Programarea calculatoarelor și limbaje de programare 1

Laborator 10
Funcții, clase de stocare și directive de compilare condiționată

Dr. Ing. Liviu-Daniel ȘTEFAN
liviu_daniel.stefan@upb.ro

Dr. Ing. Mihai DOGARIU
mihai.dogariu@upb.ro

Prof. Dr. Ing. Bogdan IONESCU
bogdan.ionescu@upb.ro

Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei
Universitatea POLITEHNICA din Bucureşti

11-13 Decembrie, 2023

Cuprins

Definiția funcției

Definiție

Funcțiile sunt construcții bloc în C. O funcție este, de regulă, dezvoltată pentru a realiza o atribuție specifică iar numele ei deseori reflectă acea atribuție.

- Definiția unei funcții specifică tipul de date returnat de funcție, numele funcției precum și tipul de date al parametrilor, urmate de parametrii formali asociaţi lor, și include corpul funcției format din declarații de variabile locale și instrucțiunile care stabilesc atribuțiile funcției.

Sintaxă

```
specificator_de_tip nume_functie(tip parametru1,
...  , tip parametruN){ secvente_de_instrucțiuni; }
```

Definiția funcției

- `specificator_de_tip` specifică tipul de date pe care îl returnează funcția. O funcție poate returna orice tip de date cu excepția unei matrice.
  - ☞ Dacă nu este specificat nici un tip, funcția returnează implicit un rezultat de tip întreg.
- Dacă o funcție urmează să transmită argumente, definiția funcției trebuie să declare variabilele care vor prelua valoarea argumentelor.
  - ☞ Toți parametrii funcției trebuie declarați individual, folosind virgula ca separator, fiind precedați de tipul de date asociat lor;
  - ☞ O funcție poate să nu aibă parametri, caz în care lista lor este vidă. Totuși, parantezele sunt necesare chiar dacă nu există parametri.

Prototipurile funcților

Definiție

Prototipul unei funcții este instrucțiunea de declarare a acesteia în care se precizează tipul de date al valorii returnate, numele funcţiei şi tipurile de date ale parametrilor, urmate de punct și virgulă. Forma generală a declarării unei funcții este următoarea:

Sintaxă

```
tip nume_functie(tip nume_parametru1optional, ... ,
tip nume_parametruNoptional);
```

- Folosirea numelor parametrilor este opțională. Prezența lor permite compilatorului să identifice prin nume orice nepotrivire de tipuri atunci când apare o eroare;
- Lista de parametri din prototipul unei funcții poate conține unul, mai mulți, sau nici un parametru;

Prototipurile funcţilor

- Prototipul unei funcţii care nu conţine parametri trebuie să folosească cuvântul cheie `void` în interiorul listei de parametri, în mod explicit. Altfel, o listă goală de parametri nu poate face nicio presupunere referitoare la tipul sau numărul de parametri.
- ☞ Chiar dacă prototipurile sunt opţionale în C, ele sunt impuse în C++;
- ☞ Standardul ANSI C99 şi următoarele actualizări cer ca tipul parametrilor şi al valorii returnate unei funcţii să fie cunoscută a priori ca programul să apeleze funcţia. Plasând funcţia înaintea porţiunii de cod care o apelează în cadrul codului programului, îi permite compilatorului să obţină informaţiile necesare înainte de a întâlni apelul de funcţie. Pe măsură ce programele devin mai complexe, plasarea funcţiilor în ordinea corectă va deveni o sarcină dificilă, astfel, standardul ANSI C permite plasarea de prototipuri de funcţii;

Prototipul funcției

- Prototipurile permit compilatorului să verifice corespondența dintre tipurile agumentelor folosite la apelarea funcției şi tipurilor parametrilor funcției, depistând astfel orice conversie nepermisă precum și diferențele dintre numărul de argumente folosit pentru apelare și numărul parametrilor funcției respective;
- Conversiile de tip sunt permise, ex., dacă prototipul funcției specifică faptul că primește la intrare o valoare reală, dar la apelare se furnizează o valoare întreagă, aceasta este convertită la tipul specificat în prototipul funcției; contrar, dacă prototipul funcției specifică faptul că primește la intrare adresa unei variabile întregi, dar la apelare se furnizează o variabilă întreagă în loc de adresa sa, compilatorul detectează și afișează conversia nepermisă de tip.

`return`

Definiție

Instrucțiunea `return` întoarce controlul înapoi (sub)programului care a apelat funcția. În plus, această instrucțiune poate să întoarcă și o valoare în (sub)programul care a efectuat apelul. Întâlnirea instrucțiunii `return` determină oprirea imediată a rulării funcției din care face parte, și ieșirea din aceasta.

Sintaxă

`return expresie`$_{optional}$`;`

- Tipul de date al valorii `expresie` trebuie să fie același cu tipul de date returnat de funcție, precizat înainte de numele funcției;
- `expresie` este opțional;
- O limitare a limbajului C în ceea ce privește funcțiile este că ele pot returna o singură valoare.

Apelarea unei funcții

Definiție

Definirea unei funcții este necesară pentru a cunoaște ce cod va executa funcția. Pentru a folosi funcții, acestea trebuie apelate. Când o funcție este apelată, controlul programului este transmis din (sub)programul apelant, către funcție. Funcția apelată execută codul său, iar la întâlnirea instrucțiunii `return` sau a sfârșitului funcției (acolada închisă) întoarce controlul (sub)programului apelant. Legătura între programul apelant și funcție se face prin intermediul parametrilor.

- Parametrii transmiși către funcție, în programul apelant, se numesc parametri reali. Variabilele funcției, care primesc valorile acestor parametri, se numesc parametri formali. Parametrii formali sunt mereu variabile. În schimb, parametrii reali pot lua și valorile unor numere, constante, expresii sau chiar valorile unor funcții.

Apelarea prin valoare și apelarea prin referință

- În ceea ce privește modul în care se transmit parametri către funcție se disting două moduri: apelarea prin valoare și apelarea prin referință.
  - Apelul prin valoare copiază valorile argumentelor transmise funcției în parametrii formali. Astfel, modificările făcute asupra variabilelor din interiorul funcției nu afectează variabilele din afara funcției;
  - Apelul prin referință copiază adresa parametrului real în parametrul formal al funcției. În interiorul funcției se va folosi adresa de memorie pentru a accesa zona de memorie unde este stocat parametrul real și se vor face modificări direct la nivelul memoriei așa că modificările pe care le va face funcția vor persista și după sfârșitul ei.

☞ Datorită faptului că funcțiile pot să returneze o singură valoare, limbajul C folosește apelarea prin referință pentru a procesa mai multe date făcând modificări direct la nivelul memoriei.

Exemplu de apelare funcție prin valoare și prin referință

```
1  #include <stdio.h>
2  void increment_prin_referinta(int  *var){
3   /* Adresa parametrului real al funncției este copiată în parametrul formal
4      var. Orice modificare suportată de var se reflectă și în parametrul real,
5      deoarece modificarea s-a realizat la adresa parametrului real. */
6      *var = *var+1;}
7  int increment_prin_valoare(int  var){
8   /* Valoarea parametrului real al funncției este copiată în parametrul formal
9      var. Orice modificare suportată de parametrul formal nu se reflectă și în
10     parametrul real */
11     return ++var;}
12 int main(){
13     int num=0;
14     increment_prin_valoare(num);
15      /* num este incrementat prin valoare, dar datorită faptului ca nu rescriem
16     valoarea lui num cu argumentul returnat de funcție, num nu își schimbă
17     valoarea. num = 0 */
18     increment_prin_referinta(&num); // num = 1
19     printf("Valoarea lui num este: %d", num); // Afișează 1
20    return 0;}
```

Sfera de influenţă a funcţiilor

Definiţie

Sfera de influenţă a unui limbaj este formată din regulile care stabilesc ce secvenţă de cod ştie sau are acces la o altă secvenţă de cod sau de date.

- Fiecare funcţie este un bloc de cod discret. Instrucţiunile şi datele unei funcţii sunt proprii ei şi nici o instrucţiune din altă funcţie nu poate să aibă acces la ele decât printr-un apel al funcţiei;
- Variabilele care sunt definite într-o funcţie sunt numite variabile locale. O variabilă locală este creată atunci când se execută acea funcţie şi este distrusă la încheiere. Aceasta inseamnă că variabilele locale nu îşi păstrează valorile între apelările funcţiei.

☞ În C şi C++, toate funcţiile au acelaşi nivel al sferei de influenţă. Aceasta înseamnă că nu puteţi defini o funcţie într-o altă funcţie.

Funcții recursive

Definiție

Recursivitatea este procesul de definire a unui obiect prin el însuși, uneori fiind numită definiție circulară.

- Se spune că o funcție este recursivă dacă o instrucțiuni din corpul ei apelează chiar acea funcție;
- O funcție recursivă este compusă din două secțiuni: condiție de stop, care determină ieșirea din recursivitate, și corpul funcției;
- Când o funcție se apelează pe ea însăși, în memoria stivă sunt alocate zone pentru un nou set de variabile locale și parametrii, iar codul funcției se execută cu aceste noi variabile din vârf. Pe măsură ce apelările recursive sunt returnate, variabilele locale vechi și parametrii sunt îndepărtați din memoria stivă iar execuția se întoarce la punctul în care funcția s-a apelat singură.

Exemplu de funcție recursivă

```c
#include<stdio.h>

void afiseaza_interval_recursiv(int, int);
void afiseaza_interval_invers_recursiv(int, int);

int main() {
    afiseaza_interval_recursiv(0, 5);
    afiseaza_interval_invers_recursiv(0, 5);
    return 0;
}
void afiseaza_interval_recursiv(int start, int stop) {
    if (start < stop) {
        printf("%d\n", start);
        afiseaza_interval_recursiv(start + 1, stop);
    }
}
void afiseaza_interval_invers_recursiv(int start, int stop) {
    if (start < stop) {
        afiseaza_interval_invers_recursiv(start + 1, stop);
        printf("%d\n", start);
    }
}
```

Discuție 1/2

Funcția `afiseaza_interval_recursiv()` este un exemplu de rezolvare a unei probleme folosind metoda Divide et Impera. Dacă nu știm cum să afișăm un interval de numere de la 0 la 5, poate putem începe prin a rezolva o problemă mai simplă de afișare a primului număr 0. După ce am realizat acest lucru, avem o nouă (sub)problemă: afișarea numerelor de la 1 la 5. Dar observăm că avem deja o funcție `afiseaza_interval_recursiv()` care va face acest lucru pentru noi. Așa că o putem apela. Deși este laborios pentru oameni, un dispozitiv de calcul va genera negreșit cele șase instanțe imbricate ale funcției `afiseaza_interval_recursiv()`:

```
afiseaza_interval_recursiv(0, 5)
afiseaza_interval_recursiv(1, 5)
afiseaza_interval_recursiv(2, 5)
afiseaza_interval_recursiv(3, 5)
afiseaza_interval_recursiv(4, 5)
afiseaza_interval_recursiv(5, 5)
```

Discuție 2/2

Acest lucru funcționează deoarece fiecare apel al funcției primește proprii parametri și propriile variabile, separate de celelalte. Deci, fiecare apelare va afișa `start` și apoi va apela o altă copie pentru a afișa `start+1`, și așa mai departe. În cele din urmă, ultimul apel al funcției determină un rezultat fals pentru testul condițional `start < stop`, astfel funcția își încheie execuția, apoi părintele ei își încheie execuția și așa mai departe până când toate apelurile din stivă sunt derulate înapoi la primul.

Diferit de `afiseaza_interval_recursiv()`, care afișează valoarea `start` și apoi determină o nouă apelare a sa, `afiseaza_interval_invers_recursiv()` determină o nouă apelare a sa, până când testul condițional `start < stop` determină un rezultat fals ce încheie execuția funcției, dar nu înainte să afișeze valoarea lui `start`. Apoi părintele ei afișează valoarea lui `start` și își încheie execuția, și așa mai departe până când toate apelurile din stivă sunt derulate înapoi la primul.

Organizarea funcțiilor în fișiere

Organizarea funcțiilor în fișiere este utilă din mai multe puncte de vedere:

- Compilarea se face mai rapid, deoarece vor fi recompilate doar fișierele în care s-au făcut modificări;
- Crește gradul de organizare și devine mai ușor de localizat funcțiile necesare;
- Reutilizarea codului devine mai facilă;
- Modularitatea ajută la dezvoltarea și menținerea codului în cadrul unei echipe;
- Funcția `main()` se păstrează la o dimensiune redusă și ușor de urmărit.

Toate aceste aspecte pot fi atinse prin separarea interfeței de implementare.

Organizarea funcțiilor în fișiere

Interfața este reprezentată de fișierele de antet (header files, extensia `.h`), iar implementarea de fișierele sursă (source files, extensia `.c` / `.cpp`).

- `Fişierele antet` sunt fişiere a căror extensie este `.h` şi conţin prototipuri de funcţii şi definiţii macro ce pot fi utilizate de mai multe fişiere sursă. Există fişiere antet de sistem, incluse cu directiva `#include <nume_fisier.h>` şi fişiere scrise de utilizator, incluse cu directiva `#include "nume_fisier.h"`. Includerea unui fişier coincide cu copierea întregului său conţinut în locul directivei care îl include;
- `Fișierele sursă` sunt fișiere cu extensia `.c` / `.cpp` și conțin definițiile (sau implementările) funcțiilor.

Exemplu de organizare a funcțiilor în fișiere

Fișier sursă `main.cpp`

```
1  /* include fişierele antet stdio
2  şi simple_math, pentru operaţii
3  de ieşire prin funcţia printf(),
4  şi respectiv, pentru operaţii
5  matematice simple. */
6  #include<stdio.h>
7  #include"simple_math.h"
8  int main(){
9      printf("%d", suma(1, 2));
10     return 0;
11 }
```

Fișier antet `simple_math.h`

```
1  /*
2  conţine definiţii de instrumente
3  */
4  int suma(int, int);
5  int diferenta(int, int);
6  int inmultire(int, int);
7  float impartire(int, int);
```

Fișier sursă `simple_math.cpp`

```
1  /*
2  conţine declaraţii de instrumente
3  */
4  #include "simple_math.h"
5
6  int suma(int a, int b){
7      return a + b;
8  }
9  int diferenta(int a, int b){
10     return a - b;
11 }
12 int inmultire(int a, int b){
13     return a * b;
14 }
15 float impartire(int a, int b){
16     return (float) a / b;
17 }
```

Variabile locale și globale

Definiție

Variabilele declarate în interiorul unei funcții sunt denumite variabile locale.

- Variabilele locale sunt inițializate de fiecare dată când este întâlnit blocul în care sunt declarate, iar daca nu sunt inițializate explicit, acestea au valori necunoscute înainte de prima atribuire;
- Variabilele locale sunt cunoscute doar în interiorul blocului de cod în care au fost declarate. Un bloc de cod începe cu o acoladă deschisă și se termină cu echivalentul ei, acolada închisă, sau la întâlnirea instrucțiunii `return`;
- Blocul de cod cel mai uzual în care sunt declarate variabilele locale este funcția.

Variabile locale și globale

- Variabilele locale există doar atât cât se execută blocul de cod în care sunt declarate. Aceasta înseamnă că o variabilă locală este creată la începerea execuției blocului său și este distrusă la încheiere. Deoarece variabilele locale sunt create și distruse la fiecare intrare, respectiv ieșire din blocul în care au fost declarate, conținutul lor se pierde o dată cu încheierea blocului, altfel spus variabilele locale nu își păstrează valoarea între apelări.

Definiție

Variabilele globale sunt variabile care se creează prin declarare în afara oricărei funcții, implicit și a funcției `main()`.

- Variabilele globale sunt inițializate la începutul programului, fie printr-o instrucțiune explicită, fie printr-o instrucțiune implicită, prin care se preia automat valoarea 0, și distruse la finalizarea execuției întregului program.

Variabile locale și globale

- Variabilele globale sunt cunoscute întregului program, astfel că orice expresie are acces la ele, indiferent de tipul blocului de cod în care se află expresia;
- Variabilele globlale își păstrează valorile pe parcursul întregii execuții a programului.

☞ Când numele unei variabilei globale intră în conflict cu cel al unei variabile locale (când sfera de influență se suprapune), compilatorul de C/C++ va folosi întotdeauna variabila locală.

Specificatori de clase de stocare

Definiție

Specificatorii de clase de stocare permit controlarea modului cum sunt stocate variabilele care le urmează. C admite patru specificatori de clase de stocare: `extern`, `static`, `register` și `auto`.

Sintaxă

```
specificator_de_stocare tip nume_variabila;
```

- `extern` oferă compilatorului informațiile despre numele și tipurile de variabile globale fără să creeze un nou loc de stocare a lor. Acest lucru permite compilarea independentă a fișierelor unui program și editarea legăturilor împreună. Altfel spus, `extern` ne permite să comunicăm tuturor fișierelor variabilele globale necesare programului.

Specificatori de clase de stocare

- Procedura presupune declararea tuturor variabilelor globale într-un fișier și folosirea declarațiilor `extern` în celelalte. Atunci când sunt editate legăturile fișierelor, este rezolvat și accesul la variabilele externe;
- `static` permite declararea de variabile permanente în interiorul funcției sau fișierului în care se găsesc;
- Variabilele statice nu sunt cunoscute în afara funcției sau fișierului, dar ele își păstrează valoarea între două apelări;
- Atunci când aplicați specificatorul static unei variabile locale, aceasta își păstrează valoarea între apelările funcției;
- Atunci când aplicați specificatorul static unei variabile globale, specificăm compilatorului să creeze o variabilă globală care este cunoscută doar în fișierul în care a fost declarată;

Specificatori de clase de stocare

- `register` specifică stocarea variabilelor în registre ale CPU;
- Standardul ANSI C stipulează că `register` este o indicație dată compilatorului, că obiectul astfel declarat va fi utilizat din plin;
- Puteți aplica `register` doar variabilelor locale;
- Cuvântul cheie `register` a fost depreciat în C++, până când a devenit rezervat și apoi ignorat în C++17 (compilatoarele moderne vor plasa variabile într-un registru, dacă este cazul, indiferent dacă indicația este dată sau nu);
- Operatorul `&` nu poate fi aplicat variabilelor `register` pentru că acestea nu au adrese;
- `auto` permite declararea de variabile locale unui bloc sau funcții. O variabila care apare în corpul unei funcții sau al unui bloc pentru care nu s-a făcut nici o specificare de clasă de memorie se consideră implicit de clasa `auto`.

Exemplu de utilizare de variabile globale și locale

Fișier `antet3.h`

```
1  // Declarația var. globale externe
2  extern int variabila_globala;
```

Fișier `sursa1.cpp`

```
1  /* Importare declarații de var.
2     globale externe și funcții */
3  #include "antet3.h"
4  #include "prog1.h"
5  // Definirea var. globale externe
6  int variabila_globala = 0;
7  int increment_var_globala(void)
8  { return ++variabila_globala; }
```

Fișier `prog1.h`

```
1  /* extern poate fi folosit
2   doar pentru consecvență*/
3  void foloseste_var_globala(void);
4  void increment_var_statica(void);
5  void increment_var_locala(void);
6  int increment_var_globala(void);
```

Fișier `sursa2.cpp`

```
1  #include "antet3.h"
2  #include "prog1.h"
3  #include <stdio.h>
4
5  void foloseste_var_globala(void){
6   printf("Var. globală: %d\n",
7   variabila_globala);
8  }
9  void increment_var_statica(void){
10  static int variabila_globala = 0;
11  printf("Var. statică incrementată: %d\n",
12  ++variabila_globala);
13 }
14 void increment_var_locala(void){
15  int variabila_locala = 0;
16  printf("Var. locală incrementată: %d\n",
17  ++variabila_locala);
18 }
```

Exemplu de utilizare de variabile globale și locale

Fișier `prog1.cpp`

```
1  #include "antet3.h"
2  #include "prog1.h"
3  #include <stdio.h>
4  int main(void){
5      printf("Variabila locală este inițializată cu 0\n");
6      increment_var_locala(); // Afișează 1
7      increment_var_locala(); // Afișează 1
8   /* Variabila locală este distrusă după fiecare apelare a funcției și nu
9      își păstrează valoarea între apelări
10  */
11     printf("Variabila statică este inițializată cu 0\n");
12     increment_var_statica(); // Afișează 1
13     increment_var_statica(); // Afișează 2
14  // Variabila locală statică își păstrează valoarea între apelările funcției
15     printf("Variabila globală este inițializată cu 0\n");
16     foloseste_var_globala(); // Afișează 0
17     variabila_globala += 19;
18     foloseste_var_globala(); // Afișează 19
19     increment_var_globala(); // Incrementează cu 1
20     foloseste_var_globala(); // Afișează 20
21     return 0;
22 }
```

Discuție

Utilizarea `extern` este relevantă numai atunci când programul pe care îl scrieți constă din mai multe fișiere sursă legate între ele, unde unele dintre variabilele definite, de exemplu, în fișierul `sursa1.cpp` trebuie să fie referite în alte fișiere, cum ar fi `sursa2.cpp`.

O modalitate de a declara și defini variabilele globale este utilizarea unui fișier antet pentru a conține o declarație externă a variabilei. Antetul este inclus de un singur fișier sursă care definește variabila și de toate fișierele sursă care fac referire la variabilă.

Pentru fiecare program, un fișier sursă (și doar un fișier sursă) definește variabila. În mod similar, un fișier antet (și doar un fișier antet) ar trebui să declare variabila.

Deși există și alte moduri de a obține acest lucru, această metodă este simplă și fiabilă. Este demonstrat de `antet3.h`, `sursa1.cpp` și `sursa2.cpp`.

Directive de compilare condiționată

Definiție

Directivele de compilare condiționată permit compilarea selectivă a unor porțiuni din codul sursă al programului.

- `#if`, `#elif`, `#else` și `#endif`. Aceste directive vă permit să introduceți condiționat porțiuni de cod bazate pe rezultatul unei expresii constante.

Sintaxă

```
#if expresie_constanta
    secventa de instructiuni
#elif expresie_constanta
    secventa de instructiuni
#else expresie_constanta
    secventa de instructiuni
#endif
```

Directive de compilare condiționată

- `#ifdef` și `#ifndef`. `#ifdef` înseamnă *dacă este definit*, iar `#ifndef` înseamnă *dacă nu este definit*.

Sintaxă `#ifdef`

```
#ifdef nume_macro
    secventa de instructiuni
#endif
```

- Dacă `nume_macro` a fost definit anterior într-o instrucțiune `#define`, blocul de cod va fi complilat.

Sintaxă `#ifndef`

```
#ifndef nume_macro
    secventa de instructiuni
#endif
```

- Dacă `nume_macro` nu a fost definit curent de o instrucțiune `#define`, blocul de cod va fi complilat.

Exemplu de utilizare de directive de compilare condiționată

Fișier antet3.h

```
1  #ifndef ANTET3_H
2  #define ANTET3_H
3  extern int variabila_globala;
4  #endif
```

Fișier sursa1.cpp

```
1  #include "antet3.h"
2  #include "prog1.h"
3  int variabila_globala = 0;
4  int increment_var_globala(void)
5  { return ++variabila_globala; }
```

Fișier prog1.h

```
1  #ifndef PROG1_H
2  #define PROG1_H
3  void foloseste_var_globala(void);
4  void increment_var_statica(void);
5  void increment_var_locala(void);
6  int increment_var_globala(void);
7  #endif
```

Fișier sursa2.cpp

```
1  #include "antet3.h"
2  #include "prog1.h"
3  #include <stdio.h>
4
5  void foloseste_var_globala(void){
6   printf("Var. globală: %d\n",
7   variabila_globala);
8  }
9  void increment_var_statica(void){
10  static int variabila_globala = 0;
11  printf("Var. statică incrementată: %d\n",
12  ++variabila_globala);
13 }
14 void increment_var_locala(void){
15  int variabila_locala = 0;
16  printf("Var. locală incrementată: %d\n",
17  ++variabila_locala);
18 }
```

Exemplu de utilizare de directive de compilare condiționată

Fișier `prog1.cpp`

```
1  #include "antet3.h"
2  #include "prog1.h"
3  #include <stdio.h>
4  int main(void){
5      printf("Variabila locală este inițializată cu 0\n");
6      increment_var_locala(); // Afișează 1
7      increment_var_locala(); // Afișează 1
8      printf("Variabila statică este inițializată cu 0\n");
9      increment_var_statica(); // Afișează 1
10     increment_var_statica(); // Afișează 2
11     printf("Variabila globală este inițializată cu 0\n");
12     foloseste_var_globala(); // Afișează 0
13     variabila_globala += 19;
14     foloseste_var_globala(); // Afișează 19
15     increment_var_globala(); // Incrementează cu 1
16     foloseste_var_globala(); // Afișează 20
17     return 0;
18 }
```

Discuție

Pe măsură ce programele utilizează din ce în ce mai multe fișiere antet, vor exista situații în care un fișier inclus va include un al doilea fișier antet, care, la rândul lui, va include primul fișier antet. Pe măsură ce preprocesorul efectuează includerile, se poate ajunge la situația unor operații circulare. Pentru a reduce posibilitatea operațiilor circulare, fișierele antet pot declara o macrocomandă care va preveni compilatorul să proceseze fișierele pentru a doua oară.

Exemplul anterior folosește o protecție de includere multiplă (include guards). Protecția de includere este folosită pentru a preveni ca un fișier, de fapt conținutul unui fișier, să fie inclus de mai multe ori.

Aici, `ANTET3_H` și `PROG1_H` sunt doar identificatori. O practică bună presupune stabilirea unor identificatori derivați din numele fișierului antet.

- ☞ Toate fișierele antet ar trebui să aibă o protecție de includere multiplă.