# LECTIA 14
# PREPROCESORUL C



## OBIECTIVE

- directive de compilare

## PREZENTARE GENERALA

Cu toate ca existenta mediului integrat este calea cea mai sigura si mai simpla de a produce programe in limbajul C++, nu este lipsit de importanta sa purtam o discutie si asupra elementelor asociate compilatoarelor utilizate in cadrul acestui mediu. Intr-un asemenea compilator la o prima parcurgere a programelor sursa, se vor include toate fisierele specificate in acest sens, se vor evalua toate expresiile conditionale ce vizeaza direct aceasta faza, deci cea a compilarii, se vor expanda toate expresiile macro si se va produce un fisier, intermediar, care va fi preluat de urmatoarea faza a compilarii. Deoarece acest fisier intermediar nu este regasit pe suportul fizic la sfirsitul etapei de compilare, atat mediul integrat, cat si compilatorul din linia de comanda sunt inzestrate cu un preprocesor, rolul acestuia fiind exact acela de a produce un astfel de fisier. Acest preprocesor este util in depanarea programelor, in inspectarea directivelor de includere, compilare conditionata si a expresiilor macro complexe.In continuare, vom enumera directivele preprocesorului, atat din punct de vedere sintactic, cat si din punct de vedere semantic. Preprocesorul C include un macroprocesor sofisticat, care scaneaza codul sursa inainte de a fi supus efectiv actiunii lui de compilare. Acesta ne furnizeaza o mare flexibilitate in:

- **definirea de expresii macro**, care reduc efortul de programare si confera mai multa claritate codului.
- **includerea textelor** provenite din alte fisiere, astfel ca fisierele header contin constante si prototipuri de functii din bibliotecile standard si definite de utilizatori.
- **stabilirea de conditii** ce trebuie sa fie indeplinite pentru ca anumite portiuni de cod sa contribuie la faza de compilare.

Astfel orice linie care incepe cu caracterul # este considerata de catre preprocesor ca o directiva ce ii este adresata.

### Directiva nula #

Aceasta directiva consta dintr-o linie care contine un singur caracter #. Aceasta este intotdeauna ignorata

de preprocesor.

## Directivele #define si #undef

Directivele **#define** definesc constante macro. Acestea reprezinta un mecanism de inlocuire globala a unor nume, insotite de o lista de parametri formali. In cazul lipsei parametrilor, sintaxa este urmatoarea:

```
#define id_macro secv_definitie_macro
```

Fiecare aparitie a identificatorului expresiei macro, id_macro, in textul sursa al programului, dupa definitia introdusa cu **#define**, se va inlocui cu secventa ce expliciteaza corpul expresiei respective, secv_definitie_macro. Exista cateva exceptii de la aceasta regula si anume: nu se vor inlocui aparitiile numelui expresiei macro din cadrul sirurilor de caractere, constantelor siruri de caractere sau al comentariilor,

```
#define SALUT " Te salut, "
#define PRIETEN " Ioane "
#define LOCALITATE " , de la Agigea!"

puts(SALUT); puts(PRIETEN); puts(LOCALITATE);
```

va produce :  " Te salut, Ioane ,de la Agigea! "

In cazul expresiilor macro cu parametri, sintaxa ce trebuie respectata in definirea acestora este:

```
#define id_macro(1_param) secv_definitie_macro
```

Nu trebuie sa existe nici un spatiu intre identificatorul expresiei macro si paranteza ce deschide lista parametrilor acestuia. In cadrul acestei liste, argumentele se vor desparti prin virgule si nu se va specifica tipul lor, din acest punct de vedere expresiile macro urmand protocolul C de declarare a functiilor. Apelul unui astfel de macro consta in doua seturi de inlocuiri. In primul rand, identificatorul expresiei macro si parantezele ce cuprind lista de argumente sunt inlocuite de sirul de definitie al expresiei, iar, incea de a doua etapa, argumentele formale ale acesteia sunt inlocuite de parametri efectivi.

```
#define SUMA(a,b) ((a)+(b))

int i,j,sum;

sum=SUMA(i,j);
```

va expanda **SUMA in ((i)+(j))**. In cadrul expresiilor macro, nu se efectueaza verificarea tipurilor argumentelor, acest element nefiind utilizat nici in definirea expresiei. Sa consideram urmatorul exemplu :

```
#define PRODUS(a,b) ((a)*(b))

int i,j,prod;

prod=PRODUS(i,j+1);
```

produsul se expandeaza in **((i)*(j+1))**. Neglijand parantezele ce inconjoara parametrii formali, produsul va fi expandat in **(i*j+1)**, fiind astfel evident ca nu aceasta a fost intentia noastra. Cateva cazuri in care se utilizeaza expresiile macro cu parametri: **parantezele imbricate -** Lista de parametri efectivi poate sa contina paranteze imbricate

```
#define SUMA(x,y) ((x)+(y))

return SUMA((f(i,j) , g(m,n));
```

**construirea de nume -** Putem obtine noi nume prin alipirea a doua nume utilizand operatorul ##, si eventual, spatii albe. Preprocesorul elimina spatiile albe si ##, combinand cele 2 nume intr-unul nou

```
#define VARIABILA(i,j) (i##j)
```

si apeland-o prin **VARIABILA(x,6)**, obtinem identificatorul **(x6)**.

**conversia de siruri prin # -** Simbolul # poate fi plasat inaintea unui argument al expresiei macro pentru a-l converti intr-un sir de caractere, dupa inlocuire. Astfel avand urmatoarea definitie de macro:

```
#define URMA(flag) printf(#flag "= %d\n",flag)
```

fragmentul de cod

```
int valmax=1024;

URMA(valmax);
```

devine

```
int valmax=1024;
```

```
printf("valmax" "= %d\n", valmax);
```

si este echivalent cu

```
printf("valmax= %d\n", valmax);
```

**backslash pentru continuarea liniei -** Un identificator prea lung poate fi descris pe mai multe linii utilizand caracterul "\" (backslash) ca terminator de linie intermediara. Preprocesorul elimina atat caracterul linie noua, cat si caracterul "\", pentru a furniza identificatorul real:

```
#define AVERTISMENT "Acesta este un avertisment !"

puts(AVERTISMENT);
```

Directiva **#undef** este opusa directivei anterioare **#undef id_macro**, anuleaza definirea anterioara a expresiei macro, astfel ca orice utilizare ulterioara a acesteia va fi considerata eronata.

## Incluziunea fisierelor. Directiva #include

Exista trei metode de a include fisiere unele in altele

```
#include <nume_fisier>
#include "nume_fisier"
#include id_macro
```

Preprocesorul inlatura linia **#include** si o inlocuieste, conceptual, cu continutul fisierului indicat, printr-una din cele trei variante. Este normal ca sursa codului nu se va modifica, dar compilatorul vede fisierul extins. Din acest motiv, pozitia directivei poate influenta scopul si durata identificatorilor. In cazul in care este specificata o cale explicita pentru fisierul inclus, atunci numai acel fisier, daca exista, va fi inclus. Si acum, despre despre metodele de a include fisiere. Diferenta dintre primele doua metode consta in faptul ca, in timp ce prima varianta specifica un fisier standard pentru includere, localizarea sa facandu-se in fiecare director cu acest rol, in ordinea in care acestia s-au definit, in cea de a doua varianta localizarea fisierului de inclus va incepe cu directorul curent si se va continua abia apoi, cu directoarele destinate incluziunii de fisiere. In sfarsit, cea din urma varianta de includere presupune ca nici < si nici " nu vor aparea ca prim caracter in cadrul identificatorului. Mai mult se presupune ca explicitarea expresiei macro va produce un nume de fisier valid, bineinteles cuprins intre ghilimele sau paranteze <,>, reducand deci directiva la una din primele variante.

```
#include <stdio.h>
#include "newgraph.h"
```

```
#define myinclud "c:\users\my_heads\graph.h"
#include myinclud
```

## Compilarea conditionata

Toate directivle de compilare conditionata trebuie sa fie incluse in fisierul in care s-a inceput efinirea lor. Directivele conditionale **#if, #elif , #else si #endif** functioneaza asemeni operatorilor conditionali ai limbajului. Ei sunt tilizati in urmatoarele variante:

```
#if expr_const_1
    sectiunea_1
#elif expr_const_2
    sectiunea_2
    . . .
#elif expr_const_n
#else sectiunea_finala
#endif
```

Daca **expr_const_1** este nenula, in urma expandarii, liniile de cod din **sectiunea_1** se vor procesa, altfel **sectiunea_1** este ignorata. In cazul in care se proceseaza **sectiunea_1**, directiva de compilare conditionata, dupa procesarea sectiunii se incheie cu **#endif**. Daca **expr_const_1** este nula deci neindeplinita, se va procesa in functie de **sectiunea_2**, respectiv **n**, sau cea finala. Directiva optionala **#else**, in cazul existentei sale, va fi considerata ca varianta in cazul in care nici una din conditiile anterioare nu a fost verificata. Sectiunea procesata poate contine si alte clauze conditionale, singura regula care trebuie urmata este aceea ca pentru fiecare **#if** existent sa existe un singur **#endif**. Dintre expresiile constante cele mai frecvente trebuie sa mentionam acele expresii ce sunt constituite prin intermediul operatorului defined acesta putand aparea intr-una din formele **defined**(id_const), sau **defined** id_const, amble fiind echivalente. Evaluarea unei expresii ce contine acest operator va furniza 1, in cazul in care **id_const** a fost definit si 0 in caz contrar. Astfel, directiva

```
#if defined(MYCONST)
```

este echivalenta cu

```
#ifdef MYCONST
```

Avantajul uitilizarii operatorului **defined** este acela ca acesta poate fi utilizat rpetat in cadrul unei aceleasi linii de directiva pe cand **#ifdef** sau **#ifndef** nu, cum ar fi in exemplul de mai jos:

```
#ifdefined (O_CONST) && ! defined (ALTA_CONST)
```

Mai mult, aceste din urma directive trebuie sa fie incheiate cu directiva **#endif**. Asa dupa cum observam putem forma expresii conditionale utilizand operatorul **defined** si operatorii relationali uzuali(**&&,!**).

## DIRECTIVA #line

Comanda **#line** poate fi utilizata pentru a numerota liniile unui program, numarul liniilor fiind util in raportarea erorilor si, in special atunci cand programul analizat este multi-fisier, numerotarea fisierelor incluse incepand cu indicele liniei din programul principal, in care s-a efectuat includerea. Astfel sintaxa

```
#line ct_intreaga "nume_fisier"
```

indica faptul ca liniile urmatoare acesteia se vor numerota cu indicele ct_intreaga in fisierul nume_fisier. Odata folosit un nume de fisier, adresarile ultrioare ale directivei **#line** se vor referi la acelasi fisier. Expresiile macro sunt expandate in aceasta directiva ca si in cazul directivei **#include**.

## DIRECTIVA #error

Aceasta directiva are urmatoarea sintaxa:

```
#error mesaj_eroare
```

si genereaza mesajul:

```
Error: nume_fisier line # :Error directive : mesaj_eroare
```

Aceasta directiva furnizeaza un mesaj atasat unei erori in compilare. De exemplu, presupunand ca am definit O_COND prin **#defined** O_COND, putem include urmatoarea conditie intr-unul din programele noastre pentru a testa o eventuala valoare incorecta a lui O_COND

```
#if (O_COND!=0 && O_COND!=1)
#error O_COND trebuie sa aiba valoarea 0 sau 1
#endif
```

## DIRECTIVA #pragma

Cu ajutorul acestei directive, programatorul poate implementa alte directive de forma:

```
#pragma nume_directiva
```

Cu ajutorul ei, C isi poate defini propriile directive, fara a genera conflicte cu alte compilatoare care

admit aceleasi directive. Cand compilatorul nu recunoaste directiva indicata prin nume_directiva, el va ignora directiva **#pragma**, fara sa genereze nici macar un mesaj de avertisment. Borland C admite urmatoarele directive **#pragma**:

**#pragma argsused** - est valabila numai in definirea functiilor si actioneaza numai asupra functiei imediat urmatoare.

**#pragma exit** - permite programatorului de a stabili functii ce vor fi apelate numai inainte de iesirea din program. Sintaxa este:

```
#pragma exit nume_functie nr_prioritate
```

Functia indicata trebuie sa fie declarata si sa posede un prototip de genul:

```
void nume_functie (void)
```

In ceea ce priveste numarul optional, de prioritate, acesta trebuie sa fie un intreg din [64,255]. Cea mai mare prioritate este 0. Functiile cu prioritati mari sunt apelate ultimele la iesire. In cazul in care numarul de prioritate lipseste, se va considera numarul de prioritate implicit, acesta fiind 100.

**#pragma hdrfile** - stabileste numele fisierului in care se vor depune headeri precompilati. Implicit, acest nume este TCDEF.SYM, iar sintaxa este:

```
#pragma hdrfile "nume_fisier.SYM"
```

In cazul in care nu utilizati headeri precompilati, directiva nu are efect.

**#pragma hdrstop** - incheie lista fisierelor header utilizate in precompilare. Puteti utiliza aceasta directiva pentru a reduce dimensiunea spatiului utilizat de headeri precompilati.

**#pragma inline** - semnalizeaza compilatorului existenta unor linii de cod in limbaj de asamblare in cadrul programului. Sintaxa este:

```
#pragma inline
```

Cel mai potrivit loc pentru a plasa aceasta directiva este inceputul fisierului, compilatorul reluand analiza la atingerea acestei directive. Scopul acestor dirctive este de a micsora timpul acordat compilarii.

**#pragma intrinsic** - rescrie comutatoare ale liniei de comanda sau optiuni ale mediului integrat pt a controla expandarea inline a functiei utilizate in functii proprii. De exemplu

```
#pragma intrinsic strcpy
```

anuleaza expandarea inline a functiei indicate in functii proprii. Atunci cand se doreste expandarea unei functii includeti prototipul functiei inainte de a utiliza functia respectiva.

**#pragma option** - include optiuni ale liniei de comanda in cadrul programului. Sintaxa este:

```
#pragma option optiune
```

Optiunile pot aparea oricat de multe, dar intr-o singura directiva.

**#pragma saveregs** - asigura ca orice functie huge nu va modifica valoarea nici unui registru in momentul apelarii sale. Aceasta directiva este uneori necesara pentru interfatarea cu codul limbajului de asamblare. In plus, pozitionarea sa se va efectua in imediata vecinatate a definitiei functiei in cauza bineinteles inaintea acesteia. Directiva actioneaza asupra primei functii intalnite dupa ea.

**#pragma startup** - permite programatorului sa stabileasca functii ce vor fi apelate chiar inainte de funtia main(). Sintaxa este:

```
#pragma startup nume_functie nr_prioritate.
```

**#pragma warn** - permte redefinirea comenzilor din linia de comanda -wxxx

```
#pragma warn +xxx
#pragma warn -yyy
```