Programarea calculatoarelor
Limbajul C

CURS 13

Directive preprocesor

Tipuri generice

Fflush

Directive preprocesor

- sunt interpretate într-o etapă preliminară compilării (traducerii) textului C, de un preprocesor
- nu se foloseste caracterul ';' pentru terminarea unei directive!
- #define ident text

  inlocuieşte toate aparițiile identificatorului "ident" prin şirul "text"
- #define ident (a1,a2,...) text

  defineşte o macroinstructiune cu argumente
- #include "fişier"

  include în compilare conținutul fişierului sursa "fişier"
- #if expr

  compilare condiționată de valoarea expresiei "expr"
- #if defined ident

  compilare condiționată de definirea unui identificator (cu #define)
- #endif

  terminarea unui bloc introdus prin directiva #if

Macrouri

- Au aspect de funcție
- Pentru compilarea mai eficientă a unor funcții mici, apelate în mod repetat.
- Pot conține şi declaratii şi se pot extinde pe mai multe linii
- Pot fi utile în reducerea lungimii programelor sursă şi a efortului de programare.

Exemple:

```
# define max(A,B)  ( (A)>(B) ? (A):(B) )
#define randomize()   srand ((unsigned)time(NULL))
#define abs(a) (a)<0 ? -(a) : (a)
#define random(num) (int)(((long) rand() * (num)) /
    (RAND_MAX+1))
// generează un număr aleator între 0 şi num !
```

Exemplu

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

Atenție!

9+1%2==0 va conduce la 9+1 ==0 care este fals – 0
”este impar”

Ar trebui:

```
#define PAR(a) (a)%2==0 ? 1 : 0
```

Programare generică în C. Colecții de date generice

- Colecția poate conține:
  - valori numerice de diferite tipuri şi lungimi sau
  - şiruri de caractere sau
  - alte tipuri de date agregat (structuri), sau
  - pointeri (adrese).
- Problemă: operațiile cu un anumit tip de colecție să poată fi scrise ca funcții generale, adaptabile pentru fiecare tip de date ce va face parte din colecție!
- Rezolvare:

1. utilizarea de tipuri generice (neprecizate) pentru elementele colecției în subprogramele ce realizează operații cu colecția.
2. utilizarea unor colecții de pointeri la un tip neprecizat (*void* * în C); înlocuirea cu un alt tip de pointer (la date specifice aplicației) se face la execuție.

1. Utilizarea de tipuri neprecizate

```
typedef  int T;                    // tip componente multime
typedef  struct {
    T m[M];                        // multime de intregi
    int n;                         // dimensiune multime
} Set;
     // operatii cu o multime
int findS ( Set a, T x) {          // cauta pe x în multimea a
  int j=0;
  while ( j < a.n  &&  x != a.m[j] )
      ++j;
   if ( j == a.n) return 0;        // negasit
   return 1;                       // gasit
}
int addS ( Set* pa, T x) {         // adauga pe x la multimea a
  if ( findS (*pa, x) )
      return 0;                    // nu s-a modificat multimea a
  pa→ m[pa→n] = x;
  pa→n++;
  return 1;                        // s-a modificat multimea a
}
```

Observație

Operații valabile pentru orice tip:

a) Definirea unor operatori generalizați, modificați prin macro-substituție :

```
#define EQ(a,b)  (a == b)      // equals
#define LT(a,b)  (a < b)       // less than
#define AT(a,b)  (a = b)       // assign to
int findS ( Set a, T x) {      // cauta pe x în multimea a
   int j=0;
   while ( j < a.n  &&  ! EQ(x,a.m[j]) )
      ++j;
   if ( j==a.n) return 0;      // negasit
   return 1;                   // gasit
}
int addS (Set* pa, T x) {       // adauga pe x la o multime
  if ( findS (*pa,x) )
     return 0;                  // nu s-a modificat multimea
  AT(pa→m[pa→n++],x);           // adaugare x la multime
  return 1;                     // s-a modificat multimea
}
```

Observație

- Pentru o mulțime de şiruri de caractere trebuie operate următoarele modificări în secvențele anterioare :

```
#define EQ(a,b)  ( strcmp(a,b)==0) // equals
#define LT(a,b)  (strcmp(a,b) < 0)  // less than
#define AT(a,b)  ( strcpy(a,b) ) // assign to
typedef char * T;
```

Observație

b) Utilizarea unor funcții de comparație cu nume predefinite, care vor fi rescrise în funcție de tipul T al elementelor mulțimii:

```
typedef  char * T;
     // comparare la egalitate şiruri de caractere
int  comp  (T a, T b ) {
   return strcmp (a, b);
}
int findS ( Set a, T x) {      // cauta pe x în multimea a
  int j=0;
  while ( j < a.n  &&  comp(x,a.m[j]) ==0 )
     ++j;
  if ( j==a.n) return 0;
  return 1;
}
```

Observație

c) Transmiterea funcțiilor de comparare, atribuire, ş.a ca argumente la funcțiile care le folosesc (fără a impune nume fixe acestor funcții)

```
typedef  char * T;      // definire tip T

  // tip funcție de comparare
typedef  int (*Fcmp) ( T a, T b) ;

   // cauta pe x în multimea a
int findS ( Set a, T x, Fcmp cmp ) {
  int j=0;
  while ( j < a.n  &&  cmp(x,a.m[j]) ==0 )
      ++j;
  if ( j==a.n) return 0;
  return 1;
}
```

2. Utilizarea de pointeri la “void”

- Colecție generică = colecție de pointeri la orice tip (***void ***), care vor fi înlocuiți cu pointeri la datele folosite în fiecare aplicație
- Funcția de comparare trebuie transmisă ca argument funcțiilor de prelucrare (inserare, căutare, etc) a colecției

- Avantaj: funcțiile pentru operatii cu colecții pot fi compilate şi puse într-o bibliotecă şi nu este necesar codul sursă
- Dezavantajul unor colecții de pointeri apare în aplicațiile numerice: pentru fiecare număr trebuie alocată memorie la execuție ca să obținem o adresă distinctă ce se memorează în colecție!

Exemplu: Mulțime ca vector de pointeri

```
#define M 100
typedef int (*Fcmp) (void*, void* ) ;            // tip funcție de comparare
typedef void (*Fprint)  (void* );      // tip funcție de afisare
typedef struct  {
    void* v[M];     // un vector de pointeri la elementele multimii
    int n;          // nr elem în multime
} * Set;

// afisare date din multime
void printS ( Set a, Fprint print) {
  int i;
  for (i=0;i<a→n;i++)
      print ( a→v[i] );  // depinde de tipul argumentului
  printf ("\n");
}
```

Exemplu

```
 // initializare multime
void initS (Set a) {
  a→n=0;
}
// cautare în multime
int findS ( Set a, void* p, Fcmp comp ) {
  int i;
  for (i=0; i<a→n; i++)
    if (comp(p,a→v[i]) == 0 )
      return 1;
  return 0;
}
 // adaugare la multime
int addS ( Set a, void* p, Fcmp comp) {
  if ( findS(a,p,comp) )
    return 0;                          // multime nemodificata
  a→v[a→n++] = p;  // adaugare la multime
  return 1;                // multime modificata
}
```

Exemplu de creare şi afişare a unei mulțimi de întregi

```
void  print ( void* p) {          // afisare număr intreg
      printf ("%d ", *(int*)p );
}
int intcmp ( void* a, void* b) {          // comparare de intregi
  return *(int*)a - *(int*)b;
}
int main () {
      Set a; int x; int * p;
      a=(Set) malloc(sizeof(*Set));
      initS(a);
      printf ("Elem. multime: \n");
      while ( scanf ("%d", &x) > 0) {
            p= (int*) malloc (sizeof(int));
            *p=x;
            add (a, p, intcmp);
      }
      printS (a);
      return 0;
}
```

Funcții generice predefinite

- “stdlib.h”: funcții generice pentru sortarea şi căutarea binară într-un vector cu componente de orice tip
- Ilustrează o modalitate simplă de generalizare a tipului unui vector: argumentul formal de tip vector al acestor funcții este declarat ca *void** şi este înlocuit cu un argument efectiv pointer la un tip precizat (nume de vector).
- Un alt argument al acestor funcții este adresa unei funcții de comparare a unor date de tipul celor memorate în vector, funcție furnizată de utilizator şi care depinde de datele folosite în aplicația sa.

bsearch

- **void * bsearch(const void *key, const void *base, size_t nelem, size_t width, int (*fcmp)(const void *, const void *));**
  - returnează adresa primei intrări din tablou care coincide cu parametrul căutat şi zero – NULL, dacă acesta nu există în tablou (căutare binară)
  - key- adresa cheii căutate
  - base - începutul tabloului
  - nelem - nr.elemente din tablou
  - width - dim. unui elem. de tablou
  - fcmp - funcția de comparare definită de utilizator şi care primeşte doi parametri
- Atenție: Vectorul trebuie sa fie sortat conform funcției de comparație!

qsort

- **void qsort (void *base, size_t nelem, size_t width, int (*fcmp)(const void *, const void *));**
- sortează tabloul dat (algorimtul Quicksort)
  - base - începutul tabloului
  - nelem - nr.elemente din tablou
  - width - dim. unui elem. de tablou
  - fcmp - funcția de comparare definită de utilizator şi care primeşte doi parametri

Utilizarea funcției Qsort

- ordonarea un vector de numere întregi cu funcția “qsort” :

```
 // comparare numere intregi
int intcmp (const void * a, const void * b) {
  return *(int*)a -*(int*)b;
}
void main () {
   int a[]= {5,2,9,7,1,6,3,8,4};
   int i, n=9;                                  //n=dimensiune vector
   qsort ( a,9, sizeof(int), intcmp);   // ordonare vector
   for (i=0;i<n;i++)                          // afisare rezultat
     printf("%d ",a[i]);
}
```

Exemplu

- Să se scrie un program care execută în mod repetat următoarele operații:
  - Preia informațiile (nume şi nota) pentru o grupa de studenți. Citirea se face de la tastatură.
  - Verifică prezența unui student în grupă utilizând bsearch.
  - Listează grupa în ordine alfabetică (qsort) şi afisează media grupei.
  - Termină program.

```
# include <stdio.h>
# include <stdlib.h>
# include <ctype.h>
# include <string.h>
# define MAX_S  30
# define MAX_L  20
```

Exemplu - continuare

```
int cmp (const void*a, const void *b) {
    return strcmp ( (char*)a, (char*)b );
}

float citire (char tab[][MAX_L], int *nrstud) {
  float nota;
  int i;
  float medie=0;
  i=0;
  while ( scanf ("%s%f", tab[i++], &nota) != EOF)
    medie+=nota;
  *nrstud=i-1;
  medie/=(*nrstud);
  qsort (tab, *nrstud, MAX_L, cmp);
  return medie;
}
```

Exemplu - continuare

```
int cauta(char *s, char tab[][MAX_L], int nrstud){
  char *t;
  t = (char*) bsearch (s, tab, nrstud, MAX_L, cmp);
  return (t-tab[0])/MAX_L;
}

int main(){
 char stud[MAX_S][MAX_L];
 int nrstud=0;
 float medie;
 char c,s[12];
 int i;
 while(1){
    printf("\nOptiunea:\nCitire, Prezenta_student, Listare, Iesire\n >");
    fflush(stdin);
    c=getchar();
     switch(toupper(c)){
```

Exemplu - continuare

```
case 'C': medie = citire (stud, &nrstud);
                    break;
case 'P': printf("nume student:"); fflush(stdin); gets(s);
              if ( (i=cauta(s,stud,nrstud)) <0 )
                   printf("nu exista studentul %s în grupa",s);
              else
                   printf("studentul %s este al %d-lea din grupa",s,i+1);
              break;
     case 'L': printf("Lista studenti:\n");
              for (i=0; i<nrstud; i++)
                   printf("%d %s\n", i+1, stud[i]);
              printf("\n\n media grupei = %4.2f\n\n",medie);
              break;
     case 'I': printf("terminare program\n");
               exit(0);
   } /*switch*/
  } /*while*/
   return 0;
}
```

Exercițiu

- Modificați exemplul anterior astfel:
  - Se va defini o structură student cu nume şi notă
  - Se va utiliza un vector de astfel de structuri
  - Funcția de listare va afişa ordonat acest vector (nume şi nota)
- Observație: se vor utiliza tot qsort şi bsearch

Exemplu

```
# include <stdio.h>
# include <stdlib.h>
# include <ctype.h>
# include <string.h>
# define MAX_S  30
# define MAX_L  20

typedef struct{
     char nume[MAX_L];
     int nota;
}Stud;

int cmp (const void*a, const void *b) {
    Stud* sa=(Stud*) a;
    Stud* sb=(Stud*) b;
    return strcmp ( sa->nume, sb->nume );
}
```

Exemplu - continuare

```
float citire (Stud tab[], int *nrstud) {
  int nota;
  int i;
  float medie=0;
  i=0;
  while ( scanf ("%s%d", tab[i].nume, &tab[i].nota) != EOF){
    medie+=tab[i].nota;
     i++;
   }
  *nrstud=i;
  medie/=(*nrstud);
  qsort (tab, *nrstud, sizeof(Stud), cmp);
  return medie;
}
```

Exemplu - continuare

```
int cauta(char *s, Stud tab[], int nrstud){
  Stud *t, aux;
  strcpy (aux.nume, s);
  t = (Stud*) bsearch (&aux, tab, nrstud, sizeof(Stud), cmp);
  return t-tab;
}

int main(){
 Stud stud[MAX_S];
 int nrstud=0;
 float medie;
 char c,s[12];
 int i;
 while(1){
    printf("\nOptiunea:\nCitire, Prezenta_student, Listare, Iesire\n >");
    fflush(stdin);
    c=getchar();
     switch(toupper(c)){
```

Exemplu - continuare

```
case 'C': medie = citire (stud, &nrstud);
                    break;
case 'P': printf("nume student:"); fflush(stdin); gets(s);
              if ( (i=cauta(s,stud,nrstud)) <0 )
                   printf("nu exista studentul %s în grupa",s);
              else
                   printf("studentul %s este al %d-lea din grupa",s,i+1);
              break;
     case 'L': printf("Lista studenti:\n");
              for (i=0; i<nrstud; i++)
                   printf("%d %s %d\n", i+1, stud[i].nume, stud[i].nota);
              printf("\n\n media grupei = %4.2f\n\n",medie);
              break;
     case 'I': printf("terminare program\n");
               exit(0);
   } /*switch*/
  } /*while*/
   return 0;
}
```

Alte observații

- Nu orice apel al unei funcții de citire sau de scriere are ca efect imediat un transfer de date între exterior şi variabilele din program!
- Citirea efectivă de pe suportul extern se face într-o zonă tampon asociată fişierului, iar numărul de octeți care se citesc depind de suport: o linie de la tastatură, unul sau câteva sectoare disc dintr-un fişier disc, etc.
- Cele mai multe apeluri de funcții de I/E au ca efect un transfer între zona tampon (anonimă) şi variabilele din program.
- Funcția “fflush” are rolul de a goli zona tampon folosită de funcțiile de I/E, zonă altfel inaccesibilă programatorului C.
- Are ca argument variabila pointer asociată unui fişier la deschidere, sau variabilele predefinite “stdin” şi “stdout”.

Exemple de situații în care este necesară folosirea funcției “fflush”:

- Citirea unui caracter după citirea unui câmp sau unei linii cu “scanf” :

```
int main () {
  int n; char s[30]; char c;
  scanf (“%d”,&n);                    // sau  scanf(“%s”,s);
// fflush(stdin);              // pentru corectare
  c= getchar();               // sau  scanf (“%c”, &c);
  printf (“%d \n”,c);                 // afiseaza codul lui c
  return 0;
}
```

- va afişa 10 care este codul numeric al caracterului terminator de linie ‘\n’, în loc să afişeze codul caracterului “c”!
- după o citire cu “scanf” în zona tampon rămân unul sau câteva caractere separator de câmpuri (‘\n’,‘\t’,’ ‘), care trebuie scoase de acolo prin fflush(stdin) sau prin alte apeluri “scanf”.

Exemple de situații în care este necesară folosirea funcției “fflush”:

- Funcția “scanf” opreşte citirea unei valori din zona tampon ce conține o linie la primul caracter separator de câmpuri sau la un caracter ilegal în câmp (de ex. literă într-un câmp numeric)!
- In cazul repetării unei operatii de citire (cu “scanf”) după o eroare de introducere în linia anterioară (caracter ilegal pentru un anumit format de citire) în zona tampon rămân caracterele din linie care urmau după cel care a produs eroarea!

```
do {
        printf ("x, y= ");
        err = scanf ("%d%d", &x, &y);
        if ( err == 2 ) break;
        fflush (stdin);
  } while (err != 2);
```

- După citirea unei linii cu funcțiile “gets” sau “fgets” nu rămâne nici un caracter în zona tampon şi nu este necesar apelul lui “fflush”!

Exemple de situații în care este necesară folosirea funcției “fflush”:

- Se va folosi periodic “fflush” în cazul actualizării unui fişier mare, pentru a evita pierderi de date la producerea unor incidente (toate datele din zona tampon vor fi scrise efectiv pe disc):

```
int main () {
  FILE * f;   int c ;   char numef[]="TEST.DAT";
  char x[ ] = "0123456789";
  f=fopen (numef,"w");
  for (c=0;c<10;c++)
     fputc (x[c], f);
   fflush (f);                  // sau fclose(f);
   f=fopen (numef,"r");
   while ( (c=fgetc(f)) != EOF)
     printf ("%c", c);
   return 0;
 }
```

- Este posibil ca să existe diferențe în detaliile de lucru ale funcțiilor standard de citire-scriere din diferite implementări (biblioteci), deoarece standardul C nu precizează toate aceste detalii!