LUCRAREA 1

Scopul lucrării *este prezentarea unui scurt istoric a limbajului C și a noțiunilor legate de lucrul pe calculator (limbaj de programare, compilare, baze de numerație) iar în final inițierea în folosirea compilatorului Developer C++ prin rularea unor aplicații tipice de conversie dintr-o bază de numerație în alta.*

I. OBSERVAŢII TEORETICE

1.1. Scurt istoric C

Limbajul C a fost creat prima dată de către Dennis Ritchie şi Brian Kernighan la laboratoarele BELL Labs. din SUA. Versiunea premergătoare lui C a fost B. Odată creat el a fost folosit - împreună cu limbajul de asamblare - la scrierea completă a sistemului de operare UNIX. Aceasta arată că limbajul are o putere şi flexibilitate aparte.

1.2. Limbajul C - caracteristici generale

Limbajul C se bucură de următoarele caracteristici:

- este de **uz general**, adică poate fi folosit atât în scrierea de programe tip aplicaţie cât şi în situaţii *dedicate* (cum sunt metodele numerice - calcule ştiinţifice, grafică, editoare de text ş.a.);
- **de nivel mediu** - cu *relativ puține* instrucţiuni se pot spune relativ multe lucruri; pe scara aceasta de clasificare a limbajelor de programare C se află deasupra limbajului de asamblare, dar sub limbajele de nivel înalt: Pascal, C++, Java, SmallTalk, Python, Eiffel, ADA, Fortran etc. (**figură** cu clasificarea limbajelor de programare din punct de vedere al nivelului lor - scăzut, mediu şi înalt);
- **compilat** - în urma etapelor de compilare şi editare a legăturilor rezultă un fişier executabil (.exe sau .com); dacă se şterg de pe hard-disk toate fişierele *cu excepția* acestui executabil despre care vorbim - presupunând că nu se şterg şi fişierele sistem - acest program va rula fără probleme majore; C este diferit de limbajele interpretate (Visual Basic sau Java) în cazul cărora în urmă procesului de traducere a sursei în limbaj apropiat de limbajul maşinii pe care rulăm nu avem un fişier executabil concret. Ce se întâmplă în acest caz? Odată lansat codul sursă, datorită asocierii, este lansat automat interpretorul acelui limbaj, care preia sursa şi o execută, succesiv, linie cu linie. Efectele instrucţiunilor se reflectă ca şi în cazul unui program executabil. De obicei compilatoarele sunt mai rapide decât interpretoarele, în sensul că generează cod obiect mai rapid. În ultima vreme, deoarece limbajul Java se

impune ca un standard de facto în programarea Internet (s-a şi impus, de altfel), unde este nevoie de rapiditate - suntem foarte aproape de cerinţele unui software în timp real -, nu ar fi deloc de acceptat o întârziere în încărcarea paginilor Web, tocmai datorită timpului de interpretare datorat applet-urilor Java cuprinse de acele pagini. Programele compilate sunt, în plus, şi mai compacte decât cele interpretate;
- **portabil** - adică odată dus de pe o platformă hardware pe o alta programul nu are nevoie de modificări, sau are nevoie de foarte puţine modificări pentru a rula corect. Comportamentul programului aste analog (se poate spune aproape perfect analog) pe cele două platforme. La aceasta contribuie şi standardizarea limbajului.
- **flexibil şi puternic** - C combină flexibilitatea limbajului de asamblare cu puterea nivelului său mediu.

1.3. Baza de numeraţie 2 (sistemul *BINAR*)

Ca în orice bază de numeraţie, cifrele folosite în reprezentarea numerelor sunt cuprinse în intervalul:

```
[0, baza-1]
```

Rezultă că în baza 2 avem o reprezentare a numerelor folosind doar cifrele 0 şi 1.

Fiecare dintre cifrele semnificative ale unei baze de numeraţie poartă denumirea de *digit*. În baza 2 deoarece sunt doar doi digiţi posibili aceştia au preluat denumirea de *binary digit (bit)*. De aici proveniența cuvântului bit.

În baza 10 aceste cifre sunt *0...9*. Alte baze de numeraţie folosite în legătură cu sistemul binar sunt: *4, 8 (octal) şi 16 (hexazecimal)*. Pentru baza 16 cifrele de reprezentare sunt:

```
0..9 şi A..F
```

deoarece nu s-ar putea spune dacă, de exemplu, combinaţia 12 este obţinută prin alăturarea cifrelor 1 şi 2 ca cifre utile de reprezentare, sau este luată ca atare (*12*). Ar rezulta un mod de reprezentare echivoc, din moment ce 12 este şi valoare separată, dar şi o alăturare de alte două cifre semnificative ale bazei de numeraţie *16*. S-a convenit atunci folosire primelor litere ale alfabetului, cu semnificaţia:

```
A ţine locul lui 10
B lui 11
...
E lui 14
F lui 15.
```

În calculatoarele actuale baza de numeraţie este *2*. Au existat încercări de creare a unor calculatoare în bază *10*, dar nu s-au putut ridica la performanţele calculatoarelor binare. S-a păstrat astfel **sistemul binar** ca standard pentru calculatoarele digitale.

Să luăm un **exemplu** de număr în baza 2:

```
0110 1101
```

Ce înseamnă acesta? Cum poate fi interpretat astfel încât să poată fi înţeles de către noi (adică tradus în baza *10*)?

La aceste întrebări se răspunde plecând de la ***regula de*** **reprezentare** în ***orice*** bază de numeraţie (poziţională): fiecărei poziţii în număr îi corespunde o putere a acelei baze de numeraţie.

Astfel, primei poziţii din dreapta îi corespunde puterea 0 a lui 2, următoarei poziţii îi corespunde puterea *1* a lui *2*, iar ultimei poziţii (prima din stânga) îi corespunde puterea *7* a lui *2*. Atunci putem genera valoarea acestui număr în baza 10 - plecând de la dreapta spre stânga - astfel:

$1*2^0 + 0*2^1 + 1*2^2 + 1*2^3 + 0*2^4 + 1*2^5 + 1*2^6 + 0*2^7 = 1 + 4 + 8+ 32 + 64 = 109|_{10}$

La fel procedăm cu un număr în baza 10 când dorim să-i aflăm valoarea. Dar, în *baza 10* interpretăm natural şi aproape instantaneu orice număr, care ne sugerează şi o puternică semnificaţie "cantitativă" (adică **mărimea** acelui număr).

Exemplu:

578

Ştim imediat că avem de-a face cu '*cinci sute şaptezeci şi opt*' şi că acesta se situează cam la jumătatea gamei *[0-1000]*.

Bitul va fi notat de acum înainte cu **b**. Combinaţia de 8 biţi succesivi se numeşte **Byte (octet)** şi va fi reprezentat prin litera **B** de acum înainte. Combinaţia de *4* biţi poartă denumirea de *nibble*.

Am amintit mai sus şi de bazele de numeraţie 4, 8 şi 16. Vom transforma numărul dat în baza *2* în aceste baze de numeraţie, dar înainte *'să numărăm'* până la *8* în baza *2*.

```
 0000     // 0|10
 0001     // 1|10
 0010     // 2|10
 0011     // 3|10

 0100     // 4|10
 0101     // 5|10
 0110     // 6|10
 0111     // 7|10

 1000     // 8|10
 1001     // 9|10
 1100     //10|10
...
 1111     //15|10

10000     //16|10
```

Putem continua în aceeaşi manieră, dar deja se pot trage anumite *concluzii*.

Observaţi că s-a făcut o *delimitare* (o linie liberă) între valorile 3 şi 4, şi între 7 şi 8 (zecimal). Sunt poziţiile în care *numărul de biți crește cu o unitate*. Între 3 şi 4 trecem de la o reprezentare pe 2 digiţi (sau *biți*) la o reprezentare pe *3* biţi. Aceasta deoarece până la valoarea *3* am acoperit

toate combinaţiile posibile de 1 şi 0 pe cele două poziţii avute la dispoziţie. Câte sunt aceste combinaţii pentru *2* biţi? Sunt *4* adică 2^2. Până aici avem **baza 4** - deoarece cifrele semnificative, după cum se vede, sunt *0, 1, 2 și 3*.

Continuând, trecem aşadar la o *reprezentare pe trei biți*, pentru care combinaţiile posibile sunt *8*, adică 2^3. Până la linia liberă între *7* şi *8* în zecimal avem baza de reprezentare *8*, cu cifrele de la *0* la *7*.

Următoarea trecere se face între valorile zecimale *15* şi *16*, moment în care bitul al cincilea (din dreapta) intră în joc.

Concluzie:

Fiecare nou bit adăugat dublează gama de reprezentare, deoarece fiecare nou digit trebuie să ia valorile de 0 şi de 1 posibile pentru acea nouă poziţie, dar în legătură cu combinaţiile de 1 şi 0 din dreapta sa. Am marcat cu alb apariţia digitului al treilea şi valorile de 0 şi 1 pe care acesta ele ia, şi cu roşu bitul al patrulea nou adăugat.

Într-un număr în baza 2 sunt importante două poziţii:

- Prima din dreapta - care poartă denumirea de **Least Significant bit (LSb)**;
  - 0 are semnificaţia de număr par;
  - 1 are semnificaţia de număr impar;
- Prima din stânga - care poartă denumirea de **Most Significant bit (MSb)**. Poziţia **MSb** are de obicei *rolul de semn* al numărului:
  - 0 are semnificaţia de *plus*;
  - 1 are semnificaţia de *minus*.

Vom vedea la tipurile de date exact ce am vrut să spunem.

Aşadar să traducem în bazele *4, 8 și 16* combinaţia dată anterior:

0110 1101

Pentru baza 4 am văzut că sunt necesari 2 biţi pentru a reprezenta cifrele semnificative ale acestei baze de numeraţie. Atunci, pornind din dreapta, vom grupa doi câte doi biţii daţi, rezultând patru combinaţii de câte *2* biţi:

01 10 11 01

Traducem apoi în zecimal combinaţiile obţinute şi astfel ajungem la reprezentarea în baza *4*. Obţinem, de la stânga la dreapta:

1 2 3 1

Pentru baza 8 sunt necesari *3* biţi pentru a reprezenta cifrele semnificative ale acestei baze de numeraţie. Pornind din dreapta, vom grupa de această dată trei câte trei biţii daţi, rezultând combinaţii de câte *3* biţi. Dacă nu avem un umăr de biţi care să se împartă exact la *3*, vom considera că poziţiile lipsă până la trei poziţii - în cazul poziţiei din extrema stângă - sunt completate cu 0 - sunt *nesemnificative*:

001 101 101

Traducem apoi în zecimal combinaţiile obţinute şi astfel ajungem la reprezentarea în baza *8*. Obţinem, de la stânga la dreapta:

```
  1   5   5
```

Pentru **verificare** putem scrie:

$1*8^2 + 5*8 + 5 = 64 + 40 + 5 = 109|_{10}$

Obţinem aceeaşi valoare cu cea din traducerea directă în baza 2 astfel că am convertit corect în bază 8 numărul dat.

Pentru baza 16 sunt necesari 4 biţi pentru a reprezenta cifrele semnificative ale sistemului hexazecimal. Ca şi până acum, pornind ***din dreapta***, vom grupa biţii patru câte patru, rezultând în cazul de faţă două combinaţii a 4b.

```
0110 1101
```

Rămâne să traducem aceste combinaţii în zecimal şi avem:

```
   6    D
```

Verificarea ne dă:

$6*16 + 13 = 96 + 13 = 109|_{10}$

Deci este corect.

1.4. Structura *oricărui* program C - scurte detalieri

Aceste scurte detalieri au rolul de a furniza minimul de informaţie pentru a putea scrie chiar de acum un program C.

Aşadar:

A. **Comentariu** - cu rolul explicării în câteva cuvinte a rolului şi scopului programului;

În C există următorul stil de comentariu:

```
        /* ... */    pentru comentariu pe o linie
sau
        /* ...
        ...
        ... */       pentru comentariu pe mai multe linii.
```

În C++ există şi varianta unică:

```
        //
```

prin care linia astfel începută va fi *ignorată* în totalitate.

B. **Fişierele header** - cu extensia **.h** – sau **fişierele antet**. Rolul lor este de a încuraja **modularizarea** programului. Vom vedea imediat ce conţin aceste fişiere. Fişierele header apar la începutul unui program. Prezenţa lor este impusă de anumite funcţii ce urmează să fie folosite în acel fişier, şi al căror prototip (declaraţie de funcţie) aparţine acelui header.

Aceste fişiere sunt incluse în codul sursă al programului cu ajutorul directivei preprocesor #include

Sintaxa valabilă este:

```
        #include<nume.h>
sau
```

```
#include"nume.h"
```

În prima variantă compilatorul caută acel fişier **în subdirectorul INCLUDE** al directorului unde a fost instalat compilatorul. Acest director conţine **toate fişierele antet** cu care "vine" compilatorul (la instalarea acestuia).

În cea de-a doua variantă fişierul antet specificat este căutat mai întâi **în directorul curent** de lucru - adică în directorul unde a fost salvat codul sursă al programului la care se lucrează. Dacă nu este găsit în acel loc, căutarea sa se face în acel subdirector implicit al fişierelor header (*INCLUDE*).

C. Definiţii de constante, definiţii de tip, declaraţii/definiţii de variabile globale.

- ***Definiţiile de constante*** sunt însoţite de directiva preprocesor:
  `#define`

Sintaxa valabilă este:

```
#define NUME_CONSTANTA valoare
```

prin care se asociază numelui *NUME_CONSTANTA* valoarea *valoare*. Din acest moment, oriunde avem dreptul de a folosi valoarea numerică asociată, se va înlocui prezenţa numelui *NUME_CONSTANTA* cu valoarea asociată acesteia.

Se recomandă folosirea literelor mari pentru a se şti că acea denumire a fost undeva *definită de către programator*.

- **Definiţii de tip**

  Se fac cu ajutorul operatorului *typedef*. Un exemplu doar, urmând să detaliem sintaxa la momentul potrivit.

  ```
  typedef int INTREG;
  ```

  Prin această instrucţiune (se încheie cu;) se dă un alt nume (sau *alias*) tipului de bază C **integer** (specificat prin cuvântul-cheie **int**). Folosirea noii denumiri se traduce de fapt prin folosirea tot a lui `int`, dar în *mod indirect*. Construcţia este utilă pentru definirea de *tipuri mai complexe (cum sunt tipurile vector, matrice sau structură)*, ca şi pentru utilizarea unor denumiri mai sugestive în locul celor deja existente în limbaj.

  Este recomandat ca denumirea asociată să fie scrisă cu *majuscule*, pentru a se face diferenţierea faţă de denumirile de variabile care de obicei apar în scriere cu *minuscule*.

- **Declaraţii/definiţii de variabile globale**

  Sintaxa este neschimbată faţă de aceeaşi declarare/definire de variabile locale. Doar poziţia în care apare declaraţia/definiţia impune denumire de *global*. **Global** are semnificaţia de a fi cunoscut în întreg fişierul curent. Toate funcţiile din acest fişier vor cunoaşte aceste variabile. O denumire sugestivă pentru ele este de *publice*. De obicei se evită folosirea acestora, datorită **efectelor secundare** pe care le generează atunci când sunt folosite (*comentariu*: două funcţii apelate succesiv).

1.5. Detalii de programare

Pentru înţelegerea programelor utilizate în lucrare studiaţi ANEXA lucrării.

1.6. Comenzile compilatorului

Înainte de compilare codul sursă trebuie salvat (cu combinaţia de taste CTRL+S), sau, echivalent, din meniul File, cu comanda Save as....

Compilarea se va face cu comanda CTRL+F9, iar rularea cu comanda CTRL+F10. Compilarea şi rularea se pot executa, succesiv, printr-o singură tastă, anume F9. Acţiunile echivalente din meniu: meniul Execute comanda Compile, respectiv Compile + Run.

Observaţii:

1. Orice nouă modificare a codului-sursă trebuie urmată obligatoriu de salvarea acestuia (CTRL+S). Abia după aceea pot urma celelalte acţiuni dorite.
2. Dacă nu se realizează aducerea la zi a programului, compilarea şi rularea ce urmează se fac tot pe varianta veche, adică ce de dinaintea ultimelor modificări.

II. DESFĂŞURAREA LUCRĂRII

2.1. Să se convertească din sistemul binar în sistemul zecimal numerele reprezentate prin combinaţiile de biţi următoare:

a) 0111 1010
b) 110 0111

Conversia se va face:

- manual, pe hârtie,
- prin rularea programului *baza2_10.cpp*.

Verificaţi apoi cele două rezultate.

2.2. Să se convertească din sistemul zecimal în sistemul binar numerele reprezentate prin:

a) 2110
b) 137
c) 256

Conversia se va face:

- manual, pe hârtie,
- prin rularea programului de conversie asociat.

Verificaţi apoi cele două rezultate.

2.3. Să se convertească din sistemul `zecimal` în sistemul *hexa-zecimal* numerele reprezentate prin:

a) `127`
b) `1024`
c) `749`

Conversia se va face:

- manual, pe hârtie,
- prin rularea programului de conversie asociat.

Verificaţi apoi cele două rezultate.

2.4. Să se convertească din sistemul *hexa-zecimal* în sistemul `binar` numerele reprezentate prin:

a) `ABC`
b) `FFFF`
c) `1F0`
d) `1FFF`

Conversia se va face:

- manual, pe hârtie,
- prin rularea programului de conversie asociat.

Verificaţi apoi cele două rezultate.

2.5. Calculaţi complementul de 1 şi de 2 pentru numerele:

a) `127`
b) `-256`
c) `-1023`
d) `234`
e) `0011 1011`
f) `1111 0011`

Conversia se va face:

- manual, pe hârtie,
- prin rularea programului dedicat complementului.

Verificaţi rezultatele.

2.6. Adunaţi şi scădeţi în sistemul binar şi în cel hexazecimal numerele:

a) `123 + 1FF`
b) `AAAA – FFFF`
c) `5210 – 0A7B`
d) `0111 – 1101`
e) `0011 + 1111`
f) `1010 - 0101`

Conversia se va face:

- manual, pe hârtie,
- prin rularea programelor de conversie între diferitele baze de numeraţie.

Verificaţi apoi rezultatele.

2.7. Înmulţiţi şi împărţiţi în sistemul binar:

a) `0111 * 1101`
b) `0011 / 1111`
c) `1010 * 0101 / 0010`

Conversia se va face:

- manual, pe hârtie,
- prin rularea programelor de conversie între diferitele baze de numeraţie.

Comparaţi rezultatele.

III. ÎNTREBĂRI

3.1. Ce este şi ce conţine un fişier `header`?
3.2. În ce zonă din program pot apărea fişierele `header`?
3.3. `typedef` este directivă de pre-procesor?
3.4. `# define` este directivă de pre-procesare?
3.5. Constantele pot fi definite cu `typedef`?
3.6. Constantele pot fi definite cu `# define`?
3.7. Cum trebuie încadrate numele fişierelor `header`?
3.8. Pot apărea spaţii înainte sau după numele fişierelor `header`?
3.9. Pentru numerele `-128` şi `+12345` arătaţi rolul `LSb` şi `MSb`.
3.10. Ce semnifică baza de numeraţie poziţională?

IV. ANEXA - sursele complete ale programelor

```
// 10 in 2 - nerecursiv
#include<stdio.h>
#include<conio.h>

int b10_b2(int, int);

int main(void)
{
 int b10 = 1022, b2 = 2;

 clrscr();
 printf("\n Numarul de impartiri: %i", b10_b2(b10, b2) );

 getch();
}
```

```
int b10_b2(int b10, int b2)
{
 int pas = 0;
 int d, cat, rest;

 if(b10 < b2) b10 = b2;   // pentru a putea face impartire corecta,
            // deimpartitul este mai mare decat impartitorul.

 cat = b10;  // primul deimpartit.
do
{
  d = cat;
  cat = d/b2;          // catul impartirii a doi intregi.
  rest = d % b2;   // restul impartirii a doi intregi.
  printf("\n rest: %i", rest);
  pas++;
} while(cat != 0);

 return pas; // numarul de executii ale ciclului.
}

// baza 10 în baza 16
#include<stdio.h>
#include<conio.h>

#define SAU ||

typedef enum depasire9 {A=10, B, C, D, E, F} Baza16;

int baza10_16(int);       // prototipul functiei de calcul.
int _baza10_16(int, int*);      // prototipul functiei de calcul.
// parametrul al doilea este pentru resturile calculate,
// adica bitii numarului.

int main(void)
{
 int nr10 = 508;
 int i, l; // i pentru ciclul for(), iar l pentru lungimea vectorului.
 int rest[8]; // retine resturile calculate in pasii de apel recursiv.

 clrscr();
// scriu 0 in vector, astfel incat sa am valoare neutra in situatia
// scrierii lui 1. La scrierea unui 0 nu se intampla nimic.

 l = sizeof(rest)/sizeof(int);
 printf("\n Lungimea vectorului rest: %i", l);
 for(i=0; i<l; i++) rest[i] = 0;
```

```
// Apelurile functiilor.
// prima functie
 puts("\n Prima functie scrie: ");
 baza10_16(nr10);
// odata pornit apelul recursiv, el se va opri din interiorul
// functiei baza10_2().
 getch();

// a doua functie
 puts("\n\n A doua functie scrie: ");
 nr10 = nr10;      // o noua initializare.
 _baza10_16(nr10, rest);
// odata pornit apelul recursiv, el se va opri din
// interiorul functiei baza10_2().
 for(i=l-1; i>=0; i--)
      printf("\n pozitia %i, restul: %i", i, rest[i]);
 getch();
}

int baza10_16(int nr10)
{
 static int nrApel=0;

// printf("\n Nr apel %i, restul: %i", nrApel, rest[nrApel]);
 printf("\n Nr apel %i, restul: %i", nrApel, nr10 % 16);

 if(nr10 < 16) return nr10;
// ultimul cat este considerat primul bit util.
// La un nou apel, nr10 are rol de cat.
 else {
  nrApel++;
  baza10_16(nr10/16);
  }
}

int _baza10_16(int nr10, int *rest)
{ // al doilea parametru tine resturile
 static int nrApel1=0;
 Baza16 var16;

// Verific daca resturile depasesc numarul 9, pentru a codifica corect in
// baza 16.
 var16 = nr10 % 16;
 switch (var16)
{
  case A: {
       rest[nrApel1] = var16;
       break;
       }
```

```
  case B: {
       rest[nrApel1] = var16;
       break;
       }
  case C: {
       rest[nrApel1] = var16;
       break;
       }
  case D: {
       rest[nrApel1] = var16;
       break;
       }
  case E: {
       rest[nrApel1] = var16;
       break;
       }
  case F: {
       rest[nrApel1] = var16;
       break;
       }
 }

 rest[nrApel1] = nr10 % 16;
 if(nr10 < 16) return nr10;
// ultimul cât este considerat primul bit util.
// La un nou apel, nr10 are rol de cât.
 else {
  nrApel1++;
  _baza10_16(nr10/16, rest);
  }
}
```