Tehnici de programare

compilare condiţionată; testare; depanare; contracte; aserţiuni

Preprocesorul C include facilităţi pentru adaptarea codului la cerinţe specifice, astfel încât codul să nu trebuiască modificat atunci când se schimbă anumiţi factori, cum ar fi:

- compilatorul - diverse compilatoare de C au caracteristici specifice, de care se poate ţine cont în program
- platforma hardware - aşa cum s-a discutat la fişiere binare, din punctul de vedere al unui program C, diverse microprocesoare (CPU) diferă de exemplu prin *endianness*
- platforma software - anumite funcţii sau biblioteci pot fi disponibile pe un sistem de operare (SO), dar absente pe altul. Mai mult decât atât, unele funcţii pot să se comporte diferit pe diverse SO sau chiar să aibă diverse buguri pe o anumită versiune de SO sau de bibliotecă
- compilare pentru diverse scopuri - se poate compila codul în mod diferit în faza de dezvoltare şi depanare (DEBUG) faţă de codul final, care este livrat clientului (RELEASE)

Pentru adaptarea codului la toate aceste cerinţe, se foloseşte următorul mecanism în preprocesor:

- compilatorul adaugă automat în program un set de definiţii predefinite Printre aceste definiţii sunt: ce compilator se foloseşte (ex: gcc, msvc, clang, etc), microprocesorul şi sistemul de operare pentru care se generează cod, etc. De exemplu, pentru a se vedea ce definiţii predefinite oferă gcc, se poate căuta pe internet *"gcc predefined macros"* sau apela gcc în Linux cu o comandă de forma *"gcc -dM -E - < /dev/null > macros.txt"*. Această comandă va lista în fişierul *"macros.txt"* toate definiţiile predefinite pe care gcc le oferă programului compilat.
- programului i se pot furniza definiţii şi din exterior, folosind linia de comandă a compilatorului. De exemplu, în gcc se foloseşte opţiunea "-D" (define) pentru a se predefini definiţiile dorite, iar apoi acestea vor fi introduse automat în programul compilat. De exemplu, *"gcc -DOPT 1.c"* va compila fişierul *1.c* ca şi cum la începutul codului său s-ar fi scris *"#define OPT 1"*, deci implicit orice definiţie predefinită în linia de comandă a compilatorului va avea valoarea 1. Dacă se doreşte să se dea altă valoare, se poate folosi un apel de forma *"gcc -DOPT=3 1.c"* şi atunci OPT se va înlocui în *1.c* cu valoarea 3. Opţiunea "-D" se poate repeta în linia de comandă de oricâte ori este nevoie, pentru a se defini mai multe definiţii.
  Toate compilatoarele de C oferă acest mecanism şi, pentru uşurinţa dezvoltării folosind IDE, există ferestre de dialog în IDE pentru setarea opţiunilor de compilare.
  În Code::Blocks, dacă s-a deschis un proiect, opţiunile de compilare pentru acel proiect sunt în *"Project -> Properties… -> Build targets -> Build options…"*. În această fereastră, pentru a se introduce definiţii, se selectează tabul *"#defines"* şi acolo se introduc definiţiile dorite, fiecare pe cate o linie. Pentru atribuiri de valori definiţiilor, ele se introduc de forma *"OPT=3"*, adică se atribuie definiţiei valoarea dorită.
  În Visual Studio directivele preprocesorului se pot seta în *"PROJECT -> Properties -> C/C++ -> Preprocessor"*, căsuţa *"Preprocessor Definitions"*, de forma *"OPT"* sau *"OPT=3"*.
- În interiorul programului se vor folosi directivele **#if #elif #else #endif #ifdef #ifndef #error** pentru a se testa existenţa definiţiilor şi valoarea lor specifică. În funcţie de rezultatele testelor, se va selecta codul corespunzător.

Exemplul 1: Să se scrie un program care trebuie compilat cu o definiţie numită EPS (epsilon) ce va defini precizia cu care se face testul de egalitate pentru numere reale. Dacă nu există definiţie pentru EPS, se va afişa o eroare:

```
// 1.c
#include <stdio.h>
```

```c
#include <math.h>

#if !defined(EPS)
   #error "EPS trebuie setat in linia de comanda a compilatorului"
#endif

int egal(double a,double b)
{
   return fabs(a-b)<=EPS;
}

int main()
{
   double a=0.001,b=0.002;
   printf("%g==%g cu precizie %g => %d\n",a,b,EPS,egal(a,b));
   return 0;
}
```

Codul dintre **#if** şi **#endif** este inclus în program doar dacă condiţia *if*-ului este îndeplinită, altfel este ca şi cum acest cod nu ar exista. Funcţia **defined** există doar în preprocesor şi returnează *true* dacă există definiţia cu numele dat, altfel *defined* returnează *false*. În acest caz, *defined(EPS)* returnează *true* dacă *EPS* este definit.

Directiva **#error** afişează un mesaj de eroare şi termină compilarea fişierului curent. În exemplu, dacă EPS nu a fost definit, *#if* lasă să fie compilat codul de după el, iar când se ajunge la *#error* se va afişa mesajul de eroare şi se va termina compilarea.

Dacă vom compila programul folosind gcc de forma *"gcc -Wall -o 1 1.c"*, din cauză că nu este definit EPS în program, se ajunge în linia #error, care va afişa mesajul respectiv şi compilarea se va termina.

Dacă vom compila programul folosind *"gcc -Wall -DEPS=0.001 -o 1 1.c"*, din cauză că EPS a fost adăugat în cod direct de către gcc, în program condiţia lui *#if* va fi false şi codul dintre *#if…#endif* nu se va compila, deci nu va exista nici linia cu *#error*. În acest caz, programul se compilează cu succes şi rezultă fişierul executabil "1" (1.exe în Windows). Dacă se execută acest fişier (./1), se va afişa textul *"0.001==0.002 cu precizie 0.001 => 1"*. Se constată că EPS a fost definit în program cu valoarea specificată în linia de comandă (0.001), iar pentru această valoare testul de egalitate este *true*. Dacă vom compila programul cu opţiunea "-DEPS=0.0001", se va afişa *"0.001==0.002 cu precizie 0.0001 => 0"*.

Deoarece expresii de forma *"#if defined(NUME)"* sau *"#if !defined(NUME)"* sunt des folosite, else se pot înlocui cu "**#ifdef** NUME", respectiv "**#ifndef** NUME":

```c
#ifndef EPS
   #error "EPS trebuie setat in linia de comanda a compilatorului"
#endif
```

Atenţie: A nu se confunda linia de comandă a compilatorului (gcc) cu linia de comandă a programului compilat. În linia de comandă a compilatorului se furnizează diverse opţiuni compilatorului (gcc) pentru a compila programul. Aceste opţiuni nu au nimic de-a face cu ce va primi ulterior programul compilat de la utilizator în linia sa de comandă.

Un neajuns al programului anterior este faptul că la fiecare compilare trebuie furnizată o definiţie pentru *EPS*. Am putea îmbunătăţi aceasta dacă vom defini în program o valoare implicită pentru *EPS*, care va fi folosită atunci când nu există dată la compilare altă valoare. Pentru aceasta, modificăm programul în felul următor:

```c
#ifndef EPS
   #define EPS 0.00001
```

```
#endif
```

În acest caz, dacă vom compila programul cu *"gcc -Wall -o 1 1.c"*, deci nu furnizăm nicio valoare pentru *EPS*, la începutul programului *EPS* nu va fi definit. Din acest motiv, se intră în *#ifndef* şi se va defini *EPS* cu valoarea dată.

Dacă este nevoie de mai multe condiţii succesive se poate folosi **#elif** (else if) între *#if* şi *#endif*. Analogic, dacă este nevoie de cod care să fie compilat dacă condiţia nu se îndeplineşte, se va folosi **#else**:

```
#if conditie_1
   // cod compilat daca conditie_1 este true
#elif conditie_2
   // cod compilat daca conditie_2 este true
...
#elif conditie_n
   // cod compilat daca conditie_n este true
#else
   // cod compilat daca niciuna dintre conditiile anterioare nu este indeplinita
#endif
```

Dezvoltarea, testarea şi depanarea programelor folosind compilarea condiţionată

Compilarea condiţionată este foarte folositoare în fazele de dezvoltare, testare şi depanare ale programelor. În aceste faze este de dorit să avem cât mai multe informaţii disponibile despre ce se petrece în program, dacă anumite variabile au valorile pe care le aşteptăm sau nu, etc. În schimb, când programul este definitivat şi va fi livrat beneficiarului, aceste informaţii nu mai sunt necesare şi ar fi foarte bine ca ele să dispară cu totul din program, pentru a nu-i reduce viteza de execuţie, pentru a nu încărca în mod inutil codul şi pentru a nu produce rezultate auxiliare (ex: afişări de teste) fără semnificaţie pentru beneficiar.

Eliminarea instrucţiunilor de testare şi depanare se poate face manual din codul final, dar este mai simplu să automatizăm acest proces, folosind compilarea condiţionată. În general, IDE-urile de C oferă două moduri (profile) de compilare:

- **Debug** - acest mod de compilare este destinat dezvoltării, testării şi depanării programelor. Sunt incluse tot felul de teste şi informaţii suplimentare, destinate programatorului. În acest mod de compilare se definesc macrouri specifice, de exempul **DEBUG** sau **_DEBUG**, care pot fi folosite în program pentru a testa în ce mod suntem.
- **Release** - acest mod de compilare este destinat compilării programului în varianta finală, care va fi livrată beneficiarului. În acest mod dispar toate testele şi informaţiile suplimentare destinate testării şi depanării şi totodată compilatorul optimizează pe cât posibil codul final, din punct de vedere al vitezei de execuţie şi/sau al memoriei ocupate. Unele IDE definesc macroul **NDEBUG** (no debug) pentru acest mod.

IDE-urile actuale au integrate depanatoare (debuggers) complexe, cu multe funcţii integrate, gen: execuţie pas cu pas sau până la un punct de oprire (breakpoint), afişarea valorilor variabilelor, afişarea stivei de apeluri, etc. Cu toate acestea, în cazul algoritmilor mai complecşi, care implică recursivitate sau structuri de date complexe, este mult mai simplu să depanăm programul folosind tehnicile prezentate în continuare.

Exemplul 2: Vom folosi un exemplu de program care conţine unele erori şi vom urmări să le identificăm şi să le eliminăm:

```
// program cu erori (prg.c)
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
double vmax(double *v,int n)
{
   int i;
   double m=0;
   for(i=1;i<n;i++){
      if(m<v[i])m=v[i];
      }
   return m;
}

int main()
{
   double *v;
   int i,n;
   printf("n=");scanf("%d",&n);
   if(v=(double*)malloc(n)==NULL){
      printf("memorie insuficienta");
      exit(EXIT_FAILURE);
      }
   for(i=0;i<n;i++){
      printf("v[%d]=",i);scanf("%g",&v[i]);
      }
   printf("maximul este: %g\n",vmax(v,n));
   free(v);
   return 0;
}
```

După cum ne putem da seama, programul citeşte *n* numere de tip *double* de la tastatură într-un vector alocat dinamic, apelează funcţia *vmax* care returnează maximul acestora şi afişează maximul rezultat.

Aplicaţia 7.1: Înainte de a trece mai departe, consultaţi în mod atent programul anterior şi notaţi toate bugurile găsite. La sfârşitul acestei secţiuni din laborator, comparaţi bugurile care au fost discutate pe parcurs cu cele găsite anterior, pentru a vedea astfel pe care dintre ele le-aţi identificat.

Vă rugăm ca secţiunile următoare să le şi executaţi pe măsură ce le citiţi, pentru a fi mai uşor să asimilaţi aspectele prezentate.

Dacă apelăm programul cu n=3, pe măsură ce introducem numere este posibil să primim un mesaj de genul *"Segmentation fault"* sau pur şi simplu programul să se termine fără niciun mesaj.

Mesajele gen *"Segmentation fault"* sau terminări abrupte de program, fără niciun mesaj, apar atunci când programul încearcă să acceseze zone de memorie la care nu are acces sau care sunt ocupate de alte variabile, ceea ce duce la coruperea acestora. Aceasta se poate petrece din mai multe motive: folosirea greşită a pointerilor, folosirea pointerilor NULL sau folosirea indecşilor care ies din vector, intrând astfel în zonele de memorie vecine vectorului.

În general, când se face depanare, primul aspect de la care se pleacă este izolarea erorii. Pentru aceasta se poate inspecta ieşirea programului pentru a se determina unde a ajuns execuţia sau se pot introduce în diverse puncte din program mesaje care să indice locaţia curentă.

În cazul nostru, programul a dat eroare undeva în *for*-ul de citire date, deci ar trebui să verificăm dacă în acest *for* nu avem vreun bug. În acest *for* intervin variabilele *i,n,v* şi apelurile funcţiilor *printf* şi *scanf*. La o primă vedere *i* şi *n* iau valori corecte, *n* fiind citit de la tastatură iar *i* fiind setat de *for* în intervalul corect, [0,n). Ca să ne asigurăm totuşi de acest aspect, putem să le afişăm valorile la începutul fiecărei bucle *for*, împreună cu valoarea lui *v*. Am putea folosi un *printf* dar, dacă dorim ca şi pe viitor să avem această afişare activă şi ea să existe doar în caz de

compilare de tip *Debug*, vom proceda în felul următor: vom crea un macro care să fie aibă exact acelaşi rol ca *printf*, dar care să fie activ doar pentru depanare, altfel să nu existe. Un asemenea macro, numit *DBG*, se poate implementa astfel, fiind situat în program înainte de funcţii:

```
#if defined(DEBUG) || defined(_DEBUG)
   #define DBG(...)              fprintf(stderr,__VA_ARGS__)
#else
   #define DBG(...)
#endif
```

În codul de mai sus se testează dacă este definit *DEBUG* sau *_DEBUG* (diverse IDE pot defini una sau cealaltă dintre aceste definiţii) şi, dacă una dintre definiţii există, se defineşte macroul *DBG* ca generând un *fprintf*. Dacă nu este definit nici *DEBUG* şi nici *_DEBUG*, atunci *DBG* se defineşte ca având un corp vid, deci practic nu va exista în program.

Se constată că un macro poate avea un număr variabil de argumente, dacă se specifică **...** (trei puncte). În acest caz, lista valorilor argumentelor se depune în variabila *__VA_ARGS__*. În cazul nostru, am folosit *__VA_ARGS__* direct ca argument în *fprintf*, deci *fprintf* va primi ca argumente tot ceea ce primeşte *DBG*.

Pentru depanare se recomandă folosirea *stderr* şi nu a *stdout*, pentru a se separa mesajele de eroare/debugging de cele obişnuite ale programului şi pentru a fi siguri că aceste mesaje sunt afişate imediat (*stdout*, din motive de optimizare, poate amâna uneori afişarea efectivă).

În cazuri de *"Segmentation fault"*, deci de posibilă corupere a memoriei, este bine să afişăm valorile variabilelor implicate cât mai des, chiar dacă aparent acestea nu se modifică, din cauză că ele se pot totuşi modifica indirect, prin intermediul unui pointer folosit greşit. În cazul nostru, vom afişa valorile lui *i*, *n* şi *v* chiar în bucla de introducere date:

```
for(i=0;i<n;i++){
   DBG("i=%d, n=%d, v=%p\n",i,n,v);
   printf("v[%d]=",i);scanf("%g",&v[i]);
   }
```

Pentru a rula programul în mod de depanare, putem să-l compilăm cu gcc, folosind opţiunea *"-DDEBUG"* pentru a activa macroul *DBG*. Dacă folosim Code::Blocks, în fereastra *"Project build options"*, tabul *"#defines"*, se va adăuga "DEBUG". În Visual Studio nu este nevoie să adăugăm nimic, fiindcă deja macroul *_DEBUG* este definit în modul *Debug*. Cu aceste setări, la execuţie programul va afişa ceva de genul:

```
n=3
i=0, n=3, v=00000000
v[0]=1
Press any key to continue . . .
```

Constatăm că valorile lui *i* şi *n* sunt corecte, dar în mod ciudat valoarea lui *v* este 0 (*NULL*), deşi la alocare am testat ca ea să nu fie *NULL*. Mai verificăm încă o data *if*-ul în care se face alocarea *"if(v=(double*)malloc(n)==NULL){...}"* şi vedem că de fapt ce-am scris ca şi condiţie, conform precedenţei operatorilor, se execută ca *"v=((double*)malloc(n)==NULL)"*, adică îi atribuim lui *v* rezultatul comparaţiei dintre valoarea returnată de *malloc* si *NULL*. Deoarece *malloc* a returnat ceva care nu este *NULL* (alocare reuşită), rezultatul comparaţiei este *false* (0) şi deci lui *v* i se va atribui valoarea 0. Din cauză că totul este *false*, nu se intră în *if*, deci nu apare mesajul de eroare. Aici devine evident că am uitat să punem parantezele în jurul atribuirii. Corectăm if-ul să fie *"if((v=(double*)malloc(n))==NULL){...}"* şi rulăm încă o dată programul, obţinând ceva de forma:

```
n=3
```

```
i=0, n=3, v=009178D8
v[0]=1
i=1, n=3, v=009178D8
v[1]=2
i=2, n=3, v=009178D8
v[2]=3
maximul este: 6.05994e-306
```

În această situaţie programul rulează până la sfârşit, dar nu afişează valoarea dorită. Valoarea 6.05994e-306 nu apare nicăieri în program şi nici nu a fost introdusă de la tastatură. Această situaţie poate apărea din mai multe cauze: valoarea nu fost introdusă corect (conform tipului ei), valoarea nu a fost tratată conform tipului ei, o zonă de memorie a fost coruptă (ex: prin intermediul unui alt pointer care pointa acolo), etc.

Pentru a proceda metodic, vom elimina rând pe rând posibilele cauze de erori, pe măsură ce programul avansează. În acest sens, ne vom asigura prima dată că valorile sunt introduse corect de la tastatură. Pentru aceasta vom afişa fiecare valoare, imediat după ce a fost introdusă:

```
for(i=0;i<n;i++){
	DBG("i=%d, n=%d, v=%p\n",i,n,v);
	printf("v[%d]=",i);scanf("%g",&v[i]);
	DBG("=> %g\n",v[i]);
	}
```

Executând acum programul, vom obține ceva de forma:

```
n=3
i=0, n=3, v=00900748
v[0]=1
=> 6.83364e-304
i=1, n=3, v=00900748
```

Constatăm că încă de la prima valoare introdusă (*v[0]*), în vector vom avea altceva decât ne dorim. Singurul loc în care se modifică aici valoarea din vector este *scanf*, aşa că îl inspectăm cu atenţie. Dacă vom consulta o referinţă de C, ne dăm seama că de fapt nu am citit corect de la tastatură valori de tip *double*. Noi am folosit %g, dar cu acesta se citesc valori de tip *float*. Pentru *double* se foloseşte %lg, (l=long). Dacă am fi urmărit atent mesajele produse de compilator, am fi văzut că de fapt am fost atenţionaţi de acest aspect, cu un mesaj de genul *”warning: format ‘%g’ expects argument of type ‘float *’, but argument 2 has type ‘double *’”*. Acesta este un exemplu că trebuie să fim atenţi la mesajele compilatorului, chiar dacă ele sunt doar atenționări (warnings) şi nu erori. Când primim asemenea mesaje, trebuie să ne dăm seama ce anume le-a generat şi, dacă este necesar, vom corecta aspectul respectiv.

Noul program corectat va afișa la execuție:

```
n=3
i=0, n=3, v=00D9FED8
v[0]=1
=> 1
i=1, n=3, v=00D9FED8
v[1]=2
=> 2
i=2, n=3, v=00D9FED8
v[2]=3
=> 3
maximul este: 3
```

Pentru prima dată am obţinut valoarea corectă a maximului. Totodată, acum ştim că valorile sunt introduse corect, deoarece le vedem pe ecran. Înseamnă aceasta că programul este corect? Deocamdată nu putem răspunde nici da nici nu la această întrebare. Din păcate mulţi programatori, atunci când obţin rezultatul dorit, presupun în mod eronat că programul este corect şi trec mai departe, fără să testeze mai serios sau să mai facă o minimă revizie a codului scris deja. De fapt, singurul lucru pe care-l putem afirma cu certitudine este că programul s-a comportat corect în acest caz particular. Pentru testare putem introduce şi alte cazuri. Pentru a alege cazuri de test cât mai potrivite în a detecta posibile buguri, putem ţine cont de următoarele sugestii:

- valorile pot fi sortate crescător, descrescător sau aleatoare
- valori speciale (min, max, valori pentru şters sau pentru inserat) pot fi pe prima poziţie din vector, pe ultima, dublate sau să lipsească
- dacă în algoritm apar anumite constante (ex: 0), se pot da seturi de date care conţin/nu conţin aceste constante, care sunt toate sub sau deasupra constantelor date

Conform acestor sugestii, noi deja am încercat un set de valori ordonate crescător. Vom introduce acum nişte valori descrescătoare şi obţinem:

```
n=3
i=0, n=3, v=00CBFED8
v[0]=3
=> 3
i=1, n=3, v=00CBFED8
v[1]=2
=> 2
i=2, n=3, v=00CBFED8
v[2]=1
=> 1
maximul este: 2
```

Constatăm că pentru acest set de date programul nu funcţionează corect. Maximul ar fi trebuit să fie 3, dar a fost afişat 2. Din moment ce partea de introducere de date este aparent corectă, vom testa în continuare funcţia *vmax*, deoarece ea urmează în ordinea execuţiei. Pentru aceasta, vom afişa la începutul funcţiei valorile pe care le primeşte şi totodată vom afişa la începutul fiecărei iteraţii valorile implicate în iteraţie. Programul va afişa:

```
n=3
i=0, n=3, v=00DA03D8
v[0]=3
=> 3
i=1, n=3, v=00DA03D8
v[1]=2
=> 2
i=2, n=3, v=00DA03D8
v[2]=1
=> 1
vmax(00DA03D8,3)
i=1, n=3, v[1]=2, m=0
i=2, n=3, v[2]=1, m=2
maximul este: 2
```

Constatăm că deşi avem 3 valori, în interiorul lui *vmax* se execută doar 2 iteraţii. Dacă analizăm codul *for*-ului *"for(i=1;i<n;i++){...}"*, ne dăm seama că de fapt am început iteraţiile de la indexul 1, nu de la indexul 0. Corectăm codul şi obţinem "*for(i=0;i<n;i++){...}*". Cu această modificare, programul funcţionează corect şi petru cazurile în care valoarea maximă este pe prima poziţie.

Vom încerca acum un set de date care să fie toate sub valorile constantelor folosite în algoritm. În *vmax* se foloseşte *"m=0"*, deci vom introduce valori negative. Programul va afişa:

```
n=3
i=0, n=3, v=01035E18
v[0]=-1
=> -1
i=1, n=3, v=01035E18
v[1]=-2
=> -2
i=2, n=3, v=01035E18
v[2]=-3
=> -3
vmax(01035E18,3)
i=0, n=3, v[0]=-1, m=0
i=1, n=3, v[1]=-2, m=0
i=2, n=3, v[2]=-3, m=0
maximul este: 0
```

Constatăm că pentru setul {-1,-2,-3} maximul nu este cel corect, ci se afişează valoarea 0. Dacă urmărim din datele afişate funcţionarea algoritmului, ne dăm seama că la fiecare iteraţie *m* rămâne la valoarea iniţială 0, valoare care nu există în vector, deoarece în vector nu se află nicio valoare mai mare decât 0. Avem 2 variante pentru a remedia aceasta: fie iniţializăm *m* cu o valoare minimă pentru tipul de date ales (*double*), astfel încât orice valoare din vector să fie mai mare decât *m*, fie iniţializăm *m* cu prima valoare din vector şi începem interaţiile de la a doua valoare. Vom alege a doua variantă:

```
double vmax(double *v,int n)
{
   DBG("vmax(%p,%d)\n",v,n);
   int i;
   double m=v[0];
   for(i=1;i<n;i++){
      DBG("i=%d, n=%d, v[%d]=%g, m=%g\n",i,n,i,v[i],m);
      if(m<v[i])m=v[i];
      }
   return m;
}
```

În această variantă, toate seturile propuse de date funcţionează corect, deci vom concluziona că algoritmul probabil este implementat corect.

Dacă vom dori acum să obţinem versiunea finală de program, care să nu mai conţină informaţii de depanare şi să fie în acelaşi timp şi optimizată, vom compila programul în mod *Release*. În linie de comandă vom folosi *"gcc -Wall -O3 -o prg prg.c"*. În Code::Blocks sau Visual Studio vom selecta modul/configuraţia *Release*. Din moment ce nu este definit *DEBUG* sau *_DEBUG*, macroul *DBG* se va reduce la o instrucţiune vidă, deci nu va avea nicio influenţă în program. Opţiunea "-O3" înseamnă optimizare maximă pentru viteză şi se referă la anumite transformări ale codului pe care compilatorul le poate face pentru a mări viteza (ex: să înlocuiască iterarea folosind indecşi prin iterare folosind pointeri).

Dacă vom rula acum programul, vom obţine:

```
n=3
v[0]=1
v[1]=2
```

```
v[2]=3
maximul este: 3
```

Din cauză că *DBG* nu mai are conţinut, toate informaţiile de depanare ce erau afişate în versiunea de *Debug* nu mai apar, deşi instrucţiunile *DBG* au rămas în cod.

Se constată că testarea şi depanarea unui program pot fi o sarcini dificile. Mai mult decât atât, cu excepţia unei testări exhaustive sau a unei metode matematice de a determina validitatea unui algoritm şi a implementării sale, nu putem fi siguri (în cazul programelor complexe) că după testare programul va fi corect.

Chiar şi în acest program, după toate testările cu date de intrare, tot a mai rămas un bug care nu a fost descoperit până acum. Este vorba de alocarea de memorie *"if((v=(double*)malloc(n))==NULL)"*, care de fapt ar fi trebuit să fie *"if((v=(double*)malloc(n*sizeof(double)))==NULL)"*. Această eroare a rămas nedetectată deoarece alocatorul de memorie a alocat pentru program, din motive de optimizare, mai multă memorie decât a fost necesară. Astfel, chiar dacă s-a depăşit spaţiul alocat efectiv lui *v*, s-a rămas totuşi în domeniul de memorie al programului, deci nu a apărut nicio eroare. Unele compilatoare (sau prin folosirea unor biblioteci auxiliare), în modul Debug realizează teste suplimentare şi pentru depistarea acestor cazuri. De exemplu, în Visual Studio, dacă în mod *Debug* apăsăm tasta F5 (*Start Debugging*), va apărea la sfârşitul programului o fereastră de dialog cu un mesaj de genul *"HEAP CORRUPTION DETECTED"*, ceea ce ne indică faptul că nu am folosit corect memoria alocată dinamic (heap-ul).

Contracte

Un **contract** se referă la toate specificaţiile unui cod (de obicei o funcţie), începând cu condiţiile pe care trebuie să le îndeplinească datele de intrare (**precondiţii**) şi terminând cu specificarea a ce anume produce codul respectiv (**postcondiţii**). În plus, se mai pot specifica **efecte colaterale** (side effects - orice alte efecte cu excepţia valorii returnate (ex: afişare, salvare în fişier, etc)) şi **invarianţi** (valori sau expresii care rămân constante de la început şi până la sfârşitul execuţiei funcţiei).

Numele de *contract* vine de la faptul că aceste specificaţii stabilesc un set de condiţii la care trebuie să se supună ambele părţi: atât funcţia în cauză, care trebuie să se comporte conform specificaţiilor sale, cât şi codul care o foloseşte, care trebuie să-i furnizeze funcţiei date valide şi să folosească în mod corect valorile returnate.

Contractele sunt foarte utile pentru definirea cât mai exactă a unei funcţii, în special pentru situaţiile care nu pot fi specificate prin tipul ei de date. Vom lua ca exemplu funcţia *vmax* corectată:

```
double vmax(double *v,int n)
{
   int i;
   double m=v[0];
   for(i=1;i<n;i++){
      if(m<v[i])m=v[i];
      }
   return m;
}
```

În această formă, pot exista câteva cazuri despre care nu s-a precizat nimic:

- Ce trebuie să se petreacă dacă v==NULL?
- Ce trebuie să se petreacă dacă n==0?
- Ce trebuie să se petreacă dacă n<0?

La multe întrebări de acest gen nu se poate răspunde prin definiţii de tip. De exemplu, nu avem în C o definiţie de pointer care să fie nenul. Ţinând cont că toate aceste situaţii se referă la datele de intrare, ele vor trebui specificate în *precondiţiile* funcţiei.

Dacă apar asemenea întrebări, înseamnă că de fapt funcţia este incomplet specificată (*contractul* ei nu este complet). Vom urmări în continuare să specificăm cât mai exact funcţia *vmax*, cu alte cuvinte să-i stabilim un *contract* complet.

Contractul unei funcţii se poate da într-o documentaţie separată sau în comentariu, înainte de funcţie:

```
/*
preconditii:
   - v!=NULL
   - n>0
efecte colaterale: niciunul
invarianti:
   - valorile din v nu sunt modificate
postconditii:
   - se returneaza o valoare m din v, cu proprietatea ca m>=v[i], oricare ar fi i din intervalul [0,n)
*/
double vmax(double *v,int n)
```

Invariantul dat se poate specifica şi prin cod, dacă modificăm antetul funcţiei în *"double vmax(const double *v,int n)"*, ceea ce arată că valorile din *v* nu vor fi modificate de către funcţie. Ori de câte ori putem specifica în cod o clauză din contract, este bine să o facem, deoarece compilatorul va verifica pentru noi respectarea ei şi totodată oferim mai multe ocazii pentru optimizare.

Folosind compilarea condiţionată, putem să includem în cod teste şi pentru verificarea contractului unei funcţii. Pentru aceasta se folosesc **aserţiuni** (condiţii care se consideră adevărate), implementate de macroul **assert***(condiţie)*. Acest macro este declarat în antetul *<assert.h>* şi el are următorul efect: dacă condiţia este îndeplinită, atunci programul continuă ca de obicei. Dacă nu se îndeplineşte condiţia, programul se opreşte şi se raportează un mesaj de eroare care cuprinde numele fişierului şi linia de cod cu *assert*-ul care nu a fost îndeplinit.

Pentru *vmax*, putem include următoarele aserţiuni:

```
double vmax(const double *v,int n)
{
   int i;
   assert(v!=NULL);
   assert(n>0);
   double m=v[0];
   for(i=1;i<n;i++){
      if(m<v[i])m=v[i];
      }
   return m;
}
```

În aceste condiţii, dacă funcţia va primi ca argument *v==NULL*, se va afişa un mesaj de forma:

Assertion failed: v!=NULL, file C:\Users\Razvan\Desktop\tests\1\1.c, line 26

În acest mesaj se pot recunoaşte: condiţia *assert*-ului care nu s-a îndeplinit, fişierul şi linia cu *assert*-ul. Aserţiunile sunt valide doar în mod *Debug*, ele fiind şterse din program în modul *Release*. Mai exact, aserţiunile sunt şterse când se defineşte macroul *NDEBUG* (no debug). În acest fel, aserţiunile pot ajuta mult la dezvoltarea şi depanarea unui program, dar nu vor reduce cu nimic performanţa sa în versiunea *Release*.

O variantă a macroului *assert* se poate implementa în felul următor:

```
#define STRINGIFY_AUX(val)  #val
```

```
#define STRINGIFY(txt)  STRINGIFY_AUX(txt)

#ifdef NDEBUG
   #define myassert(cond)
#else
   #define myassert(cond)  do{     \
      if(!(cond)){      \
         fprintf(stderr,"Assertion failed: " #cond ", file " __FILE__ ", line " STRINGIFY(__LINE__) "\n");    \
         abort();        \
         }       \
      }while(0)
#endif
```

Se constată că dacă NDEBUG este definit, atunci practic macroul nu mai produce cod. În caz contrar, din moment ce punct şi virgulă este obligatoriu, se foloseşte *do{...}while(0)* pentru a încapsula instrucţiunea *if*.

__FILE__ este o definiţie predefinită care are ca valoare numele fişierului curent, sub formă de şir de caractere (ex: ”/home/stud/1.c”). **__LINE__** este o definiţie predefinită care are ca valoare numărul liniei curente, sub formă de constantă întreagă (ex: 21).

Pentru a se asambla şirul de caractere care va fi afişat de *fprintf*, s-a folosit proprietatea că mai multe şiruri de caractere puse unul lângă altul se vor considera ca fiind un singur şir. Condiţia *cond* a fost transformată în şir de caractere folosind *#cond* (deci s-au pus ghilimele în jurul ei). __FILE__ deja era şir de caractere.

__LINE__ nu a putut fi transformat în şir de caractere punând # în faţa lui, fiindcă # nu funcţionează decât asupra argumentelor macroului (ex: *cond*), iar *__LINE__* nu este un argument al lui *myassert*. În această situaţie am folosit macroul *STRINGIFY*, care să-l primească pe *__LINE__* ca argument şi astfel să se poată aplica operatorul #. Totuşi, nu este suficient să folosim doar *STRINGIFY(__LINE__)* sub forma *”#define STRINGIFY(txt) #txt”*, deoarece în acest caz s-ar fi pus ghilimele chiar în jurul argumentului, rezultând *”__LINE__”*. Pentru a se obţine valoarea lui *__LINE__* între între ghilimele, macroul *STRINGIFY* va apela *STRINGIFY_AUX* cu valoarea rezultată din *__LINE__* (ex: 21), iar apoi *STRINGIFY_AUX* va converti această valoare la şir de caractere (ex: ”21”).

Se foloseşte *abort* în loc de *exit*, deoarece *abort* produce o terminare mai ”abruptă” a programului, care nu trece prin toate fazele lui *exit* (sunt aspecte de amănunt, corelate şi cu C++, care nu ne interesează la acest laborator).

Aplicaţii propuse

Aplicaţia 7.2: Să se scrie o aplicaţie care implementează o funcţie *abs_real* (returnează valoarea absolută a unui *double*) în două moduri, în funcţie de prezenţa definiţiei IMPLICIT la compilare. Dacă această definiţie există, *abs_real* va folosi funcţia *fabs* din *math.h*. Dacă IMPLICIT nu există, *abs_real* nu va folosi nicio altă funcţie.

Aplicaţia 7.3: Pentru exemplul 2, să se scrie în funcţia *vmax* la început o buclă care afişează conţinutul vectorului *v*. Toată această buclă va trebui să existe în program doar dacă este definit *DEBUG* sau *_DEBUG*.

Aplicaţia 7.4: Pentru exemplul 2, să se scrie o funcţie *test* care primeşte ca parametri un vector de elemente de tip *double*, numărul de elemente din vector şi o valoare *x* de tip *double*. Funcţia va returna 1 dacă *x* este maximul elementelor din vector, altfel 0. Folosind această funcţie, să se scrie un *assert* în *main* care să verifice că *vmax* returnează într-adevăr maximul elementelor din vector.

Aplicaţia 7.5: Să se scrie o funcţie care primeşte ca argument un pointer la un vector *v* de 4 octeţi de tip *unsigned char*. Programul va putea fi compilat doar cu gcc. Dacă se foloseşte alt compilator, se va genera eroare. Dacă platforma pe care se face compilarea este *big endian*, atunci funcţia va afişa octeţii de la stânga la dreapta, altfel octeţii vor fi afişaţi de la dreapta la stânga. Se poate folosi documentaţia cu macrourile predefinite de gcc.

Aplicaţia 7.6: Să se scrie un macro cu număr variabil de argumente, care afişează pe o linie prima oară fişierul şi linia curentă, iar apoi argumentele date.
Exemplu: SRCSHOW(”x=%g, y=%g”,x,y); va afişa ceva de genul: /home/stud/1.c [21]: x=0.5, y=-7.8

Aplicaţia 7.7: Să se implementeze o funcţie *sterge(v1,n1,v2,n2)*, cu *v1* şi *v2* vectori de tip *int* şi *n1* şi *n2* numere întregi care arată câte elemente sunt în *v1*, respectiv *v2*. Funcţia va şterge din *v1* toate elementele care există în *v2* şi va returna noua dimensiune a lui *v1*. Să se scrie un contract pentru această funcţie în care să se definească tot ceea cu nu s-a specificat în descrierea de mai sus, conform viziunii programatorului despre cum ar trebui să se comporte funcţia dată. Pe baza acestui contract, să se introducă în funcţie aserţiuni care să verifice respectarea contractului.