Preprocesorul

Preprocesarea este o faza care precede compilarea. Preprocesorul limbajului C este relativ simplu si in principiu executa substitutii de texte. Prin intermediul lui se realizeaza:

- Includeri de fisiere sursa
- Definitii si apeluri de macrouri
- Compilare conditionata

Includerea unor fisiere sursa

Includerile de fisiere se realizeaza cu directiva `#include`.
Forma ei este

```
#include "nume_de_fisier"
```

sau

```
#include <nume_de_fisier>
```

O linie #include de acest tip este inlocuita cu continutul integral al fisierului specificat. Diferenta intre cele 2 moduri de scriere este urmatoarea : daca numele fisierului de inclus este dat intre ghilimele, preprocesorul cauta fisierul pornind de la directorul curent, cel in care se afla programul. Daca fisierul nu e gasit, sau daca numele este dat intre paranteze unghiulare, cautarea fisierului se face in continuare dupa reguli definitede implementarea compilatorului de C. La TurboC, se specifica prin optiuni de configurare a mediului TurboC care sunt directoarele unde se face cautarea. Fisierele antet ale bibliotecilor standard, cum sunt `<stdio.h>` care se da la o directiva include, nu sunt intotdeauna intr-adevar fisiere, pot avea si alte instantieri dependente de implementarea mediului de C.

Un fisier inclus poate contine la randul sau directive #include.

De obicei, intr-un fisier care este inclus in altele, se pun directivele #define si declaratiile de clasa extern, sau prototipurile functiilor de biblioteca. Utilizarea #include e o modalitate de a utiliza aceleasi declaratii in cazul programelor care sunt impartite pemai multe fisiere. Daca se modifica unfisier inclus, trebuie recompilate toate fisierele care il include pe acesta direct sau indirect.

Definitii si apeluri de macrouri

Definitia constantelor simbolice prin directiva `#define` e un caz particular de definitie de macro.
Un macro e o facilitate generala de prelucrare a textelor care are la baza operatia de substitutie. Un macro are o definitie si poate fi apelat de un numar oarecare de ori. La definitia unui macro se specifica de fapt textul care urmeaza a se substitui la fiecare apel al sau. Acest text poate fi variabil, in functie de anumiti parametrii.

Forma generala a unei macrodefinitii este:

```
#define nume(p1, p2, …, pn) tex_de_substituit
```

nume = numele macroului
p1, p2, …, pn = parametri macroului

text = textul cu care este inlocuit macroul la fiecare apel

In cazul unei macrodefinitii cu parametri intre numele macroului si paranteza deschisa trebuie sa nu existe spatii, altfel macroul se va considera fara parametrii si parantezele impreuna cu parametrii vr fi considerate ca facand parte din textul de substitutie. Intre paranteza inchisa si textul desubstitutie trebuie sa existe cel putin un caracter spatiu.
Textul de substitutie poate contine parametrii p1, p2, …, pn.
Apelul unui macro consta din numele lui, urmat de lista valorilor parametrilor intre paranteze. Parametri dela apel se substituie in locul parametrilor formali in textul de substitutie al macroului apelat si apoi textul rezultat se substituie apelului.

Exemplu. Macro pentru definirea maximului a doua numere:

```
#define MAX(x,y) ((x)>y() ? (x) : (y))
```

Exemple de apeluri pentru acest macro

```
int i,j,k;
k=MAX(i+j, i-j);
```

La compilare linia va fi inlocuita cu

```
k=(i+j) > (i-j) ? (i+j) (i-j))

float a,b,c;
c=MAX(a,b);
```

La preprocesare linia se inlocuieste cu

```
c=((a) > (b) ? (a) : (b));
```

Avantajele unui macro fata de o functie care sa implementeze acelasi lucru sunt urmatoarele:
In primul rand, este evitarea apelurilor ineficiente din punct de vedere al regiei pentru functiile simple care se reduc la una sau doua instructiuni. In aceste cazuri este recomandabil sa nu se mai construiasca functii care sa fie apelate, ci instructiunile respective sa fiescrise direct in locurile unde sunt necesare. Macrourile tratate de preprocesorul limbajului C realizeaza generari in line a functiilor.
In al doilea rand, macrourile pot fi mai generaledecat functiile, nedepinzand de tpul parapetrilor. In cazul macroului de determinare a maximului, acesta e general si nu depinde de tipul parametrilor, care pot fi intregi sau reali.

Exista si anumite **pericole** legate de utilizarea necorespunzatoare a macrodefinitiilor, care pot avea efecte secundare ascunse.
De exemplu, daca se apeleaza macroul de determinare a maximului cu urmatorii parametri:

```
MAX(i++, j++)
```

acesta va incrementa de 2 ori valoarea variabilei care e mai mare ! pentru ca expandarea lui se face in felul urmator:

```
(i++) > (j++) ? (i++) : (j++)
```

Fie macroul de ridicare a unui numar la patrat
#define PATRAT(x) x*x
Acesta da rezultate eronate in cazul in care este apelat de exemplu pentru x+1:
PATRAT(x+1) este expandat in x+1*x+1, ceea ce, avand in vedere precedenta operatorilor, nu calculeaza patratul expresiei (x+1).
Corect, un macro de ridicare la patrat ar trebui scris utilizand paranteze:

#define PATRAT(x) (x)*(x)

Exemple

Macro pentru transformarea unui caracter din litera mica in litera mare:

```
#define UPPER(c)  ((c)-'a'+'A')
```

Macro pentru definirea unui ciclu infinit:

```
#define FOREVER  for(;;)
```

Macro pentru interschimbarea a doua numere

```
#define SCHIMBA(X,Y) {\
      int t;\
      t=X;\
      X=Y;\
      Y=t;
}
```

Apelul

```
SCHIMBA(a,b)
```

Este substituit cu secventa:

```
{
int t;
t=a;
a=b;
b=t;
}
```

Variabila t exista numai in interiorul instructiunii compuse generata de preprocesor !

O facilitate interesanta a macrodefinitiilor este ca permit parametrizarea unei operatii cu un nume de tip, ceea ce se poate folosi ca o facilitate de realizare a unor functii generice primitive. Macro-ul de interschimbare a doua elemente poate fi rescris astfel incat sa fie parametrizat si cu tipul elementelor de interschimbat:

```
#define SWAP(TIP, X, Y) {\
      TIP t;\
      t=X;\
      X=Y;\
      Y=t;
}
```

Macrodefinitia SWAP se poate apela cu parametrii de orice tip pe care este definita operatia =.

```
int a, b;
…
SWAP(int, a, b);
…

float x, y;
…
SWAP(float, x, y);
```

…

Macro pentru alocarea dinamica a unui bloc de memorie de n elemente de un anumit tip:

```
#define ALOCA(tip, n)  (tip *)malloc(sizeof(tip)*n)

void main(void) {
int *ip;
float *fp;
ip=ALOCA(int, 10);
fp=ALOCA(float, 10);
}
```

Macroexpandarea poate fi oricand suspendata cu directiva undef:

```
#undef nume
```

la intalnirea acestei directive, macroul cu numele respective nu mai este expandat in continuarea fisierului.

In mod normal, preprocesorul nu substituie parametrii formali in interiorul sirurilor de caractere intre ghilimele. Insa daca in textul de inlocuire se plaseaza caracterul # inaintea unui nume de parametru formal, aceasta combinatie se expandeaza intr-un sir intre ghilimele, continand argumentul respective.

```
#define AFISEAZA(expr) printf(#expr "=%g\n", expr)

AFISEAZA(x/y);
```

Se expandeaza in

```
printf("x/y""=%g\n",x/y);
```

Compilare conditionata

Compilarea conditionata permite sa se aleaga dintr-un text general partile care se compileaza impreuna.

Compilarea conditionata se realizeaza folosind constructiile
#if, #ifdef si #ifndef

In cele ce urmeaza, se noteaza cu `expr` o expresie constanta (valoarea ei poate fi evaluata de preprocessor la intalnirea ei).

```
#if expr
      text
#endif
```

Daca expr are valoarea adevarat (diferita de zero), atunci text se supune preprocesarii, altfel se continua cu ceea ce urmeaza dupa #endif.

```
#if expr
      text1
#else
      text2
```

```
#endif
```

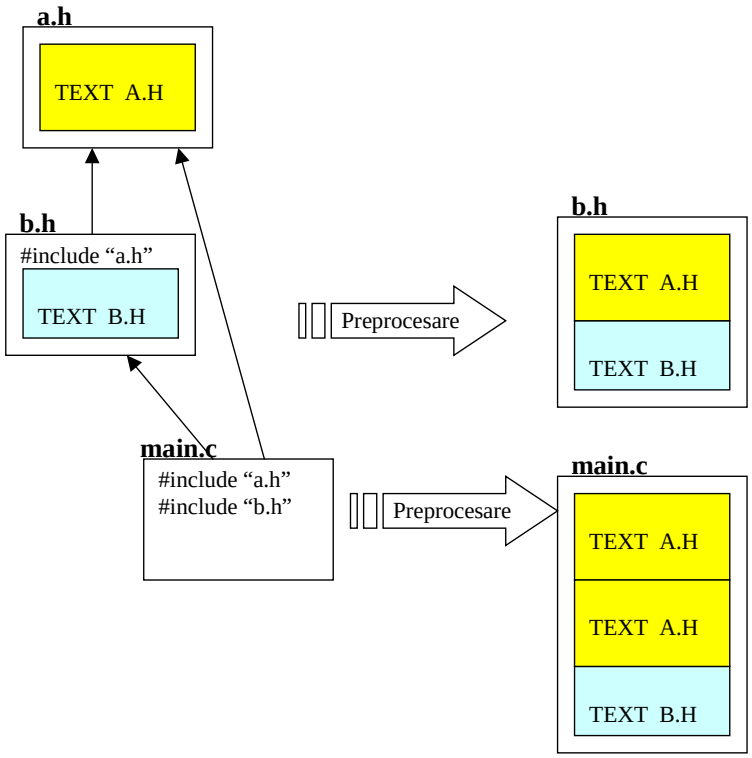
In interiorul unei directive#if, expresia defined(nume) are valoarea 1 daca nume a fost deja definit de o directiva define, sau zero in caz contrar.

De exemplu, pentru a asigura faptul ca se include o singura data continutul unui fisier antet antet1.h, continutul fisierului se poate plasa intr-o instructiune conditionala ca si

```
#if !defined(ANTET1)
#define ANTET1

…. /* continutul fisierului antet */
#endif
```

Prima includere a fisierului `antet1.h` defineste numele `ANTET1`. In cazul in care ar mai exista includeri ulterioare, indirect, prin alte fisiere, preprocesorul vede ca numele e deja definit si sare direct la `#endif`. Aceasta facilitate este utila in situatii in care un fisier antet ar fi altfel inclus de mai multe ori, direct sau indirect, ca in urmatoarea situatie de exemplu: Se considera un fisier antet `a.h` si un fisier antet `b.h` care contine o directiva de includere a lui `a.h`. Daca un alt fisier `main.c` contine directive de includere a ambelor fisiere `a.h` si `b.h`, rezulta ca fisierul `a.h` va fi inclus de doua ori. Aceasta includere dubla poate conduce la erori datorita redefinirii unor variabile si constante continute in `a.h`.



Solutia este plasarea continutului util al antelului `a.h` intr-un fisier de forma:

```
/* fisierul a.h */
#if !defined(ANTET_A)
#define ANTET_A
… TEXT A.H …
#endif
```

In aceasta situatie, in fisierul `main.c` se include de doua ori continutul textual al fisierului a.h, dar textul util din fisierul a.h (partea notata TEXT A.H) va ajunge sa fie inclusa doar o data,datorita directivei #if: prima data, se defineste numele ANTET_A si se include TEXT_A.H, dar a doua oara numele ANTET_A este deja definit, deci TEXT_A.H nu mai este inclus !

Exista si formele #ifdef si #ifndef, forme specializate ale directivei #if, care testeaza daca un nume estesau nu este definit.

```
#ifndef(ANTET1)
#define ANTET1

… /* continutul fisierului antet */
#endif
```

O alta forma de exploatare a facilitatii de compilare conditionata este de includere a unor fisiere diferite in functie de anumite variabile sistem:

```
#if SYSTEM==SYSV
      #define ANTET “sysv.h”
#elif SYSTEM==BSD
      #define ANTET “bsd.h”
#elif SYSTEM==MSDOS
      #define ANTET “msdos.h”
#else
      #define ANTET “default.h”
#endif
#include ANTET
```

Un exemplu in care e necesara facilitatea de includere conditionata sunt definirea de tipuri mutual recursive: se considera oua tipuri structurate T! si T2, fiecare dintre acestea contine un camp de tip pointer la celalalt tip.

```
typedef struct {T2 a; int b;} *T1;
typedef struct {T1 a; int b;} *T2;
```

In acest caz, se pune problema ordinii corecte in care trebuie sa fie definite aceste tipuri. Daca T1 e definit inaintea lui T2, compilatorul nu il cunoaste pe T2 cand intalneste definitia luiT1, care are un camp (a) de acest tip. Problema exista si invers, daca T2 ar fi definit inaintea lui T1. Pentrua rezolva problema referirilor reciproce, se utilizeaza declaratii incomplete de structuri:

```
typedef struct t_T1 * T1;
typedef struct t_T2 *T2;
struct t_T1 {T2 a; int b;};
struct t_T2 {T1 a; int b;};
```

In cazul in care cele 2 tipuri, T1 si T2 , sunt definite in fisiere separate t1.h si t2.h:

```
/* fisierul t1.h */
#if !defined(tip_T1)
      #define tip_T1
      typedef struct t_T1 *T1;
      #include "t2.h"
      struct t_T1 {T2 a; int b;};
#endif
```

```
 /* fisierul t2.h */
 #if !defined(tip_T2)
      #define tip_T2
      typedef struct t_T2 *T2;
      #include "t1.h"
      struct t_T2 {T1 a; int b;};
 #endif

/* fisierul main.c */
#include "t1.h"
#include "t2.h"

void main() {
…

}
```