Capitolul IB.12. Convenţii şi stil de programare

Cuvinte cheie
Stil de programare, convenţii de scriere,
identificatori, indentare, spaţiere, directive preprocesor, macrouri

IB.12.1 Stil de programare – coding practices

Comparând programele scrise de diverşi autori în limbajul C se pot constata diferenţe importante în:

- modul de redactare al textului sursă (utilizarea de acolade, utilizarea de litere mici şi mari, etc.). Acest mod de redactare poate fi supus utilizării anumitor convenţii de programare
- modul de utilizare a elementelor limbajului (instrucţiuni, declaraţii, funcţii, etc.), aşa numitul stil de programare, propriu fiecărui programator, dar care poate fi supus totuşi unor reguli de bază.

O primă diferenţă de abordare este alegerea între a folosi cât mai mult facilităţile specifice oferite de limbajul C sau de a folosi construcţii comune şi altor limbaje (Pascal de ex.).
Exemple de construcţii specifice limbajului C:

- Expresii complexe, incluzând prelucrări, atribuiri şi comparaţii.
- Utilizarea de operatori specifici: atribuiri combinate cu alte operaţii, operatorul condiţional, etc.
- Utilizarea instrucţiunilor *break* şi *continue*.
- Utilizarea de pointeri în locul unor vectori sau matrice.
- Utilizarea unor declaraţii complexe de tipuri, în loc de a defini tipuri intermediare, mai simple.

Exemplu:

```
// definire vector de pointeri la functii void f(int,int)
void (*tp[M])(int,int);            // greu de citit!

// definire cu tip intermediar pointer la functie
typedef void (*funPtr) (int,int); // pointer la o functie cu 2 argumente int
funPtr tp[M];                     // vector cu M elemente de tip funPtr
```

O alegere oarecum echivalentă este între programe sursă cât mai compacte (cu cât mai puţine instrucţiuni şi declaraţii) şi programe cât mai explicite şi mai uşor de înţeles. În general este preferabilă calitatea programelor de a fi uşor de citit şi de modificat şi mai puţin lungimea codului sursă şi, eventual, lungimea codului obiect generat de compilator.
Deci se recomandă programe cât mai clare şi nu programe cât mai scurte.

Exemplu de secvenţă pentru afişarea a n întregi câte m pe o linie:

```
for (i=1; i<=n; i++)
    printf ( "%5d%c", i, ( i%m==0 || i==n)? '\n':' ');
```

O variantă mai explicită dar mai lungă pentru secvenţa anterioară:

```
for (i=1; i<=n; i++){
  printf ("%6d ", i);
     if (i%m==0)
        printf("\n");
```

```
}
printf("\n");
```

Alte recomandări

- Se recomandă utilizarea standardului ANSI C pentru portabilitate
- Programele nu trebuie să depindă de caracteristicile compilatorului (ordinea de evaluare a expresiilor
  Exemple:

```
k = ++i + i;      /* gresit */
y = f(x) + z_glb; /* gresit daca f() schimba valoarea lui z_glb*/
k = ++i + j++;    /* OK */
a[i++] = j++;     /* OK */
```

- Orice definire (de structură, enumerare, tip, etc) utilizată în mai multe fişiere va fi inclusă într-un fişier antet (***.h***) care va fi apoi inclus în fişierele care folosesc acea definiţie
- Toate conversiile de tip vor fi făcute explicit
- Variabilele structură se vor transmite prin adresa
- Pentru constantele utilizate pentru activarea / dezactivarea unor instrucţiuni se va verifica definirea acestora utilizând compilările condiţionate

```
// Nerecomandat:
#define DEBUG 4 /* folosit pentru a indica nivelul dorit de debug */
for (i = 0; i < 5 && DEBUG; i++)
{
      printf("i = %d\n", i);
}

// Recomandat:
#define DEBUG
#ifdef DEBUG
      for (i = 0; i < 5; i++)
      {
            printf("i = %d\n", i);
      }
#endif
```

- Pentru un simbol testat cu *#ifdef*, *#ifndef sau #if defined* nu se va defini o valoare

```
// Nerecomandat
#define DEBUG 0
#ifdef DEBUG
for (i = 0; i < 5; i++)
{
      printf("i = %d\n", i);
}
#endif

//Recomandat
#define DEBUG
#ifdef DEBUG
      for (i = 0; i < 5; i++)
      {
```

```
            printf("i = %d\n", i);
      }
#endif
```

A se vedea anexa ***Directive preprocesor utile în programele mari. Macrouri***

- Elementele unui vector vor fi accesate utilizând [] şi nu operatorul de dereferențiere *.

```
// Nerecomandat
int array[11];
*(array + 10) = 0;
// Recomandat
int array[11];
array[10] = 0;
```

- Transmiterea parametrilor prin pointeri va fi evitată ori de cate ori este posibil

```
x = f(a, b, c);
// şi
x = f(a, b, c, x);
//sunt mai uşor de înțeles decât:
f(a, b, c, &x);
```

- Toate instrucțiunile *switch* vor avea clauza *default* care întotdeauna va fi ultima

```
switch (variabila_int)
{
      case:..
            break;
      case:..
            break;
      default:..
}
```

- Operatorul virgulă (‘,’) va fi utilizat doar în instrucțiunea *for* şi la declararea variabilelor
- Modificarea unui cod existent se va face conform standardului existent deja în acel cod
- Modulele unui program nu trebuie să depăşească un anumit grad de complexitate şi un anumit număr de linii (maxim o jumătate de pagină)
- Se va utiliza evaluarea condiției afirmative mai degrabă decât a celei negative (!)
- Condițiile logice vor fi scrise explicit:

```
//Nerecomandat
if (is_available)
if (sys_cfg__is_radio_retry_allowed())
if (intermediate_result)
//Recomandat
if (is_available == FALSE)
if (sys_cfg__is_radio_retry_allowed() == TRUE)
if (intermediate_result != 0)
```

- Nu se recomandă utilizarea variabilelor globale; daca vor fi utilizate trebuie indeplinite următoarele cerințe:
  - Toate variabilele globale pentru un proiect vor fi definite într-un singur fişier
  - Variabilele globale vor fi inițializate înainte de utilizare
  - O variabilă globală va fi definită o singură dată pentru un program executabil

Funcţii

- Se vor folosi pe cât posibil funcţii standard în loc de funcţii specifice sistemului de operare
  - System Dependent: *open(), close(), read(), write(), lseek(),* etc.
  - ANSI Functions: *fopen(), fclose(), fread(), fwrite(), fseek(),* etc.
- Toate funcţiile vor avea tip definit explicit
- Funcţiile care întorc pointeri vor returna NULL în cazul neîndeplinirii unei condiţii
- Numărul parametrilor unei funcţii ar trebui limitat la cinci sau mai puţin
- Toate funcţiile definite trebuie însoţite de antete ce vor conţine lista tipurilor parametrilor

Constante

- Toate constantele folosite într-un fişier vor fi definite înainte de prima funcţie din fişier
- Dacă se definesc constantele TRUE şi FALSE, acestea trebuie să aibă valoare 1, respectiv 0:

```
#ifndef TRUE
      #define TRUE 1
#endif
#ifndef FALSE
      #define FALSE 0
#endif
```

- Definirea de constante simbolice (*#define*) va fi preferată utilizării directe a valorilor:

```
//Recomandat:
#define NMAX 100
int a[NMAX];

//Nerecomandat:
int a[100];
```

Variabile

- Toate variabilele se definesc înainte de partea de cod propriu-zis (instrucţiuni)
- Variabilele locale se definesc câte una pe linie; excepţie fac indecşii, variabilele temporare şi variabilele iniţializate cu aceeaşi valoare

```
int   Zona;
int   i, j, contor;
int   Mode = k = 0;
```

- Variabilele locale ar trebui iniţializate înainte de utilizare
- Se va evita utilizarea variabilelor globale pe cât posibil

Dimensiune vectori

- Nu se transmite dimensiunea maximă a unui vector când acesta este parametru al unei funcţii; aceasta se transmite separat într-o variabilă!

```
// Nerecomandat
char *substring(char string[80], int start_pos, int length)
{…
}
// Recomandat
char *substring(char string[], int start_pos, int length)
{…
```

```
}
```

Includerea fişierelor

- Fişierele antet vor defini o constantă simbolică pentru a permite includerea multiplă. Dacă fişierul se numeşte *file.h* constanta se poate numi *FILE_H*

```
// fisierul example.h
#ifndef EXAMPLE_H
#define EXAMPLE_H
…
#endif
```

- Se recomandă ca fişierele antet să nu includă alte fişiere antet
- Pentru includerea unui fişier antet definit de utilizator se vor utiliza “ ”, iar pentru o bibiloteca standard < >

Macrouri

- În macrourile tip funcţie parametrii vor fi scrişi între paranteze

```
// Nerecomandat
#define prt_debug(a, b) printf("ERROR: %s:%d, %s\n", a,__LINE__, b);

// Recomandat
#define prt_debug(a, b) printf("ERROR: %s:%d, %s\n", (a),__LINE__,(b));
```

- Macrourile complexe vor fi comentate
- Un macrou nu va depăşi 10 linii

IB.12.2. Convenţii de scriere a programelor

Programele sunt destinate calculatorului şi sunt analizate de către un program compilator. Acest compilator ignoră spaţiile albe nesemnificative şi trecerea de la o linie la alta.
Programele sunt citite şi de către oameni, fie pentru a fi modificate sau extinse, fie pentru comunicarea unor noi algoritmi sub formă de programe. Pentru a fi mai uşor de înţeles de către oameni se recomandă folosirea unor convenţii de trecere de pe o linie pe alta, de aliniere în cadrul fiecărei linii, de utilizare a spaţiilor albe şi a comentariilor.
Respectarea unor convenţii de scriere în majoritatea programelor poate contribui la reducerea diversităţii programelor scrise de diverşi autori şi deci la facilitarea înţelegerii şi modificării lor de către alţi programatori.
O serie de convenţii au fost stabilite de autorii limbajului C şi ai primului manual de C.

Beneficiile utilizării unor convenţii de programare:

- Uniformizează modul de scriere a codului
- Facilitează citirea şi înţelegerea unui program
- Facilitează întreţinerea aplicaţiilor
- Facilitează comunicarea între membrii unei echipe ceea ce duce la un randament sporit al lucrului în echipă

Observaţie:
Chiar dacă unele persoane pot resimţi ca o “îngrădire” aceste convenţii, ele permit totuşi

manifestarea creativităţii programatorului deoarece orice convenţie poate fi imbunătăţită şi adoptată ca standard al echipei respective de programatori.

IB.12.2.1 Identificatori

- Identificatorii trebuie să îndeplinească standardul ANSI C (lungimea<31, caractere permise: litere, cifre, _)
- Simbolul '_' nu va fi folosit ca prim caracter al unui identificator
- Nu se vor folosi nume utilizate de sistem decât dacă se doreşte înlocuirea acestora (constante, fişiere, funcţii)
- Toate fişierele header vor avea extensia ***.h***
- Toate constantele simbolice definite cu *#define* vor fi scrise cu litere mari
- Numele de variabile şi de funcţii încep cu o literă mică şi conţin mai mult litere mici (litere mari numai în nume compuse din mai multe cuvinte alăturate, cum sunt nume de funcţii din *MS-Windows*)
- În ceea ce priveşte numele unor noi tipuri de date părerile sunt împărţite
- Numele unui fişier nu trebuie să depşească 14 caractere în lungime (cu extensie).
- Variabilele locale, definiţiile de tip (*typedef*), numele de fişiere cât şi membrii unei structuri vor fi scrişi cu litere mici
- Numele variabilelor şi funcţiilor trebuie să fie semnificative şi în concordanţă cu ceea ce reprezintă.

Exemple:

Tip	Exemplu nume
Contor, index	i, j, k, l, g, h,
Pointeri	ptr_var, var_ptr, var_p, p_var, p
Variabile temporare	tmp_var, var_tmp, t_va
Valori returnate	status, return_value
Funcţii	readValues, reset_status, print_array
Fişiere	initialData.txt, entry.txt, setFuncţions.c, set.h

- Variabilele globale trebuie să se diferenţieze de cele locale (fie primul caracter literă mare, fie un sufix de genul *global*): **Vec_initial, vec_global, set_glb**

IB.12.2.2 Funcţii

- Se va scrie o singură instrucţiune pe linie
- Tipul unei funcţii şi numele acesteia vor fi pe aceeaşi linie

```
// Nerecomandat
int
err_msg(int error_code)
{
…
}

// Recomandat
int err_msg(int error_code)
{
…
}
```

IB.12.2.3 Spaţierea

Nu vor fi spaţii albe:

- După un cast explicit de tip

```
// Nerecomandat
int x = 1;
double y = 3.0;
y = (double) x + 16.7;

// Recomandat
int x = 1;
double y = 3.0;
y = (double)x + 16.7;
```

- Între operatorii unari (&, *, -, ~, ++, --, !, cast, sizeof) şi operanzii lor
- Înainte sau după operatorii primari (“()”,”[]”,”.”,”->”)
- Între caracterul # şi directiva de preprocesare

```
// Nerecomandat
#
define TEST 0
// sau
# define TEST 0

// Recomandat
#define TEST 0
```

- Între numele unei funcţii şi paranteza care îi urmează
- între primul argument al funcţiei şi paranteza deschisă
- între ultimul argument al funcţiei şi paranteza închisă

```
// Nerecomandat
just_return ( arg1, arg2 );

// Recomandat
just_return(arg1, arg2);
if (x == y)…
```

- Între parantezele deschisă, respectiv închisă şi expresia unei instrucţiuni condiţionale

```
if (x == y)…
```

Un singur spaţiu

- va exista între expresia condiţională a unei instrucţiuni şi numele ***if***

```
// Nerecomandat
if(x == y)

// Recomandat
if (x == y)
```

- precede şi urmează operatorii de atribuire, operatorii relaţionali, operatorii logici, operatorii aritmetici (excepţie cei unari) operatorii pe biţi şi operatorul condiţional

```
a = 3;
```

- va urma unei virgule

```
int a, b, c;
```

Alte recomandări:

- Va exista cel puţin o linie goală care să separe definirea variabilelor locale de instrucţiuni

```
int just_return(int first_arg, int second_arg)
{
      /*-------------- LOCAL VARIABLES -------------*/
      int i = 0;  /* Loop counter. */
      int j = 0;  /* Loop counter. */
      /*-------------------- CODE ------------------*/
      /*
      Body of funcţion just_return.
      */
      return(0);
}
```

- Membrii unei structuri, uniuni, enumerări vor fi plasaţi pe linii distincte la declararea lor
- Componentele logice ale unei expresii condiţionale vor fi grupate cu paranteze chiar dacă acestea nu sunt necesare

```
if ((x == y) && (a == b))
```

IB.12.2.4 Utilizarea acoladelor şi parantezelor

Una dintre convenţii se referă la modul de scriere a acoladelor care încadrează un bloc de instrucţiuni ce face parte dintr-o funcţie sau dintr-o instrucţiune *if, while, for* etc. Cele două stiluri care pot fi întâlnite în diferite programe şi cărti sunt ilustrate de exemplele următoare:

- **Stil Kernighan & Ritchie**

```
// exemplu bucla for
for (i = 0; i < loop_cntrl; i++){
      /* Corp for. */
}

// descompunere in factori primi
int main(){
  int n, k, p ;

  printf("\n n= ");
  scanf("%d",&n);
  printf("1");                        // pentru simplificarea afisarii
  for (k=2; k<=n && n>1; k++) {
     p=0;                          // puterea lui k in n
     while (n % k == 0) {          // cat timp n se imparte exact prin k
       p++;
       n = n / k;
     }
     if (p > 0)                    // nu scrie factori la puterea zero
       printf (" * %d^%d",k,p);
   }
}
```

- **Stil Linux:** Toate acoladele vor fi câte una pe linie

```
// exemplu bucla for
for (i = 0; i < loop_cntrl; i++)
{
      /* Corp for. */
}

// descompunere in factori primi
int main()
{
 int n, k, p ;

 printf("\n n= ");
 scanf("%d",&n);
 printf("1");                  // pentru simplificarea afisarii
 for (k=2; k<=n && n>1; k++)
 {
    p=0;                       // puterea lui k in n
    while (n % k ==0)          // cat timp n se imparte exact prin k
    {
       p++;
       n = n / k;
    }
    if (p > 0)                 // nu scrie factori la puterea zero
       printf (" * %d^%d",k,p);
 }
}
```

- Uneori se recomandă utilizare de acolade chiar şi pentru o singură instrucţiune, anticipând adăugarea altor instrucţiuni în viitor la blocul respectiv.

```
if (p > 0) {            // scrie numai factori cu putere nenula
     printf(" * %d^%d",k,p);
}
```

IB.12.2.5 Indentarea

Pentru alinierea spre dreapta la fiecare bloc inclus într-o structură de control se pot folosi caractere Tab (‘\t’) sau spaţii, dar evidenţierea structurii de blocuri incluse este importantă pentru oamenii care citesc programe.

Recomandări:

- Toate definiţiile de funcţii încep în coloana 1
- Acoladele ce definesc corpul unei funcţii vor fi în coloana 1 sau imediat după antet
- Acoladele corespunzătoare unei instrucţiuni, unei iniţializări de structură, vector, etc vor fi în aceeaşi coloană cu instrucţiunea sau iniţializarea respectivă

```
for (i = 0; i < loop_cntrl; i++)
{
      /* corp for */
}
```

- Instrucţiunile aflate la acelaşi nivel de includere vor fi indentate la aceeaşi coloană

```
if (conditie == TRUE)
{
```

```
        /* corp prim if */
}
else
        if
        {
                /* corp al doilea if */
        }
        else
        {
                /* else al doilea if */
        }
```

- Toate blocurile incluse în alt bloc vor fi indentate cu 2 până la 4 spații albe

Vor fi indentate cu 2 - 4 spaţii:

- câmpurile unui tip de date

```
struct example_type
{
    int x;
    double y;
};
```

- ramurile *case* ale unei instrucțiuni *switch*
- continuarea unei linii, față de operatorul de atribuire sau față de paranteza deschisă în cazul unei instrucțiuni sau a unui apel de funcție

```
num = this_example_test_structure.example_struct_field1 *
    this_example_test_structure.example_struct_field2;

if ((very_long_result_variable_name >=
     lower_specification_value))
```

IB.12.2.6 Comentarii

O serie de recomandări se referă la modul cum trebuie documentate programele folosind comentarii.

Astfel, fiecare funcție C ar trebui precedată de comentarii ce descriu

- rolul acelei funcții
- semnificația argumentelor funcției
- rezultatul funcției pentru terminare normală şi cu eroare
- precondiții - condiții care trebuie satisfăcute de parametri efectivi primiți de funcție (limite, valori interzise, etc.) şi care pot fi verificate sau nu de funcție
- plus alte date despre:
  - autor
  - data ultimei modificări
  - alte funcții utilizate sau asemănătoare, etc.

Exemplu:

```
/*
 Functie de conversie numar întreg pozitiv
 din binar în sir de caractere ASCII terminat cu zero
 "value" = numar intreg primit de functie (pozitiv)
 "string" = adresa unde se pune sirul rezultat
```

```
 "radix" = baza de numeratie (intre 2 şi 16, inclusiv)
 are ca rezultat adresa sir sau NULL in caz de eroare
 trebuie completata pentru numere cu semn
*/
char *itoa(int value, char *string, int radix) {
      char digits[] = "0123456789ABCDEF";
      char t[20], *tt=t, * s=string;
      if ( radix > 16 || radix < 0 || value < 0) return NULL;
      do {
            *tt++ = digits[ value % radix];
      } while ( (value = value / radix) != 0 );
      while ( tt != t)
               *string++= *(--tt);
      *string=0;
      return s;
}
```

Alte observaţii legate de comentarii:

- Vor completa codul, nu îl vor dubla!
- Explică mai mult decât este subînţeles din cod
- Nu trebuie să fie foarte multe comentarii (îngreunează codul) dar nici foarte puţine (nu este explicat codul)

Pot fi comentarii bloc, pe o linie, sau in-line:

- Comentarii bloc: descriu secţiunile principale ale programului şi vor fi indentate la acelaşi nivel cu codul pe care il comentează

```
/*
Acesta este un format care poate fi folosit pentru comentariile bloc
*/

/*********************************************************
*
Si acesta este un format care poate fi folosit pentru comentariile bloc
*
*********************************************************/

/*
*********************************************************
*
Si acesta este un format care poate fi folosit pentru comentariile bloc
*
*********************************************************
*/
```

- Comentarii pe o linie : Se indentează la acelaşi nivel cu codul pe care îl comentează

```
if (argc > 1)
{
      /* ia numele fisierului de intrare din linia de comanda. */
      if ((freopen(argv[1], 'r', stdin) == NULL)
      {
            /* Corp if */
      }
}
```

- Comentarii in-line (pentru descrierea declaraţiilor): trebuie să fie indeajuns de scurte încât să intre pe aceeaşi linie cu codul comentat

```
int i = 0;              /* Contor bucla */
int status = TRUE;      /* Rezultatul funcţiei*/
```

- Pentru comentarii pe o linie sau in- line se pot folosi şi comentarii C++:

```
int i = 0;              // Contor bucla
int status = TRUE;      // Rezultatul funcţiei
```

Reguli generale privind comentariile

- Comentariile nu vor fi incluse unele în altele
- Fiecare variabilă locală va avea un comentariu ce va descrie utilizarea ei dacă aceasta nu reiese din nume
- Comentariile in-line trebuie să fie aliniate pe cât posibil la stânga în cadrul unei funcţii

```
j = 5;      /* Assign j to the starting string position */
k = j + 9;  /* Assign k to the ending string position  */
```

- Instrucţiunile condiţionale sau buclele complexe (mai mult de 10 linii necomentate) vor avea ataşat un comentariu la acolada de închidere ce va indica închiderea instrucţiunii şi unul la începutul sau în interiorul blocului ce va indica scopul acestuia

```
if (a>b){
      /* scop
      ….
      ….*/
      …
} // end of if(a>b)
```

IB.12.3. Anexa: Directive preprocesor utile în programele mari. Macrouri

Directivele preprocesor C au o sintaxă şi o prelucrare distinctă de instrucţiunile şi declaraţiile limbajului, dar sunt parte a standardului limbajului C.
Directivele sunt interpretate într-o etapă preliminară compilării (traducerii) textului C, de către un preprocesor.
O directivă începe prin caracterul # şi se termină la sfârşitul liniei curente (dacă nu există linii de continuare a liniei curente).
Nu se foloseşte caracterul **;** pentru terminarea unei directive!

Cele mai importante directive preprocesor sunt:

Sintaxa	Descriere
#define *ident text*	înlocuieşte toate apariţiile identificatorului *ident* prin şirul *text*
#define *ident* (a1,a2,...) *text*	defineşte o macroinstrucţiune cu argumente
#include *fişier*	include în compilare continutul fişierului sursa *fişier*
#if *expr*	compilare condiţionată de valoarea expresiei *expr*

#if defined ***ident***	compilare condiționată de definirea unui identificator (cu *#define*)
#endif	terminarea unui bloc introdus prin directiva *#if*

Directiva define are multiple utilizări în programele C:

- Definirea de constante simbolice de diferite tipuri (numerice, text)
  Exemple:

```
#define begin {      // unde apare begin acesta va fi înlocuit cu {
#define end }        // unde apare end acesta va fi înlocuit cu }
#define N 100        // unde apare N acesta va fi înlocuit cu 100
```

- Definirea de **macrouri** cu aspect de funcție, pentru compilarea mai eficientă a unor funcții mici, apelate în mod repetat.
  Exemple:

```
// maxim dintre a si b
#define max(A,B)   ( (A)>(B) ? (A):(B) )

// generează un număr aleator între 0 şi num!
#define random(num)(int) (((long)rand()*(num))/(RAND_MAX+1))

// initializare motor de generare numere aleatoare
#define randomize() srand((unsigned)time(NULL))

// valoarea absoluta
#define abs(a) (a)<0 ? -(a) : (a)

// numar par cu utilizare!
#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+0 == 0 F ->”este impar”
Ar trebui:
#define PAR(a) (a)%2==0 ? 1 : 0
```

Macrourile pot conține şi declarații, se pot extinde pe mai multe linii şi pot fi utile în reducerea lungimii programelor sursă şi a efortului de programare.
În standardul din 1999 al limbajului C s-a preluat din C++ cuvântul cheie *inline* pentru declararea funcțiilor care vor fi compilate ca macroinstrucțiuni în loc de a folosi macrouri definite cu *define*.

- Definirea unor identificatori specifici fiecărui fişier şi care vor fi testați cu directiva ***ifdef***. De exemplu, pentru a evita declarațiile *extern* în toate fişierele sursă, mai puțin fişierul ce conține definițiile variabilelor externe, putem proceda astfel:

- Se defineşte în fişierul sursă cu definiţiile variabilelor externe un nume simbolic oarecare:

```
// fişier ul DIRLIST.C
#define MAIN
```

- În fişierul ***dirlist.h*** se plasează toate declaraţiile de variabile externe, dar încadrate de directivele if şi endif:

```
// fişier ul DIRLIST.H
#if !defined(MAIN)            // sau ifndef MAIN
      extern char path[MAXC], mask[MAXC], opt[MAXC];
#endif
```

Directiva *include* este urmată de obicei de numele unui fişier antet (de tip *H = header*), fişier care grupează declaraţii de tipuri, de constante, de funcţii şi de variabile, necesare în mai multe fişiere sursă (C sau CPP).

- Fişierele antet nu ar trebui să conţină definiţii de variabile sau de funcţii, pentru că pot apare erori la includerea multiplă a unui fişier antet.
- Un fişier antet poate include alte fişiere antet.
- Pentru a evita includerea multiplă a unui fişier antet (standard sau nestandard) se recomandă ca fiecare fişier antet să înceapă cu o secvenţă de felul următor:

```
#ifndef  HDR
#define HDR
  // continut fişier  HDR.H ...
#endif
```

Fişierele antet standard (*stdio.h*, etc.) respectă această recomandare.
O soluţie alternativă este ca în fişierul ce face includerea să avem o secvenţă de forma următoare:

```
#ifndef STDIO_H
#include <stdio.h>
#define _STDIO_H
#endif
```

Directivele de compilare condiţionată de forma ***if...endif*** au şi ele mai multe utilizări ce pot fi rezumate la adaptarea codului sursă la diferite condiţii specifice, cum ar fi:

- dependenţa de modelul de memorie folosit (în sistemul MS-DOS)
- dependenţa de sistemul de operare sub care se foloseşte programul (de ex., anumite funcţii sau structuri de date care au forme diferite în sisteme diferite)
- dependenţa de fişierul sursă în care se află (de exemplu *tcalc.h*).

Directivele din grupul ***if*** au mai multe forme, iar un bloc ***if ... endif*** poate conţine şi o directivă ***elseif***.