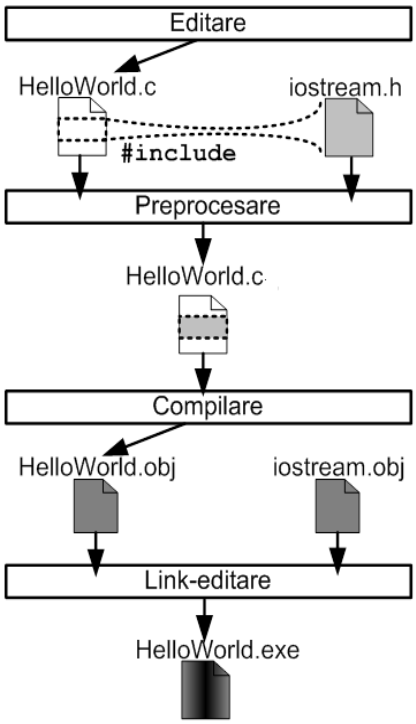
CURS 2. Etapele realizării unui program. Instructiuni preprocesor. Afisarea la ecran si citire de la tastatura

2.1. Etapele realizării unui program



Realizarea unui program presupune implicarea mai multor etape. Aceste etape sunt independente de limbajul de programare utilizat şi implică existenţă câtorva restricţii cu privire la computerul/limbajul utilizat.Etapele realizării unui program sunt:

1. Studierea detaliată a cerinţelor aplicaţiei. Este foarte important ca cerinţele impuse de aplicaţie să fie foarte bine explicitate. Adică înainte de a trece la realizarea unui program pentru o anumită aplicaţie trebuie ca cea aplicaţie sa fie foarte bine analizată şi cerinţele pe care aceasta le impune trebuie să fie complete şi consistente. De exemplu o cerinţă de genul “scrie un program care sa rezolve ecuaţiile” este evident că este incompletă şi se impun întrebări de genul “ce tip de ecuaţii”, “câte ecuaţii”, “care este precizia”, etc.

2. Analiza problemei şi determinarea rezolvării acesteia. In această etapă se decide asupra unei metode de rezolvare a problemei (*algoritm*).

3. Traducerea algoritmului realizat la etapa anterioară într-un limbaj de programare evoluat corespunzător. Forma scrisă a acestui program este denumită *program sursă* (PS, *source program* sau *source code*). În această etapă programul trebuie citit şi verificat pentru a i se stabili corectitudinea. Aceasta se face prin introducerea unui set de valori şi verificarea dacă programul furnizează valorile corespunzătoare corecte. Odată verificat programul este scris într-un anumit limbaj prin intermediul unui Editor.

4. *Compilarea* programului în limbaj maşină. Astfel programul obţinut în limbaj maşină se numeşte cod obiect (*object code).* În această etapă compilatorul poate determina erori de sintaxă ale programului. O eroare de sintaxă este o greşeală în gramatica limbajului. De exemplu în C trebuie ca fiecare linie să se termine cu ; Dacă se uită plasarea ; atunci compilatorul va semnala eroarea de sintaxă. Astfel se repetă compilarea până la eliminarea tuturor erorilor de sintaxă.

5. Programul obţinut în urma compilării, *object code*, este apoi corelat (*linked*) cu o serie de biblioteci de funcţii (*function libraries*) care sunt furnizate de sistem. Toate acestea se petrec cu ajutorul unui program numit *link-editor* (*linker*) iar apoi programul *linked object code*este încărcat în memoria computerului de către un program numit *loader.*

6. Rularea programului compilat, link-editat şi încărcat cu un set de date pentru testare. Astfel se vor pune în evidenţă erorile de logică ale programului. Erorile de logică sunt erori care sunt produse de metoda de rezolvare a problemei. Astfel deşi programul este scris corect din punct de vedere al sintaxei acesta poate executa ceva ce este incorect în contextul aplicaţiei. Poate fi ceva simplu, de exemplu realizarea unei operaţii de scădere în loc de adunare. O formă particulară a erorilor de logică este apariţia erorilor de rulare (*run-time error*). O eroare de rulare va produce o oprire a programului în timpul execuţiei pentru că nu anumite instrucţiuni nu pot fi realizate. De exemplu o împărţire la zero sau încercarea de accesare a datelor dintr-un fişier inexistent.

Astfel se impune ca în această etapă programul să fie reverificat şi apoi erorile să fie recorectate prin intermediul editorului ceea ce impune ca etapele 3,4, şi 5 să fie repetate până la obţinerea rezultatelor satisfăcătoare.

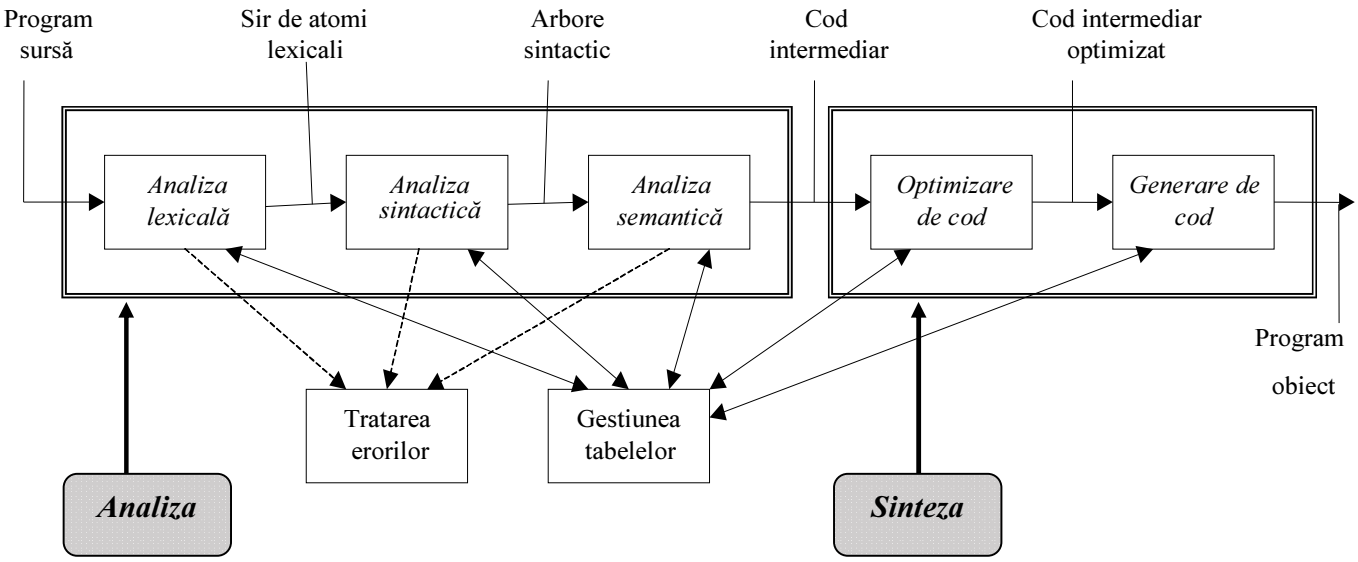
7. Programul poate fi pus în execuţie pentru rezolvarea problemei pentru care a fost conceput. Este posibil ca pe parcursul execuţiei sale să se mai depisteze anumite erori de logică. Astfel se impune reformularea algoritmului şi reluarea etapelor de realizare a programului.

Programul poate fi rulat in mod debug – adică pas-cu-pas pentru a putea verifica execuția punctuală a fiecărei variabile. In acest sens, in mod debug, putem insera breakpoints, iar programul va rula pana la intalnirea acestora, sau putem executa programul pana la linia unde se afla cursorul.
Pentru executia pas cu pas, avem 2 optiuni:

- Step-in – care permite intrarea si execuția pas-cu-pas in funcțiile din linia curentă
- Step-over – care permite executia intr-un singur pas (totul o sigura data) a funcțiilor din linia curentă de cod.

La executia in mod debug avem activată fereastra Watch in care putem solicita afisarea valorilor pentru variabilele din program.

Prezentăm mai jos schema unui compilator.



În procesul de comunicare om-calculator intervine un program intermediar, translatorul, care asigură traducerea programelor scrise de utilizator din cod sursă într-un alt limbaj mai apropiat de calculator. Dacă limbajul ţintă este codul maşină, translatorul se numeşte *compilator.*
Astfel, execuţia unui program sursă se realizează, în cadrul limbajelor compilative, se face în 2 faze:

- compilare, care traduce codul sursă în program obiect
- execuţie, care rulează codul obiect pe calculator, folosind datele iniţiale ale programului şi produce rezultate

Conceptual, compilatorul realizează două mari clase de operaţii: *analiza textului sursă* şi *sinteza codului obiect.*

Compilatorul unui limbaj de asamblare se numeşte asamblor.

În practică, pe lângă compilatoarele obişnuite, există şi alte tipuri de compilatoare.

Astfel, *preprocesoarele* sunt translatoare care traduc dintr-un limbaj de nivel înalt în alt limbaj de nivel înalt.

2.2.Funcţii de intrare/ieşire

Funcţia printf

converteşte, formatează şi tipăreşte argumentele sale la ieşirea standard sub controlul şirului de control. Este o funcție de bibliotecă care realizează ieșiri cu format (in stdio.h).

Formatul general al acestei instrucţiuni este:

```
printf(control,param1, param2, ..., paramn);
```

`control` este un şir de caractere ce conţine 2 tipuri de obiecte:
1)caractere ordinare care sunt simplu copiate în şirul de ieşire
2)semnificaţii de conversie care cauzează conversia şi tipărirea următoarelor argumente succesive ale lui printf.
`param1, param2, …, paramn` sunt expresii ale căror valori se scriu conform specificatorilor de format prezenti in parametrul de control.

Un specificator de format începe prin caracterul % şi se încheie prin caracterul de conversie. Între caracterul % şi caracterul de conversie pot apărea:

- *semnul minus* care specifică cadrarea la stânga în câmp a argumentului convertit

- un *şir de cifre zecimale optional,* care specifică dimensiunea minimă a câmpului în care se editează data. Dacă data necesită mai puţine poziţii decât câmpul descris de acest şir atunci el va fi completat la stânga cu caractere nesemnificative (sau la dreapta dacă s-a folosit semnul minus în specificatorul de format respectiv). Caracterele nesemnificative implicite sunt spaţiile. Dacă şirul de cifre începe cu un zero nesemnificativ, caracterele nesemnificative vor fi zerouri.
- *Un punct opțional* separă şirul de cifre ce defineşte dimensiunea minimă a câmpului de şirul următor de cifre ce semnifică precizia
- un *şir de cifre* ce defineşte precizia care specifică numărul maxim de caractere acceptate dintr-un şir de caractere sau numărul de zecimale care se vor scrie în cazul numerelor reale.

Deci semnificaţiile de conversie au structura:

```
%[-][sir_de_cifre][.][sir_de_cifre]c
```

Caracterul de conversie c poate avea următoarele valori:

Car	Semnificaţie
d	Data se convertește din int în întreg zecimal cu semn
i	Întreg zecimal cu semn
o	Data se convertește din int în întreg octal fără semn
x	Data se convertește din int în întreg hexagesimal fără semn (cu litere mici)
X	Data se convertește din int în întreg hexagesimal fără semn (cu litere mari)
u	Data de convertește din unsigned în întreg zecimal fără semn
c	caracter
s	şir de caractere
e	număr în virgulă flotantă sau flotant dublă precizie şi e convertit în zecimal sub forma: [-] m.nnnnnn e [±]xx. Implicit numărul de zecimale este 6 iar dacă se specifică precizia, numărul de zecimale va fi indicat de aceasta.
E	analog dar se foloseşte E
f	număr în virgulă mobilă simplă sau dublă precizie iar argumentul va fi convertit în notaţie zecimală
g	în acest caz se va folosi forma cea mai scurtă dintre %e şi respectiv %f
G	forma cea mai scurtă dintre %E şi respectiv %f
p	afişează un pointer (o adresă)
n	plasează în argumentul asociat numărul de caractere afişat până în acel moment de funcţia printf

Exemple:

"%5f" arată ca numărul are o lungime de cel puţin 5 caractere
"%05d" determină umplerea cu zerouri a unui număr mai mic de 5 cifre, astfel încât lungimea totală să fie 5
"%.2f" solicită două poziţii după punctul zecimal, dar lungimea lui nu este supusă restricţiilor
"%-5.2f" determină afişarea pe cel puţin 5 poziţii din care exact 2 zecimale aliniat la stânga (datorită prezentei semnului '-')
"%.0f" suprimă tipărirea părţii fracţionare
"%5.10s" determină afişarea unui şir pe cel puţin 5 poziţii dar nu mai mult de 10 caractere
"%%" determină afişarea caracterului %

```
printf("Folosim%n caracterul %%n",&p);    //determină încărcarea în variabila
                                          //pointată de pointerul p a valorii 7 ('Folosim' are 7 litere)
```

Există posibilitatea de a utiliza doi modelatori de format pentru a afişa întregi de tip short şi long. Aceşti doi modelatori sunt h şi l. Se foloseşte %hd pentru a tipări un short int şi %ld pentru a tipări un long int.

Modelatorul L se aplică numerelor în virgulă mobilă permiţând afişarea unui long double. Astfel, se foloseşte L în faţa caracterului de conversie e, f sau g.

Prezenţa modelatorului # în faţa caracterului de conversie g, G, f, e sau E determină afişarea unui punct zecimal chiar dacă nu există cifre zecimale.

Funcţia scanf

Citeşte date de la intrarea standard pe care le converteşte în format intern conform specificatorilor. *Prototipul funcţiei scanf* se găseşte în headerul stdio.h şi are forma:

```
scanf(control,param1, param2, ..., paramn);
```

Funcţia scanf returnează numărul câmpurilor citite sau EOF dacă se întâlneşte prematur marca de sfârşit de fişier (detalii despre EOF la capitolul despre fisiere).
Şirul de control poate conţine 3 tipuri de obiecte: specificatori de format, caractere albe sau alte caractere.
Specificatorul de format este precedat de caracterul % şi specifică tipul valorii ce urmează a fi citite. Aceştia sunt identici cu cei prezentaţi la funcţia printf.

Un caracter alb în şirul de control determină neglijarea unuia sau mai multor caractere albe din intrare. Un caracter care nu este alb determină ca scanf să citească şi să neglijeze caracterul respectiv din intrare.

Exemplu:
`scanf("%d,%d",&a,&b);`//determină citirea unui întreg, apoi caracterul virgulă care se neglijează şi în final citirea unui alt întreg
Dacă caracterul specificat nu se găseşte în intrare, se termină execuţia funcţiei scanf. Toate variabilele a căror valori se citesc cu funcţia scanf trebuie transmise prin adresă. Astfel, pentru variabilele simple se foloseşte operatorul &.
Pentru un şir de caractere citit cu %s se poate scrie doar numele şirului. Un element de tablou precum a[i] se poate citi folosind &a[i] sau simplu folosind a+i.

Datele din intrare trebuie separate prin caractere albe. Caracterele de punctuaţie punct, virgulă şi punct şi virgulă nu sunt considerate separatori. Caracterele albe sunt folosite ca şi separatori dar pot fi citite şi ca orice caractere folosind codul de format %c.

Caracterul * plasat după % şi înainte de litera de format determină citirea datei de tipul respectiv dar nu se asignează nici unei variabile.

Exemplu:
`scanf("%d%*c%d",&x,&y);`
-dacă în intrare vom avea 10/20 va determina citirea în variabila x a valorii 10, citirea caracterului / care nu se alocă nici unei variabile după care variabilei y i se atribuie valoarea 20.

În specificatorul de format poate fi indicată lungimea maximă a câmpului care poate fi citit.

Exemplu:
`scanf("%2d%3d",&x,&y); //`dacă în intrare avem 121234 atunci x=12, y=123
după care următoarea instrucţiune scanf:

`scanf("%d",&z);` //va conduce spre z=4.

Un număr se consideră citit complet când s-a întâlnit un caracter care nu poate participa la scrierea numărului respectiv sau când s-a luat din zonă numărul de caractere specificat în format.

În cazul şirurilor de caractere, scanf citeşte până la întâlnirea primului spaţiu alb. În cazul în care se doreşte citirea unui şir de caractere până la întâlnirea caracterului sfârşit de rând, se va folosi funcţia gets().

Specificatorul %n determină ca funcţia scanf să atribuie variabilei spre care indică argumentul corespunzător numărul de caractere citit din stream-ul de intrare până în momentul întâlnirii specificatorului %n.

O facilitate a funcţiei scanf este aşa-numitul *scanset*. Scanset defineşte o listă de caractere ce vor fi selectate de `scanf` şi memorate într-o variabilă de tip tablou de caractere. Când se întâlneşte un caracter ce nu apare în `scanset`, scanf pune caracterul `null` la sfârşitul şirului de caractere corespunzător şi trece la următorul câmp. Scanset se indică între paranteze drepte. Pentru specificarea unui interval în scanset se poate utiliza caracterul deci [0123456789] este echivalent cu [0-9]. Pentru litere mici şi mari se poate folosi [a-zA-Z], iar [^0-9] înseamnă orice caractere în afara cifrelor.

Exemplu:
```
#include<stdio.h>
int main()
{ char s1[80], s2[80];
  scanf("%[0123456789]%s",s1,s2);
}
```
Efectul acestui program este citirea unui şir de cifre în şirul de caractere s1 iar la întâlnirea primului caracter care nu este cifră se trece la citirea şirului de caractere în s2 până la întâlnirea primului caracter alb.

2.3.Funcţii de citire/scriere caractere

```
int getchar(void)
```

În stdio.h

Efect: Funcţia aşteaptă apăsarea unei taste citind de la intrarea standard codul ASCII al caracterului din poziţia curentă şi returnând codul ASCII al acestuia (tipul valorii returnate este int) sau constanta simbolică EOF dacă s-a întâlnit marca de sfârşit de fişier (perechea CTRL/Z, detalii in cap. destinat fisierelor). Tasta apăsată are automat ecou pe ecran.

*Exemplu:*Se citeste o litera mare din fisierul de intrare si se rescrie ca litera mica.
```
#include <stdio.h>
int main()
{
   putchar(getchar()-'A'+'a');
}
```

Exemplu: Se testeaza daca s-a citit o litera mare si numai in acest caz se aplica transformarea in litera mica. In cazul in care la intrare nu se afla o litera mare, se rescrie caracterul respectiv.
```
#include <stdio.h>
int main()
{
```

```
    int c;
    putchar(((c=getchar())>='A'&&c<='Z')?c-'A'+'a':c);
}
```

```
int putchar(int c)
```

În stdio.h

Efect: afişează caracterul al cărui cod ASCII este egal cu valoarea expresiei dintre paranteze; returnează codul ASCII al caracterului scris la ieşirea standard sau EOF la detectarea unei erori. Se foloseşte pentru a scrie un singur caracter în fişierul standard de ieşire în poziţia curentă a cursorului.

Exemplu:

```
putchar('a');    //trimite la ieşirea standard caracterul 'a'
putchar('\n');   //trimite la ieşirea standard codul '\n' care are ca efect trecerea
                 //cursorului pe linia următoare în prima coloană
putchar('A'+10);//scrie caracterul de cod A + 10 =65+10=75 adica litera K
```

2.4 Compilare si Precompilare

Fisierul este unitatea de compilare. Un program poate fi compus din mai multe fisiere care se compileaza, iar fiecare fisier care se compileaza trebuie sa aiba extensia c. Compilatorul tratează diferit fisierele care au extensia c de fisierele care au extensia cpp. De accea este absolut necesar ca pentru a se invoca compilatorul corect fisierele sa aiba extensia C.
Fiecare fisier al programului este compus din funcţii. Fiecare funcţie se compilează separat şi pentru acestea se generează cod obiect. La compilare se genereaza cod obiect pentru toate functiile din program.
Dupa compilare umeaza linkeditarea care colectezeaza toate codurile compilate ale functiilor utilizate in program (inclusiv a celor referite prin include-uri de funcţii system) si apoi se creaza prima imagine a zonei de memorie de date si cod a programului

Compilatorul C conţine un preprocesor capabil să facă substituţii lexicale, macroinstrucţiuni, compilări condiţionate şi includeri de fişiere.
Prepocesorul execută operaţii anterioare compilării propriu-zise, operaţii ce sunt specificate prin directive preprocesor. Preprocesorul este apelat automat înainte de a începe compilarea.
Preprocesorul limbajului C este relativ simplu şi el, în principiu, execută substituţii de texte.
Prin intermediul lui se pot realiza:

- Includeri de texte;
- Definiţii şi apeluri de macrouri simple;
- Compilare condiţionata.

Acestea au întotdeauna ca şi prim caracter simbolul # şi permit efectuarea de manipulări în textul programului sursă înaintea compilării sale.
Liniile care încep cu simbolul # au o sintaxă independentă de restul limbajului, pot apărea oriunde în cadrul programului sursă (dar se recomandă plasarea la începutul lui) şi au efecte cu remanenţă până la sfârşitul execuţiei programului sau până când o nouă directivă anulează acest lucru. Fiecare din directivele preprocesor trebuie să se găsească pe o linie separată.

Substituţia lexicală se face prin directiva:

```
#define identificator [text]
```

Dacă identificatorul este recunoscut ca element lexical în analiza programului sursă, preprocesorul va înlocui toate apariţiile ulterioare ale identificatorului cu secvenţa text precizată. Dacă textul trebuie să se extindă pe mai multe linii, liniile intermediare vor fi precedate de un caracter '\'. Textul poate fi şi vid ceea ce va avea ca şi efect suprimarea din textul sursă a tuturor apariţiilor identificatorului. Pentru a distinge mai uşor identificatorul substituit, se recomandă folosirea majusculelor.

Exemple:

```
#define void          //suprimarea din textul sursă a tuturor apariţiilor identificatorului void
#define then          //suprimarea tututor apariţiilor identificatorului then
#define begin { //înlocuirea identificatorului begin cu {
#define end }   // înlocuirea identificatorului end cu }
#define N 100   //înlocuieşte identificatorul N cu valoarea 100
```

O dată făcută o substituţie lexicală, ea poate fi folosită ca parte a unei alte substituţii lexicale. Astfel, având ultima substituţie lexicală din exemplul precedent, este corect să scriem:

```
#define NPATRAT N*N
```

Un identificator care a apare deja într-o linie #define poate fi ulterior redefinit ulterior în program cu o altă linie #define. Preprocesorul va substitui în continuare identificatorul cu noua definiţie a lui.
Substituirea nu se face în cazul în care identificatorul apare între ghilimele, în acest caz el nejucând practic rolul unui identificator.

Exemplu:

```
#define PI 3.14
…
printf("valoarea lui PI are un numar infinit de zecimale!\n");
…
```

În acest caz, nu se face expandarea.

Macroinstrucţiuni

Sunt substituţii cu parametri formali şi se realizează cu ajutorul directivei:

```
#define identificator(lista_param_formali) text
```

Parametri formali sunt înlocuiţi de cei din lista parametrilor reali. Nu trebuie să existe nici un spaţiu între identificator şi lista_param_formali.

Exemple:

```
#define max(a,b) (a)>(b)?(a):(b)
#define abs(a) (a)<0?-(a):(a)
```

Se obţine o scriere asemănătoare unui apel de funcţie dar care este tratată mai direct prin înlocuire în textul sursă şi nu printr-un apel de funcţie ceea ce este mai avantajos din punct de vedere al performanţelor de viteză. Macroinstrucţiunile se execută mai repede decât funcţiile lor echivalente deoarece compilatorul nu necesită depunerea pe stivă a parametrilor şi apoi returnarea valorilor. Având în vedere că macroinstrucţiunile măresc dimensiunea codului, este indicat a ne limita doar la calcule simple pentru a nu creşte talia programului.
Se remarcă utilizarea parantezelor rotunde la încadrarea parametrilor formali. Dacă acestea sunt omise, evaluarea se poate face eronat dacă parametri formali sunt substituiţi de expresii.

Exemplu:

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

Efectul utilizării în acest mod a macroinstrucţiunii este eronat deoarece:
9+1%2= =0 va conduce la 9+0 = =0 adică fals ceea ce va genera mesajul “este impar” în loc să obţinem mesajul “este par”.

Cauza este lipsa parantezelor la definirea macroinstrucţiunii. Utilizarea sub forma:
`#define PAR(a) (a)%2= = 0?1:0` va conduce spre un rezultat corect deoarece:
(9+1)%2= = 0 conduce spre 0 = = 0 ceea ce este adevărat iar mesajul generat va fi “este par”.

Includerea fişierelor se face prin una din directivele:

```
#include <nume_fisier>
#include ”nume_fisier”
```

Această directivă are rolul de a solicita compilatorului ca, in faza de preprocesare, acesta să includă în textul sursă în care este folosită această directivă, întregul conţinut al fişierului desemnat, cu scopul de a fi compilate împreună. Este validă prezenţa unei directive include într-un fişier care este apelat la rândul lui printr-o altă directivă include. Standardul ANSI C prevede existenţa unui număr minim de opt niveluri de includere imbricate.

Fişierul este căutat după cum urmează:

a)dacă fişierul se delimitează între paranteze unghiulare se va căuta fişierul în directorul specificat de mediul de programare Dacă fişierul nu este găsit, se continuă căutarea în directoarele standard ale mediului.

b) dacă fişierul se delimitează între ghilimele fişierul este căutat:

b1)Dacă se specifică calea ca şi parte componentă a numelui fişierului doar în acel director se va face căutarea.

b2)Dacă nu se specifică calea, fişierul va fi căutat în directorul curent (cel în care am fost în momentul în care am lansat mediul de programare).

În loc ca programatorul să folosească propriile sale declaraţii pentru funcţiile de bibliotecă, se recomandă includerea în cadrul programului sursă a unor fişiere antet (header), care conţin declaraţiile funcţiilor de bibliotecă apelate. Aceste fişiere au în general extensia .h şi se găsesc în directorul INCLUDE.

Fisierele *include* vor contine doar DECLARATII si nu definitii de funcţii. Declaratiile si definitiile pot fi de functii sau variabile globale. Daca un fisier include contine definitii, este dincolo de controlul programatorului ca aceste fisiere sa apara incluse de 2 ori intr-un proiect, ceea ce genereaza eroare de linkeditare in sensul ca o definitie este gasita (la linkeditare) ca si duplicata.,

Includerea fişierelor se foloseşte pentru gestionarea corectă a unui proiect, în general de mari dimensiuni, care se compune din mai multe module. Pentru încorporarea la începutul fiecărui modul de de fişiere de descriere se pot crea:
-fişiere conţinând definiţii generale ale proiectului: declaraţii de tip şi directive ale preprocesorului, în particular directive define
-fişiere conţinând declaraţii de variabile externe în cazul în care definirea lor a fost regrupată într-un modul special.

Programele C mari pot fi structurate pe mai multe fişiere, care pot fi compilate distinct. Aceasta prezintă mai multe avantaje:
-în cazul erorilor sintactice şi semantice, doar fişierul eronat trebuie recompilat, după modificările sau corecţiile aduse. Programatorul poate profita de faptul că unitatea de compilare este fişierul, repartizând funcţiile programului său în mai multe fişiere. În acest fel, modificarea unei funcţii va fi urmată de recompilarea doar a fişierului care o conţine şi a fişierelor care depind de el, nu a întregului program, obţinându-se astfel un câştig de timp însemnat.
-structurarea programelor pe mai multe fişiere permite simularea mecanismului de încapsulare a datelor. Variabilele şi funcţiile având clasa de memorare static pot fi accesate doar în cadrul fişierului în care au fost definite. Variabilele şi funcţiile declarate cu clasa de memorare extern sunt definite în alt fişier decât cel curent, dar ele pot fi accesate în fişierul curent.
Observaţie: Dacă se modifică un fişier care este inclus printr-o directivă `#include`, atunci toate fişierele care depind de el trebuie recompilate.