Arhitectura Sistemelor de Calcul
Laborator - Introducere in C

Radu-Tudor Vrînceanu
Bogdan Macovei

Octombrie 2024

Cuprins

1 Introducere

1.1 Introducere

Scopul acestui suport este de a va introduce in limbajul de programare C nu din punctul de vedere al sintaxei, ci al conceptelor teoretice pe care le vom folosi ulterior in cadrul laboratoarelor viitoare. Motivatia alegerii unui limbaj precum C este data de usurinta cu care putem gestiona memoria interna si, totodata, vizualiza corespondenta cu GNU Assembly x86.

Nota: Daca nu stapaniti anumite elemente de sintaxa ale limbajului C sau nu l-ati studiat in trecut, va rugam sa vizualizati materialele celor de la **Harvard CS50's Introduction to Computer Science**[1] unde se regasesc atat materiale video cat si prezentari cu sintaxa de baza a limbajului.

Odata ce stiti sintaxa de baza a limbajului C, dupa parcurgerea acestui material veti putea intelege mai bine anumite concepte ale limbajului care va vor fi de folos in viitor:

- *entry-pointul* unui program, functia `main(...)`;
- functii de baza pentru I/O (*input / output*, `scanf`, `printf`, `fprintf`, `fscanf`;
- pointeri;
- apeluri de sistem;
- biblioteca *string.h* pentru sirurile de caractere;
- transmiterea prin referinta si prin valoare *(pass by reference, pass by value)*;
- returnarea prin referinta si prin valoare *(return by reference, return by value)*.

Pentru instalarea compilatorului de C, **GCC (GNU Compiler Collections)**, daca folositi un sistem de operare bazat pe UNIX (i.e. macOS, Linux-based) atunci il veti avea deja instalat. Pentru cei care folosesc Windows, daca aveti WSL instalat, atunci puteti lucra in cadrul WSL-ului. Daca nu aveti WSL instalat va recomandam sa urmariti acesti pasi pentru instalare:

- descarcati si rulati installerul pentru **MinGW (Minimalist GNU for Windows)**[2];
- dupa finalizarea instalarii, va aparea o ferestrea noua **MinGW Instalation Manager** si trebuie sa marcati pentru instalare optiunea *mingw32-gcc-g++*;
- salvati selectia si instalati optiunea selectata selectand din coltul stanga sus optiunea **Installation > Apply Changes** si confirmati la aparitia ferestrei de instalare apasand **Apply**;
- pentru a putea rula in **Command Prompt** comanda **gcc** trebuie sa ii spunem sistemului de operare calea catre acesta editand registrul cu variabile de mediu;
- deschiderea registrului cu variabilele de mediu pe Windows se face astfel **Windows Explorer > Click dreapta pe This PC > Advanced system settings > Environment Variables...**;
- odata intrati in registrul variabilelor de mediu vom cauta in tabelul de **System variables**, deseori tabelul de jos, o variabila denumita **Path**;

[1] **Harvard CS50's Introduction to Computer Science**, `https://cs50.harvard.edu/x/2024/weeks/1/`
[2] **MinGW (Minimalist GNU for Windows)**, `https://sourceforge.net/projects/mingw/`

- dupa identificarea ei o vom selecta si vom apasa pe **Edit...** (mare grija aici sa nu va stergeti / suprascrieti variabila **Path** pentru ca anumite programe s-ar putea sa nu mai functioneze ulterior);
- se va deschide o ferestrea denumita **Edit environment variable** si vom selecta optiunea **New**;
- vom introduce in noul rand creat calea catre directorul **bin** unde am instalat **MinGW**, daca ati lasat calea prestabilita la instalare atunci vom introduce **C:\MinGW\bin**;
- vom apasa pe **Ok**, iar dupa inchiderea ferestrei de editare a variabilei **Path** mai apasam **Ok** pentru a inchide ferestrea cu variabilele de mediu si inca un **OK** pentru interfata de setari avansate;
- deschideti un **Command Prompt** si introduceti in el *gcc −−version*, daca primiti un text care va afiseaza versiunea compilatorului atunci instalarea este finalizata.

In acest laborator vom cunoaste mai bine bazele limbajului de programare C, cat si conceptele fundamentale care inrudeste acest limbaj cu GNU Assembly x86.

2 Introducere in C

2.1 Introducere in limbajul de programare C

Limbajul C este unul dintre cele mai vechi si influente limbaje de programare, dezvoltat de Dennis Ritchie la AT&T Bell Labs in 1972. Scopul sau initial a fost de a oferi un limbaj de programare care sa fie eficient si suficient de flexibil pentru a fi folosit in dezvoltarea sistemului de operare UNIX, care a fost si el creat la Bell Labs.

Caracteristici importante ale limbajului de programare C:

- **Limbaj compilat:** Codul scris in C este transformat intr-un fisier executabil de catre un compilator, ceea ce permite rularea rapida a programelor si optimizari la nivel de cod.
- **Tipizare statica:** C este un limbaj cu tipizare statica, ceea ce inseamna ca tipurile de date trebuie specificate explicit si sunt verificate la momentul compilarii.
- **Limbaj procedural:** C se bazeaza pe paradigma de programare procedurala si modulara, permitand programatorilor sa imparta programele complexe in functii mai mici si mai gestionabile.
- **Management manual al memoriei:** Spre deosebire de limbaje moderne care ofera **garbage collectors**[3], C lasa gestionarea memoriei in responsabilitatea programatorului. Aceasta implica folosirea functiilor precum *malloc* si *free* pentru alocarea si eliberarea manuala a memoriei.

In inchiere, limbajul C a influentat multe alte limbaje moderne, cum ar fi C++, Java, C# si Objective-C. Este considerat un limbaj de nivel mediu, oferind un echilibru intre puterea de control a limbajelor de asamblare si complexitatea oferita de limbajele de nivel inalt.

[3] **garbage collectors**, `https://craftinginterpreters.com/garbage-collection.html`

2.2 *Hello, world!* in C

In cadrul acestui program vom invata, de fapt, ce importanta are functia *main* pentru a prelua informatii din exterior si a le transmite catre programul nostru. Programul de baza in limbajul C arata astfel:

```c
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

Dupa rularea comenzii de compilare si rulare a acestui cod vom observa ca in *terminal*, respectiv *cmd* obtinem un mesaj *Hello, world!* si un rand nou, datorita utilizarii caracterului special $\backslash n$.

- **UNIX Terminal:**

```
user@PC:~$ gcc main.c -o main
user@PC:~$ ./main
Hello, world!

user@PC:~$
```

- **Windows Command Prompt:**

```
C:\Users\user> gcc main.c -o main
C:\Users\user> main.exe
Hello, world!

C:\Users\user>
```

Functia *main* reprezinta punctul de start pe care *runtime-ul* C se bazeaza pentru rularea unui program. Aceasta functie, in mod normal ar trebui sa aiba o serie de parametri, iar scrierea ei completa este urmatoarea:

```c
int main(int argc, char** argv, char** arge)
```

- *argc* reprezinta numarul de argumente (i.e. *argument counter*);
- *argv* este un *array* de siruri de caractere, reprezentand valorile celor *argc* argumente. *argv[argc] = NULL*, iar *argv[0]* este numele programului executabil (i.e. *argument values*);
- *arge* este un *array* de siruri de caractere, reprezentand variabile de mediu si valorile acestora. (i.e. *argument environment*).

Pentru o vizualizare mai buna a acestor argumente, vom scrie in fisierul *main.c* urmatoarea secventa de cod:

```c
#include <stdio.h>

int main(int argc, char** argv, char** arge)
{
    printf("%d\n", argc);

    for (int i = 0; i < argc; i++)
    {
        puts(argv[i]);
    }

    int e = 0;
    while (arge[e] != NULL)
    {
        puts(arge[e++]);
    }
    return 0;
}
```

Vom explica functia *printf* in cadrul urmatorului capitol, functia *puts* scrie catre *stdout (i.e. standard output)* un sir de caractere cu \n la final. Astfel putem afisa valorile argumentelor cat si argumentele de mediu.

- **UNIX Terminal:**

```
user@PC:~$ gcc main.c -o main
user@PC:~$ arge0="Demo ENV Variable" ./main argv1 argv2 argv3
4
./main
argv1
argv2
argv3
... #other system ENV variables
arge0=Demo ENV Variable
... #other system ENV variables
```

- **Windows Command Prompt:**

```
C:\Users\user> gcc main.c -o main
C:\Users\user> set ENV="some value" & main.exe argv1 argv2 argv3 & set ENV=
4
./main
argv1
argv2
argv3
... #other system ENV variables
ENV="some value"
... #other system ENV variables
```

Nota: *argc* nu este folosit pentru a determina cate variabile de mediu au fost transmise programului, el doar contorizeaza cate argumente s-au stocat in *argv*.

3 Concepte fundamentale ale Limbajului C

In acest capitol vom acoperi cele mai importante functii si notiuni de *I/O* (i.e. fluxul de intrare si iesire a datelor) din cadrul librariei standard de C, vom invata despre tipurile de date existente in acest limbaj, alocarea dinamica a memoriei, manipularea sirurilor de caractere si mecanismele de transmitere a variabilelor in functii.

3.1 Tipurile de date si pointeri

3.1.1 Tipurile de date

Tipurile de date in limbajul C sunt clasificate in mai multe grupe precum:

- **Tipuri de date primitive:** aceste tipuri de date sunt cele de baza pentru a reprezenta valori simple precum *int*, *char*, *float*, *double*, *void*;
- **Tipuri de date derivate:** aceste tipuri de date se construiesc cu ajutorul primitivelor sau sunt *built-in*: *array*, *pointer*, *function*;
- **Tipuri de date definite de utilizator:** acestea sunt definite de utilizator, cel care scrie codul. Aceste tipuri in limbajul C sunt redate prin *struct*, *enum* si *union*.

Vom analiza dimensiunile acestor tipuri de date, in special pentru cele primitive cu ajutorul documentatiei de la Microsoft pentru **Storage of basic types**[4] si **C and C++ Integer Limits**[5].

Observatie: cu ajutorul documentatiei, veti putea observa ca exista mai multe derivate de reprezentare a unui numar intreg in memorie prin *short int* sau *long long int*.

Observatie: putem pune conditia ca *MSB-ul* (i.e. *Most Significant Bit*) sa nu fie luat in calcul ca bit de semn ci folosit in reprezentarea numarului prin cuvantul cheie *unsigned*.

Observatie: diferenta dintre double si float este redata de precizia de aproximare a numarului, intrucat un calculator care are o memorie finita nu poate reprezenta ceva infinit.

```c
#include <stdio.h>

int main() {
   short int x = 1;
   long long int y = 1;

   printf("%d %d", sizeof x, sizeof y);
   return 0;
}
```

Folosind operatorul *sizeof*, putem vedea care este diferenta de memorie (reprezentata in *Bytes*) intre un *short* si un *long long*.

[4] **Storage of basic types**, https://learn.microsoft.com/en-us/cpp/c-language/storage-of-basic-types?view=msvc-170

[5] **C and C++ Integer Limits**, https://learn.microsoft.com/en-us/cpp/c-language/cpp-integer-limits?view=msvc-170

```c
#include <stdio.h>

int main() {
   float x = 0.1, y = 0.2;

   printf("%.16f", x + y);
   return 0;
}
```

Rulati acest cod si observati ce problema apare in momentul in care se afiseaza rezultatul adunarii, ce se intampla daca inlocuim tipul *float* cu *double*? Toate aceste aspecte se leaga de o problema denumita *Floating point imprecision* si o veti discuta pe larg la curs in saptamanile urmatoare.

3.1.2 Pointer

In C orice variabila este declarata static, i.e. memoria este prealocata pentru acea variabila la inceputul executiei programului si este dezalocata la sfarsitul zonei de vizibilitate a variabilei.

In cazul in care ne dorim sa nu se intample acest lucru, putem folosi pointeri pentru a putea aloca memoria cand avem nevoie de ea. *Un pointer este o adresa catre o zona din memorie.* Un pointer se poate declara in urmatorul fel `<data-type> * <pointer-name>`.

Cand declaram un pointer, compilatorul aloca doar 4 sau 8 *bytes* (in functie de arhitectura) necesari pentru a putea salva valoarea adresei de memorie. Pentru a putea aloca un pointer, folosind *malloc*:

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
   int *p;

   p = (int *)malloc(sizeof(int));
   *p = 3;

   printf("Memory address %x, memory value %d", p, *p);
   return 0;
}
```

Dupa definirea pointerului si dupa ce nu mai avem nevoie de el in programul nostru, trebuie dezalocat utilizand `free`, altfel s-ar produce un fenomen denumit *Memory Leakage*.

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
   int *p;

   p = (int *)malloc(sizeof(int));
   *p = 3;
```

```
    printf("Memory address %x, memory value %d", p, *p);

    free(p);
    return 0;
}
```

3.2 Functii de baza I/O

Aceste functii de baza pentru manipularea fluxului de intrare si de iesire a datelor de la *stdin* (i.e. *standard input*) si *stdout* (i.e. *standard output*) se regasesc in biblioteca *stdio.h* pe care o avem deja inclusa prin sintagma `#include <stdio.h>` regasita la inceputul codului.

3.2.1 scanf

Citirea datelor se face prin functia `int scanf(const char *format, ...)`. Aceasta functie accepta ca parametru un format care va expune ce tip de data va trebui sa populeze. In tabelul de mai jos regasiti lista cu cele mai utilizate siruri de format:

Format	Descriere
%d	numar intreg in baza 10 (decimal)
%i	numar intreg in baza 10, 8 sau 16 bazat pe scriere
%u	numar intreg in baza 10 (decimal) fara semn, MSB-ul va fi luat in calcul
%o	numar intreg in baza 8 (octal)
%x	numar intreg in baza 16 (hexadecimal)
%c	caracter
%s	sir de caractere pana la intalnirea primului spatiu
%f	numar real

Vom scrie urmatorul cod in cadrul *main.c* unde vom putea analiza cum se apeleaza functia `scanf`

```
#include <stdio.h>

int main(int argc, char** argv, char** arge)
{
        int x, y, z = 0;
        char c, name[256];

        scanf("%d %i %x %c %s", &x, &y, &z, &c, name);

        return 0;
}
```

Nota: functia `scanf` se ocupa de popularea datelor in timpul rularii programului. A nu se intelege ca citirea datelor de intrare se realizeaza prin pasarea argumentelor in *argv*.

Observatie: dupa inserarea formatului de citire trimitem variabilele unde dorim sa stocam informatiile preluate. Totusi observam ca in fata numelui variabilelor punem caracterul &, acesta este operator de referentiere si indica adresa de memorie a variabilei. Totodata, pentru variabila *name*

nu am pus acest operator, intrucat `char name[256];` se poate rescrie ca `char* name;`. Cunoastem despre pointeri ca indica deja catre o adresa de memorie, astfel *&name* ar rezulta intr-o eroare.

3.2.2 printf

Functia `int printf(const char *format, ...)` este asemanatoare functiei *scanf*, doar ca de aceasta data, dupa ce trimite sirul cu formaturi nu mai trebuie sa trimitem adrese de memorie, pentru ca ne intereseaza sa vedem valorile acestor variabile.

```c
#include <stdio.h>

int main(int argc, char** argv, char** arge)
{
    int x, y, z = 0, a = 8;
    float pi = 3.14159;
    char c, name[256];

    scanf("%d %i %x %c %s", &x, &y, &z, &c, name);
    printf("Dec: %d, Int: %i, Hexa: %x, Char: %c, String: %s\n", x, y, z, c, name);
    printf("Special formats: %02d, %.2f", a, pi);
    return 0;
}
```

```
$: ./main
010 010 0xbeef c ASC
Decimal: 10, Integer: 8, Hexadecimal: beef, Character: c, String: ASC
Special formats: 08, 3.14
```

Observatie: valoarea lui x si a lui y desi am dat-o la fel pentru ambele, formatul *%i* al variabilei y vazand un *0* in fata valorii a considerat ca este scris in baza 8 (octal). Astfel $10_8 = 8_{10}$ unde n_b inseamna reprezentarea lui n in baza de numeratie b.

3.2.3 fscanf

Probabil ca uneori nu dorim sa citim datele doar de la *stdin* sau sa le afisam catre *stdout*, astfel putem folosi o functie precum *fscanf* pentru a citi informatii din fisiere text. Aceasta are urmatorul prototip: `int fscanf(FILE *stream, const char *format, ...)`. Putem vedea ca aceasta mai primeste un parametru si anume un pointer la un tip denumit *FILE*[6] care este o structura definita in libraria *stdio.h*.

Observatie: Pentru a citi dintr-un fisier acesta trebuie deja sa fie creat pe disc, altfel citirea va esua. Putem simula acest comportament schimband numele fisierului *input.txt* dupa ce il cream.

```c
// '''main.c
#include <stdio.h>

int main() {
   const char *filename = "input.txt";
```

[6] FILE, https://github.com/openbsd/src/blob/master/include/stdio.h#L99

```
    FILE *file = fopen(filename, "r"); // il deschidem in modul citire (read)

    int number;
    char word[50];

    if (fscanf(file, "%d %16s", &number, word) == 2) {
        printf("Number: %d, Word: %s\n", number, word);
    } else {
        printf("Failed to read data from file.\n");
    }

    fclose(file); // foarte important sa inchidem fisierul dupa ce am terminat de scris / citit
    return 0;
}

// '''input.txt
32 ASCeMateriaMeaPreferata

$: ./main
Number: 32, Word: ASCeMateriaMeaPr
```

3.2.4 fprintf

Functia `int fprintf(FILE *stream, const char *format, ...)` este foarte asemanatoare cu *printf*, doar ca de aceasta data mai avem un parametru in plus care denota fisierul de unde vom scrie rezultatele din sirul de caractere de format.

Observatie: Pentru a scrie intr-un fisier acesta **NU** trebue sa fie creat pe disc. Recomandam pentru a vedea functionalitatea din codul urmator sa stergeti fisierul text *input.txt* creat anterior.

```
#include <stdio.h>

int main() {
    const char *filename = "input.txt";
    FILE *file = fopen(filename, "r");

    // daca fisierul nu exista pe disc in modul de scriere, atunci pointerul va fi null
    if (file == NULL) {
        file = fopen(filename, "w"); // il deschidem in modul de scriere (write)
        if (file == NULL) {
            perror("Error creating file");
            return 1;
        }
        fprintf(file, "123 HelloWorld\n");
        fclose(file);
        file = fopen(filename, "r");
        if (file == NULL) {
            perror("Error opening file");
            return 1;
        }
```

```
    }

    int number;
    char word[50];

    if (fscanf(file, "%d %16s", &number, word) == 2) {
        printf("Number: %d, Word: %s\n", number, word);
    } else {
        printf("Failed to read data from file.\n");
    }

    fclose(file);
    return 0;
}
```

```
$: ./main
Number: 123, Word: HelloWorld
```

3.3 Apeluri de sistem

In limbajul C, un *apel de sistem* (i.e. *system call*) este o functie speciala oferita de *nucleul sistemului de operare* (i.e. *Kernel's OS*) care permite unui program sa interactioneze direct cu resursele hardware sau sa efectueze actiuni privilegiate care nu pot fi realizate doar din *spatiul utilizatorului* (i.e. *user space*).

Apelurile de sistem în C se regasesc in cadrul bibliotecii *sys/syscall.h*[7]. Puteti observa ca fiecare apel sistem are atribuit un cod, numar, care ne va fi folos de viitor. Cele mai comune apeluri de sistem sunt:

- *open*: deschide un fisier pentru citire sau scriere;
- *read*: citește date dintr-un fisier sau dispozitiv;
- *write*: scrie date intr-un fisier sau dispozitiv;
- *close*: inchide un fișier;
- *fork*: creeaza un nou proces, copiat dupa cel care a facut apelul;
- *exec*: incarca si executa un alt program in spatiul de adresa al procesului curent;
- *wait*: asteapta terminarea unui proces copil;
- *exit*: termina executia unui program;
- *kill*: trimite un semnal catre un proces (de exemplu, pentru a-l termina).

3.3.1 *open*, *close* si descriptori de fisier

Tabela de descriptori. Fiecare proces are o tabela de descriptori, constituita dintr-un *array* de structuri, indexat de la 0 la cel mai mare descriptor posibil. Daca procesul a asociat unui fisier un descriptor i (numar natural), intrarea i din aceasta tabela e completata cu informatii specifice, printre care un pointer la o intrare in tabela de fisiere deschise. De regula, primii trei descriptori sunt asignati automat, dupa cum urmeaza: 0 = intrarea standard, 1 = iesirea standard, 2 = iesirea standard pentru erori. Ei pot fi desemnati si prin urmatoarele constante simbolice, definite in *unistd.h*: STDIN_FILENO (=0), STDOUT_FILENO (=1), STDERR_FILENO (=2).

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
    int file_descriptor;
    char buffer[100];

    // apel de sistem pentru deschiderea unui fișier
    file_descriptor = open("input.txt", O_RDONLY);
```

[7] *sys/syscall.h*, https://github.com/openbsd/src/blob/master/sys/sys/syscall.h

```
    if (file_descriptor < 0)
    {
        perror("Eroare la deschiderea fisierului");
        return 1;
    }

    // apel de sistem pentru citirea datelor din fișier
    ssize_t bytes_read = read(file_descriptor, buffer, sizeof(buffer) - 1);
    if (bytes_read < 0)
    {
        perror("Eroare la citire");
        return 1;
    }
    buffer[bytes_read] = '\0';  // terminare string

    // afisam continutul citit
    printf("Date citite: %s\n", buffer);

    // inchidem fisierul
    close(file_descriptor);
    return 0;
}
```

3.3.2 Ce inseamna, de fapt, `return 0`?

Pe un sistem de tip UNIX, cand se apeleaza *return 0* din functia `main()`, se intampla urmatoarele lucruri:

- controlul revine la punctul de intrare al programului, *_start*, care face parte din *runtime-ul* C pentru acel sistem;
- codul din *_start* apeleaza *exit(0)*, sau ceva care are acelasi efect;
- functia *exit()* invoca apelul de sistem *exit*, care transfera controlul catre kernel;
- kernel-ul recupereaza toate resursele pe care procesul le utiliza;
- procesul parinte al programului, la un moment dat, apeleaza *wait()* sau un apel de sistem similar;
- se vor curata ultimele inregistrari si se va reporta statusul (i.e. codul) de iesire.

3.4 Siruri de caractere

Sirurile de caractere reprezinta defapt *array-uri* de caractere care pot fi alocate dinamic sau static: `char *str = (char *)malloc(sizeof(char) * 10)` sau in cazul in care avem o alocare statica `char str[10]`. Intern *str* este defapt un *pointer* catre un *char* pentru ca el indica adresa de memorie unde acest array este stocat. Intotdeauna in momentul cand declaram un sir de caractere trebuie sa avem in vedere ca la finalul acestuia trebuie sa lasam un loc disponibil pentru *null-terminated byte string* (i.e. '\0'). In limbajul C exista o biblioteca speciala pentru manipularea acestor siruri de caractere denumita *string.h*.

Vom vedea in urmatoarele liste cateva functii importante din cadrul librariei *string.h* pe care le veti folosi cand aveti de lucrat cu siruri de caractere.

- `size_t strlen(const char *s)`: extrage lungimea unui sir de caractere. **Nota:** aveti grija deoarece *strlen* se executa in timp liniar, folositi-l in afara structurilor de control iterative ori de cate ori aveti ocazia;
- `char *strcat(char *dest, const char *src)`: concatenarea, lipirea a doua siruri de caractere;
- `char *strcpy(char *dest, const char *src)`: copierea unui sir de caracter in altul;
- `char *strchr(const char *s, int c)`: cauta prima aparatie a caracterului *c* in *s*;
- `char *strstr(const char *haystack, const char *needle)`: cauta prima aparatie a unui subsir de caractere *needle* in sirul de caractere *haystack*;
- `int strcmp(const char *s1, const char *s2)`: compara doua siruri de caractere, daca sirurile de caractere sunt identice atunci rezultatul returnat este 0, altfel se returneaza o valoare negativa daca sirul *s1* este lexicografic mai mic decat *s2*, analog se poate returna o valoare pozitiva;
- `char *strtok(char *str, const char *delim)`: extrage dintr-un sir de caractere cate un subsir delimitat de caractere din sirul *delim*.

Pentru citirea sirurilor de caractere de la *stdin* putem folosi functia *gets*, pentru citirea sirurilor de caractere din fisier putem folosi functia *fgets*. *scanf* este recomandat doar daca sirul de caractere nu contine spatii. Afisarea se poate face fie cu *puts* sau cu *printf* la stdout, analog pentru fisiere se poate folosi *fprintf*.

Rulati acest cod pentru a vedea cum functioneaza aceste functii din libraria *string.h*:

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

int main() {
        char *str = (char *)malloc(sizeof(char) * 128);
        puts("Introdu un sir de caractere: ");

        gets(str);
```

```
        printf("Sirul de caractere introdus: %s\n", str);

        size_t length = strlen(str);
        printf("Dimensiunea sirului de caractere: %d\n", length);

        char *correct_dimension_str = (char *)malloc(sizeof(char) * (length + 1));

        strcpy(correct_dimension_str, str);
        printf("Sirul de caractere copiat: %s\n", correct_dimension_str);
        printf("Sunt identice?: %d\n", strcmp(str, correct_dimension_str));

        strcat(str, " lorem, ipsum, dolor");
        printf("Sirul de caractere dupa concatenare: %s\n", str);
        printf("Mai sunt identice?: %d\n", strcmp(str, correct_dimension_str));

        for (size_t i = 0; i < length; i++) {
                if (strchr("aeiou", str[i])) {
                        printf("Am gasit prima vocala din sir la pozitia %d, %c\n", i, str[i]);
                        break;
                }
        }

        char *res = strstr(str, "ASC");
        printf("Sirul ASC a fost gasit? %s, secventa %s\n", (res != NULL ? "DA" : "NU"), res);

        char sep[] = " ,";
        char *token = strtok(str, sep);
        while(token != NULL)
        {
                puts(token);
                token = strtok(NULL , sep);
        }

        free(correct_dimension_str);
        free(str);
        return 0;
}
```

3.5 Operatori pe biti

In aceasta sectiune vom prezenta un tabel cu operatorii pe biti cei mai uzuali si semnificatia acestora. Trebuie sa stiti ca orice numar se poate reprezenta in baza 2, pe un numar finit de biti din punct de vedere al tipurilor de date pe care le folositi (i.e. declararea unui *short int* $x = 2$ se reprezinta intern ca *00000000 00000010*).

Denumire	Simbol
Operatorul pe biti SAU	$x \mid y$
Operatorul pe biti SI	$x \ \& \ y$
Operatorul pe biti SAU EXCLUSIV	$x \ \hat{} \ y$
Operatorul pe biti de NEGATIE	$\sim x$
Operatorul pe biti de DEPLASARE la DREAPTA si la STANGA	x<<y, x>>y

b_i^x	b_i^y	$(b_i^x \mid b_i^y)$	$(b_i^x \ \& \ b_i^y)$	$(b_i^x \ \hat{} \ b_i^y)$
1	1	1	1	0
1	0	1	0	1
0	1	1	0	1
0	0	0	0	0

Unde cu b_i^x s-a notat al *i-lea* bit din reprezentarea binara a lui x, iar cu b_i^y s-a notat al *i-lea* bit din reprezentarea binara a lui y.

Observatie: x>>N inseamna ca deplasam N biti la dreapta; daca e sa ne gandim ce inseamna acest lucru din punctul de vedere al reprezentarii binare, inseamna $\lfloor x/2^N \rfloor$

Observatie: x<<N inseamna ca deplasam N biti la stanga; daca e sa ne gandim ce inseamna acest lucru din punctul de vedere al reprezentarii binare, inseamna $x \cdot 2^N$

Observatie: $x \ \hat{} \ x = 0$.

3.6 Transmiterea prin referinta si prin valoare a parametrilor

Intr-o functie parametrii pot fi trimisi in doua modalitati:

- prin valoare – intainte de apel se face o copie a parameterului care este folosita pe toata durata executiei functiei. La intoarcerea din apel valoarea parametrului ramane neschimbata: orice modificare efectuata pe parametru in corpul functiei nu este vazuta dupa finalizarea apelului.
- prin referinta – functia este apelata direct cu variabila pasata ca parametru si orice modificare facuta in corpul functiei pe acel parametru este vizibil dupa revenirea din apel. Valorile constante nu pot fi pasate prin referinta catre o functie. Un parametru este trimis prin referinta daca are & inainte de numele parameterului.

In schimb exista anumite tipuri care se trimit implicit datorita mecanismului de functionare prin referinta, putem sa observam in exemplu de mai jos ce se intampla cu stringul nostru dupa rularea codului:

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

int by_value_and_reference(int x, double y, char *str) {
    x = 3;
    y = 3.141592653589793238;
    str[0] = 'B';

    printf("Valoarea din by_value a lui str este %s\n", str);
    printf("Valoarea din by_value a lui y este %.15f\n", y);
    return x;
}

int main(int argc, char *argv[], char *arge[]) {
    char *something_dynamic = (char*)malloc(sizeof(char*) * 4);
    strcpy(something_dynamic, "ASC");

    int x = 6; float pi = 3.1415;
    int result = by_value_and_reference(x, pi, something_dynamic);

    printf("Valoarea lui something_dynamic este %s\n", something_dynamic);
    printf("Valoarea lui pi este %.15f\n", pi);
    printf("Rezultatul este %d, iar x este %d", result, x);

    free(something_dynamic);
    return 0;
}
```

Astfel, putem observa cum toate primitivele sunt pasate prin valoare, iar variabilele dinamice sunt transmise prin referinta pe baza mecanismului intern. Totusi, cum putem trimite o primitiva

prin referinta? Vom rescrie codul de mai sus pentru a permite modificarea valorii variabilei x in functia *by_value_and_reference*.

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

int by_value_and_reference(int *x, double y, char *str) {
    *x = 3;
    y = 3.141592653589793238;
    str[0] = 'B';

    printf("Valoarea din by_value a lui str este %s\n", str);
    printf("Valoarea din by_value a lui y este %.15f\n", y);
    return *x;
}

int main(int argc, char *argv[], char *arge[]) {
    char *something_dynamic = (char*)malloc(sizeof(char*) * 4);
    strcpy(something_dynamic, "ASC");

    int x = 6; float pi = 3.1415;
    int result = by_value_and_reference(&x, pi, something_dynamic);

    printf("Valoarea lui something_dynamic este %s\n", something_dynamic);
    printf("Valoarea lui pi este %.15f\n", pi);
    printf("Rezultatul este %d, iar x este %d", result, x);
    free(something_dynamic);
    return 0;
}
```

Observatie: oferind ca parametru &x, tipul din antetul functiei se va modifica in *int ** pentru ca avem o adresa transmisa. Astfel putem acesa valoarea ei prin operatorul de derefentiere ***.

4 Exercitii propuse

- Scrieti un program in limbajul C care permite interschimbarea a doua valori a unor variabile fara a folosi o variabila auxiliara pentru acest lucru.
- Scrieti un program in limbajul C care primeste intr-un fisier un numar de $2n + 1$ numere, dintre care cele $2n$ sunt egale doua cate doua. Sa se determine numarul care nu are pereche dezvoltand un algoritm optim din punct de vedere al spatiului si al timpului de executie.
- Scrieti un program in limbajul C care primeste prin argv un string care contine in interiorul sau o data in format ll.zz.aaaa, utilizand functia *sscanf*, extrageti si afisati catre *stdout* ziua, luna si anul (i.e. ./main *Astazi, 02.10.2024, am fost la laboratorul de ASC*)
- Scrieti un program in limbajul C care declara *un array bidimensional* de dimensiune $N \times N$ (i.e. o matrice patratica) intr-o maniera dinamica utilizand *malloc* si *free*, populand diagonala secundara cu numere de la 1...N.