Robin RIEUTORD

Portfolio

Table of Contents

Introduction 3
TP Réalisée : 3
D'où je pars : 3
Les compétences : 4
I/O : gérer les entrées/sorties en C 4
Types : utiliser correctement les types de base 6
Programme : notion de programme 7
Modules : établir des architectures de sources modularisées 8
Compilation : savoir compiler correctement les programmes 10
Récursivité : utiliser la récursivité en C 11
Tableaux : maîtriser la notion de tableau en C 14
Pointeurs : utiliser adresses et valeurs pointées des variables 15
Fichier : gérer les fichiers et leur contenu en C 18
Bibliothèques : bibliothèques et packaging en C 22

Introduction

TP Réalisée :

- TP2
- TP4
- TP5
- TP6

D'où je pars :

J'ai fait du C dans mes études précédentes en école d'ingénieur à EPITA mais il y'a eu une longue pause (2018 à 2020). Le but était donc de me rappeler mes notions de C, c'est pourquoi je n'aie pas fait les TP1 et TP3. J'ai ensuite essayé d'aller le plus loin possible. J'ai eu beaucoup de mal sur le TP5. Maintenant que j'ai rafraichit mes connaissances, je dois essayer d'accélérer le rythme. Je prévois de faire notamment des projets pour le second semestre.

Les compétences :

I/O : gérer les entrées/sorties en C

Cette compétence représente pour moi la capacité a rentrée et à afficher les données exploiter par le programme. Le plus utilisée est pour moi le printf qui consiste à écrire dans la console une chaine de caractères. Cette fonction est issue de la bibliothèque <stdio.h>. Elle a pour prototype :

```
int printf(const char* format, ...);
```

Le premier paramètre contient la chaine de caractères à envoyer. Les paramètres suivant sont optionnels. Ce sont les valeurs a affiché qui sont indiqué dans la chaîne de caractères de la façon suivante :

A chaque caractère « % » on regarde la lettre d'après pour savoir le type de la valeur comme indiquer dans ce tableau

Type	Lettre
int	%d / %i
long	%ld
float/double	%lf / %f
char	%c
string (char*)	%s
pointeur (void*)	%p
short	%hd
entier hexadécimal	%x

Printf retourne la taille de la chaîne de caractères ou une valeur négative s'il y a une erreur.

La fonction scanf sert à récupérer une ou plusieurs valeurs rentrées au clavier et à la stocké dans une ou plusieurs variables. Elle vient également de la bibliothèque <stdio.h>. Elle a pour prototype :

```
int scanf(const char * format, …);
```

Le premier paramètre contient la chaîne de caractères avec le type des différentes valeurs à regarder indiquer par un « % » suivit d'une lettre qui indique le type. Il est possible d'indiquer la taille de la variable rajoutant un nombre entre le % et la lettre indiquant le type. Les autres paramètres contiennent les variables qui vont stockés les valeurs rentrées.

Scanf retourne le nombre de variable affectées par la saisie. Cela permet de voir si la fonction c'est bien exécuté.

Types : utiliser correctement les types de base

Les types classiques en C utilisés dans mes TP sont :

- Pour les entiers : int
- Pour les flottant : float

Il existe 5 type pour les entiers :

- char ;
- short int, ou plus simplement short ;
- int ;
- long int, ou long ;
- long long int, ou long long

Ils diffèrent par leur nombre de bit ce qui veut dire que leur taille mais du coup leur valeur minimal et maximal et également impacté.

- Char est sur 8 bits ; les valeurs vont de -128 à 127 ;
- Short est sur 16 bits ; les valeurs vont de -32 768 à 32 767 ;
- Int est sur 16 bits ou 32 bits en fonction du processeur; les valeurs vont de -32 768 à 32 767 ou de -2 147 483 648 à 2 147 483 647 ;
- Long est sur 32 bits ; les valeurs vont de -2 147 483 648 à 2 147 483 647 ;
- Long long est sur 64 bits ; les valeurs vont de −9 223 372 036 854 775 807 à 9 223 372 036 854 775 807 ;

Char sert a conservé des caractères, ils ont une valeur numérique mais à l'affichage il ressort un caractère en fonction de la valeur du char.

Il existe 3 type pour les flottants :

- float ;
- double ;
- long double.

Comme pour les entiers le type indique une précision différente en fonction du nombre de bit (et donc de la taille)

- float est sur 32 bits ; La précision est de 6 chiffres ;
- double est sur 64 bits ; La précision est de 15 chiffres ;
- long double est sur 80 bits ; La précision est de 17 chiffres

Programme : notion de programme

Un programme c'est l'exécution d'une fonction appelé main. Elle a pour prototype :

```
int main(int argc, char *argv[]) ;
```

- argc est le nombre d'élément dans la table argv.
- argv est une liste contenant les valeur rentrer lors du lancement du programme. Le 1 er argument est toujours le nom du programme.

Une fois le programme compiler, on lance le programme depuis le Terminal en tapant

./prg test.txt 2

Alors le programme affichera dans le Terminal :

Nombre d'arguments passes au programme : 3

- argv[0] : './prg.exe'
- argv[1] : 'test.txt'
- argv[2] : '2

A la fin de la fonction main on retourne la valeur 0 pour dire que tout s'est bien passé.

Voici un exemple issu de mes TP

TP4

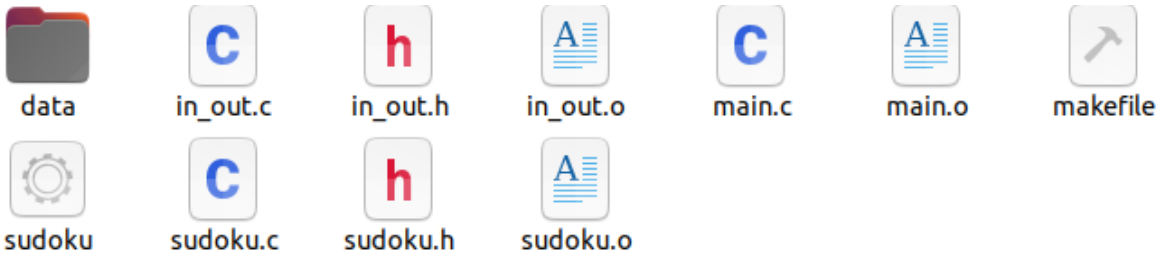
```c
int main()
{
    int empty = 0;
    stack_init();
    printf("size= %i\n", stack_size());

    empty = stack_is_empty();
    if (empty == 1)
        printf("VIDE\n");
    else
        printf("NON VIDE\n");

    stack_push(2);
    printf("size= %i\n", stack_size());

        empty = stack_is_empty();
    if (empty == 1)
                printf("VIDE\n");
        else
                printf("NON VIDE\n");


    stack_push(6);
    stack_push(7);
    stack_push(7);

    printf("top = %i\n", stack_top());

    stack_display();

    printf("element de 1 est: %i\n", stack_get_element(1));

    printf("element poper est: %i\n", stack_pop());

    stack_display();
    printf("size= %i\n", stack_size());

    return 0;
}
```

Modules : établir des architectures de sources modularisées

Pour organiser son programme, il est nécessaire de séparer son code en plusieurs fichier. Cela permettra de ranger les fonctions. Pour communiquer entre les fichiers, on utilise des headers qui contienne les prototypes des fonctions contenu dans le fichier c du même nom. On utilise pour à des :

#include « mesfunction.h »

Voici un exemple :



```c
#include <stdio.h>

#include "sudoku.h"
#include "in_out.h"

int main(int argc, char* argv[]){
  Board B;
  /*int i;*/

  fread_board(argv[1], B);

  print_board(B);
  printf("\n");
  solver(B);

  /*for (i = 0; i < 9; i++){
    printf("%i\n", i/3);
  }*/


  return 0;
}
```

Compilation : savoir compiler correctement les programmes

On compile en C grâce à gcc :

```
gcc -o nom_du_programme
```

On peut rajouter des option (des flags) pour s'imposer des restrictions sur la manière de coder afin d'éviter les erreurs et d'avoir un code plus propre :

Voici ceux que j'utilise

- W
- Wall
- Ansi
- pedantic

Afin de faciliter la compilation on utilise généralement de Makefile. Il facilite la compilation et permet de rajouter des commandes supplémentaires comme effacer les fichiers inutiles, ou nettoyer tous les exécutable.

Voici un exemple de Makefile issus de mes TP

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=exe

all: $(EXEC)

exe: main.o
    $(CC) -o exe main.o $(LDFLAGS)

main -o: main.c
    $(CC) -o main.o -c main.c $(CFLAGS)

clean:
    rm -rf *.o

mrproper: clean
    rm -rf $(EXEC)
```

Récursivité : utiliser la récursivité en C

La récursivité, c'est tout simplement le fait que la fonction s'appelle elle-même. Le raisonnement est toujours le même :

- Mettre en place les différents cas
- Prévoir le cas d'arrêt
- Penser au paramètre de la fonction appelée

Voici des exemples issus de mes TP

TP4

```c
int* merge_sort(int* array){
    int* first;
    int* second;

    if (array_size(array) == 1)
    {
        return array;
    }
    else
    {
        split_arrays(array, &first, &second);
        return merge_sorted_arrays(merge_sort(first),merge_sort(second));
    }


}
```

TP5

```c
void permutations(int buffer[], int current, int max)
{
  int i;
  if (current > max){
    print_tab(buffer);
  }
  else
  {
    for (i = 0; i < N; i++){
      if (buffer[i] == 0){
        buffer[i] = current;
        /*print_tab(buffer);*/
        permutations(buffer, current +1, max);
        buffer[i] = 0;
      }
    }
  }

}
```

TP6

```
void add_elem(Board grid){
  int *height = (int*)malloc(sizeof(int));
  int *width = (int*)malloc(sizeof(int));
  int i;

  parcourir(grid, height, width);
  /*printf("height1 = %i; width1 = %i;\n", *height, *width);*/
  /*printf("\n");*/
  if (*height != -1 && *width != -1)
  {
    for (i = 1; i < 10; i++)
    {
      grid[*height][*width] = i;

      if (elem_valide(grid, *height, *width) == 1)
      {
        /*print_board(grid);
        printf("height1 = %i; width1 = %i;\n", *height, *width);*/
        /*printf("oui\n");*/
        add_elem(grid);
        grid[*height][*width] = 0;
      }
      else
      {
        grid[*height][*width] = 0;
      }
    }
  }
  else
  {
    print_board(grid);
    printf("fini\n");
  }
}
```

Tableaux : maîtriser la notion de tableau en C

Les tableaux sont des listes permettant de stockée des variables du même type à diffèrent indice du tableau. Le tableau à une taille max. On définit un tableau comme ci-dessous

```
int tab[20];
```

20 est la taille du tableau.

Dans cette exemple le tableau est à 1 seul dimension mais il n'y a pas de limite à cette dernière.

Si l'on initialise avec des valeurs, pas besoin de donner la taille du tableau, celle-ci est implicite.

```
int tab[] = { 1, 2, 3 };
```

On entre une valeur à l'indice 2 d'un tableau comme ci-dessous

```
int tab[20];

tab[2] = 4 ;
```

Voici quelque utilisation possible d'un tableau.

TP6

```
int* init_tab()
{
  int* buffer = malloc(sizeof(int) * N);
  int i;
  for (i = 0; i<N; i++){
    buffer[i] = 0;
  }
  return buffer;
}
```

```
void print_tab(int buffer[]){
  int i;
  printf("[");
  for (i = 0; i < N - 1; i++){
    printf("%i, ", buffer[i]);
  }
  printf("%i]\n", buffer[i]);
}
```

Pointeurs : utiliser adresses et valeurs pointées des variables

Les pointeurs permettent de commencer à jouer avec la mémoire. Un pointeur est une valeur nommé adresse qui indique l'adresse d'une valeur dans la mémoire. Un pointeur à un type qui est le type de l'objet pointé. On accédé à la valeur de l'adresse d'une variable en précèdent la variable du caractère « & ». On accède à la valeur pointée par un pointeur grâce au caractère « * »

```
int i = 3;
int *p;

p = &i;
```

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000

Il est possible de parcourir les pointeurs comme un tableau, on parcourt ainsi les objets stockés à des adresse voisine.

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
{
  int i;
  int *p;
  p = tab;
  for (i = 0; i < N; i++)
    printf(" %d \n", p[i]);
}
```

Voici quelque exemple d'utilisation de pointeur issus de mes TP

TP5

```c
int array_size(int* array){
    int compteur = 0;
    int* tmp = array;
    if (array == NULL)
        return compteur;
    while (*tmp != -1)
    {
        tmp++;
        compteur++;
    }
    return compteur;
}

void print_array(int* array){
    int* tmp = array;
    if (array != NULL)
    {
        printf("(");
        while (*tmp != -1 && *(tmp+1) != -1)
        {
            printf("%d, ", *tmp);
            tmp++;
        }
        if (*tmp != -1)
            printf("%d", *tmp);
        printf(")\n");
    }
}
```

TP6

```c
void split_arrays(int* array, int** first, int** second){
    int compteur;
    int compteur2;

    *first = allocate_integer_array(array_size(array)/2);
    *second = allocate_integer_array(array_size(array) - (array_size(array)/2));

    for (compteur = 0; compteur < array_size(array)/2; compteur++)
    {
        (*first)[compteur] = array[compteur];
    }
    (*first)[compteur]= -1;

    for (compteur2 = 0; compteur2 < array_size(array) - (array_size(array)/2); compteur2++)
    {
        (*second)[compteur2] = array[compteur + compteur2];
    }
    (*second)[compteur2] = -1;
}
```

```c
void parcourir(Board grid, int *height, int *width){
  int i, j;
  *height = -1;
  *width = -1;
  for(i = 0; i < 9; i++)
  {
    for (j = 0; j < 9; j++)
    {
      /*printf("height = %i; width = %i;\n", i, j);*/
      if (grid[i][j] == 0)
      {
        *height = i;
        *width = j;
        /*printf("height = %i; width = %i;\n", *height, *width);*/
        return;
      }
    }
  }

}
```

Fichier : gérer les fichiers et leur contenu en C

Il est possible de lire un fichier pour récupérer des valeurs ou en afficher dans ce dernier.

Il faut d'abord ouvrir un fichier. Pour ça on utilise la fonction fopen de la bibliothèque <stdio.h>. Cette fonction renvoie un pointeur sur un fichier. Elle prend en paramètre le nom du fichier qu'on veut ouvrir, et le mode d'ouverture.

Les mode d'ouverture permette de dire ce qu'on va faire avec le fichier :

- **"r": lecture seule**. Vous pourrez lire le contenu du fichier, mais pas y écrire. *Le fichier doit avoir été créé au préalable.*
- **"w": écriture seule**. Vous pourrez écrire dans le fichier, mais pas lire son contenu. *Si le fichier n'existe pas, il sera créé.*
- **"a": mode d'ajout**. Vous écrirez dans le fichier, en partant de la fin du fichier. Vous ajouterez donc du texte à la fin du fichier. *Si le fichier n'existe pas, il sera créé.*
- **"r+": lecture et écriture**. Vous pourrez lire et écrire dans le fichier. *Le fichier doit avoir été créé au préalable.*
- **"w+": lecture et écriture, avec suppression du contenu au préalable**. Le fichier est donc d'abord vidé de son contenu, vous pouvez y écrire, et le lire ensuite. *Si le fichier n'existe pas, il sera créé.*
- **"a+": ajout en lecture / écriture à la fin**. Vous écrivez et lisez du texte à partir de la fin du fichier. *Si le fichier n'existe pas, il sera créé.*

Le prototype de la fonction fopen :

```
FILE* fopen(const char* nomDuFichier, const char* modeOuverture);
```

Voici ci-dessous un exemple de l'utilisation fopen

```
FILE *fichier = fopen("sort.dat", "w+");
if (fichier != NULL)
{
    for (size = 10; size <= MAX_SIZE; size+=100)
    {
        tab1 = create_array(size);
        tab2 = create_array(size);
        tab3 = create_array(size);
        fill_random_array(tab1, size, max_value);
        copy_array(tab1, tab2, size);
        copy_array(tab1, tab3, size);

        nb_less = 0;
        nb_swap = 0;

        selection_sort(tab1, size);
        fprintf(fichier, "\n%d %d", size, nb_less);

        nb_less = 0;
        nb_swap = 0;

        insertion_sort(tab2, size);
        fprintf(fichier, " %d", nb_less);

        nb_less = 0;
        nb_swap = 0;

        quick_sort(tab3, size);
        fprintf(fichier, " %d", nb_less);

        free(tab1);
        free(tab2);
        free(tab3);
    }
    /*printf("%d comparaisons\n", nb_less);
```

Après l'avoir ouvert, il faut fermer le fichier pour cela ou utilise la fonction fclose. Cela permet de libérer la mémoire en supprimant le fichier chargé dans la mémoire vive.

Son prototype est :

```
int fclose(FILE* pointeurSurFichier);
```

Elle renvoie une valeur indiquant si la fermeture à réussit :

- 0: si la fermeture a marché ;
- EOF: si la fermeture a échoué. EOF est undefine situé dans <stdio.h> qui correspond à un nombre spécial, utilisé pour dire soit qu'il y a eu une erreur, soit que nous sommes arrivés à la fin du fichier. Dans le cas présent cela signifie qu'il y a eu une erreur.

Pour écrire dans un fichier il existe plusieurs fonction.

- fputc: écrit un caractère dans le fichier (UN SEUL caractère à la fois) ;
- fputs: écrit une chaîne dans le fichier ;
- fprintf: écrit une chaîne « formatée » dans le fichier, fonctionnement quasi-identique à printf.

Elles ont pour prototype :

```
int fputc(int caractere, FILE* pointeurSurFichier);
```

Elle renvoie un int pour indiquer si la fonction à bien marché, sinon elle renvoie EOF. Caractere est un int qui marche comme un char sauf que le nombre de caractères utilisable est plus grand.

```
char* fputs(const char* chaine, FILE* pointeurSurFichier);
```

Elle marche comme fputc sauf qu'elle écrit une chaîne de caractère.

```
int fprintf(FILE* pointeurSurFichier, const char* format, ...);
```

Elle marche comme printf à la différence qu'elle prend en 1er argument le fichier ou elle va écrire.

Il existe 3 fonctions équivalente pour lire un fichier :

- fgetc: lit un caractère ;
- fgets: lit une chaîne ;
- fscanf: lit une chaîne formatée.

```
int fgetc(FILE* pointeurDeFichier);
```

Elle renvoie la valeur du caractere lu.

Cette fonction avance le curseur (la position actuelle dans le fichier) d'un caractère.

```
char* fgets(char* chaine, int nbreDeCaracteresALire, FILE*
pointeurSurFichier);
```

Elle prend en paramètres 1 chaine da caractère qui va récupérer la lecture du fichier, un int pour le nombre de caractère à récupérer, et le fichier à lire. La fonction lit au maximum une ligne. Pour lire plusieurs lignes, il faut faire une boucle jusqu'à ce que la fonction renvoie NULL.

```
int fscanf(FILE* pointeurSurFichier, const char * format, …);
```

Elle fonctionne comme scanf mais elle prend en paramètre le pointeur du fichier

Bibliothèques : bibliothèques et packaging en C

Une bibliothèque est un ensemble de fonction déjà construite qui peuvent être utilisé grâce à un :

#include <stdio.h>

Il est possible de créer soi-même des bibliothèques mais je ne sais pas encore comment faire.