. Pentru a fi folosit în *printf*, el a trebuit transformat în șir de caractere, adică să fie puse ghilimele în jurul său. Pentru aceasta, preprocesorul ne pune la dispoziție operatorul unar *#* (diez), care se pune în fața unui argument și-l transformă în șir de caractere, prin adăugare de ghilimele înainte și după conținutul argumentului.

Pentru *INTRO_E* am avut nevoie să scriem definiția pe mai multe linii. Aceasta se realizează punând la sfârșitul fiecărei linii ** (backslash). Backslash, când apare pe ultima poziție a unei linii, are rol de concatenare a liniei curente cu cea următoare, ca și când nu ar fi existat *\n*. Astfel, de fapt preprocesorul vede toate cele 4 linii ale definiției lui *INTRO_E* ca fiind o singură linie.

Atenție: ** trebuie să fie strict pe ultima poziție din linie, fără niciun caracter după el, nici măcar spațiu sau comentariu. Dacă urmează orice altceva după el, dispare semnificația sa de concatenare.

Se constată că prin folosirea macrourilor codul s-a simplificat mult, el devenind aproape schematic, ca un fel de pseudocod. Mai mult decât atât, preprocesorul permite scrierea unui cod care diferă de sintaxa standard a limbajului C, de exemplu să nu mai trebuiască acolade sau punct și virgulă.

Exemplul 4: Să se scrie un macro care primește ca argumente un text, un placeholder pentru *printf/scanf* și un nume de variabilă. Macroul va genera o secvență *printf* și *scanf* pentru citirea unei valori în variabila specificată:

```
#include <stdio.h>

// varianta 1 - macro implementat ca o instructiune care nu necesita punct si virgula
#define CITIRE1(TEXT,P,VAR)    {printf(TEXT ": ");scanf("%" #P, &VAR);}

// varianta 2 - macro implementat ca o instructiune care necesita punct si virgula
#define citire2(TEXT,P,VAR)    do{printf(TEXT ": ");scanf("%" #P, &VAR);}while(0)

// varianta 3 - macro implementat ca o expresie care in plus returneaza valoarea citita
#define citire3(TEXT,P,VAR)    (printf(TEXT ": "), scanf("%" #P, &VAR), VAR)

int main()
{
    float kg;
    CITIRE1("Introduceti greutatea, intre 1 si 10 kg",g,kg)
    printf("greutatea=%g\n",kg);

    int n;
    citire2("nr elemente",d,n);
    printf("n=%d\n",n);

    getchar();

    char ch;
    printf("ch=%c\n",citire3("litera",c,ch));

    return 0;
}
```

Deoarece argumentul *TEXT* poate conține virgulă, s-a preferat introducerea sa direct ca text, fără a se folosi *#*. Altfel, dacă s-ar fi scris *"CITIRE1(Introduceti greutatea, intre 1 si 10 kg,g,kg)"*, din cauza virgulei din text s-ar fi considerat că se încearcă apelarea macroului cu 4 argumente.

În limbajul C, dacă se întâlnesc două constante șir de caractere una lângă alta, acestea se concatenează automat într-o singură constantă. De exemplu, șirurile "%" "g" vor deveni "%g". Pe această proprietate s-au bazat în cod concatenările TEXT ": " și "%" #P (P a fost anterior transformat în șir de caractere, prin folosirea #).

Când macrourele se substituie prin mai multe instrucțiuni, este indicat să se folosească o modalitate de tratare a acestora ca fiind o singură instrucțiune. Altfel pot apărea situații neașteptate la folosirea lor. De exemplu, în codul "if(a<b)AFISARE" să presupunem că macroul AFISARE generează două instrucțiuni: "instr1;instr2;". În această situație, codul generat va fi "if(a<b)instr1;instr2;", ceea ce este o eroare, deoarece doar instr1 va fi condiționată de if, nu și instr2.

Pentru a se grupa mai multe instrucțiuni ca și când ar fi una singură, avem trei posibilități:

- punem instrucțiunile între acolade {...} și obținem astfel un bloc care este tratat ca o singură instrucțiune. Deoarece după blocuri nu se pune punct și virgulă, în această formă nu se mai adaugă după macro punct și virgulă.
- punem instrucțiunile în interiorul unei instrucțiuni do{...}while(0). Aceasta se va trata tot ca o singură instrucțiune, iar din cauza condiției care este mereu falsă, nu se va repeta, deci vom avea o singură execuție. Deoarece do{...}while necesită punct și virgulă, în această situație macroul va avea nevoie de punct și virgulă după el.
- punem instrucțiunile între paranteze rotunde și le separăm prin virgulă (operatorul de secvențiere). Obținem astfel o expresie a cărei valoare este valoarea ultimei componente. Această formă de implementare este posibilă doar pentru secvențe de funcții, nu și pentru cazurile în care în corpul macroului se folosesc instrucțiuni gen for.

Folosirea oricărei variante ține mai mult de stilul de programare ales: unii programatori doresc să știe clar unde au în cod macroure, și atunci le evidențiază prin litere mari și detalii sintactice gen absența punct și virgulei, pe când alți programatori vor ca aspectul codului să fie cât mai unitar și atunci definesc macrourele cu litere mici și cer ca după ele să se pună punct și virgulă, la fel ca la o expresie obișnuită. Ultima variantă, cu folosirea parantezelor rotunde, este cea mai apropiată de folosirea unei funcții, deoarece i se poate prelua valoarea rezultată.

Când se scrie un macro mai complex, este util ca prima oară să se scrie codul efectiv pe care vrem să-l genereze acel macro. Abia după ce am testat codul respectiv, îl vom transforma într-un macro. Altfel, dacă începem direct cu scrierea macroului, depanarea eventualelor buguri devine mult mai complicată.

Cod generic

Exemplul 5: Să se scrie un macro care implementează o funcție generică de căutare a unui element într-un vector. Funcția va primi ca argumente un pointer la vectorul de elemente, numărul de elemente din vector și elementul de căutat. Pentru un tip oarecare de date, funcția se va numi "caut_tip" (ex: caut_int, pentru tipul int):

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

#define FN_CAUT(TIP) \
    int caut_##TIP(const TIP *p,int n,TIP e) \
    { \
        for( ; n ; n--, p++) { \
            if(*p==e) return 1; \
        } \
        return 0; \
    }

FN_CAUT(int)
FN_CAUT(double)
```

```
int main()
{
    int vi[]={7,0,-1,3,2};
    printf("%d\n",caut_int(vi,5,8));    // => 0

    double vf[]={3.14,9,0.5,-1.2e3};
    printf("%d\n",caut_double(vf,4,-1200));    // => 1

    return 0;
}
```

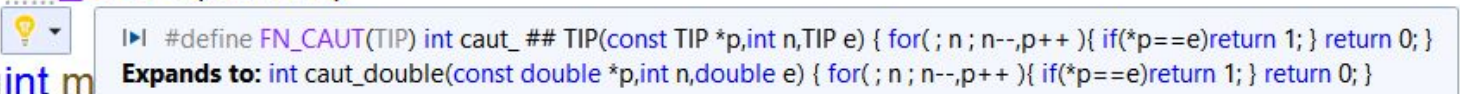
Funcția de căutare în sine este destul de simplă: se refolosește pointerul primit ca argument pentru a itera în toate elementele. Dacă elementul căutat a fost găsit se returnează 1 altfel, în final, se returnează 0.

În corpul macroului, pentru a se concatena două elemente, se folosește operatorul binar **##**. Acesta concatenează argumentul din stânga cu cel din dreapta, rezultând astfel un singur element. În cazul de mai sus, **##** a fost folosit pentru a concatena identificatorul *caut_* cu tipul specificat, pentru a rezulta astfel numele funcției.

După ce am implementat acest macro, îl putem folosi pentru a genera orice funcții de căutare dorim. Putem genera funcții pentru căutarea elementelor de tip *int*, *double*, *float*, *char*, etc. Singura cerință este ca algoritmul de căutare să fie identic. De exemplu, nu vom putea căuta șiruri de caractere într-un vector de șiruri, pentru că aceasta necesită folosirea funcției *strcmp*. În C++ funcțiile sau tipurile de date generice se implementează folosind mecanismul *template*.

Pentru simplificarea implementării și a folosirii macrourilor, unele IDE oferă facilitatea ca atunci când se poziționează cursorul mausului peste o folosire a macroului respectiv, să se afișeze textul pe care acesta îl generează. De exemplu, în Visual Studio, această facilitate este implementată astfel:

FN_CAUT(int)
FN_CAUT(double)



```
#define FN_CAUT(TIP) int caut_ ## TIP(const TIP *p, int n, TIP e) { for(; n; n--, p++) { if(*p==e) return 1; } return 0; }
Expands to: int caut_double(const double *p, int n, double e) { for(; n; n--, p++) { if(*p==e) return 1; } return 0; }
```

Tabele cu valori precalculate

Uneori, din motive de economie de spațiu sau de optimizare a vitezei, se preferă ca anumite valori să fie precalculate și incluse direct în codul programului, sub forma unor tabele. Macrourele pot ajuta mult la această sarcină.

Exemplul 6: Să se creeze o tabelă de valori precalculate pentru toate valorile pe un octet [0,255] care, la fiecare poziție, să specifice dacă acea valoare este o literă mică sau nu. Se va defini un macro care, folosind această tabelă, va implementa funcția *islower*.

```
#include <stdio.h>

#define VAL(v) ((v)>='a'&&(v)<='z'?1:0)

#define LINIE(baza) VAL(16*baza+0),VAL(16*baza+1),VAL(16*baza+2),VAL(16*baza+3), \
                    VAL(16*baza+4),VAL(16*baza+5),VAL(16*baza+6),VAL(16*baza+7), \
                    VAL(16*baza+8),VAL(16*baza+9),VAL(16*baza+10),VAL(16*baza+11), \
                    VAL(16*baza+12),VAL(16*baza+13),VAL(16*baza+14),VAL(16*baza+15)

const char tabela[]={
    LINIE(0),LINIE(1),LINIE(2),LINIE(3),
```

```

    LINIE(4),LINIE(5),LINIE(6),LINIE(7),
    LINIE(8),LINIE(9),LINIE(10),LINIE(11),
    LINIE(12),LINIE(13),LINIE(14),LINIE(15)
};

#define ISLOWER(c) (tabela[(unsigned char)(c)])

int main()
{
    printf("%d\n",ISLOWER('\n'));    // => 0
    printf("%d\n",ISLOWER('a'));    // => 1
    printf("%d\n",ISLOWER('A'));    // => 0
    return 0;
}

```

Pentru simplificarea codului, tabela a fost implementată ca o serie de linii, fiecare linie adăugând 16 valori. Calculul efectiv al unei valori din tabelă se face în macroul `VAL(v)`. Acesta returnează 1 sau 0, dacă valoarea este literă mică sau nu. Singura acțiune pe care o are de făcut `ISLOWER(c)` este să returneze valoarea din tabelă corespunzătoare argumentului dat. Conversia la *unsigned char* a fost necesară, deoarece tipul de date *char* este cu semn (intervalul [-128,127]) și atunci valorile peste 127 nu s-ar fi indexat corect în tabelă.

Deoarece toate aceste valori sunt constante, rezultatele operațiilor cu ele se calculează încă din faza de compilare. De exemplu, expresia `"16*0+1"` se înlocuiește cu valoarea 1. Compilatorul de C precalculează automat rezultatele tuturor expresiilor constante, astfel încât în codul final nu va mai fi nicio operație. Tabela va conține direct valorile finale, fără nicio operație de adunare, înmulțire, comparare, etc.

Aplicații propuse

Aplicația 6.1: Să se scrie un macro care returnează maximul a 3 argumente.

Aplicația 6.2: Să se scrie un macro care să genereze o funcție de sortare pentru un tip de date dat. Funcția va primi ca argumente un vector de elemente și dimensiunea sa și va sorta vectorul în ordine crescătoare.

Exemplu: `FN_SORTARE(unsigned)` -> va genera o funcție de sortare pentru valori de tip *unsigned*

Aplicația 6.3: Să se modifice exemplul 6 astfel încât fiecare valoare din tabelă să aibă următoarea semnificație: bitul 0==1 => literă mică, bitul 1==1 => literă mare, bitul 2==1 => cifră. Să se scrie 3 macroui, `ISLOWER`, `ISUPPER` și `ISDIGIT` care, folosind tabela, să testeze încadrarea argumentului lor în aceste clase de caractere.

Aplicația 6.4: Să se scrie un macro care primește ca argumente un text, un placeholder pentru *print/scanf*, un nume de variabilă și 2 valori, *min* și *max*. Macroul va trebui să ceară de la tastatură o valoare în mod repetat, până când ea se încadrează în intervalul închis dat.

Exemplu: `CITIRE("x=",g,x,0,100)` -> citește variabila x până când valoarea citită se încadrează în [0,100]

Aplicația 6.5: Să se scrie un macro care generează o structură *Punct* cu componentele {x,y} de un tip specificat în macro și totodată o funcție care primește ca parametru o variabilă de tipul *Punct* și returnează distanța de la punct la origine.

Exemplu: `PUNCT(float)` -> va genera structura *Punct_float* și funcția `"float len_float(Punct_float pt)"`

Aplicația 6.6: Să se scrie un macro care primește ca argumente două tipuri, *tip_dst* și *tip_src*. Macroul va genera o funcție care primește ca argument un vector cu elemente de tipul *tip_src* și numărul de elemente din vector. Funcția va alocă dinamic un nou vector cu elemente de tipul *tip_dst* și va depune în el toate elementele din vectorul inițial, convertite la *tip_dst*. Funcția va returna vectorul alocat.

Exemplu: `V_CONV(int,float)` -> va genera funcția `"int *v_conv_int_float(float *v,int n)"`

Aplicația 6.7: Folosind macrouri, să se genereze o tabelă cu valori precalculate corespunzătoare unui octet [0,255]. La fiecare poziție se va afla numărul de biți de 1 din indexul poziției. Să se scrie un macro *NBITI4(u)* care primește ca argument o valoare de tip *unsigned* pe 4 octeți și returnează numărul total de biți de 1 din ea.

Exemplu: NBITI4(0x01FF0325) -> 14