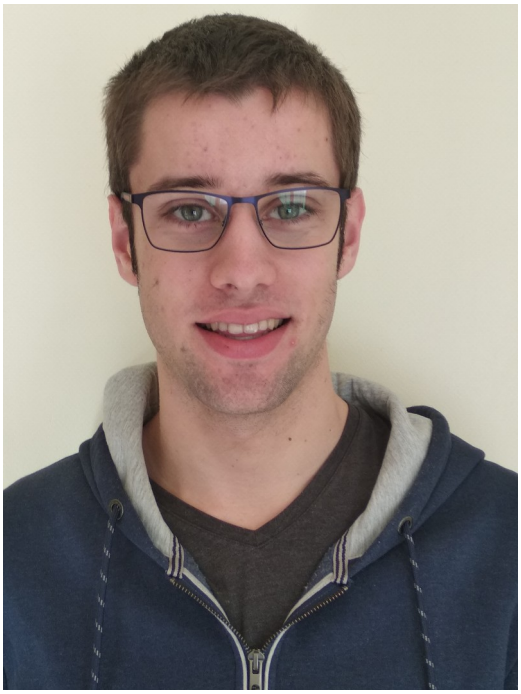
PortFolio

sur

la programmation en C

De

VALADE Jérémy



Étudiant à l'ESIPE

(École Supérieure d'Ingénieurs à l'université Gustave Eiffel)

Sommaire

Présentation....3
Qui suis-je ?....3
Quel est l’objectif de ce portfolio ?....3
Quel outil j’utilise pour programmer en C ?....3
Quel est mon objectif en C?....3
Acquis d’Apprentissage Visés pour la programmation en langage C....4
Qu’est ce que c’est ?....4
Compétences....4
Programmation impérative avec le langage C - niveau 1....5
Concevoir et écrire des fonctions C et des petits programmes C simples s’exécutant en console....5
Produire du code source C de qualité standard....12
Maîtriser son environnement de production de code source C....13
Finaliser et documenter un petit livrable écrit en C....15
Programmation impérative avec le langage C - niveau 2....19
Pratiquer la programmation modulaire en C....19
Concevoir et écrire des programmes C de taille moyenne....22
Maîtriser l’approche binaire de la manipulation des données dans le langage C....27
Recourir correctement à l’allocation dynamique de mémoire lorsque c’est nécessaire ou demandé....28

Présentation

Qui suis-je ?

Avant d'arriver à l'Esipe, j'ai fait un **BTS SIO** (Service Informatiques aux Organisations) **SLAM** (Solutions logicielles et applications métiers forme des spécialistes des logiciels) au lycée René Descartes à Champs-sur-Marne. Dans cette formation j'ai appris à programmer dans les langages suivants :

- Java
- les langages de programmation web (**HTML**, **CSS**, **Javascript**, **PHP**)
- Base de données (**SQL** et **PSQL**)
- des frameworks (**CodeIgniter** (PHP), **Bootstrap** (Css) et **Hibernate** (java)).

Pour en savoir plus, vous pouvez aller sur mon PortFolio du BTS qui se situe à l'adresse web suivante : **http://jeremy.valade.free.fr/PortFolio**.

Néanmoins je n'ai jamais programmer en C pendant cette formation.

Quel est l'objectif de ce portfolio ?

L'objectif de ce portfolio est d'écrire un recueil structuré de preuve illustrant l'effectivité de mon apprentissage sur la programmation en C. Celui-ci doit me permettre de prouver que je possède des compétences en programmation C.

Afin de valider des compétences dans ce domaine, l'ESIPE a mis en place les **AAV (Acquis d'apprentissage Visé)** qui seront définis ultérieurement. Néanmoins l'ESIPE a fixé trois niveaux en programmation C où le dernier niveau me permettrait de travailler en tant que développeur en C dans le monde professionnel.

Quel outil j'utilise pour programmer en C ?

Afin de pouvoir programmer facilement en C, il faut utiliser **Linux**. Par conséquent j'ai choisi la distribution **Ubuntu** parce que je suis très familiarisé avec celle-ci. De plus pour développer en C, j'utilise l'éditeur de texte **Vim** inclus dans la distribution qui est un outil très puissant.

Quel est mon objectif en C?

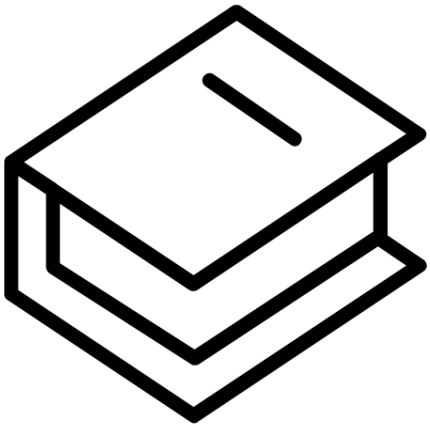
Mon objectif est d'acquérir des compétences en programmation C afin de devenir un bon développeur. Cela passe par la validation des trois niveaux des AAV.

Comment les obtenir ?

Dans ma formation, on peut soit choisir soit créer nos projets afin de justifier les compétences acquises. Néanmoins étant néophyte sur le langage C, j'ai décidé de suivre les TP du fil rouge proposés par notre professeur Nicolas Borie.

Pour conclure la démarche de l'ESIPE est de justifier que tu sais faire et pas que ça fonctionne.

Acquis d’Apprentissage Visés pour la programmation en langage C



Qu’est ce que c’est ?

C’est juste la description d’une compétence, d’une capacité ou d’un comportement qui correspond à une tache standard que relève naturellement d’un futur diplômé. C’est quelque chose qu’il faut savoir faire, voire, dont vous devez démontrer la maîtrise.

Les AAV, c’est ce qu’on attend de vous pour pouvoir certifier votre apprentissage. Il faut le voir comme un objectif durant votre développement personnel et comme un achèvement une fois que vous être capable de rassembler des preuves illustrant votre bonne maîtrise de la compétence attendue.

Il est important que ces AAV soient très clairs pour vous sinon cela signifie que soit l’objectif est mal formulé, soit que vous ne comprenez pas ce que l’on attend de vous. Plus c’est clair, plus le cap est identifié, plus simplement et rapidement on arrive à attendre son but.

Compétences

Les 4 grandes compétences ESIPE info :

- **Conception** : Concevoir une solution informatique argumentée (avant le clavier)
- **Production** : Mettre en œuvre une solution informatique adaptée (avec le clavier)
- **Projet** : Agir en collaborateur responsable au sein d’un projet informatique (équipe et outils)
- **Com/doc** : Assurer une communication technique adaptée (para-productions autour des livrables)

Programmation impérative avec le langage C - niveau 1

Concevoir et écrire des fonctions C et des petits programmes C simples s'exécutant en console

AAV(conception) : Expliquer oralement en une minute ou en quelques lignes à l'écrit un choix de prototype de fonction C réaliste à partir d'un petit texte en français de la description d'une fonctionnalité simple.

Le prototype d'une fonction est un mode d'emploi pour le programmeur voulant utiliser celle-ci. Ainsi bien choisir son prototype est primordiale dans un programme. La fonction doit répondre aux questions suivantes :

- Qu'est ce qu'effectue la fonction ?
- Quels sont les paramètres de la fonction ?
- Qu'est ce que la fonction retourne ?

Afin de justifier que j'ai acquis cette compétence, je vais l'illustrer avec l'exercice 1 du TP3.

L'objectif du programme de cet exemple est de calculer la puissance d'un nombre entier **n** à la puissance d'un nombre entier **a** supérieur à 0 de manière itérative et récursive. Par exemple si **n = 2 et a = 3**, la fonction devrait retourner **8 (2^3).**

Grâce à l'énoncé de l'exercice, j'en déduis que la fonction aura deux entiers comme paramètres et elle retournera le résultat $\mathbf{n^a}$. Ainsi le prototype de la fonction serait **int puissance(int n, int a);**

Test du programme avec n = 2 et a = 3

```
Fonction iterative_puissance(2, 3) : 8
Fonction recursive_puissance(2, 3) : 8
```

AAV(conception) : Établir plusieurs prototypes de fonctions élémentaires C (fonctions réalisant une seule tâche) dont l'assemblage va réaliser une fonctionnalité de taille moyenne décrite en français.

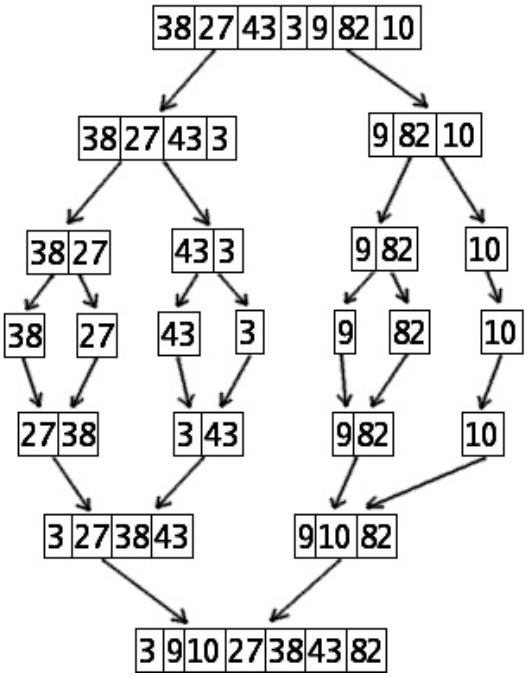
En programmation ou dans d'autres domaines, plus les tâches seront réparties plus le travail sera facilité. Par exemple les bugs seront plus centré, le code et la répartition du travail sera plus clairs, ...

Afin de valider cette compétence, je vais implémenté le tri fusion qui correspond au TP5 du fil rouge.

Qu'est ce que le tri fusion ?

Le tri fusion est un algorithme de tri. De plus sa complexité pour une entrée d'un tableau de **taille n** est de l'ordre de **n log n**. Cet algorithme est basé sur la technique algorithmique **diviser pour régner**. (Wikipédia)

Comment fonctionne l'algorithme ?



Pour réaliser le tri par fusion, le programme possède trois fonctions :

- **split_arrays** : cette fonction divise un tableau en deux sous-tableaux
- **merge_sorted_arrays** : cette fonction retourne un tableau trié en fusionnant deux tableaux
- **merge_sort** : cette fonction retourne un tableau trié en appelant successivement les fonctions **split_arrays** et **merge_sorted_arrays** de manière récursive

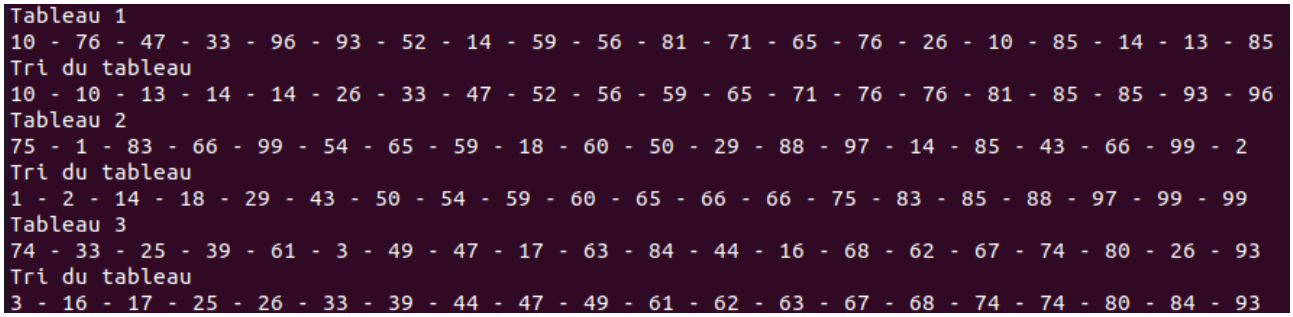
Fonction merge_sort soit le tri fusion

```
/* Cette fonction trie un tableau en utilisant le tri fusion */
int* merge_sort(int* array) {
        if (array_size(array) <= 1)
                return array;
        else {
                int* tab4 = NULL;
                int* tab5 = NULL;
                split_arrays(array, &tab4, &tab5);

                return merge_sorted_arrays(merge_sort(tab4), merge_sort(tab5));

        }
}
```

Test du tri fusion sur des tableaux aléatoires



En conclusion on observe que l'assemblage de ces différentes fonctions permettent d'appliquer l'algorithme de tri fusion.

AAV(production) : En 5 minutes, écrire, compiler et exécuter un programme C minimal qui produit un affichage simple à l'écran durant son exécution.

Pour valider cette compétence, je vais m'appuyer sur le programme du TP2 de l'exercice 5. Celui-ci consiste à afficher son propre code source sur la console (l'écran).

Le programme se déroule ainsi :

1. Ouvrir le fichier du programme
2. Afficher caractère par caractère du fichier à l'aide d'une boucle **while** et de la fonction **fgetc**
3. Arrêter le programme lorsqu'il n'y a plus de caractères à afficher
4. Fermeture du fichier

Afin de compiler le programme, tapez la commande **make**. Ensuite vous pouvez exécuter le programme en lançant la commande **./exe** . Enfin vous pouvez supprimer les fichiers **.o** et **l'exécutable** en tapant la commande **make clean**.

AAV(conception) : Concevoir intégralement en moins de 30 minutes l'organisation d'un programme C simple qui s'exécute en console et qui opère des interactions (écran/claviers) avec l'utilisateur durant son exécution. Le cahier des charges de ce programme sera donné par un texte court en français qui explicite ses fonctionnalités et son comportement.

Afin de justifier de l'obtention de cette compétence, j'ai décidé de programmer le jeu du **Nombre mystère.**

Déroulement du Nombre mystère

1. Le jeu récupère un nombre aléatoire entre deux bornes (Dans l'exemple : 1 et 100).
2. L'objectif du joueur est de trouvé le nombre mystère en saisissant des nombres dans le programme.
3. Le joueur saisi un nombre
4. Le programme vérifie si ce dernier n'est pas une chaîne de caractère et que le nombre soit compris entre les bornes incluses sinon retour à l'étape 3.
5. Ensuite le programme indique si le nombre mystère est strictement plus petit ou strictement plus grand que le nombre saisi. Sinon cela signifie que le nombre est égale au nombre mystère. Ainsi on passe à l'étape 6 sinon retour à l'étape 3.
6. Enfin quand le joueur a trouvé le nombre mystère, on lui indique le nombre de coups qu'il a fait pour le trouver.

Exemples de parties

```
Nombre Mystère !!!
Saisir un nombre entre 1 et 100
50
Le nombre mystère est plus petit

Saisir un nombre entre 1 et 100
25
Le nombre mystère est plus grand

Saisir un nombre entre 1 et 100
37
Le nombre mystère est plus petit

Saisir un nombre entre 1 et 100
31
Le nombre mystère est plus grand

Saisir un nombre entre 1 et 100
34
Le nombre mystère est plus petit

Saisir un nombre entre 1 et 100
33

Vous avez gagné!!! Félicitation
Nombre de coups = 6
```

```
Nombre Mystère !!!
Saisir un nombre entre 1 et 100
50
Le nombre mystère est plus grand

Saisir un nombre entre 1 et 100
75
Le nombre mystère est plus grand

Saisir un nombre entre 1 et 100
87
Le nombre mystère est plus grand

Saisir un nombre entre 1 et 100
95
Le nombre mystère est plus petit

Saisir un nombre entre 1 et 100
90
Le nombre mystère est plus grand

Saisir un nombre entre 1 et 100
93
Le nombre mystère est plus petit

Saisir un nombre entre 1 et 100
92
Le nombre mystère est plus petit

Saisir un nombre entre 1 et 100
91

Vous avez gagné!!! Félicitation
Nombre de coups = 8
```

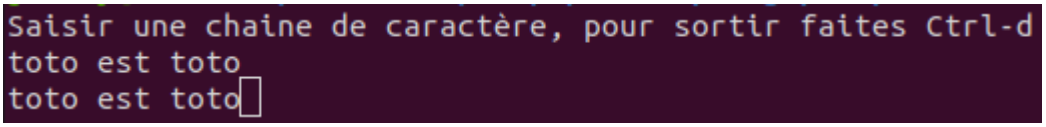
AAV(production) : Produire des programmes C récupérant, exploitant et permettant la visualisation de données simples (nombres divers et chaînes de caractères) fournis au clavier par l'utilisateur durant l'exécution.

Pour justifier l'acquisition de cette compétente, je vais utiliser l'exercice 2 du TP3. Cet exercice consiste à reproduire la commande **wc** de **linux** qui compte le nombre de caractères, de mots et de lignes écrit par l'utilisateur.

Exécution du programme

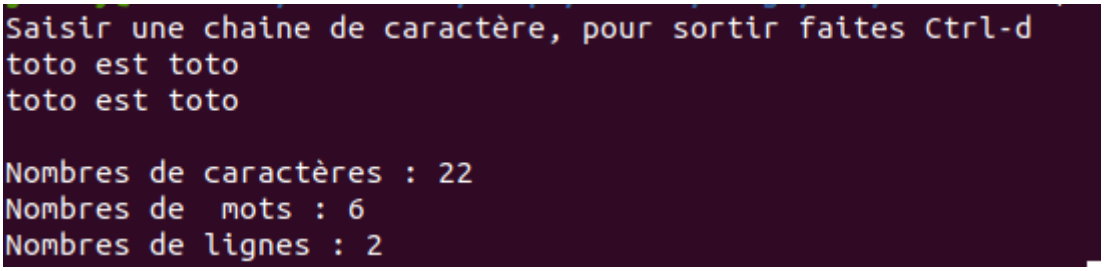
1. Lancement

2. la personne écrit ce qu'elle souhaite



3. Résultat du programme

Pour connaître le nombre de caractères, de mots et de lignes, on va récupérer les caractères saisi par l'utilisateur avec la méthode **getchar().** Tant que ce dernier n'a pas tapez **Ctrl-d**, le programme continu. A la fin du programme, on affiche le nombre de caractères, de mots et de lignes.



AAV(production) : Produire en une quinzaine de minutes une fonction C simple (affichage, moyenne, min, max, somme, occurrences, etc…) exploitant des tableaux sur des types de base (nombres ou caractères). Mettre aussi en scène la fonction dans un programme pour illustrer son fonctionnement.

Afin de justifier que je sais créé en moins d'une quinzaine de minutes une fonction C simple. J'ai décidé de créé un projet **Tableau** où il y a les fonctions permettant de calculer la moyenne, le minimum, le maximum d'un tableau. Enfin une dernière permettant de l'afficher. Ces fonctions sont écrites dans le fichier **tableau.c**.

Arborescence du projet

```
jeremy@valade:~/Documents/Esipe/IINFO1/ProgC/Projet/Tableau$ ll
total 24
drwxrwxr-x 2 jeremy jeremy 4096 janv. 24 19:27 ./
drwxrwxr-x 4 jeremy jeremy 4096 déc.  24 10:59 ../
-rw-rw-r-- 1 jeremy jeremy  623 déc.  23 15:39 main.c
-rw-r--r-- 1 jeremy jeremy  180 déc.  23 15:34 Makefile
-rw-rw-r-- 1 jeremy jeremy  850 janv.  7 10:39 tableau.c
-rw-rw-r-- 1 jeremy jeremy  331 déc.  23 15:39 tableau.h
```

Test des diverses fonctions

```
Affichage tableau
[15, 16, 17, 18, 19, 20]
Moyenne du tableau : 17
Minimum du tableau : 15
Maximum du tableau : 20
Affichage tableau
[0, 6, 7, 8, 19, 14]
Moyenne du tableau : 9
Minimum du tableau : 0
Maximum du tableau : 19
Affichage tableau
[15, 13, 6, 1, 19, 20]
Moyenne du tableau : 12
Minimum du tableau : 1
Maximum du tableau : 20
```

Pour conclure dans ce projet, les fonctions renvoient bien les résultats attendus (capture d'écran ci-dessus). De plus il y a un makefile permettant de compiler (**make**) le programme et de nettoyer l'arborescence (**make clean**). Enfin les fichiers sont bien définis.

AAV(production) : Utiliser correctement les fonctions simples de la librairie standard portant sur les chaînes de caractère (strlen, strcpy, strcmp et leur dérivés).

Afin de connaître les diverses fonctions de la librairie portant sur les chaînes de caractères (**string.h**), on tape la commande **man 3 string** dans le terminal afin d'obtenir des informations sur celles-ci.

Afin de justifier que j'ai acquis cette compétence, j'ai créé un programme contenu dans le répertoire **String** en testant divers fonctions de la librairie **string.h**.

Les fonctions testées sont les suivantes :

- strlen
- strcopy
- strcmp
- strcat
- strchr
- srfry

Exécution du programme

```
Affichage de la chaine pour les tests : TEST
Test de la fonction strlen
Longueur de la chaine doit être égale à 4 : vrai

Test de la fonction strcopy
newChaine n'est pas initialisé : ô◆◆◆
Copie de la chaine 'TEST' dans la variable newChaine : TEST

Test de la fonction strcmp
Cette fonction compare deux chaines de caractères
Vérifions en comparant une chaine caractère à elle-même
Comparons la variable chaine à elle-même : les chaines sont identiques
Comparons la variable chaine à la chaine 'TOTO' : les chaines ne sont pas identiques

Test de la fonction strcat
Fonction qui ajoute une chaine à une autre
Test sur la chaine 'TEST', le programme ajoutera ' SUCCESS'
Résultat de la chaine finale : TEST SUCCESS

Test de la fonction strchr
Chaine : TOTO
On va remplacer les lettres 'O' de la chaine par les lettres : 'A'
Chaine remplacé : TATA

Test de la fonction srfry qui change aléatoirement une chaine
Chaine 'TEST' :
Echange 1 : SETT
Echange 2 : ESTT
Echange 3 : TTES
```

Ce programme permet de justifier que je sais recherché les informations nécessaires dans la documentation afin de pouvoir utiliser les fonctions de la librairie.

Produire du code source C de qualité standard

AAV(production) : Savoir indenter agréablement (avec constance) un code source C d'une centaine de lignes (avec des doubles espaces ou bien des tabulations).

Pour rappel, j'utilise l'éditeur de texte **vim** pour programmer en C. Pour indenter un programme en C sur cet éditeur il faut tapez **échap** puis lancer la commande suivante **gg=G** qui indentera toutes les lignes du fichiers.

• AAV(production) : Produire du code source C dont la compilation réussit et ne produit pas de warning même en activant les flags de contrôle de qualité (-Wall et -ansi).

Tous les projets que j'ai rendu excepté l'exercice 4 du TP2 (Exercice écrit par le professeur montrant comment utiliser les pointeurs), ces derniers compilent avec les flags -**Wall, -ansi et -pedantic**.

AAV(production-projet) : Organiser le contenus des fichiers sources avec rigueur et constance (bibliothèques, directives préprocesseur, sauts de lignes entre fonctions, placement de la documentation, entêtes et méta-informations)

Les différents fichiers d'un projets doivent contenir des fonctions précises. Ainsi l'organisation du contenu dans chaque fichier est important.

Par exemple dans le TP7 qui est la conception d'un sudoku graphique avec la librairie MLV. J'ai divisé ce projet en trois parties :

- la partie graphique
- la gestion des variables du jeu
- l'exécution du jeu

Ainsi les différentes parties sont indépendantes des unes des autres. De plus la lecture d'un fichier ou des reviews vous montrera que mon code est aéré et simple à lire.

AAV(projet-com/doc) : Produire du code source C dans lequel tous les identifiants (nom de fonctions, noms de variable, nom de macros) ont du sens et se suffisent pour deviner ce qu'ils représentent et quelle est leurs utilités dans le contexte

et

AAV(projet) : Commenter son code de manière à optimiser toute relecture par une autre personne qui voudrait comprendre en profondeur votre code source

Un code doit être lisible et compréhensible par d'autres personnes que juste par celui qu'il a codé. Pour que ce dernier soit lisible, les variables utilisés doivent être clair, signifier quelque chose.

Par exemple pour la taille d'un tableau, si une variable contenant la taille d'un tableau est **a**, on ne le saura pas au premier coup d'œil contrairement au nom suivant qui pourrait être **taille**. Petite anecdote : le développement du noyau Linux est très peu commenté. Les noms des différentes acteurs suffisent à la compréhension du code.

De plus on peut rajouter des commentaires dans le code permettant d'expliquer celui-ci ou des points précis dans ce dernier. Néanmoins il ne doit pas en avoir en trop, cela risque de nuire à la lisibilité du code.

Afin de vérifier que j'ai acquis cette compétence, j'ai demandé à plusieurs de mes camarades de faire des reviews sur mes TP et vous pouvez le vérifier en regardant les fichiers.

Maîtriser son environnement de production de code source C

AAV(projet) : Utiliser un éditeur capable de colorer les mots clés du langage C

L'éditeur **vim** permet de colorer les mots clé en C. On peut changer la couleur des mots clé avec la commande **:colorsheme**.

Capture d'écran du fichier opération.c du TP7

```c
#include <stdio.h>
#include "objet.h"

#define N 9

int fread_board (const char *file, Board board)
{
        FILE *f;
        int i, j;
        int entry;

        f = fopen (file, "r");
        if (f == NULL)
        {
                fprintf (stderr, "Erreur d'ouverture du fichier %s\n", file);
                return 0;
        }

        for (i = 0; i < N; i++)
        {
                for (j = 0; j < N; j++)
                {
                        fscanf (f, "%d", &entry);
                        board[i][j] = entry;
                }
        }
        return 1;
}
```

AAV(projet) : Être capable en moins d'une minute de renommer une variable dans l'intégralité d'un code source et quel que soit sa longueur à l'aide d'une fonctionnalité du type 'search-replace'.

Pour renommer une variable dans l'intégralité du code source dans l'éditeur **vim**, on doit utiliser la commande suivante **:%s/nom_var/var_renomme/g**.

Davantage d'explications sur les différentes fonctionnalités de search-replace à cette adresse web : https://vim.fandom.com/wiki/Search_and_replace .

AAV(projet) : Organiser efficacement le stockage de vos productions dans un système de fichiers, utiliser des noms de fichiers/dossiers vous permettant rapidement de retrouver vos productions.

Savoir organiser les ressources d'un programme est très important car cela permet d'appréhender plus facilement le programme. Étant donnée que les TP du fil rouge ne sont pas des projets où on a besoin de créer plein de fichiers **.c ou .h**, j'ai décidé de stocké tous les fichiers à la racine du TP. Le faible nombre de fichiers apporte une grande compréhension sur l'exercice.

De plus vous pourriez regarder les différentes reviews de mes camarades afin de vous forger votre propre avis sur la validation de cette compétence.

AAV(projet) : Utiliser un compilateur C pour générer vos exécutables à partir des sources. Contrôler les arguments et options donnés au compilateur.

Pour compiler mes programmes C, j'utilise le compilateur **GCC**. Ensuite pour contrôler mes arguments et options données au compilateur, j'utilise des variables contenant ces dernières comme **CFLAGS** (contient les flags (-Wall, ...), **OBJ** (fichier .o), ...

Par exemple dans le sudoku graphique du TP7, le makefile contient le nom du compilateur, les flags, les objets à compiler et les autres flags pour les libraires. (**-lMLV** : libraire graphique).

Capture d'écran du makefile du TP7 ci-dessous

```
# Makefile TP 7

CC=gcc
CFLAGS=-Wall -ansi -pedantic
OBJ=interface.o main.o operation.o
LDFLAGS=-lMLV

calc: $(OBJ)
	$(CC) -o calc $(OBJ) $(LDFLAGS)

interface.o: interface.c interface.h objet.h
	$(CC) -c interface.c -o interface.o $(CFLAGS) -lMLV

operation.o: operation.h operation.c objet.h interface.h
	$(CC) -c operation.c -o operation.o $(CFLAGS) -lm

main.o: main.c objet.h interface.h operation.h
	$(CC) -c main.c -o main.o $(CFLAGS)

clean:
	rm -f *.o
	rm -f calc
```

AAV(projet) : Retrouver un moins d'une minute une information simple à propos d'une fonction de la librairie standard dans le manuel du programmeur Linux (via le terminal Unix ou le web).

Étant donnée que je suis sur Ubuntu, afin de retrouver une fonction de la librairie du **C**, on utilise la commande suivante **man 3 nomFonction**. Astuce : si on a oublié la fin d'une fonction on peut utiliser la **tabulation** pour **l'autocomplétion**.

Finaliser et documenter un petit livrable écrit en C

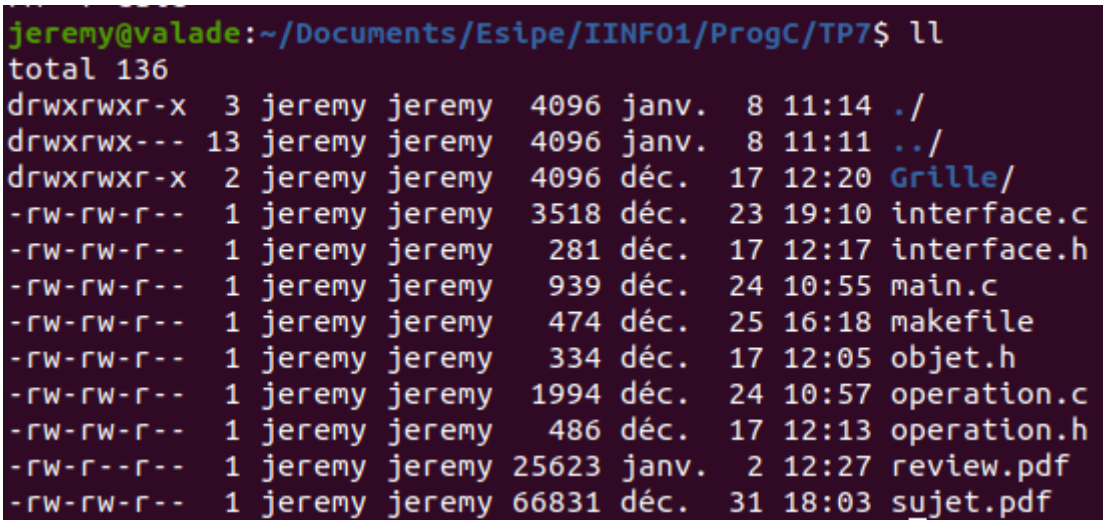
AAV(production-com/doc) : Mettre en place des instructions de compilation ou produire un makefile simple pour générer les programmes à partir de leurs sources.

Chaque TP ou Projet possède un makefile simple permettant de compiler le programme facilement. De plus il y a juste à appuyer sur **make** pour compiler le programme ou sur **make clean** pour nettoyer le répertoire. Aussi les variables contenu en haut du fichier comme les fichiers .o ou les flags permettent de rajouter des fichiers .o à généraux ou des flags sans modifier le makefile.

AAV(production) : Produire des archives complètes et concises, contenant sources et documentations optimisant la réception des solutions C livrables.

En C, produire une archive complète et concise revient à transmettre uniquement les sources (fichier.c), la documentation des différents projets. Pour être précis étant donné que j'ai réalisé les TP du fil rouge, la documentation revient au sujet de ces derniers.

Par exemple pour le TP7 du sudoku graphique on a les fichiers ci-dessous

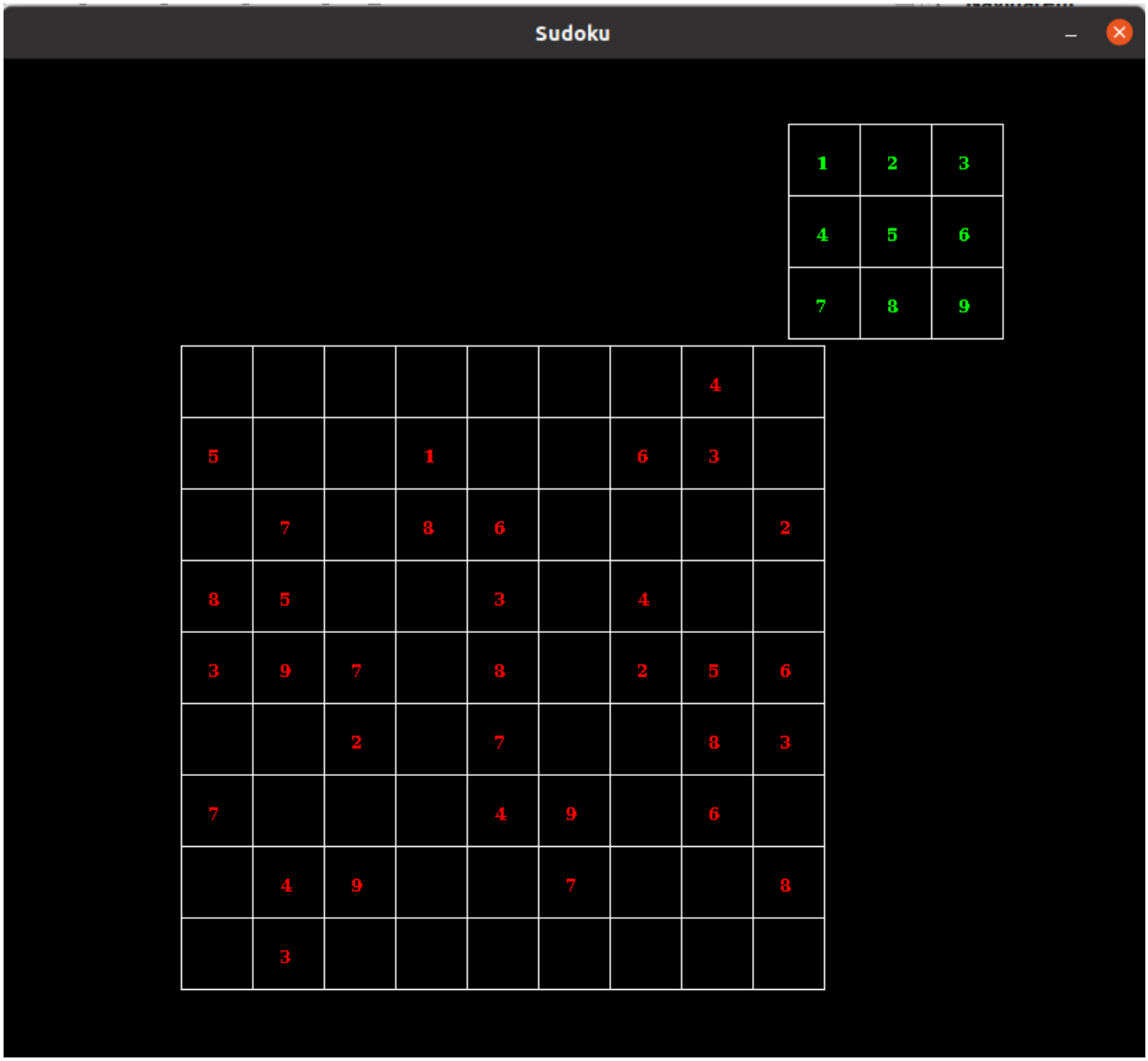


AAV(com/doc) : Être capable d'expliquer et de justifier simplement ses choix de programmation lors de la réalisation d'un petit projet informatique en C.

Un projet informatique ne se code pas sans réfléchir même un petit. Il faut réfléchir comment organiser les différentes ressources, fonctions, ... Cela permet d'être efficace sur la conception et la direction du projet.

Je vais illustrer cette compétence par le TP7 qui est de réaliser un sudoku graphique.

Capture d'écran du Sudoku



Pour réaliser ce jeu, je l'ai diviser en plusieurs parties qui sont les suivantes : l'exécution du programme, la gestion du jeu et la partie graphique.

J'ai divisé ces trois parties dans des fichiers distincts, le **main.c** correspond à l'exécution des différentes étapes du jeu, le fichier **operation.c** correspond à la gestion du jeu, le fichier **interface.c** correspond à l'affichage du sudoku sur une fenêtre et le fichier **objet.h** contenant les différents objets permettant l'exécution du sudoku.

1. Fichier main.c

Ce fichier permet d'aligner les étapes du sudoku :

1. lancement du jeu
2. on clique sur une case
3. on met un nombre dans cette case
4. on recommence l'étape 2 jusqu'à qu'on est remplit la grille
5. on ferme le jeu

2. Fichier operation.c

Ce fichier possède les fonctions permettant d'initialiser le jeu. Les grilles du sudoku sont dans des fichiers intitulé **grid*.txt** (Remplacé le * par un chiffre). Lors de l'exécution du programme, on implémente cette grille dans un tableau de taille [9][9]. Puis on récupère la solution de cette grille avec la fonction **solveSudoku** du TP6 qui résout une grille du sudoku.

Enfin pour savoir si l'utilisateur a gagné, on compare la solution avec la grille qui l'est en train de remplir avec la méthode **array_equals**.

3. Fichier interface.c

Ce fichier contient toutes les fonctions permettant de tracer les éléments graphiques du sudoku et les fonctions de la librairie MLV (MLV_clear_window(), MLV_create_window, …) que j'ai besoin. Ainsi dans mon fichier **main**, il n'y a pas la ligne suivante : **#include <MLV/MLV_all.h>** ce qui permet de réduire le nombre de dépendances dans le fichier main.

De plus j'ai créé une structure intitulé sudoku dans le fichier **objet.h** qui stocke toutes les données caractéristiques du sudoku graphique.

```
struct graphiqueSudoku {
        char titre[40];
        int hauteur;
        int largeur;
        int longueurCote;

        int abscisseGrille;
        int ordonneeGrille;
        int abscisseProposiion;
        int ordonneeProposition;
};
```

Ainsi un utilisateur peut changer la ligne suivante dans le fichier **interface.c** ou dans une **interface graphique** (en théorie) sans connaître la programmation :

```
static const struct graphiqueSudoku sudoku =  {"Sudoku", 700, 800, 50, 125, 200, 550,  45};
```

La variable sudoku contient les différentes caractéristiques de la fenêtre (taille de celle-ci, position grille, …). Étant donnée que les fonctions du fichiers sont dépendantes de cette variable. Le fait de modifier celle-ci, on n'aura pas besoin de modifier le code des fonctions.

Pour conclure avec ce projet, la répartition des différentes parties du programme sont indépendantes les unes des autres. Ainsi on peut en modifier une sans que cela a des répercussions sur le reste du programme.

AAV(production-com/doc) : Démontrer le bon fonctionnement de vos livrables écrit en C. Mettre en place des tests simples dans vos programmes permettant à un correcteur/client de reproduire les preuves de bon fonctionnement aux premières exécutions.

Les programmes sont transmis seulement avec la documentation, les fichiers sources et les différentes reviews (critique du code).

Pour exécuter chacun de mes programmes, il faut juste lancer la commande **make** dans le répertoire où se situe le programme. De plus il y a une fonctionnalité **clean** permettant de nettoyer les fichiers **.o** et **l'exécutable**.

Ensuite pour chaque exercices de TP, il y a des tests automatiques justifiants le bon fonctionnement du programme, vous pouvez rajouter les vôtres dans le fichier **main.c**.

Par exemple dans le TP8, on implémente la suite de Syracuse. On veut connaître la longueur maximale du vol pour les nombres allant de 1 jusqu'à 200 millions.

Exemple de test :

Résultat du programme du professeur

```
nborie@perceval:$ time ./test
MAX_FLY : 953

real 0m9.151s
user 0m9.007s
sys 0m0.132s
```

Résultat de mon programme

```
jeremy@valade:~$ time ./exe
Max : 953

real    0m8,112s
user    0m7,314s
sys     0m0,797s
```

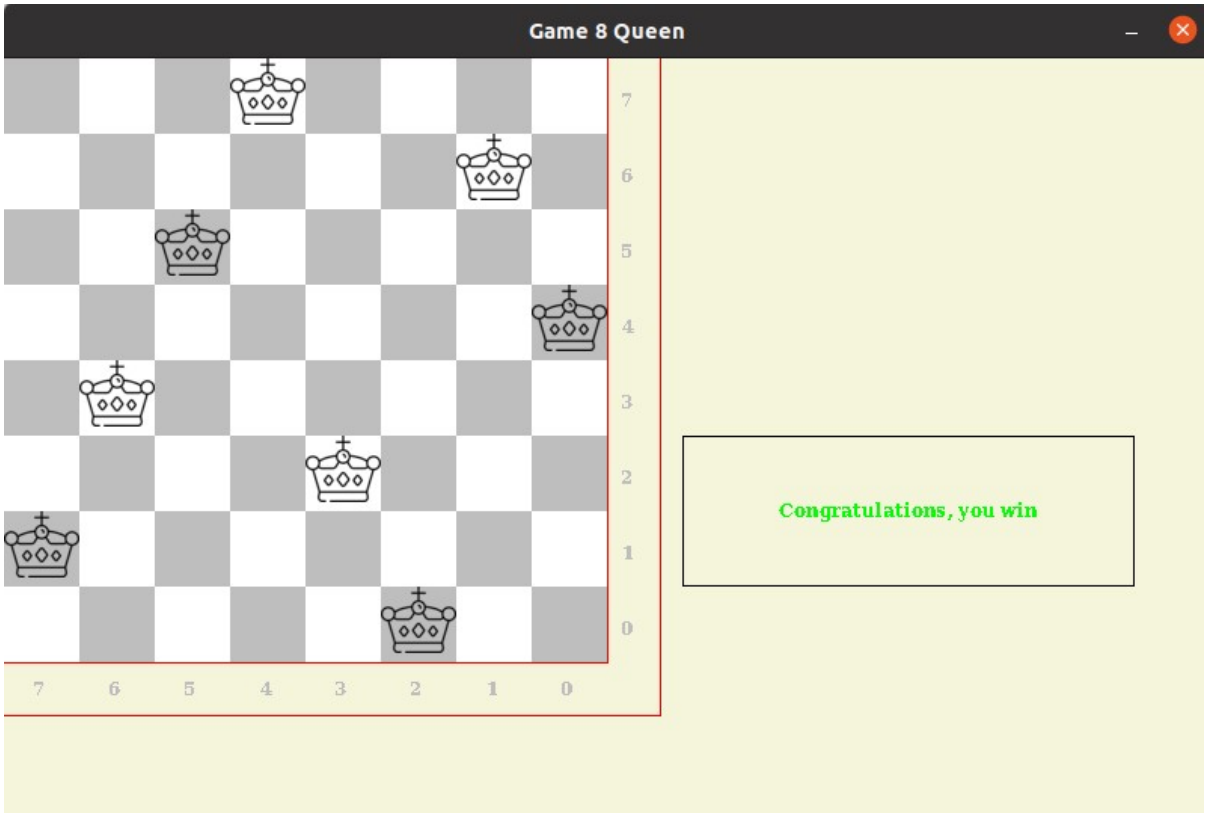
Programmation impérative avec le langage C - niveau 2

Pratiquer la programmation modulaire en C

AAV(conception) : Découper en modules et donner une liste de prototypes de fonctions pour chacun de ces modules un programme C dont la liste des fonctionnalités attendues à été communiquer dans un document d'une ou deux pages écrit en français.

En informatique, la programmation modulaire consiste à décomposer une application complexe en modules. Plus précisément en groupes de fonctions, de méthodes, de structure, de traitement… Grâce à cette répartition, les modules sont indépendant des uns des autres. Cela signifie que si on a besoin de modifier une fonction dans un module, on n'aura pas besoin de faire évoluer tous le programme à cause de ce changement.

Afin de vous montrer que j'ai acquis cette compétence, j'ai décidé de vous montrer le TP13 soit le jeu des 8 dames.



Présentation

Le jeu des 8 dames consiste à placer 8 dames sur un échiquier sans qu'elle puisse s'attaquer entre-elles. Par exemple dans la capture d'écran ci-dessus, vous pouvez constater que les 8 dames ne se menacent pas.

Répartition du programme :

Dans ce TP, l'échiquier était représenté par un nombre de 64 bits, tel que chaque bit correspondent à une case. Si c'est case vaut 0, alors cette case est libre sinon elle est menacé. Enfin j'ai fait une interface graphique pour représenter l'échiquier.

Afin de construire ce jeu, je l'ai réparti en 3 fichiers distincts qui sont les suivants **operation.c**, **interface.c** et **dame.c.**

Explication des fichiers

Le fichier **operation.c** s'occupe de la gestion des cases de l'échiquier. Cela signifie qu'il est en charge de savoir si une case est menacé par une dame ou non.

Le fichier **interface.c** s'occupe de la mise graphique de l'échiquier. Afin que ce fichier puisse connaître la position des dames, il va demander au fichier **dame.c** si la case sélectionnée est une dame.

Le fichier **dame.c** fait le lien entre le partie graphique et l'échiquier. Toutes les opérations de modification sur l'échiquier se fera à partir du fichier **interface.c** qui demandera au fichier de **dame.c** d'effectuer une action qui peut être sur le fichier **operation.c.**

Justification de la modularité

Tous ces fichiers ont des prototypes de fonctions, cela signifie que si l'on change le code d'une de ces fonctions, on n'aura pas besoin de changer l'autre partie du code. Par exemple dans le fichier **dame.c**, pour connaître la position des dames posées par le joueur, j'ai choisi d'utiliser une un tableau de 8 cases qui est globale. On pourrait alors changer ce tableau en une autre structure comme une liste par exemple. Ainsi on aura besoin de changer que le code du fichier dame.c.

```c
/*
 * Le joueur peut poser 8 dames sur l'échiquier.
 * Chaque case du tableau représente une dame posé par le joueur.
 * Si la case vaut -1 alors le joueur n'a pas encore posé sa n ème dame.
 * */
int dames[8] = {-1, -1, -1, -1, -1, -1, -1, -1};
```

AAV(production-projet) : Produire des fichiers entêtes correctement standardisés et en phase avec l'état de l'art (sécurisation inclusion - dépendances - bibliothèques non confinables - déclaration des types - prototypes des fonctions - fin de sécurisation)

Un fichier entête correctement un standardisés est un fichier ayant les lignes suivantes. Ainsi si on appelle plusieurs fois la même dépendances comme **stdio.h**, par exemple, il n'y aura pas de problème de redéfinition méthode. De plus il ne pourra pas avoir des structures ou comportant le même nom. Enfin les méthodes seront bien unique.

```c
#ifndef NOM
#define NOM
code...
#endif
```

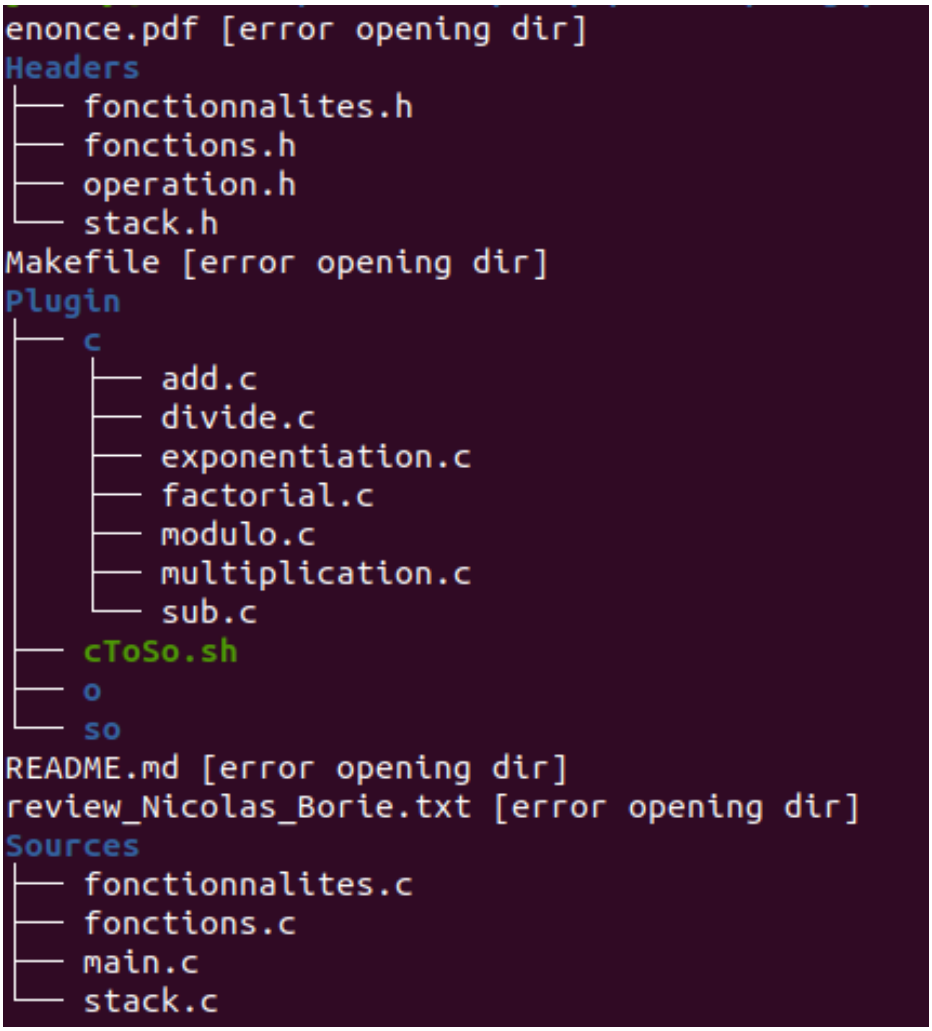
AAV(production-projet) : Mettre en place des Makefile tirant parti de la compilation séparée et gérant correctement les dépendances entre vos différents modules.

Un Makefile est un fichier permettant de compiler un projet C entre tous ses fichiers. De plus ce dernier doit être facilement modifiable comme par exemple en ajoutant des flags et facilement lisible par une autre personne.

Afin de justifier que je sais produire des Makefile tirant partie de la compilation séparée, je vais vous montrer le Makefile du TP15.

Le TP15 consiste à produire une calculatrice (Polonais inversées). Cette calculatrice doit avoir le même comportement que dc (Programme linux) excepté que dans ce tp, on ne prend que les entiers en compte. De plus les opérations de la calculatrice sont situés dans des fichiers .so.

Arborescence du projet



On observe que dans ce projet, les différents types de fichiers sont placés dans des répertoires différents.

```
CC=gcc
CFLAGS= -lreadline -lm -ldl -Wall -ansi
OBJ=main.o stack.o fonctions.o fonctionnalites.o
sources=Sources/
headers=Headers/
library=Plugin/

all: calc plugin

calc : $(OBJ)
    $(CC) -o calc $(OBJ) $(CFLAGS)

stack.o: $(headers)stack.h
    $(CC) -c $(sources)stack.c -o stack.o $(CFLAGS)

fonctions.o: $(headers)fonctions.h
    $(CC) -c $(sources)fonctions.c -o fonctions.o $(CFLAGS)

fonctionnalites.o: $(headers)fonctionnalites.h
    $(CC) -c $(sources)fonctionnalites.c -o fonctionnalites.o $(CFLAGS)

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

plugin:
    ./Plugin/cToSo.sh $(library)

clean:
    rm -f $(library)o/* $(library)so/*
    rm -f *.o
    rm -f calc
```

Dans ce Makefile, on observe qu'il appelle le script **cToSo.sh** pour générer les bibliothèques des opérations comme l'addition, soustraction… Ensuite il compile le reste du programme.

Pour conclure, ce Makefile est clair et lisible. De plus il compile le programme alors que celui-ci est dans des projets séparés.

Concevoir et écrire des programmes C de taille moyenne.

AAV(conception-com/doc) : Justifier à l'écrit les choix de structures supplémentaires établit vis à vis des fonctionnalités et algorithmes d'un projet informatique.

Avant de développer un programme, les programmeurs doit réfléchir à une manière efficace d'implémenter le programme. Ainsi le choix d'une structure dans un programme est très important.

Le TP 14 consiste à produire un exécutable (**larger_file**) récupérant des informations sur les fichiers des systèmes Unix. Plus précisément, ce programme consiste à afficher les 10 fichiers les plus gros d'un répertoires.

Par exemple, les 10 plus gros fichiers du répertoire ***usr*/lib/** :

```
jeremy@valade:~/Documents/Esipe/IINFO1/ProgC/TP14$ ./larger_file /usr/lib
142112588 /usr/lib/jvm/java-11-openjdk-amd64/lib/modules
134874968 /usr/lib/thunderbird/libxul.so
64302006 /usr/lib/thunderbird/omni.ja
37567616 /usr/lib/virtualbox/UICommon.so
29352240 /usr/lib/virtualbox/vboxwebsrv
27778576 /usr/lib/snapd/snapd
24664384 /usr/lib/virtualbox/ExtensionPacks/Oracle_VM_VirtualBox_Extension_Pack/solaris.amd64/VBoxPuelMain.so
22850224 /usr/lib/virtualbox/webtest
20434408 /usr/lib/firmware/liquidio/lio_23xx_vsw.bin
20197488 /usr/lib/grass78/lib/libgdal.so
```

Pour stocker les informations des fichiers, j'ai dû créé une structure contenant le chemin avec le nom du fichier et sa taille en octets nommé position. Ensuite j'ai dû créé une structure contenant un tableau de fichier (position).

```c
typedef struct position {
    char path[8192];
    unsigned long int size;
} Position;

typedef struct tableau {
    Position* index[MAX_SIZE];
    int size;
} Tableau;
```

Étant donnée que la consigne du TP était de récupérer les 10 fichiers les plus volumineux, j'ai créé un tableau car ce type de variable est un ensemble fini contrairement à une liste. Ainsi pour trier de manière décroissante les fichiers en fonction de leurs tailles. J'ai besoin d'utilisé une fonction qui insère les éléments de manières décroissante dans le tableau et je supprime le 11^{e} élément ci-besoin.

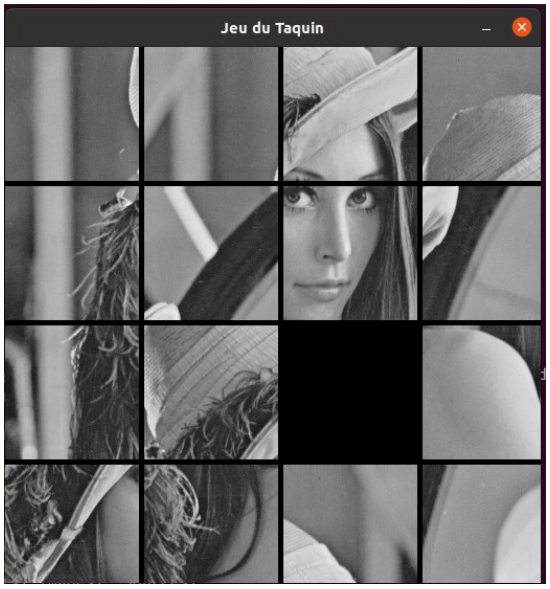
Ainsi en changeant le code source, on peut modifier aisément les informations sur les fichiers par exemple en rajoutant d'autres informations comme le propriétaire, la date de création … De plus en modifiant la variable MAX_SIZE, je peux afficher soit moins soit plus de fichier sur la console..

Donc pour ce programme, cette structure est la plus approprié me semble-t'il.

AAV(production) : Produire des programmes définissant leur propres types et exploitant ces nouveaux types de manière naturelle et judicieuse (type mini jeu avec plateau).

Dans des programmes, on a besoin d'utiliser des structures afin de représenter les différents objets du programme.

Dans le TP 11, j'ai utilisé des structures pour créer le jeu du Taquin. Ce dernier consiste à diviser une image en n images (4*4), de mélanger ces dernières entre-elles. Le but du joueur est de reconstituer l'image finale comme le montre la capture d'écran ci-dessous.



Afin de facilement manipuler le jeu, j'ai créé les structures suivantes :

```c
typedef struct carre {
    int ligne;
    int colonne;
    MLV_Image* image;
} Carre;

/* Structure contenant toutes les parties de l'image*/
typedef struct plateau {
    int ligneCaseNoire;
    int colonneCaseNoire;
    Carre bloc[NB_LIGNE][NB_COLONNE];
} Plateau;

/* Caractéristique du jeu */
typedef struct Taquin {
    int width;
    int height;
    int nbCase;
    int epaisseurDuTrait;
    Plateau* p;
} Taquin;
```

La structure du **Taquin** correspond au propriétés de l'interface graphique du jeu. Cette dernière possède un plateau. Le plateau contient un tableau de 15 images provenant de l'image originelle.

Grâce au type carré qui contient l'image, il est très facile d'échanger deux images entre-elles. Au lieu de prendre pixel par pixel par exemple.

Enfin en modifiant les propriétés du taquin, je peux diviser l'image par 3 au lieu de 4. Ce qui est très pratique.

Pour conclure, ces structures me permettrait d'avoir un fichier de configuration pour lancer le jeu. Par exemple : est-ce que le joueur veut jouer en 3*3 au lieu de 4*4 ou la taille de la fenêtre...

AAV(production) : Produire des programmes manipulant les données et les variables de manière efficace. Éviter les copies inutiles et favoriser l'utilisation des pointeurs sans tomber dans un excès inverse donnant des codes sources inutilement compliqués.

Dans l'exercice 2 du TP10, on récupère un fichier contentant une liste de personne. On souhaite trier les personnes en fonction de leur âge ou par ordre alphabétique. Pour effectuer ce tri, j'utilise une structure de liste chaînée. A chaque insertion d'une personne dans la liste, on ajoute directement la personne dans la liste selon la fonction qui est passée en paramètre. Par exemple voici le prototype de la fonction d'insertion :

```c
void ordered_insertion(List* liste, Cell* new, int order_func(Cell*, Cell*));
```

L'argument **int order_func(Cell*, Cell *)**, on peut passer en paramètre les fonctions suivantes :

```c
while (fscanf(fichier, "%s %s %d", nom, prenom, &age) == 3) {
    Cell* c = allocate_cell(nom, prenom, age);
    if (choix == 1)
        ordered_recursive(&l,  c, age_order);
    else
        ordered_recursive(&l,  c, name_order);
}
```

AAV(production) : Produire des programmes C manipulant des fichiers en lecture et/ou en écriture.

Aujourd'hui la plupart des applications utilisent des fichiers pour lire ou traiter des données. Ainsi il est important de pouvoir lire et écrire dans un fichier facilement.

Je vais me servir de l'exercice 2 du TP 10 qui consiste à lire un fichier de personnes (nom, prénom et âge). Le programme doit trier de manière alphabétique ou par âge les personnes provenant du fichier. Enfin le programme affichera la liste triée des personnes.

Par exemple on souhaite trier le fichier suivant par ordre alphabétique :

```
Emile Mercier 37
Chloe Lavigne 90
Appoline Delattre 35
Gilles Barre 39
Louis Chauvin 35
Eugene Boyer 7
Beatrice Ancona 3
Sophie Dumas 17
Elga Faure 15
Robert Dumas 40
Martine Cruz 25
Elga Fourrier 33
```

Affichage de la liste après le trie des personnes par ordre alphabétique, j'ai choisi le choix 2.

```
jeremy@valade:~/Documents/Esipe/IINFO1/ProgC/TP10/Exercice2$ ./exe |  more
1 - Trier la liste des pesonnes par âge
2 - Trier la liste des pesonnes par ordre alphabétique
Votre choix
Liste :
Abdel Ancona 52
Abdel Angelie 84
Abdel Blondel 14
Abdel Bouvier 79
Abdel Boyer 87
Abdel Brun 49
Abdel Chevalier 15
```

Pour conclure, on observe que la liste des personnes est bien triées par ordre alphabétiques.

AAV(production-projet-com/doc) : Qualifier toutes informations relatives à un programme C méritant d'être consignées et savoir où les placer dans un projet (commentaires et documentation pour développeur ou utilisateur, informaticien ou pas…).

Lorsqu'un programmeur développe une application, il y a plusieurs types d'utilisation. La première utilisation est l'utilisation cliente, c'est à dire que la personne ne connaît rien en informatique. Cette documentation a pour objectif d'apprendre à comment utiliser l'application. Ainsi le fichier **README.md** permet d'indiquer au client de comment exécuter le programme.

La seconde documentation est lorsque l'on créait une bibliothèque. Il y aura une documentation pour savoir ce que font les méthodes et comment les utiliser par un autre développeur. Par exemple avec la bibliothèque de la libMLV.

Enfin la dernière documentation est la documentation du code source. Ce dernier doit permettre à un autre développeur de comprendre le code source afin qu'il puisse le modifier.

AAV(production-projet) : Produire un programme C utilisant un interface graphique conçu par vos soins basé sur une librairie non standard (sdl ou libMLV)

La bibliothèque MLV est une bibliothèque Multimédia simplifiée qui permet au développeur de produire des programmes avec des interfaces graphiques.

Pendant cette année, j'ai produit trois interfaces graphiques qui sont le sudoku (TP7), le jeu du Taquin (TP 11) et le jeux des 8 dames (TP 13). Dans ces TP, j'ai pu me familiarisé avec les méthodes de la libMLV. J'ai appris à afficher des images, à lancer des fichiers son…

Pour conclure, j'ai appris à a lier le programme et son interface graphique de manière efficace.

Maîtriser l'approche binaire de la manipulation des données dans le langage C

AAV(conception) : Comprendre le fonctionnement des contenus binaires des types basiques du langage C. Déterminer le faisable et l'infaisable en binaire avec les types de bases du C.

Avec les opérateurs &, |, ^, <<, >> , on peut tous faire sur les types de bases.

AAV(production) : Produire un programme C utilisant correctement des opérateurs bits à bits pour implanter une fonctionnalité standard (swap, miroir, tri, copie, déplacement).

Étant donnée que le langage C est un langage de bas niveau, on peut manipuler les bits des variables afin de les modifier. Grâce à cela, le programme est plus rapide (moins d'opérations complexes).

Pour justifier que j'ai acquis ces compétences, j'ai fait les exercices sur les opérateurs bit à bit du logiciel Platon. Dans ces exercices j'ai appris à manipuler les bit pour trouver par exemple le plus grand nombre de bits consécutif commun, aussi le plus grand nombre d'alternance, à échanger les bits d'un nombre, …

Plus particulièrement, j'ai écrit un programme qui permet de donner le PGCD entre deux nombres sans utilisés les divisions, les modulos ou les multiplication (Répertoire bit).

Voici le programme suivant :

```c
unsigned int pgcd(unsigned int a, unsigned int b) {
    if(a == 1 || b == 1)
        return 1;
    if (a == b)
        return a;
    int resteA = (a & (1<<0))  ? 1 :0;
        int resteB = (b & (1<<0))  ? 1 :0;

    if (resteA == 0 && resteB == 0) {
        a = a>>1;
        b = b>>1;
        return pgcd(a, b)<<1;
    }
    if (resteA == 0) {
        a = a>>1;
        return pgcd(a, b);
    }
    if (resteB == 0) {
        b = b>>1;
        return pgcd(a, b);
    }

    if (a > b)
        return pgcd(a-b ,b);
    else
        return pgcd(a, b-a);
}
```

Dans ce programme, j'ai appris à déterminer si un nombre est pair ou impair sans utiliser de division. Pour cela, il faut regarder le bit de poids faible, si ce dernier est égale à 0 alors le nombre est pair sinon il est impair. De plus j'ai appris à décaler les bits vers la gauche ou la droite en fonction de mes besoins. Par exemple pour effectuer une division par deux d'un nombre il faut décaler les bits de poids forts à droite et inversement pour le double.

Recourir correctement à l'allocation dynamique de mémoire lorsque c'est nécessaire ou demandé

AAV(conception-production) : Vos programmes C utilisent des structures de données adaptées à la résolution complète des problèmes (sans limite de taille par exemple).

En informatique, on va souvent créer une structure afin de répondre au besoin de l'application. Ainsi il est important d'avoir une structure adaptée pour résoudre le problème auquel vous être confronté.

Par exemple dans le TP12, le programme doit compter le nombre de mot différents dans un texte. En faisant l'exercice 1 du TP12, la solution naïve, le programme met pratiquement 1 minute sur ma machine pour compter un texte de taille 1006K octets (Germinal.txt). On observe que la structure n'est pas optimal.

Au lieu d'utiliser la méthode naïve, on peut utiliser une table de hachage. Cette dernière est implanté dans l'exercice 2 du TP. Ainsi on observe que pour calculer le nombre de mots différents dans Germinal, le programme met à peine 1 secondes.

Enfin dans la table de hachage, on peut mettre un nombre infini de valeur. Attention le nombre de case de la table de hachage doit être cohérent avec le nombre de mot différent sinon le programme sera moins rapide.

AAV(production) : Produire des programmes manipulant une quantité de mémoire qui s'adapte durant ses exécutions. Utiliser correctement cette mémoire dans ces programmes (allocation/libération).

En fonction des machines utilisées, c'est important de bien manipuler la mémoire utilisé par le programme. Par exemple un programme peut marcher sur une machine mais pas sur une autre à cause de la différence de mémoire. Ainsi il est important de savoir bien allouer et la libérer la mémoire.

Dans le TP 15 qui est une calculatrice en polonais inversée, j'utilise une pile pour stocker les nombres saisis par l'utilisateur et effectuer les opérations sur ces derniers.

Voici la structure de la pile :

```c
typedef struct dynamic_stack {
    int* values;    /* Tableau d'entiers de la pile */
    int capacity;   /* Taille maximale du tableau */
    int current;    /* Taille courante du tableau */
} Stack;
```

Dans un premier, j'alloue le tableau de la pile avec une constante MAX_SIZE. Si la taille du tableau arrive à la capacité maximale, on réalloue la tableau avec la fonction ci-dessous :

```c
/* Fonction qui réalloue le tableau de la pile si il est plein */
void realloc_values(Stack* pile) {
    pile->capacity = pile->capacity*2;
    pile->values = (int*) realloc(pile->values, pile->capacity * sizeof(int));
    if (pile->values == NULL) {
        fprintf(stderr, "Not enough meomory!\n");
        exit(1);
    }
}
```

Enfin à la fin de l'exécution du programme, je libère la mémoire de la pile.