Tehnici de programare

compilare independentă

Pe măsură ce programele devin din ce în ce mai mari, apare nevoia de a împărți codul în diverse module care să fie mai ușor de gestionat. De exemplu, un program care implementează o bază de date poate fi împărțit în: interfața utilizator (ferestre de introducere date, rapoarte), operarea efectivă asupra bazei de date (adăugare, ștergere, modificare) și operații de intrare/ieșire (salvare/încărcare bază de date, import, export). Unele proiecte pot fi foarte mari. În cazurile extreme, cum este kernelul de Linux, versiunea 5.6 a acestuia are aproximativ 33 milioane de linii de cod, dintre care 60% în diverse drivere.

Pentru gestionarea proiectelor mari, limbajul C pune la dispoziția programatorului mecanismul de **compilare independentă**. Acest mecanism permite următoarele:

- Codul proiectului poate fi împărțit în mai multe fișiere de cod (**.c**), fiecare fișier implementând un set de funcții sau chiar și numai o singură funcție.
- Fiecare fișier poate avea funcții care sa fie vizibile doar în interiorul acelui fișier sau funcții care să fie vizibile global, din toate fișierele proiectului.
- Pentru ca funcțiile și variabilele implementate într-un fișier să fie vizibile și din alte fișiere, se folosesc **declarații** de funcții și de variabile. De obicei aceste *declarații* se grupează în fișiere antet (**.h** - header)
- Fișierele de cod pot fi compilate independent, separat, doar atunci când este nevoie de compilarea lor. De exemplu, dacă avem un proiect cu 5 fișiere și lucrăm doar la unul dintre ele, doar acesta va fi recompilat la execuția programului. Celelalte fișiere nu mai au nevoie să fie recompilate, deoarece ele nu au fost modificate de la ultima compilare.
- Un proiect mai mare poate conține proiecte mai mici, unele dintre ele fiind aplicații separate (ex: programul de instalare al aplicației), iar altele fiind **biblioteci** (libraries) care se pot folosi în aplicațiile proiectului. O *bibliotecă* nu este o aplicație propriu-zisă ci doar un set de funcții, tipuri de date, variabile, etc., pe care le putem folosi în diverse aplicații.

În continuare, vom discuta folosind un exemplu cum funcționează mecanismul de compilare independentă.

Exemplul 1: Să se implementeze o bază de date care conține persoane, fiecare persoană fiind definită prin nume (max 30 caractere) si salariu de tip *float*. La pornirea aplicației baza de date va fi încărcată automat de pe disc. La ieșirea normală din aplicație (fără eroare), baza de date va scrisă pe disc. Programul va afișa un meniu cu următoarele opțiuni: 1-adăugare, 2-ștergere, 3-listare (în ordine alfabetică), 4-ieșire.

Vom implementa aplicația sub forma unui proiect, denumit ”bazadate”, format din 6 fișiere (2 antete și 4 de cod), listate mai jos. La începutul fiecărui fișier este numele său și o scurtă descriere.

```
// util.h - functii utilitare de uz general
#pragma once

// afiseaza un text de eroare formatat si iese din program
void err(const char *fmt,...);

// citeste in "dst" o linie de text de maxim "max" caractere
// inainte de a se citi linia, se afiseaza "text"
void citesteText(const char *text,char *dst,int max);
```

```
// citeste si returneaza o valoare de tip float
// inainte de a se citi valoarea, se afiseaza "text"
float citesteFloat(const char *text);
```

```
// util.c - functii utilitare de uz general
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>

void err(const char *fmt,...)
{
    va_list va;
    va_start(va,fmt);
    fprintf(stderr,"eroare: ");
    vfprintf(stderr,fmt,va);
    fputc('\n',stderr);
    va_end(va);
    exit(EXIT_FAILURE);
}

void citesteText(const char *text,char *dst,int max)
{
    printf("%s: ",text);
    fgets(dst,max,stdin);
    dst[strcspn(dst,"\r\n")]='\0';        // elimina \n de la sfarsitul liniei
}

float citesteFloat(const char *text)
{
    float f;
    printf("%s: ",text);
    scanf("%g",&f);
    return f;
}
```

```
// bd.h - interfata cu baza de date
#pragma once

#define MAX_NUME        31

typedef struct Persoana{
    char nume[MAX_NUME];
    float salariu;
    struct Persoana *urm;
    }Persoana;

extern Persoana *bd;

// adauga o noua persoana la baza de date
```

```
// adaugarea se face in asa fel incat baza de date este mereu sortata dupa nume
void adauga(const char *nume,float salariu);

// sterge o persoana din baza de date
// returneaza 1 daca persoana a fost gasita, altfel 0
int sterge(const char *nume);

// elibereaza memoria ocupata de baza de date
void elibereaza();

// salveaza baza de date in fisier
void salveaza();

// incarca baza de date din fisier
void incarca();
```

```
// bd.c - functii de operare cu baza de date
#include "bd.h"
#include "util.h"

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

Persoana *bd;

// creaza un nou element de tip Persoana
static Persoana *nou(const char *nume,float salariu, Persoana *urm)
{
   Persoana *p=(Persoana*)malloc(sizeof(Persoana));
   if(!p)err("memorie insuficienta");
   strcpy(p->nume,nume);
   p->salariu=salariu;
   p->urm=urm;
   return p;
}

void adauga(const char *nume,float salariu)
{
   Persoana *pred=NULL;          // predecesorul nodului curent
   Persoana *crt;                // nodul curent
   // itereaza toate numele din bd
   for(crt=bd;crt;pred=crt,crt=crt->urm){
     // daca s-a ajuns la un nume identic cu cel de inserat sau care trebuie sa fie dupa acesta,
     // atunci  termina iterarea
      if(strcmp(crt->nume,nume)>=0)break;
      }
   if(pred){                     // inserare in interiorul sau la sfarsitul listei
      pred->urm=nou(nume,salariu,crt);
      }else{                     // inserare la inceputul listei
      bd=nou(nume,salariu,crt);
      }
```

```
}

int sterge(const char *nume)
{
   Persoana *pred=NULL;              // predecesorul nodului curent
   Persoana *crt;                    // nodul curent
   for(crt=bd;crt;pred=crt,crt=crt->urm){
      if(!strcmp(crt->nume,nume)){
         if(pred){                   // stergere din interiorul sau de la sfarsitul listei
            pred->urm=crt->urm;
            }else{                   // stergere de la inceputul listei
            bd=crt->urm;
            }
         free(crt);
         return 1;
         }
      }
   return 0;
}

void elibereaza()
{
   Persoana *crt,*urm;
   for(crt=bd;crt;crt=urm){
      urm=crt->urm;
      free(crt);
      }
   bd=NULL;                          // pentru a se putea refolosi
}
```

```
// fis.c - functii de salvare si incarcare din fisier a bazei de date
#include "bd.h"
#include "util.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void salveaza()
{
   FILE *fis=fopen("bd.txt","wb");
   if(!fis)err("nu se poate crea bd.txt");
   Persoana *p;
   for(p=bd;p;p=p->urm){
      // fiecare persoana este salvata in fisier pe cate o linie,
      // cu ';' ca separator intre nume si salariu
      fprintf(fis,"%s;%g\n",p->nume,p->salariu);
      }
   fclose(fis);
}

void incarca()
```

```
{
    elibereaza();
    FILE *fis=fopen("bd.txt","rb");
    // daca nu s-a putut deschide fisierul nu este eroare,
    // deoarece este posibil sa fie prima rulare a programului
    if(!fis)return;
    char linie[MAX_NUME+32];    // o dimensiune acoperitoare pentru o linie din fisier (nume+';'+salariu+\n)
    // citeste linie cu linie
    while(fgets(linie,MAX_NUME+32,fis)){
        char *sep=strchr(linie,';');                    // cauta separatorul dintre nume si salariu
        if(!sep)continue;                               // daca o linie nu contine separatorul, nu o ia in considerare
        int nNume=sep-linie;                            // numarul de caractere din nume
        if(nNume>=MAX_NUME)nNume=MAX_NUME-1;    // ne asiguram ca numele citit nu este prea mare
        char nume[MAX_NUME];
        memcpy(nume,linie,nNume);
        nume[nNume]='\0';       // adauga separator
        float salariu=(float)atof(sep+1);
        adauga(nume,salariu);
        }
    fclose(fis);
}
```

```
// main.c - meniu utilizator si comportament global
#include "util.h"
#include "bd.h"

#include <stdio.h>

int main()
{
    char nume[MAX_NUME];
    float salariu;
    Persoana *p;

    incarca();
    for(;;){
        printf("1. adaugare\n");
        printf("2. stergere\n");
        printf("3. listare\n");
        printf("4. iesire\n");
        int op;           // optiunea
        printf("optiune: ");scanf("%d",&op);
        switch(op){
            case 1:
                getchar();
                citesteText("nume",nume,MAX_NUME);
                salariu=citesteFloat("salariu");
                adauga(nume,salariu);
                break;
            case 2:
                getchar();
                citesteText("nume",nume,MAX_NUME);
```

```
            if(sterge(nume)){
               printf("%s a fost sters din baza de date\n",nume);
               }else{
               printf("%s nu exista in baza de date\n",nume);
               }
            break;
         case 3:
            for(p=bd;p;p=p->urm){
               printf("%s\t%g\n",p->nume,p->salariu);
               }
            break;
         case 4:
            salveaza();
            elibereaza();
            return 0;
         default:printf("optiune invalida\n");
         }
      }
}
```

Declaraţii şi definiţii

Un **simbol** este nume definit de programator, de exemplu o funcţie, variabilă, nume de tip, structură, etc. În C, **declaraţia** şi **definiţia** unui simbol sunt termeni diferiţi:

- **declaraţia** - este specificarea tipului unui simbol, fără a se furniza o *implementare* (sau *corp*) pentru acel simbol. Diverse simboluri se declară în felul următor:
  - *funcţii* - se pune **;** (punct şi virgulă) după lista de parametri ai funcţiei (ex: *int max(int a,int b);*)
  - *structuri* - se pune **;** (punct şi virgulă) după numele structurii (ex: *struct Punct;*)
  - *variabile* - se pune **extern** în faţa variabilei sau, pentru vectori, nu se mai trece numărul de elemente (ex: *extern int a; float v[];*)
- **definiţia** - este implementarea propriu-zisă a unui simbol, de exemplu o structură sau funcţie cu tot cu acoladele corpului ei, o declaraţie de vector cu dimensiune, etc.

Înainte de a folosi un simbol, compilatorul de C trebuie să ştie tipul acelui simbol. De exemplu, nu se poate folosi o funcţie dacă nu avem înainte de folosirea ei o *declaraţie* sau *definiţie* pentru ea. Din acest motiv, dacă avem o funcţie definită într-un fişier .c şi dorim să o folosim într-un al doilea fişier .c, trebuie prima oară să o *declarăm* în al doilea fişier, pentru ca compilatorul să ştie despre ce este vorba. Pentru aceasta folosim fişiere antet (**.h** - header). Aceste fişiere conţin doar *declaraţii* de simboluri, la care se pot adăuga şi *definiţii* de structuri. Aşa cum am discutat la preprocesorul de C, directiva *#include* se substituie direct cu corpul fişierului specificat, ca şi cum am face copy/paste cu conţinutul acelui fişier. Astfel, dacă într-un fişier antet punem o declaraţie de funcţie şi includem acel antet într-un alt fişier .c, este ca şi cum am scrie declaraţia de funcţie direct în fişierul .c. În acest mod putem specifica faptul că în proiect există o funcţie cu numele şi tipul declarat, dar care se află implementată (*definită*) în alt fişier.

În exemplul de mai sus, funcţia *err* este implementată în fişierul *util.c*. Pentru a folosi această funcţie şi în alte fişiere, am pus *declaraţia* ei în antetul *util.h* şi am inclus acest antet în toate fişierele care folosesc *err*. Astfel, ea a devenit vizibilă şi în alte fişiere.

Dacă avem nevoie ca unele variabile să fie vizibile din mai multe fişiere de cod, putem să le declarăm în antet punând în faţa lor cuvântul cheie **extern**. Acesta înseamnă că avem de-a face doar cu *declaraţia* variabilei, nu cu *definiţia* ei. În exemplul nostru, variabila *bd* este folosită în mai multe fişiere, astfel încât a trebuit să o facem vizibilă pentru toate aceste fişiere cu declaraţia ”*extern Persoana *bd;*” (în antetul *bd.h*). Dacă nu am fi pus *extern* în faţă, în fişierele de cod ar fi apărut ”*Persoana *bd;*”, ceea ce înseamnă de fapt *definiţia* variabilei. În acest caz,

compilatorul ar fi dat o eroare de redefinire, fiindcă ar fi constatat că variabila *bd* este definită în mai multe fişiere de cod.

Pentru variabile de tip vector se preferă ca declararea lor să se facă prin omiterea numărului de elemente din vector. De exemplu, *"int v[];"* este considerată ca fiind o *declaraţie*, nu o *definiţie*, deoarece lipseşte numărul elementelor.

Organizarea fişierelor antet şi a celor de cod

Fişierele antet specifică **interfaţa** unui anumit modul. Tot ceea ce într-un anumit modul este destinat folosinţei publice (utilizare în alte fişiere) se va *declara* şi în fişierul antet pentru acel modul. Dacă o anumită funcţie sau variabilă nu face parte din interfaţă (ex: ea este folosită doar pentru implementarea internă a modulului), declaraţia acelei funcţii nu se va regăsi în antet. Mai mult decât atât, limbajul C permite ca la implementarea unei funcţii să punem în faţă cuvântul cheie **static**, ceea ce înseamnă că funcţia respectivă se va folosi strict doar în cadrul acelui fişier şi nu este vizibilă în alt fişier. De exemplu, funcţia *nou* din *bd.c* a fost considerată ca fiind un auxiliar intern pentru implementarea bazei de date şi este folosită doar în acest fişier. Din acest motiv, a fost definită ca fiind *static*. Numele funcţiilor statice se pot repeta şi în alte module, fără a fi considerate erori de redefinire (ex: putem avea o altă funcţie numită *nou* în fişierul *main.c*).

Nu există o corespondenţă de unu la unu între un antet şi un fişier de cod. Putem avea un antet a cărui funcţii să fie implementate într-un singur fişier de cod (ex: toate funcţiile declarate în *util.h* au fost implementate în *util.c*) sau putem să avem funcţiile dintr-un antet implementate în mai multe fişiere de cod (ex: funcţiile declarate în *bd.h* au fost implementate în fişierele *bd.c* şi *fis.c*). Putem să avem chiar antete la care nu le corespunde niciun fişier de cod (ex: un antet care defineşte doar tipuri de date: structuri, nume de tipuri, enumerări).

Criteriile de împărţire pe fişiere sunt date mai mult de legătura logică care formează un anumit set de funcţii şi de mărimea implementării lor. De exemplu, funcţiile din *util.h* sunt funcţii utilitare de uz general, care pot fi refolosite fără nicio modificare şi în alte proiecte. Dacă analizăm funcţia *err*, constatăm că ea nu are niciun fel de legătură cu baza de date. Funcţiile din *bd.h* implementează baza de date, dar am considerat că este prea mult cod de scris într-un singur fişier, aşa că l-am împărţit în două fişiere, folosind un subcriteriu de organizare: dacă sunt funcţii legate de fişiere sau nu.

În general este bine ca modulele să fie cât mai independente unele de altele (cât mai decuplate). În acest fel, pe de o parte putem să le refolosim uşor şi în alte aplicaţii, iar pe de altă parte, dacă modificăm ceva într-un modul, nu va trebui să facem multe modificări şi în alte module, dacă nu se schimbă interfaţa dintre ele. În exemplul nostru, toate funcţiile de operare asupra bazei de date (interfaţa definită de *bd.h*) sunt complet *agnostice* (neutre, independente) în privinţa modului de introducere sau de afişare a datelor. Nu vom găsi în *bd.c* sau în *fis.c* nicio folosire de *printf*, *scanf*, *fgets*, etc, decât eventual la apariţia unei erori (prin funcţia *err*). Din acest motiv, toată implementarea bazei de date se poate folosi practic nemodificată şi dacă vom dori ca aplicaţia noastră să aibă o interfaţă utilizator grafică. Pentru aceasta vor trebui să fie modificate doar funcţiile din *util* şi din *main*.

Când avem în faţă un program făcut de altcineva, este bine să începem analiza sa cu fişierele antet. În acestea vom găsi interfaţa de comunicare dintre module şi totodată tipurile de date cele mai importante, care se folosesc în mai multe module. După ce ne-am familiarizat cu interfaţa, putem pe urmă să inspectăm şi fişierele de cod (.c) pentru a vedea cum anume s-au implementat diverse funcţii. Din acest motiv, este bine să punem comentariile care explică funcţiile în fişierele antet, fiindcă acolo ne vom uita prima oară să vedem cum se folosesc acele funcţii.

Includerea antetelor în fişierele de cod

Când includem antete în fişierele de cod, este bine ca prima oară să includem antetele aplicaţiei respective (între ghilimele, "...", ceea ce înseamnă că prima oară se caută aceste fişiere în directoarele aplicaţiei) şi abia apoi antetele standard (între <...>, ceea ce înseamnă că antetele se caută prima oară în directoarele bibliotecii standard C). Prin faptul că includem prima oară antetele aplicaţiei, ne asigurăm de faptul că ele sunt complete, adică din ele nu lipsesc declaraţiile unor tipuri folosite în antet.

De exemplu, să presupunem că un fişier antet al nostru foloseşte la o declaraţie tipul de date *size_t* şi acest antet nu mai include la rândul lui alte fişiere. În acest caz am avea o eroare, deoarece tipul *size_t* este definit în antetele standard C (mai exact în *stddef.h*), pe care antetul nostru nu le include. Este posibil ca în anumite fişiere sursă să putem folosi acest antet (de exemplu, dacă este precedat de *string.h*, care include ceea ce este necesar pentru *size_t*), iar în alte fişiere sursă să avem o eroare de genul "*size_t* este nedefinit". Dacă vom include prima oară antetul nostru, deoarece înainte de el nu este niciun alt antet care să definească *size_t*, vom primi imediat mesajul de eroare şi vom putea astfel să remediem eroarea. Un fişier antet poate la rândul lui să includă alte fişiere antet, deci, în acest caz, la începutul antetului va trebui să includem *stddef.h*, în care este definit *size_t*.

Asigurarea unicităţii includerii fişierelor antet

În proiecte pot exista situaţii mai complexe, în care mai multe fişiere antet să includă fiecare acelaşi antet. Dacă vom include aceste fişiere în cod, antetul comun va fi inclus de mai multe ori, ceea ce poate duce la anumite erori şi la creşterea timpului de compilare. De exemplu, dacă antetul comun defineşte o structură, acea definiţie va fi inclusă de mai multe ori şi se va genera o eroare de genul "redefinire structură". Din aceste motive, este bine să ne asigurăm că un antet va fi inclus o singură dată într-un fişier de cod (dar evident că el se poate include în oricâte fişiere de cod).

Pentru a se asigura unicitatea includerii unui antet, se foloseşte directiva **#pragma once**, pusă chiar la începutul antetului respectiv. Această directivă lasă să fie inclus antetul la prima incluziune, dar, dacă ulterior se va încerca din nou includerea sa în acelaşi fişier de cod (chiar dacă această incluziune se face prin intermediul altor antete), nu se va mai include nimic din acel antet.

#pragma once nu a existat în primele versiuni ale limbajului C, astfel încât iniţial a fost folosită o altă modalitate de asigurare a unicităţii includerii, modalitate care este redată în codul de mai jos şi pe care este posibil să o găsiţi în multe antete:

```
#ifndef util_h
   #define util_h
   //... corp antet ...
#endif
```

Cu această metodă, la prima incluziune a unui antet defineşte un simbol, pe cât posibil unic pentru acel antet (de exemplu numele lui). Se poate constata că corpul antetului este inclus doar dacă acest simbol unic nu este definit, deci antetul încă nu a mai fost inclus în acel fişier de cod. Dacă se constată că simbolul este deja definit, deci antetul a fost deja inclus, atunci nu se mai compilează nimic din corpul antetului.

Compilarea aplicaţiilor formate din mai multe fişiere de cod

Când un proiect C este compilat la cod maşină, de fapt au loc două faze distincte:

1. *Compilarea fiecărui fişier de cod la **cod obiect*** - în această fază, din fiecare fişier .c rezultă un **fişier obiect** (object, cu extensia **.o** în Linux sau **.obj** în Windows). Un *fişier obiect* conţine codul maşină pentru funcţiile din acel fişier, plus o serie de informaţii suplimentare, de exemplu numele funcţiilor, variabilelor, etc din fişier.
2. *Legarea (link - asamblarea, linkeditarea) codurilor obiect împreună* - în această fază se preia doar codul util (cel al funcţiilor folosite de aplicaţie) din fiecare fişier obiect. Toate aceste coduri se leagă împreună (link) şi astfel rezultă aplicaţia finală (fişierul executabil).

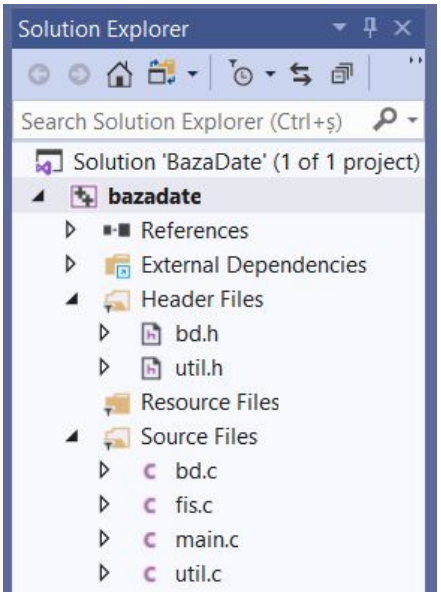
Până acum, când am folosit în linia de comandă *gcc*, aceste faze au avut loc automat, rezultând direct aplicaţia finală. Dacă vrem să compilăm la cod obiect un fişier .c, vom folosi în linia de comandă argumentul "**-c**" (compile only), ceea ce are ca efect doar generarea codului obiect pentru fişierul respectiv. Nu contează ordinea în care compilăm fişierele de cod. În final, după ce avem toate fişierele obiect, legarea lor (link) se face tot cu *gcc*, înşirând în linia de comandă toate fişierele obiect pe care dorim să le legăm împreună. Nu contează ordinea în care sunt înşiruite fişierele obiect. Pentru exemplul de mai sus, secvenţa de operaţii este:

```
gcc -Wall -c util.c
gcc -Wall -c bd.c
gcc -Wall -c fis.c
gcc -Wall -c main.c
gcc -Wall -o bazadate util.o bd.o fis.o main.o
```

Această secvenţă de operaţii va genera aplicaţia executabilă (*bazadate* în Linux sau *bazadate.exe* în Windows).

Există şi unele programe utilitarele care automatizează procesul de compilare al proiectelor. Printre acestea sunt *cmake*, *make*, etc. Pentru folosirea acestora, se scriu nişte fişiere care cuprind regulile de compilare (ex: aplicaţia *bazadate* este compusă din codul obiect rezultat din compilarea fişierelor *util.c*, *bd.c*, *fis.c*, *main.c*). Utilitarele respective vor invoca *gcc*-ul (şi alte programe), vor compila doar fişierele modificate între timp, vor ţine cont dacă un fişier s-a compilat cu succes sau nu, etc.

Toate IDE-urile moderne pot lucra cu proiecte. Unele IDE-uri pot lucra chiar cu proiecte ce conţin subproiecte. O *soluţie* (solution) în *Visual Studio* poate conţine oricâte proiecte. De exemplu, un proiect poate fi o bibliotecă de funcţii, iar altul aplicaţia principală, care utilizează acea bibliotecă. Exemplul de mai sus poate arăta în Visual Studio în felul următor:



IDE-urile pot dispune de algoritmi complecşi de compilare independentă. De exemplu, ele pot analiza fişierele .c să vadă fiecare fişier ce antete include şi vor recompila acel fişier .c doar dacă s-a modificat între timp unul dintre antetele incluse de el. În acelaşi timp, dacă se dispune de un CPU cu mai multe *nuclee* (cores), se pot compila simultan mai multe fişiere .c, pentru a se utiliza toate nucleele disponibile. Aceste mecanisme pot salva mult timp de aşteptare la compilarea proiectelor mari.

Biblioteci (libraries)

Codul C poate fi compilat pentru a rezulta o *bibliotecă de funcţii* (nu o aplicaţie executabilă). O bibliotecă este formată din mai multe fişiere obiect arhivate împreună. În acest fel putem avea la dispoziţie funcţii (uneori foarte complexe), pe care nu mai trebuie să le scriem noi şi pe care le putem include în aplicaţiile noastre. Inclusiv funcţiile standard C (ex: *printf*) sunt puse la dispoziţia programatorului tot sub forma unei biblioteci de funcţii.

Există foarte multe biblioteci, pentru cele mai variate scopuri: lucrul cu imagini bitmap (libjpeg, libpng), arhive (zlib), grafică 3D (OpenGL, Vulkan, DirectX), baze de date SQL (sqlite), interfeţe utilizator grafice (gtk), folosirea protocoalelor de reţea (libcurl), etc. Folosirea unor biblioteci existente poate duce la economii substanţiale de muncă. De exemplu, la unele dintre aceste biblioteci au lucrat zeci sau chiar sute de programatori, de-a lungul a mai multor ani. Când începem lucrul la o nouă aplicaţie, putem într-o primă fază să identificăm ce gen de

funcţionalităţi sunt necesare, iar apoi să căutăm dacă găsim bibliotecile potrivite sau dacă putem adapta un cod deja existent.

Aplicaţii propuse

Aplicaţia 11.1: La exemplul din laborator să se adauge funcţia “listeaza după salariu”, care va lista persoanele în ordinea crescătoare a salariului.

Aplicaţia 11.2: La exemplul din laborator să se adauge funcţia “modifică”. Pentru aceasta se cere numele persoanei a cărei înregistrare va fi modificată, iar apoi se vor cere noul nume şi salariu. Baza de date trebuie să rămână sortată alfabetic, chiar dacă noul nume diferă de cel anterior.

Aplicaţia 11.3: La exemplul din laborator să se adauge câmpul *”sex”* de tip *char* la structura *Persoana*, cu două valori: *‘m’* - masculin, *‘f’* - feminin. Se vor modifica funcţiile de adăugare, salvare, încărcare şi listare pentru a opera şi cu acest câmp.

Aplicaţia 11.4: Să se implementeze un modul care defineşte următoarele funcţii pe şiruri de caractere: *strCat(s1,s2)* - concatenează şirurile *s1* şi *s2* într-o zonă alocată dinamic şi o returnează; *strExtract(begin,end)* - copiază şirul de caractere care începe de la pointerul *begin* şi se termină la pointerul *end* (exclusiv) într-o zonă alocată dinamic şi o returnează; *strTrim(s)* - elimină caracterele *\n\r\t* şi *spaţiu* de la începutul şi de la sfârşitul şirului *s* şi returnează noul şir într-o zonă alocată dinamic. Să se scrie o aplicaţie care testează funcţiile din acest modul, folosind şiruri de caractere introduse de la tastatură.

Aplicaţia 11.5: Să se scrie o aplicaţie care implementează tipul de date *VecInt*, un vector redimensionabil automat cu elemente de tip *int*. Acest tip se va implementa într-un fişier .c separat, având ca interfaţă funcţii pentru: crearea unui vector nou, adăugarea unui element la vector, ştergerea unui vector, returnarea elementului de la o poziţie dată, setarea elementului de la o poziţie dată, returnarea numărului de elemente din vector. Aplicaţia va citi de la tastatură valori într-un *VecInt* până la apariţia valorii 0, exclusiv, le va sorta şi va afişa valorile sortate.