10. ELABORAREA PROGRAMELOR COMPLEXE

Comparativ cu exemplele prezentate, programele aplicative de calcul destinate rezolvării unor probleme concrete sunt în general programe de mari dimensiuni cu un grad ridicat de complexitate. Elaborarea unor astfel de programe în maniera plasării tuturor instrucțiunilor într-un singur fişier-sursă, compilarea acestuia şi apoi crearea fişierului executabil prezintă numeroase dezavantaje. În primul rînd, datorită dimensiunilor mari, programul este greu de urmărit şi de înțeles, iar eliminarea erorilor sintactice şi logice este cel mai adesea imposibil de realizat. Un alt dezavantaj rezultă din faptul că orice modificare survenită în cadrul fişierului sursă va necesita recompilareea întregului program. În sfîrşit, prin această manieră de programare suntignorate principiile structurării şi modulizării programelor de calcul, precum şi posibilitatea lucrului în echipă la realizarea unui program de aplicație complex.

10.1. COMPILĂRI SEPARATE

Înainte de a trece la scrierea primei instrucțiuni din cadrul oricărui program de calcul, este necesară mai întîi specificarea funcțiilor acestuia şi a structurilor de date cu care va lucra. Specificarea funcțiilor se realizează pe principiul "top-down" (de sus în jos) care se bazează pe faptul că orice program de calcul,simplu sau complex, se compune dintr-o funcție principală care apelează alte funcții. La rîndul lor, aceste funcții apelează alte funcții şi aşa mai departe pînă cînd funcția apelată devine o funcție elementară şi nu mai necesită apelul altor funcții. Specificarea oricărei funcții trebuie să includă scopul acesteia, parametrii pe care îi va primi, tipul vatorii pe care o va returna şi funcțiile pe care le va apela.

Programele de calcul complexe vor conține numeroase funcții care, în scopul modulizării, pot fi plasate în diferite fişiere. Mediul de programare Turbo C++/Borland C oferă posibilitatea compilării separate a mai multor fişiere sursă şi apoi link-editarea acestora în vederea obținerii programului executabil. În acest sens, după ce au fost create fişierele sursă se mai crează încă un fişier, numit **fişier proiect**, care are extensia "prj" şi în care vor fi incluse numele tuturor fişierelor sursă ce alcătuiesc programul de calcul.

Crearea şi actualizarea (modificarea) fişierelor proiect se realizează utilizînd opțiunile meniului "**Project**". Astfel, pentru a crea un nou fişier proiect, se utilizează opțiunea "**Open project**".

La deschiderea fişierului proiect mediul Turbo C++/Borland C deschide fereastra "**Project**", care devine fereastră activă. Dacă s-a creat un fişier proiect nou,conținutul acestei ferestre este vid,iar dacă s-a selectat un fişier proiect existent,atunci în fereastră sunt afişate numele fişierelor sursă conținute în acesta.

Modificarea conținutului fişierului proiect se poate realiza fie cu ajutorul comenzilor meniului "**Project**", fie cu ajutorul tastelor <Del> şi <Ins>. Astfel, pentru a insera în fişierul proiect numelor fişierelor sursă se apasă <Ins> ("**Add item**"). Pe ecran este afişată fereastra "**Add to project list**" prin intermediul căreia se aleg numele fişierelor sursă necesare. Pentru eliminarea unui fişier susră din fişierul proiect se utilizează tasta <Del>.

Pentru a exemplifica modul de lucru cu mai multe fişiere, considerăm că dorim realizarea unui program de calcul cu ajutorul căruia să efectuăm operațiile aritmetice a numerelor complexe. Pentru realizarea acestui program parcurgem următoarele etape:

Etapa 1. Definirea funcțiilor programului pe principiul "top-down".

Programul de calcul se va numi Oper_cmp şi va conține următoarele 7 funcții:

- funcția princpală **main**( );
- funcția **meniu** care are rolul de a afişa pe ecran un meniu din care se va selecta operația dorită şi de a returna funcției principale varianta selectată;
- funcția **citire** destinată citirii celor două valori complexe pe care le primeşte ca argumente;
- funcția **adun** furnizează suma a două valori complexe pe care le primeşte ca argumente;
- funcția **difer** furnizează diferența numerelor complexe, primite ca argumente;
- funcția **produs** furnizează produsul numerelor complexe, primite ca argumente;
- funcția **raport** furnizează rezultatul împărțirii celor două numere complexe, primite ca argumente.

Funcția principală va apela funcția **meniu** şi în rapord de varianta selectată de utilizator va decide fie continuarea programului prin apelul funcției **citire** şi al funcției corespunzătoare operației selectate, fie oprirea execuției şi redarea controlulului sistemului de operare folosind funcția **exit** definită în fişierul antet <dos.h>.

Etapa 2. Definirea structurii de date şi a variabilelor globale.

Pentru efectuarea operațiilor cu numere complexe, în limbajul C este necesară definirea tipului de date complex. În acest sens, se va crea un fişier antet care va conține descrierea structurii de date de tipul complex. Programul va utiliza variabila globală *rezultat*, de tipul complex, pentru a transmite funcției principale rezultatul obținut în urma operației efectuate.

Etapa 3. Editarea fişierului antet şi a fişierelor sursă.

Fişierul antet se va numi **complex.h**, iar pentru specificare considerăm că funcția principală şi funcțiile **meniu** şi **citire** sunt grupate în acelaşi fişier numit **oper_com.c**, în timp ce funcțiile **adun**, **difer**, **produs** şi **raport** sunt grupate în fişierul **operatii.c**. La această etapă fiecare fişier avînd extensia C poate fi compilat separat în vederea eliminării eventualelor erori sintactice.

Conținutul fişierelor complex.h, oper_com.c şi operatii.c este prezentat în programul din exemplul 10.1.

Exemplul 10.1.

```
/* continutul fisierului complex.h */
typedef struct {
        float re, im;
} complex;
/* continutul fisierului oper_com.c */
#include <process.h>
#include <conio.h>
#include <stdio.h>
complex rezultat;
int meniu (void);
complex citire (void);
```

```
void adun (complex,complex);
void difer (complex,complex);
void produs (complex,complex);
void raport (complex,complex);
void main (void) {
        complex z1, z2;
        int oper;
        while(1)
        {
                oper=meniu();
                if(oper==0) exit(0);
                clrscr();
                gotoxy(30,5);           cputs("z1=");      z1=citire();
                gotoxy(30,6);           cputs("z2=");      z2=citire();
                switch(oper)
                {
                    case 1:
                                adun(z1,z2);
                                gotoxy(30,7); cprintf("z1+z2=(%f)+i (%f)" , rezultat.re, rezultat.im); getch();
                    break;
                    case 2:
                                difer(z1,z2);
                                gotoxy(30,7); cprintf("z1-z2=(%f)+i (%f)" , rezultat.re, rezultat.im); getch();
                    break;
                    case 3:
                                produs(z1,z2);
                                gotoxy(30,7); cprintf("z1*z2=(%f)+i (%f)" , rezultat.re, rezultat.im); getch();
                    break;
                    case 1:
                                raport(z1,z2);
                                gotoxy(30,7); cprintf("z1/z2=(%f)+i (%f)" , rezultat.re, rezultat.im); getch();
                    break;
                }
        }
}
int meniu() {
        int oper; clrrscr();
        gotoxy(20,5); cputs("SIMULAREA OPERATIILOR CU NUMERE COMPLEXE");
        gotoxy(30,7);       cputs("1 Adunare");
        gotoxy(30,8);       cputs("2 Scadere");
        gotoxy(30,9);       cputs("3 Produs");
        gotoxy(30,10);      cputs("4 Impartire");
        gotoxy(30,11);      cputs("0 Stop");
        gotoxy(25,13);      cputs("Selectati operatia dirita > ");          scanf("%d", &oper);
        return(oper);
}
complex citire() {
        complex z;
        scanf("%f %f", &z.re, &z.im);
        return(z);
}
/* continutul fisierului operatii.c */
#include <process.h>
#include <conio.h>
#include "complex.h"
extern complex rezultat;
void adun (complex z1,complex z2) {
        rezultat.re = z1.re+z2.re;
        rezultat.im = z1.im+z2.im;
}
void difer (complex z1,complex z2) {
        rezultat.re = z1.re-z2.re;
        rezultat.im = z1.im-z2.im;
}
void produs (complex z1,complex z2) {
        rezultat.re = z1.re*z2.re;
        rezultat.im = z1.im*z2.im;
}
```

```
void raport (complex z1,complex z2) {
        double modul;
        modul = z2.re*z2.re + z2.im*z2.im;
        if (modul == 0) {
                rezultat.re = (z1.re*z2.re + z1.im*z2.im)/modul; rezultat.im = (z1.im*z2.re - z1.re*z2.im)/modul;
        }
}
```

Etapa 4. Crearea fişierului proiect.

Se selectează din meniul opţiunii "**Project**" comanda "**Open project**", iar la solicitarea numelui proiectului se va tasta *Oper_com.prj*. Din lista fişierelor afişate se selectează fişierele *oper_com.c* şi *operatii.c*.

Etapa 5. Crearea fişierului executabil.

Pentru crearea programului executabil şi lansarea acestuia în execuţie se apasă simultan tastele <Ctrl>+<F9>. În urma acestei acţiuni se compilează fişierele-sursă, se efectuează link-editarea şi, dacă nu apar erori,se lansează programul în execuţie. Meniul integrat Turbo C++/Borland C, în urma lansării comenzii <Ctrl>+<F9>, va analiza dependenţele dintre fişierele ce alcătuiesc proiectul şi va efectua numai operaţiile necesare. De exemplu, dacă fişierele conţinute în proiect au fost compilate separat şi ulterior nu s-au efectuat modificări în acestea, atunci se va trece direct la faza de link-editare. Dacă, însă, într-unul dintre fişierele sursă s-au efectuat modificări, atunci, înainte de a trece la link-editare, mediul de programare va lansa comanda de compilare numai a fişierului modificat.

Fişierul executabil va avea numele fişierului proiect şi extensia **exe** şi poate fi creat cu ajutorul comenzilor "**Make exe file**" sau "**Bild all**" din submeniul opţiunei "**Compile**". Deosebirea dintre aceste comenzi constă în faptul că, în timp ce comanda "**Make exe file**" va analiza dependenţele şi va compila doar fişierele modificate, comanda "**Bild all**" va determina recompilarea tuturor fişierelor sursă din proiect şi apoi va crea programul executabil.

Etapa 6. Închiderea sesiunii de lucru.

Pentru închiderea sesiunii de lucru, mai întîi se închide fişierul proiect cu ajutorul comenzii "**Close project**".

10.2. Clase de memorare

În limbajul C fiecare variabilă posedă o caracteristică, numită **clasă de memorare**, care se referă pe de o parte la durata de existenţă a variabilei sau durata de viaţă, iar pe de altă parte la vizibilitatea sau scopul variabilei.

10.2.1. Durata de existenţă a variabilelor

Durata de existenţă, numită şi **durata de viaţă** a unei variabile, reprezintă intervalul de timp, din cadrul timpului de execuţie al programului, în care aceasta are alocat un spaţiu de memorie şi poate memora o anumită valoare.

Din punctul de vedere al duratei de existenţă, variabilele din cadrul programelor C pot fi împărşite în următoarele 2 categorii:

- *Variabile de tipul automatic*. Aceste variabile sunt cele mai utilizate în cadrul programelor scrise în limbajul C şi cuprind toate variabilele declarate în interiorul perechii de acolade care delimitează corpul funcţiilor. Ele sunt create (li se alocă spaţiu în memorie) atunci cînd este apelată funcţia în care sunt declarate şi sunt distruse (se eliberează spaţiul de memorie ce le-a fost alocat) atunci cînd funcţia se termină şi redă controlul funcţiei apelante. De exemplu, în cadrul următoarei secvenţe de program:

```
func() {
        int alfa;
        auto int beta;
        register gama;
}
```

Toate cele 3 variabile sunt de tipul *automatic* deoarece sunt create în momentul în care este apelată funcţia **func** ş sunt distruse atunci, cînd aceasta se termină şi redă controlul funcţiei care a apelat-o. Variabila *alfa* este în mod implicit variabilă automatic, iar *beta* a fost declarată explicit variabilă automatic folosind cuvîntul cheie **auto**. Efectul instrucţiunii *auto int beta;* este identic cu cel al instrucţiunii *int beta;*, dar în anumite situaţii, pentru a evita confuziile, se preferă declararea în mod explicit a variabilei de tipul automatic. Variabila *gama* este un tip special de variabilă automatic, numită variabilă **registru**, căreia compilatorul îi va atribui ca adresă unul dintre registrele microprocesorului. Utilizarea variabilelor de tipul registru prezintă avantajul că măreşte considerabil viteza de execuţie a progrmului, dar este limitată de numărul registrelor libere.

- *Variabile statice şi variabile externe*. Dacă dorim ca o variabilă declarată în cadrul corpului unei funcţii să nu fie distrusă în momentul în care funcţia se termină, atunci în faţa cuvîntului cheie care defineşte tipul variabilei se va folosi prefixul **static**. De exemplu:

```
func()
    {
    int delta;
    static int kapa;
    }
```

În acest exemplu, variabila *delta* ca şi variabila *kapa* sunt recunoscute numai în interiorul funcţiei **func**. Doesebirea între cele 2 tipuri de variabile constă în faptul că în timp ce variabila *delta* va fi distrusă în momentul terminării funcţiei, variabila *kapa* va exista şi după acest moment, astfel încît la o nouă revenire în funcţie, valoarea pe care a memorat-o anterior va fi regăsită şi deci va putea fi reutilizată.

O altă modalitate de a păstra valoarea unei variabile pe tot timpul execuţiei unui program o constituie declararea variabilei în afara corpului oricărei funcţii. O astfel de variabilă se numeşte **variabilă externă** şi are, ca şi variabilele statice, durata de existenţă egală cu durata de execuţie a programului.

Este recomandabil, ca în cadrul programelor de calcul complexe să se evite pe cît este posibil utilizarea variabilelor de tipul static sau extern şi să se folosească, cu preponderenţă, variabile de tipul automatic, care permit o utilizare mult mai eficientă a spaţiului de memorie disponibil.

10.2.2. Vizibilitatea variabilelor

Vizibilitatea unei variabile, sau scopul acesteia, este o caracteristică ce defineşte părţile unui program de calcul care vor putea recunoaşte variabila respectivă. Astfel, o variabilă poate fi recunoscută în interiorul unui bloc, al unei funcţii, al unui fişier, al unui grup de fişiere sau în tot programul.

Variabilele de tipul automatic şi cele statice sunt recunoscute numai în interiorul corpului funcţiei în care au fost declarate, motiv pentru care se mai numesc şi **variabile locale**. În interiorul unei funcţii, vizibilitatea variabilelor locale poate fi resdtrînsă la un bloc de cod care constituie un set de instrucţiuni grupate în interiorul unei perechi de acolade.

În următoarea secvenţă de cod

```
func()
    {
    int alfa;
     {
     float beta;
     }
    }
```

variabila *alfa* este recunoscută în tot corpul funcţiei, iar variabila *beta* este recunoscută doar în interiorul blocului definit de a doua pereche de acolade.

Variabilele **externe** sunt vizibile în toate funcţiile definite după instrucţiunea dedeclarare a acestora. De exemplu, în secvenţa de cod următoare:

```
    int alfa;
main()
    {
    . . .
    }
            func
    {
    . . .
    }
```

variabila externă *alfa* este vizibilă atît în funcţia **main**, cît şi în funcţia **func**. Dacă modificăm poziţia instrucţiunii de declarare a variabilei *alfa* ca în secvenţa următoare:

```
main()
{
. . .
}
int alfa;
func ()
    {
. . .
    }
```

atunci variabila *alfa* va fi vizibilă doar în corpul funcţiei **func**.

În cazul cînd funcţiile unui program sunt plasate în cadrul unor fişiere sursă distincte Limbajul C oferă posibilitatea ca o variabilă externă declarată într-un fişier să fie vizibilă şi în alte fişiere. în acest sens, în fişierele în care se deoreşte ca variabila să fie recunoscută se va insera o instrucţiune de declarare a acesteia avînd prefixul **extern**. Acesta indică faptul că variabila a fost declarată în alt fişier şi că va fi recunoscută de funcţiile conţinute în fişierul curent.

Acest mecanism a fost utilizat în cadrul exemplului 10.1, în care variabila *rezultat*, declarată în fişierul **oper_com.c** pentru a fi recunoscută şi de funcţiile conţinute în fişierul **operatii.c**, a fost declarată şi în acest fişier, dar folosind prefixul **extern**.

În tabelul 10.1 sunt prezentate durata de existenţa şi vizibilitatea variabilelor utilizate în limbajul C, în funcţie de modul şi locul în care sunt declarate.

Tabelul 10.1

Locul declarării variabilei	Modul de declarare	Durata de existenţă	Domeniul de vizibilitate
Corpul unei funcţii	declararea obişnuită a tipului precedată sau nu de prefixul auto	funcţia	corpul funcţiei
	register	funcţia	corpul funcţiei
	static	programul	corpul funcţiei
În afara corpului oricărei funcţii	declararea obişnuită a tipului fără prefix	programul	fişierul în care au fost declarate
	declararea obişnuită a tipului precedată de prefixul static	programul	fişierul în care au fost declarate
	declararea obişnuită a tipului precedată de prefixul extern	programul	mai multe fişiere

10.3. Modele de memorie şi tipuri de pointeri

Memoria internă a calculatoarelor personale compatibile IBM este organizată în celule de cîte 8 biţi, numite octeţi, fiecare avînd asociat un cod, memorat pe 20 de biţi, numit **adresă fizică**.

Microprocesoarele din familia Intel 80x86 funcţionînd în mod "real" pot adresa un spaţiu real de memorie de pînă la 1MO împărţit în blocuri de 64 kO, numite **segmente**. În acest sens se foloseşte conceptul de **adresă logică**, care are la bază tehnica segmentării memoriei şi faptul că adresa fizică a fiecărui segment de 64 kO este un muliplu de 16.

O adresă logică constă din următoarele două componente memorate fiecare pe 16 biţi:

- **adresa_segment** conține valoarea adresei fizice a primului octetdin segment împărțită la 16, adică valoarea obținută prin eliminarea din valoarea adresei fizice a aultimilor 4 biți (biți mai puțin semnificativi);

- **deplasamentul de segment** numit **offset**, în care se memorează poziția unui octet din segment față de primul octet al acestuia.

Pentru adresa logică se folosește notația:

adresa_logică = adresă_segment : offset

iar pentru a obține valoarea adresei fizice, cunoscînd adresa_segment şi offsetul, se foloseşte relația:

adresa_fizică = 16·adresă_segment + offset

De exemplu, dacă adresa logică exprimată în hexazecimal este 1ED7:0536, atunci adresa fizică ce îi corespunde este 16·1ED7+536 = 1ED70 + 536 = 1F3A6.

Pentru efectuarea calculelor s-a ținut cont că înmulțirea cu 16 este echivalentă cu o translatare la stînga cu 4 biți, adică, cu adăugarea unui 0 la sfîrşitul valorii hexazecimale multiple.

Constatăm faptul, că pentru o adresă fizică, a cărei valoare este unică, pot exista mai multe adrese logice.

Programele de calcul scrise în limbajul C++ conțin următoarele categorii de informații:

- **codul executabil**, memorat într-un segment de memorie, numit **segmentul de cod**;
- **datele statice**, memorate pe toată durata de execuție a programului, într-un segment de memorie, numit **segmentul de date**;
- **datele automatice**,memorate pe durata lor de existență într-un segment de memorie,numit **stivă** sau în registrele microprocesorului;
- **datele dinamice**, pentru care se alocă sau se eliberează spațiul de memorie necesar într-o zonă special rezeratăm, numită **heap**.

Pentru memorarea adreselor celor 4 segmente de memorie prezentate anterior, microporcesorul dispune de următoarele 4 registre specializate:

- **registrul CS** (Code Segment), destinat memorării adresei segmentului de cod. Pentru memorarea offsetului în segmentul de cod este utilizat registrul **IP** (Instruction Pointer) care este incrementat în mod automat, astfel că adresa logică CS:IP indică întotdeauna adresa următoarei instrucțiuni de executat;
- **registrul DS** (Data Segment), destinat memorării adresei segmentului de date;
- **registrul SS** (Stack Segment), destinat memorării adresei segmentului stivă. Offsetul vîrfului stivei este memorat în registrul **SP** (Stack Pointer), astfel că adresa logică corespunzătoare vîrfului stivei este SS:SP;
- **registrul ES** (Extra Segment), destinat memorării adresei segmentului de date suplimentar (heap).

Într-un program de calcul pot exista mai multe segmente logice de cod, de date şi de stivă, dar numai 4 dintre acestea sunt accesibile la un momemt dat. Avînd în vedere faptul că o aceeaşi adresă fizică se poate obține din mai multe adrese logice diferite, segmentele logice se pot suprapune total sau parțial.

Mecanismul de segmentare a memoriei se reflectă şi asupra tipurilor de pointeri, folosiți în limbajul C. Avînd în vedere categoriile de informații conținute într-un program C++, pointerii pot fi:

- **pointeri de funcții**, destinați memorării adreselor dintr-un segment logic de cod;
- **pointeri de date**, destinați memorării adreselor dintr-un segment logic de date.

Dacă toate datele şi funcțile unui program C sunt în acelaşi segment, se pot utiliza pointeri pe 16 biți, numiți **pointeri near** (apropiați), în care se vor memora deplasările de segment. În acest caz pentru calculul adresei fizice se foloseşte valoarea fixă conținută în registrul de segment corespunzător.

Dacă zona de date sau funcții ocupă mai multe blocuri de 64 kO, se utilizează variabila pointer pe 32 biți, numite **pointeri far**. Un pointer far conține adresa logică. Utilizarea pointerilor far necesită o atenție deosebită, deoarece operațiile aritmetice modifică doar deplasamentul, rezultatul fiind modulo 65536, adică incrementarea offsetului peste valoarea 65535 nu v-a determina trecerea în alt segment, ci revenirea la începutul segmentului curent. În plus, operatorii relaționali >, <, >=, <= compară doar deplasarea, iar operatorii == şi != compară pointerii ca întregi de tipul **long undigned**. Avînd în vedere faptul că o aceeaşi adresă fizică se poate obține din mai multe adrese logice diferite, în cele mai multe cazuri rezultatul oerațiilor aritmetico-logice cu pointer de tipul **far** este eronat. Eliminarea acestui inconvenient se realizează prin utilizarea pointerilor de tipul **huge** (uriaş) în care se memorează adresa logică normalizată. Într-o adresă normalizată adresa segment conține numai ultimii 4 biți ai adresei fizice, avînd valori cuprinse între 0 şi 15. Operația de normalizare se efecuează în mod automat, calculîndu-se mai întîi adresa fizică, din care apoi se extrag adresa segment şi depplasamentul. Astfel, pentru a normaliza adresa logică 1ED7:0536 se evaluează mai întîi adresa fizică cu relația: 16·1ED7+0536 = 1F3A6, şi se scrie rezultatul obținut sub forma 1F3A0+6. Deci, adresa segment normalizată este 1F3A, iar deplasamentul 0006.

Ca urmare a mecanismului de normalizare, pointerii **huge** conțin valori distincte pentru fiecare adresă fizică, iar operațiile aritmetice şi logice se efectuează corect. Dezavantajul utilizării pointerilor de tipul **huge** constă în faptul, că prin efectuarea operației de normalizare se reduce viteza de execuție a programului.

Pentru elaborarea programelor de calcul, mediul de programare Turbo C++/Borland C oferă 6 variante de organizare a memoriei, numite **modele de memorie**:

- **modelul tiny** - se foloseşte cînd programul se compune dintr-un singur segment de 64 kO pentru cod, date şi stivă. Registrele CS, SS, DS şi ES au aceeaşi valoare şi se utilizează în mod implicit pointeri de tipul **near** atît pentru funcții, cît şi pentru date.
- **modelul small** - se foloseşte cînd este necesară generarea de segmente distincte penrru cod, şi rspectiv, date. Acest model utilizează în mod implicit pointeri de tipul **near** şi este recomandabil pentru aplicațiile de dimensiuni medii.
- **modelul medium** - generează segmente separate pentru cod şi date. Deosebirea față de modelul **small** constă în faptul că pot exista mai multe segmente de cod, care pot ocupa pînă la un 1 MO, şi că se utilizează pointeri de tipul **far**. Acest model este recomandabil pentru programe mari (fiecare dintre fişierele sursă poate ocupa un segment de 64 kO.
- **modelul compact** - utilizabil pentru programe de dimensiuni reduse şi cu un volum mare de date. Modelul generează un singur segment de 64 kO pentru cod şi mai multe segmente de date (pînă la 1 MO), cu următoarea organizare:
  - un segment de 64 kO pentru datele statice;
  - un segment de 64 kO pentru stivă;
  - restul memoriei poate fi alocat pentru datele dinamice.

  În mod implicit, pointerii de funcții sunt de tipul **near**, iar cei pentru date sunt de tipul **far**.

- **modelul large** - utilizabil pentru programe de dimensiuni mari, în care atît codul, cît şi datele necesită mai multe segmente. egmentele de date au o organizare identică cu cea a modelului compact, iar segmentele de cod o organizare similară cu cea a modelului medium. În mod implicit se folosesc pointerii de tipul **far** atît pentru funcţii cît şi pentru date, iar adresele logice nu sunt normalizate.

- **modelul huge** - este similar modelului large, deosebirea constînd în faptul că utilizează în mod implicit pointeri de tipul **huge** şi, prin urmare, fiecare segment de date sau cod poate ocupa (în ansamblu) un spaţiu de memorie de pînă la 1 MO.

Referitor la viteza de execuţie a programelor de calcul obţinute cu diversele modele de memorie, se fac următoarele precizări:

- modelele **tiny** şi **small**, datorită faptului că folosesc în mod implicit pointeri de tipul **near** atît pentru funcţii cît şi pentru date, generează programe cu cea mai mare viteză de execuţie;
- modelele **medium** şi **comoact**, prin utilizarea, în funcţie de necesităţi atît a pointerilor de tipul **near** cît şi a pointerului de tipul **far**, generează programe cu o viteză de execuţie uşor diminuată;
- modelele **larg** şi **huge**, datorită utilizării pointerilor de tipul **far** şi **huge**, generează programe de calcul ale căror performanţe, din punctul de vedere al vitezei de execuţie, sunt cele mai reduse comparativ cu celelalte modele de memorie.

Alegerea modelului de memorie se va face în funcţie de scopul şi necesităţile programului de calcul elaborat. În cadrul mediului Turbo C++.Borland C, pentru a selecta modelul de memorie necesar se parcurge următoarea secvenţă de comenzi:

- se selectează din meniul principal opţiunea "**Option**";
- se selectează din submeniul afişat comanda "**Compiler**";
- se selectează din noul submeniu comanda "**Code generation**";
- marchează în fereastra de dialog "**Code generation**" butonul corespunzător modelului de memorie necxesar aplicaţiei.

În vederea optimizării programelor de calcul, mediul Turbo C++.Borland C oferă posibilitatea de a specifica în mod explicit tipul de pointer dorit, indiferent de tipul implicit asociat mediului de memorie selectat. De exemplu, dacă dimensiunea structurilor de date nu depăşeşte 64 kO, nu se justifică utilizarea modelului de memorie **compact** sau **large** doar pentru accesul **far** la memoria video.

Declararea explicită a tipului unui pointer se face cu ajutorul unei instrucţiuni de forma:

tip modificator_de_tip lista_variabilelor_pointer;

similară instrucţiunii de declarare a variabilelor pointer. Deosebirea constă în plasarea, între cuvîntul cheie ce defineşte tipul variabilei şi lista variabilelor pointer, a unuia dintre cuvintele cheie, numite **modificatori de tip**, care să specifice tipul, diferit de cel implicit, asociat pointerilor din lista de variabile:

- **far** este utilizat în cazul modelelor **tiny**, **small** şi **medium** pentru a declara un pointer care adresează o variabilă din afara segmentului de date. De exemplu, pentru a accesa memoria video în cadrul unui program construit cu unul dintre cele trei modele de memorie amintite, se va declara în mod explicit variabila *mem_video* d tipul pointer **far** la întreg prin instrucţiunea:

int far *mem_video;

- **huge** este similar cu **far**, dar va permite în plus efectuarea de operaţii aritmetice şi comparaţii cu variabile de tipul pointer.
- **near** este utilizat în cazul modelelor **compact large** sau **huge**, pentru a forţa utilizarea unui pointer de tipul **near**.