Fazele compilatorului de C

Mihai Budiu

ianuarie 1996

Subiect: fazele compilării programelor C

Cuvinte cheie: preprocesor, compilator, linkeditor, modul, modul obiect, header, bibliotecă, fişier executabil

Nivelul tehnic: programator cu 1-2 ani de experienţă

Cunoştinţe necesare: limbajul C, bine

Cuprins

Acest articol face parte dintr-o serie de articole care îşi propune să discute limbajul C standard (ANSI C) supunînd atenţiei felurite aspecte mai puţin înţelese / folosite / explicate. Fiecare articol este complet, în sensul că o cunoaştere a limbajului C ar trebui să fie suficientă pentru înţelegerea lui, şi nu se bazează fundamental pe altele din aceeaşi serie. Datorită însă faptului că în realitate articolele nu prezintă tehnici de programare, ci efectiv limbajul şi semantica (semnificaţia) sa (deci o construcţie unitară), referiri incrucişate între articole apar, atunci cînd aspecte clarificate cu o altă ocazie intervin în discurs.

1 Precizare

Vom folosi pe alocuri termenul *obiect* pentru a denota anumite construcţii ale unui limbaj cum ar fi variabile, constante, funcţii, metode, obiecte (în sensul limbajelor orientate pe obiecte). A nu se confunda această noţiune cu cea de obiect din limbajele orientate pe obiecte, pe care nu o folosim niciodată explicit în acest articol, şi al cărei sens îl socotim mai restrîns.

2 Introducere

Încă de la apariţia sa la începutul anilor '70 limbajul C s-a impus ca lider de necontestat pentru programarea de sistem. Aceasta din urmă cuprinde o clasă largă de programe care interacţionează foarte strîns cu calculatorul şi a căror performanţă o afectează pe a tuturor celorlalte. Un exemplu tipic de astfel de program este sistemul de operare. (El este singurul program care — de exemplu — interacţionează cu discul. Toate celelalte programe cer serviciile lui pentru acest scop, prin funcţiile de lucru cu fişiere. De aceea el "interacţionează strîns cu calculatorul".)

C are mai multe calităţi care îl fac atît de apreciat de către programatorii de sistem (şi nu numai). Una dintre ele, de care ne vom ocupa în parte în acest articol, este suportul pe care îl oferă pentru dezvoltarea de programe *mari*, eventual lucrate în echipă. Vom vedea că acest suport este oferit prin nişte mecanisme extrem de simple, dar a căror existenţă are adesea un impact foarte important asupra limbajului însuşi.

Să vedem acum ce condiţii trebuie să fie îndeplinite de un limbaj pentru ne permite scrierea unor programe mari. Să numim aceste condiţii *premize*, şi să urmărim apoi întruparea lor în cazul C-ului.

3 Premizele scrierii de programe mari

Pentru ca un program să poată fi scris de mai mulţi programatori trebuie ca el să poată fi descompus în părţi independente — cît mai mult — una de alta, care să poată fi concepute, compilate, testate eventual separat.

O categorie de astfel de părţi sunt *bibliotecile*. Unele operaţii foarte utile sunt totdeauna aceleaşi (de exemplu funcţiile matematice (sinus, cosinus, exponenţiale)) şi pot fi scrise odată pentru totdeauna de un programator, fiind apoi folosite de toţi utilizatorii. Aceste funcţii sunt livrate adesea de firme sub forma unor biblioteci de funcţii.

Chiar şi cînd un singur programator este implicat, împărţirea unui program în părţi puţin dependente una de alta este de foarte mare ajutor pentru scrierea, dar mai ales depanarea rezultatului. Studii experimentale au arătat că dificultatea scrierii unui program creşte exponenţial cu lungimea lui. Asta înseamnă ca a scrie un program de 2000 de linii este (mult) mai greu decît a scrie 2 de cîte 1000.

Este deci foarte util să poţi împărţi un program în mai multe texte, fiecare — probabil — într-un fişier separat. Această împărţire se poate face în foarte multe feluri; de preferinţă toate funcţiile / variabilele / procedurile etc. care fac ceva înrudit trebuie sa fie puse în acelaşi fişier. Numele unui astfel de fişier este *modul*. Putem acum enunţa premiza întîi necesară unui limbaj pentru scrierea de programe mari:

PREMIZA 1 : textul unui program poate fi împărţit în mai multe module.

Pentru a putea testa corectitudinea — măcar sintactică — a fiecărui modul, el trebuie să se poată compila separat de celelalte. Reamintim că în cursul compilării toate erorile sintactice sunt depistate (iar cîteodată şi posibile erori conceptuale — de exemplu variabile neiniţializate). Compilarea separată permite corectarea fiecărui modul independent. Avem deci:

PREMIZA 2 : fiecare modul se poate compila separat.

Pe de altă parte, modulele făcînd parte dintr-un tot, este de aşteptat ca unele obiecte (funcţii, proceduri, variabile, etc.) dintr-un modul să se folosească cumva de altele, dintr-un alt modul. Pentru că modulele se pot compila separat, pentru a face posibilă verificarea corectitudinii sintactice a fiecăruia în parte, cumva insuşirile obiectelor dintr-un modul care sunt accesibile şi din altele trebuie descrise modulelor utilizator.

Să numim un obiect (variabilă, funcţie, etc.) al unui modul care poate fi folosit de alte module *exportat* de modulul căruia îi aparţine. El este, prin simetrie, *importat* de modulele care îl folosesc.

Premiza care urmează este necesară pentru întreţinerea uşoară a informaţiilor care trebuie să fie folosite din mai multe locuri:

PREMIZA 3 : Declaraţiile obiectelor exportate de *fiecare* modul trebuie să fie centralizate (strînse într-un singur loc). (*Declaraţia* este o instrucţiune care descrie unele din însuşirile unui obiect; ea va fi tratată mai pe larg mai jos, în contextul limbajului C).

Avem apoi:

PREMIZA 4 : Trebuie să existe o metodă prin care declaraţiile unui modul exportator sunt preluate de modulele importatoare.

Pentru a încheia ne mai trebuie o premiză. Cineva trebuie să combine rezultatele compilărilor diferitelor module separate într-un singur program care se poate executa :

PREMIZA 5 : Trebuie să putem pune împreună mai multe module compilate (separat) pentru a obţine un executabil.

În principiu pentru a obţine programul am putea avea o nouă compilare a tuturor modulelor împreună. Însă din moment ce fiecare modul este probabil deja compilat, ar fi timp cîştigat dacă am putea folosi acest lucru.

Să observăm că remarcile din această secţiune sunt practic independente de limbajul considerat. Ele arată care trebuie să fie calităţile unui limbaj pentru a uşura scrierea unor programe multimodul.

4 Soluţia oferită de C

Compilarea unui program scris în C presupune prezenţa mai multor module. Cazul existenţei unuia singur este unul particular. (În realitate, după cum vom vedea, chiar programele care par formate dintr-un singur modul sunt adesea formate din mai multe: cele scrise de programator plus biblioteci!)

Limbajul C face o distincţie foarte precisă între *declaraţii* şi *definiţii*. O declaraţie este o construcţie care anunţă existenţa unui obiect precum şi anumite trăsături ale sale. *Definiţia* unui obiect construieşte efectiv obiectul (îi alocă un spaţiu). Orice definiţie este simultan şi o declaraţie, dar nu şi invers.

Limbajul C permite exportarea variabilelor şi funcţiilor. C mai permite şi folosirea în comun a macrodefiniţiilor (numite pe scurt *macro*uri) şi numelor de tipuri (introduse cu `typedef`) de către mai multe module. Macrourile şi tipurile nu dau naştere nici unui obiect care să existe în timpul *execuţiei* programului, cum este de exemplu o variabilă. (Cu alte cuvinte macro-urile şi tipurile au numai declaraţii şi niciodată definiţii).

Un exemplu

Pentru a ilustra terminologia să considerăm un program foarte simplu format din două fişiere:

```
/***********************
 * fisierul : header.h *
 ***********************/

#define YES 1
#define NO 0

#define max(a,b) ((a) > (b) ? (a) : (b))
typedef int NUMAR;

NUMAR abs(NUMAR);   /* valoarea absoluta */
extern NUMAR m;

/*********************
 * fisierul : main.c *
 *********************/

#include "header.h"
#include <stdio.h>

NUMAR m;

NUMAR abs(NUMAR x)
{ if (x < (NUMAR) 0) return -x; else return x; }

static NUMAR n;

int main(void)
{
        int y,z;

        /* sarim ceva din program */
        if (max (y,z) < m) /* mai sarim ceva */
}
```

O calitate importantă a obiectelor în C este *vizibilitatea*. Vizibilitatea unui obiect este totalitatea liniilor din program în care el poate fi referit.

Obiectele C care sunt declarate înafara oricărei funcţii se numesc *externe*, iar celelalte se numesc *locale*. În fişierul `main.c` din exemplul nostru sunt externe variabilele `m` şi `n`, precum şi funcţiile `main` şi `abs`.

Obiectele externe au vizibilitate începînd de la declaraţia lor şi pînă la sfîrşitul fişierului cu excepţia blocurilor care declară un alt obiect cu acelaşi nume. Deci `m` este vizibil în funcţiile `abs` şi `main`, dar `n` numai în `main`.

Obiectele declarate în interiorul unui bloc (delimitat de { }) se numesc *locale*. `y` şi `z` sunt locale lui `main`. `x` este local lui `abs`. Un obiect local este vizibil începînd de la locul declaraţiei şi pînă la sfîrşitul blocului în care este declarat (cu excepţia blocurilor interioare care declară un obiect cu acelaşi nume).

Cum arată definiţia unei variabile / funcţii în C presupunem că se ştie. Declaraţia unei variabile se face prefixînd definiţiei ei cuvîntul `extern`. Declaraţia unei funcţii se face ne-scriind blocul care formează corpul ei. Astfel în fişierul `header.h` avem declaraţii (şi nu definiţii) ale variabilei `m` şi funcţiei `abs`.

În C implicit toate obiectele externe dintr-un fişier pot fi exportate. Dacă un obiect extern este folosit numai de modulul în care este definit (adică nu este exportat) atunci definiţia sa se poate prefixa cu cuvîntul `static`. Astfel în fişierul `main.c` `m` este exportat iar `n` nu este! Obiectele care pot fi exportate se numesc cu *legare externă* (external linkage). Asta pentru că un alt modul le poate folosi fără a le defini, legîndu-se la cel în care sunt definite. Pentru modulul `main.c` `m` este cu legare externă.

Se obişnuieşte a se strînge toate declaraţiile obiectelor exportate de un modul (sau un grup de module) într-un fişier numit *header* (sau antet în română). Aceste fişiere nu conţin niciodată definiţii. Numele lor se termină prin convenţie cu `.h`. Un astfel de fişier este în exemplul nostru `header.h`.

5 Compilarea C

Compilarea este formată din trei faze independente, care sunt realizate de obicei de trei programe distincte, care se execută unul după altul sau în paralel.

Exemplele pe care le prezentăm sunt pentru două tipuri de compilatoare : de sub UNIX (practic orice UNIX, testate pentru Linux) sau MsDos (testate cu BorlandC 2.0).

```
module c si          module              module                program
headere              preprocesate        obiect                executabil
 |  ----------------    ------------------   ---------------   ^
 \->| preprocesare | -> | compilare      | -->| linkeditare |---/
    |              |    |   propriu-zisa | /->|  (legare)   |
    ----------------    ------------------ |  ---------------
                                         biblioteci
```

Prin anumite opţiuni putem ruga compilatorul să parcurgă numai unele dintre aceste faze. Cînd compilăm mai multe module, noi dăm o singură comandă, dar această comandă invocă cele trei faze pentru noi. Să presupunem că avem un program format din modulul `main.c` de mai sus şi modulul `error.c`. Sub UNIX putem crea rezultatul cu:

```
cc main.c error.c
```

Cu BorlandC o facem cu comanda

```
bcc main.c error.c
```

(Chiar cînd compilarea este lansată din mediul integrat BorlandC prin tasta F9, sau prin Compile/Build, în realitate acelaşi lucru se întîmplă : mediul integrat invocă pentru noi compilatorul cu o astfel de linie, luînd din project numele fişierelor de compilat !).

Iată evoluţia fişierelor pentru MsDos BorlandC

$$\left.\begin{array}{l}\left.\begin{array}{l} stdio.h \\ header.h \\ main.c \end{array}\right\} \xrightarrow{preprocesare} main.i \xrightarrow{compilare} main.obj \\ \qquad error.c \xrightarrow{preprocesare} error.i \xrightarrow{compilare} error.obj \\ \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad cs.lib \end{array}\right\} \xrightarrow{legare} main.exe$$

După cum vedeţi premiza 1 (mai multe module) este îndeplinită — un program se poate asambla din mai multe fişiere sursă.

Să studiem acum fiecare fază a compilării separat, identificînd şi celelalte premize:

6 Preprocesarea

Preprocesarea este o fază deosebit de interesantă, căreia intenţionăm să-i acordăm un articol special cîndva. Acum o vom privi din punctul de vedere care ne interesează.

Preprocesorul este de obicei un program separat, care se numeşte `cpp` (C PreProcessor). Numele său vine de la faptul că el realizează o procesare înainte (pre) de compilarea propriu-zisă.

Preprocesorul este un program care prelucrează texte. El primeşte un text iar rezultatul lui este tot un text. Comparaţi-l din acest punct de vedere cu celelalte două faze.

În principiu preprocesorul parcurge toate textele care-i sunt specificate de sus în jos, lucrînd pe linii, căutînd comenzi pentru el (numite directive), pe care le execută. Comenzile pentru preprocesor încep cu un semn diez (#) în prima coloană. Preprocesorul C are trei mari grupe de comenzi:

```
#include
#define / #undef
#if / #else / #endif / #ifndef
```

Cea mai interesantă pentru noi acum este **`#include`**, prezentă şi în fişierul `main.c`. Această directivă este urmată de un nume de fîsîer scris între semnele < > sau `" "`. În `main.c` avem

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

Prima formă caută fişierul specificat (stdio.h) într-o serie de directoare care depind de implementarea compilatorului. De obicei în UNIX este vorba de `/usr/lib`, iar la BorlandC există o opţiune “Directories” unde se specifică acest(e) locuri. A doua formă caută fişierul specificat (`header.h`) întîi relativ la acelaşi loc în care se găseşte `main.c` (ex. în directorul curent), iar apoi, dacă a eşuat în căutare, în aceleaşi locuri ca prima formă a lui `include` (cu `< >`).

După cum arată şi numele fişierelor (faptul că se termină cu .h), de obicei (dar nu obligatoriu) acestea sunt fişiere header, care conţin declaraţiile unor obiecte exportate de alte module, importate de modulul curent.

Cînd preprocesorul execută directiva `#include` el găseşte headerul indicat (sau dacă nu, opreşte compilarea raportînd o eroare) şi substituie linia `#include` cu textul headerului. După aceea continuă preprocesarea din acelaşi punct, prelucrînd deci textul tocmai introdus (care poate include alte headere la rîndul lui).

Iată deci cum premiza a 4-a (preluarea într-un modul a fişierelor cu declaraţii) este îndeplinită. De asemenea, fişierele header sunt o consecinţa a premizei a 3-a (centralizarea declaraţiilor).

Puteţi vedea rezultatul preprocesării rulînd comandă `cpp` pe fişierele C. Aceasta realizează numai prima fază a compilării din cele mai sus indicate. (`-P-` indică suprimarea informaţiilor despre numele fişierelor incluse. Rezultatul obţinut este fişierul `main.i`; prin convenţie extensia .i este pentru fişiere preprocesate.)

Borland:

```
cpp -P- main.c
```

sau UNIX

```
cc -E main.c > main.i
```

sau UNIX (dacă aveţi `cpp` în directorul `/lib`) :

```
/lib/cpp main.c > main.i
```

7 Compilarea propriu-zisă

Datorită includerii, fişierele preprocesate conţin deja declaraţiile obiectelor importate. Apoi ele sunt compilate separat (premiza 2). Rezultatul acestei compilări este ceea ce se numeşte *modul obiect (object file)*. Borlandc generează fişiere cu extensia `.obj`, compilatoarele din UNIX cu extensia `.o`. Se obţine cîte unul pentru fiecare modul C iniţial.

Ce este un modul (fişier) obiect? Este un amestec de program compilat (cod maşină) şi variate informaţii. Tot textul C al modulului a fost compilat în cod maşină. Însă referinţele la simboluri externe din alte module nu au putut fi satisfăcute, pentru că aceste module sunt compilate separat. De aceea modulul obiect conţine o listă a simbolurilor nesatisfăcute, indicînd şi locurile unde sunt folosite. În plus mai conţine şi o listă a simbolurilor exportate de modul.

De asemenea un modul obiect are o parte cu informaţii numite *de relocare*. Acestea sunt necesare pentru că toate salturile din codul compilat sar la nişte adrese care se vor schimba probabil atunci cînd modulul obiect va fi pus cap la cap cu altele pentru a forma executabilul.

Adesea modulul obiect mai conţine şi alte informaţii care sunt necesare pentru depanarea programului rezultat. Aceste informaţii arată căror fişiere sursă şi căror linii (C) le corespund feluritele instrucţiuni din codul maşină.

Cîteodată faza de compilare propriu-zisă este formată din două părţi independente şi făcute de programe separate. O primă fază generează un program în limbaj de asamblare, iar a doua fază asamblează programul pentru a obţine un fişier obiect.

```
  module      programe in      module
preprocesate  asamblare         obiect
---|  |------------|  |------------|  |----
   |->| compilare  |->| asamblare  |->|
   |  |            |  |            |  |
---|  |------------|  |------------|  |-----
        \__________________________/
          compilare propriu-zisa
```

Puteţi opri compilarea la module obiect invocînd compilatorul cu opţiunea `-c` (compile only); iată cum puteţi compila cele două surse la module obiect :

UNIX :

```
cc -c error.c main.c
```

BorlandC

```
bcc -c error.c main.c
```

Fişierul obiect nu arată prea interesant — pentru cititorul uman — dar puteţi încerca să opriţi compilarea la limbaj de asamblare:

UNIX

```
cc -S error.c main.c
```

BorlandC

```
bcc -S error.c main.c
```

Rezultatele sunt fişiere cu extensia `.s` în UNIX sî `.asm` în DOS. Citiţi-le! Aceste fişier pot fi asamblate pentru a obţine cod obiect prin

UNIX

```
as -c error.c main.c
```

BorlandC

```
tasm error.c main.c
```

8 Linkeditarea

După cum am văzut fiecare modul sursă este compilat separat într-un modul obiect. Modulele obiect au o mulţime de referinţe nesatisfăcute (la obiectele importate). Legarea modulelor (implicînd completarea referinţelor obiectelor globale, relocarea codului) este făcută de ultima fază a compilării, numită *linkeditare*. (Link = legătură în engleză). Ea este făcută de programul `ld` (în UNIX) sau `tlink` (BorlandC). Argumentele ei sunt module obiect.

Interesant este că aceste module obiect pot fi obţinute prin compilarea unor programe scrise în limbaje diferite; este frecventă combinarea programelor scrise în limbaj de asamblare cu a celor în C: toate sunt compilate la module obiect, după care sunt legate împreună. Lucrurile nu sunt foarte simple, pentru că trebuie respectate anumite convenţii (de exemplu compilatorul C se poate aştepta ca anumiţi regiştri să nu fie schimbaţi de nici o funcţie).

Puteţi încerca să legaţi modulele obiect obţinute mai sus astfel, deşi n-o să meargă!

UNIX :

```
ld main.o error.o
```

BorlandC :

```
tlink main.obj error.obj
```

Iată de ce: în realitate cînd linkerul este invocat de către compilatorul de C (deci nu separat, cum am făcut noi mai sus), mai primeşte ca argumente, pe lîngă modulele obiect generate prin fazele anterioare ale compilării şi cel puţin o *bibliotecă*. Este vorba de biblioteca standard C, numită în UNIX `libc.a`, iar în DOS `cs.lib` (în locul lui s poate fi o altă literă care depinde de modelul de memorie pe care l-aţi folosit). Erorile obţinute prin comanda de mai sus arată care sunt simbolurile externe importate din bibliotecă şi care au lipsit la legare.

Pentru a lega modulele trebuie specificate şi bibliotecile. Sintaxa la BorlandC pentru linker e puţin mai complicată :

BorlandC :

```
tlink c0s main error, main, ,cs
```

pentru că se specifică separate de virgule fişierele `.obj` (incluzînd `c0s`, un fişier standard), numele rezultatului, fişierele map (informaţii generate de compilator), apoi bibliotecile, şi apoi resursele (aici map şi resursele lipsesc).

Sub UNIX lucrurile pot fi foarte complicate, pentru că bibliotecile se pot afla în multe locuri. Numele bibliotecilor este dat linkerului este precedat de opţiunea `-l`; linkerul unui nume specificat îi prefixează `lib`, sufixîndu-i `.a`. Astfel `-lc` leagă biblioteca `libc.a`. O linie minimă de comandă pentru linker ar fi (dar în funcţie de sistem mai trebuie specificate şi alte opţiuni, ca `-L` care indică locul bibliotecilor) :

```
ld /usr/lib/crt0.o main.o error.o -L/usr/lib -lc
```

Fişierele `c0s` (Borland), respectiv `crt0.o` (UNIX) care apar mai sus sunt nişte module obiect care conţin codul unei funcţii cu care se începe execuţia programului, care va cheama funcţia `main()` şi care după executarea acesteia cheamă funcţia `exit()`, pentru a termina civilizat programul.

O bibliotecă (cs.lib respectiv libc.a sunt biblioteci) este de fapt o colecţie de module obiect care au fost compilate mai demult din sursele lor şi au fost strînse laolaltă într-un fişier arhivă. În libc.a (cs.lib) se află codul funcţiilor standard pe care programul dumneavoastră le poate folosi. De exemplu printf, exit, fopen, scanf. Prin convenţie declaraţiile acestor funcţii sunt grupate în fişiere header puse într-un anumit loc în sistemul de fişiere. Acestea sunt headerele incluse prin directiva #include <nume_header.h>! (De aceste headere este nevoie pentru că ele conţin declaraţille funcţiilor exportate de bibliotecă).

Şi dumneavoastră puteţi crea biblioteci folosind programul ar (în UNIX) sau tlib (în DOS). Aceste programe manipulează arhive (biblioteci) în multiple feluri. Bibliotecile pot fi explorate (pentru a vedea ce conţin) cu tdump (în DOS) sau cu nm, ranlib, objdump, etc. sub UNIX. O discuţie acestor utilitare este însă în afara acestui articol.

Rezultatul obţinut de la linker este în fine un program executabil sub sistemul de operare pentru care lucrează linkerul. Un astfel de fişier are însă o structură destul de complicată, folosită de sistemul de operare pentru două scopuri: el conţine informaţii despre structura însăşi a fişierului executabil (cît de lung e codul, cît de lungi sunt datele, sistemul de operare pentru care merge, dacă foloseşte biblioteci încărcate dinamic, eventuale date de relocare, etc.) şi informaţii de depanare, care sunt folosite cînd programul este depanat (de ex. cu td = Turbo Debugger în DOS sau cu dbx în UNIX). Aceste informaţii sunt preluate din fişierele obiect, dacă existau, şi în principiu indică din ce surse C provin instrucţiunile din executabilul final. Aceste informaţii pot fi scoase din program, dacă nu mai este nevoie ca el să fie depanat, cu tdstrip (DOS) sau strip (UNIX), pentru a face un executabil mai mic.

Două cuvinte despre *linkeditarea dinamică*. Aceasta este o tehnologie relativ nouă, prezentă în sistemul de operare Windows sub forma fişierelor-biblioteci cu extensia .dll (Dinamic Linking Libraries). Sub UNIX aceste biblioteci se numesc fie dll-uri fie Shared Libraries (biblioteci comune; numele lor conţine particula so; exemplu: lib.so.4.1). Ideea este destul de simplă. Există o clasă mare de funcţii care tind să fie prezente în majoritatea programelor. Un exemplu tipic este printf: cam toate programele au ceva de scris. Atunci în loc să se lege (linkeze) codul acestor funcţii în fiecare program, funcţiile sunt "încărcate" în memorie o singură dată de sistemul de operare, şi apoi legate de toate programele care au nevoie de ele. De aici cele două denumiri ale acestor biblioteci: dinamice pentru că legarea nu mai e o fază a compilării, ci chiar a executării programului (legarea se face cînd programul este pornit) şi comune (shared) pentru că pot aparţine mai multor programe diferite simultan!

9 Concluzii

Limbajul C pune la dispoziţie nişte mecanisme excelente pentru scrierea de programe mari. Aceste mecanisme permit

- scrierea unui program din mai multe bucăţi, care se pot compila separat.
- reutilizarea bucăţilor în programe diferite sub forma bibliotecilor.
- o gestiune relativ simplă a schimbărilor dintr-un modul datorită headerelor care grupează declaraţiile obiectelor exportate.

Cunoaşterea fazelor compilării este esenţială pentru scrierea unor programe corecte în C. Dacă un obiect este folosit fără a fi declarat, se obţin erori, sau codul generat este incorect! (Vom reveni asupra acestui aspect într-un articol despre prototipuri de funcţii). Asta poate da naştere la multe bug-uri misterioase. Este clar că un modul nu poate şti nimic despre obiectele globale (adică cu legare externă) definite în alte module pînă în clipa legării. Ori pînă la legare anumite informaţii se pierd datorită compilării (de ex. anumite informaţii de tip). Linker-ul ar putea să nu depisteze anumite erori (de exemplu folosirea unui acelaşi obiect cu tipuri diferite în module diferite).

Lecţia se poate rezuma într-o propoziţie : *nu folosiţi niciodată în C funcţii sau variabile nedeclarate!* Puteţi folosi sau nu pentru declaraţii headere.

10 Glosar

Acest articol (şi toate din această serie) folosesc o terminologie standard, foarte precisă. Iată semnificaţiile cuvintelor cheie din acest text (şi numele lor englezesc) :

asamblare: compilare din limbaj de asamblare în cod-maşină; ASSEMBLY.

ANSI: American National Standards Institute; o foarte prestigioasă instituţie americană care se ocupă cu crearea standardelor. În această serie de articole noi discutăm limbajul C aşa cum a fost el standardizat de ANSI.

bibliotecă: o colecţie de MODULE OBIECT grupate laolaltă. Conţine codul unor funcţii / variabile foarte utile; în C se consideră ca biblioteca include şi HEADERELE cu declaraţile obiectelor exportate de biblioteca; LIBRARY.

compilare: transformarea unui program dintr-un limbaj într-altul; adesea rezultatul este în cod-maşină; COMPILATION.

declaraţie: o construcţie (de program) care anunţă existenţa unui obiect fără însă a-l construi (aloca spaţiu) pentru el; DECLARATION.

definiţie: o construcţie (de program) care construieşte (alocă spaţiu) pentru un obiect; DEFINITION.

exportat: (de un modul) un obiect care este DEFINIT în acel modul dar care poate fi folosit şi în altul; EXPORTED.

extern: (în C) orice obiect care nu este LOCAL unei funcţii. În C variabilele declarate înafara corpului unei funcţii şi toate funcţiile sunt externe; EXTERNAL

header: un fişier care grupează declaraţiile obiectelor exportate de unul sau mai multe module (sursă, obiect sau biblioteci); HEADER FILE.

importat: (de un modul) un obiect folosit de acel modul dar DEFINIT în altul; IMPORTED.

linkeditare: combinarea mai multor module obiect într-unul singur (executabil) prin rezolvarea referinţelor la obiectele NEDEFINITE pentru fiecare din modulele componente; LINKING.

local: un obiect este local unui bloc dacă este vizibil numai în acel bloc. Aşa sunt în C argumentele funcţiilor şi variabilele declarate în funcţii; LOCAL

modul: o parte dintr-un program — un singur fişier, grupînd de obicei obiecte (variabile, funcţii, etc). folosite împreună; MODULE; TRANSLATION UNIT

modul obiect: un modul compilat dar cu referinţe la obiecte din alte module încă nerezolvate (referă obiecte cu adrese necunoscute în acel modul); OBJECT MODULE (OBJECT FILE)

preprocesare: o fază anterioară compilării, care prelucrează textul programului sursă C, obţinînd o nouă sursă C; PREPROCESSING

program executabil: un program care poate fi direct executat de către sistemul de operare; EXECUTABLE (BINARY) FILE.

vizibilitate: a unui obiect — porţiunea din program în care acel obiect poate fi referit; SCOPE