**Langage de programmation avancé**

**HAUTE ÉCOLE DE NAMUR-LIÈGE-LUXEMBOURG**

**DA-IA – bloc 1**

Atelier 2 – Fichiers[[1]](#footnote-1)

Objectifs

* Continuer à utiliser les éléments déjà vus
* Expliquer ce que sont fichier, enregistrement et flux
* Utiliser les fonctions liées aux flux

[Introduction 2](#_Toc64382070)

[A. Entrées/sorties, fichier et autres 3](#_Toc64382071)

[Entrées/sorties 4](#_Toc64382072)

[Fichier 4](#_Toc64382073)

[Enregistrement 5](#_Toc64382074)

[Flux 6](#_Toc64382075)

[B. Opérations sur les fichiers - mode binaire 3](#_Toc64382076)

[Ouverture 8](#_Toc64382077)

[Fermeture 10](#_Toc64382078)

[Écriture d’un enregistrement 11](#_Toc64382079)

[Lecture d’un enregistrement 12](#_Toc64382080)

[C. Code de *cartes.c* 14](#_Toc64382081)

# Introduction

Jusqu’à présent, vous avez écrit des programmes qui permettaient de traiter des informations ne pouvant provenir que de deux sources : soit elles étaient inclues dans l’algorithme lui-même, par vos soins, soit elles étaient obtenues en cours d’exécution par l’utilisateur. Mais évidemment, cela ne suffit pas aux besoins réels.

Les fichiers permettent de sauvegarder ces infos entre chaque utilisation du programme. Ils servent à stocker des informations de manière permanente.

Dans cet atelier, vous allez apprendre à mémoriser de façon permanente, dans un fichier, les informations obtenues et manipulées dans le programme.

La théorie sur les fichiers est assez étendue et parfois complexe. Vous allez donc avoir une présentation un peu plus théorique des éléments qui vous seront utiles et directement liés au langage C.

Ensuite vous verrez en quoi les structures sont intéressantes dans la gestion des fichiers.

Enfin, vous pourrez tester quelques opérations, qui seront des fonctions en C, disponibles sur les fichiers.

Dans ce document, plusieurs conventions sont utilisées :

* les mots gras désignent des termes de vocabulaire liés à l’**informatique en général**.
* les mots soulignés et gras désignent des termes de vocabulaire directement liés aux cours de **programmation**.
* le logo signifie que vous avez quelque chose à réaliser.
* le logo est associé aux cadres présentant certaines conventions.
* le logo est associé aux cadres présentant les éléments liés à la propreté/lisibilité du code (*clean code*).

Pour réaliser cet atelier, commencez par créer un projet (et une solution) sous VS2019. Donnez-lui un nom vous permettant de vous y retrouver par la suite, par exemple, Module2Atelier. N’oubliez pas de faire en sorte que le fichier principale, celui qui contient la fonction main, ait l’extension .c.

# Pourquoi des fichiers ?

Dans le cadre d’un programme, comme celui sur les cartes de l’atelier précédent, lorsque l’utilisateur a terminé d’encoder ses cartes, elle sont mémorisées dans le tableau de structures cartes. Mais dès qu’il quitte le programme, ces informations sont perdues puisque la mémoire, qui a été allouée pour que le programme puisse s’exécuter, est libérée.

Un peu de technique…

En effet, comme vous l’avez déjà vu, lorsqu’un programme est exécuté, une partie de la **mémoire vive** lui est associée.

Cette partie de mémoire est géré comme une pile d’assiette : dernier entré, premier sorti. Elle est donc appelée la **pile**, ou **pile d’appels**, ou encore **pile d’exécution**…

Comme vous le savez déjà, la fonction principale d'une pile d'appels est de stocker les adresses de retour des sous-programmes, dans le cadre du C, des fonctions. Lorsqu'une fonction est appelée, c’est l'adresse de l'instruction qui suit l’appel qui est mémorisée sur le sommet de la pile.

En plus de cette adresse, et selon le système d’exploitation et l’architecture de la machine, d’autres informations sont mémorisées dans cette pile. Ainsi, chaque appel de fonction provoque la création d’un **cadre de pile** (en anglais, « stack frame »). Dans ce cadre sont mémorisés les arguments de la fonction ainsi que les variables locales à celle-ci.

Lorsque la fonction est terminée, le cadre est supprimé de la mémoire ainsi que les données qui s’y trouvaient. C’est pour cette raison qu’il faut soit renvoyer le résultat, soit passer les informations par adresse pour les modifier directement…

Chaque programme se voit allouer une pile d’appels lors de son exécution (un programme exécuté est appelé **processus**). Mais dès que celle-ci est terminée, la pile d’appels qui lui est associée est également supprimée de la mémoire. Ainsi, d’une exécution à l’autre, l’utilisateur perd les données qui ont été traitées durant l’exécution du programme.

Pourtant, l’utilisateur veut pouvoir retrouver les cartes qu’il a déjà encodées, avec les modifications qu’il y a apportées la dernière fois qu’il a exécuté le programme. C’est pour cette raison que les informations sont mémorisées sous la forme de fichier, dans une **mémoire morte**.

# Entrées/sorties, fichier et autres

Dans cette partie, il est question de décrire plus en détail la notion d’entrées/sorties, celle de fichier et celle d’enregistrement… Ces éléments sont abordés de manière assez superficielle et uniquement au regard de ce qui est nécessaire à leur utilisation en langage C. Cette partie est assez théorique et ne permet pas de mise en pratique directe au moyen d’exercices.

## Entrées/sorties

Les **E/S** (entrées/sorties) sont des opérations permettant d'accéder aux périphériques (disques, imprimantes, écrans, clavier, réseaux...) en lecture ou en écriture.

Pour rappel, les variables sont stockées en **mémoire vive** (RAM, mémoire principale qui n’est utilisable que lorsque l’ordinateur est sous tension), alors que les fichiers sont stockés en **mémoire morte** (HDD, SSD, clé USB, cloud… et autres périphériques externes, mémoires secondaires qui gardent l’information en mémoire même s’ils ne sont plus sous tension).

L'avantage de la mémoire vive est sa rapidité d’accès, mais ses inconvénients sont que son contenu est effacé lorsqu’elle n’est plus sous tension et qu’elle coute plus cher.

Les avantages de la mémoire secondaire sont que les données restent en mémoire même hors tension et que son coût est plus faible que celui de la mémoire vive, mais son inconvénient est qu’elle est plus éloignée du processeur et que le temps d’accès est donc plus long.

Les entrées/sorties peuvent être réalisées en C de deux manières :

* soit de manière élémentaire, en accédant directement au système d'exploitation avec des instructions de « bas niveau », appelé des appels systèmes comme *open*, *read*, *write* et *close*.
* soit au travers d'un flux, en mémoire vive, en utilisant des instructions de « haut niveau ». Les fonctions des bibliothèques standards *fopen*, *fread*, *fwrite* et *fclose*, ainsi que d’autres, permettent de lire/écrire d'un coup un plus grand nombre d'octets en passant par ce flux, comme vous allez le faire par la suite.

## Fichier

Un fichier est une collection de données enregistrées de façon permanente et de façon à être manipulées par un programme.

En toute généralité, il peut contenir différents types d’information tels qu’un programme exécutable, des données, des images… Dans le cadre de ce cours, seuls les fichiers de données sont manipulés.

Les fichiers peuvent être considérés selon deux points de vue :

* **logique** : le fichier est envisagé selon le point de vue du programmeur, au niveau logiciel. Il s’agit de la façon dont les données seront structurées dans le fichier. La notion d’enregistrement décrite ci-dessous y est liée.
* **physique** *:* le fichier est envisagé selon le point de vue du système d'exploitation, au niveau matériel. Il s’agit de la façon dont le système d’exploitation organise les données selon le support sur lequel elles se trouvent.

C’est le système d’exploitation qui permet le lien entre ces deux points de vue au travers d’un de ses composants appelé le « système de gestion de fichiers ».

Seul le point de vue logique est abordé dans ce cours puisque vous passerez toujours par un tampon et que la gestion de la mémoire physique est laissée aux soins du système d’exploitation.

## Enregistrement

Un enregistrement est la représentation logique d’un ensemble de données qui concernent un même thème, une même entité. La plupart des fichiers contenant des données sont composés d’enregistrements.

En repartant de l’exercice sur les cartes, voici quelques exemples d’enregistrements…

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| "ville" | 25 | 25 | true | "maison" | "chateau" | "riviere" |
| "ile" | 15 | 23 | false | "ocean" | "palmier" | "crabes" |
| "caverne" | 32 | 48 | true | "rats" | "sombre" | "humide" |

Vous pouvez facilement faire le lien entre enregistrement et type structuré ou structure. En effet, en C, il n’existe pas de type prédéfini pour un enregistrement, c’est donc au moyen de structures que vous allez mémoriser les données dans un fichier.

Pour votre culture générale, lorsqu’il s’agit d’**enregistrement**, on parle de **champs** plutôt que de membres.

Il existe plusieurs façons de stocker les informations dans le fichier : en utilisant des enregistrements de longueur fixe ou de longueur variable... Dans la suite du cours, seuls des enregistrements de taille fixe seront abordés.

Dans le cas des cartes, la structure qui représente l’enregistrement est la suivante.

#define T\_NOM 25

#define NB\_MOTS\_CLES 3

#define T\_MOT\_CLE 10

typedef struct carte Carte;

struct carte {

char nom[T\_NOM];

int nbCasesHauteur;

int nbCasesLargeur;

bool officielle;

char motsCles[NB\_MOTS\_CLES][T\_MOT\_CLE];

};

Tous les enregistrements occupent donc autant d’octets que nécessaire pour mémoriser la structure en question, ici :

25 + 3 (alignement) + 4 + 4 + 1 + 30 + 1 (alignement) = 68 octets

Quelles que soient les valeurs mémorisées, la taille nécessaire pour un enregistrement est toujours la même, d’où le fait qu’on parle d’enregistrement de « taille fixe ».

La notion d’alignement est abordée en théorie.

## Flux[[2]](#footnote-2)

Une image contenant capture d’écran, texte, diagramme, ligne

Description générée automatiquementComme vous l’avez déjà lu, les fonctions des bibliothèques standards, dont stdio.h, n'accèdent pas directement aux fichiers, mais à des flux de données (*stream* en anglais), sorte de tampons situés en mémoire vive.

Pourquoi passer par un flux de données plutôt que de manipuler directement les fichiers ?

Tout d’abord, les fonctions élémentaires s’avèrent utiles dans certains cas, mais elles ne sont pas très efficaces car les appels systèmes sont des fonctions particulières qui ont un protocole d'appel plus complexe que de simples fonctions.

Ensuite, il existe des disparités entre systèmes d’exploitation quant à la représentation des passages à la ligne. En effet, les différents systèmes d’exploitation n’utilisent pas les mêmes caractères de fin de ligne. Les fonctions des bibliothèques standards remplacent d’elles-mêmes ces caractères de fin de ligne par un \n (et inversement). Ainsi le code est plus portable et pérenne !

« De plus, lire depuis un fichier ou écrire dans un fichier signifie le plus souvent accéder au disque dur. Or, rappelez-vous, il s’agit de la mémoire la plus lente d’un ordinateur. Dès lors, si pour lire une ligne il était nécessaire de récupérer les caractères un à un depuis le disque dur, cela prendrait un temps fou.

Pour éviter ce problème, les flux de la bibliothèque standard recourent à un mécanisme appelé la **temporisation** ou **mémorisation** (*buffering* en anglais). La bibliothèque standard fournit deux types de temporisation : la temporisation par **blocs** et la temporisation par **lignes**. »

Enfin, il existe deux catégories de flux :

* les flux de type **texte** : le contenu est lisible et compréhensible dans un éditeur de texte, comme pour les fichiers Txt, Xml ou Json, et
* les flux de type **binaire** : le contenu est peu compréhensible… mais tous les fichiers ne sont pas créés pour être lisibles sans passer par un programme.

C’est lors de l’ouverture du flux que ce choix est fait et, selon ce choix, certaines fonctions seront utilisables et d’autres pas… Seuls les flux de type binaire sont abordés dans cet atelier.

Par défaut, 3 flux standards (déclarés dans stdio.h) sont ouverts lors du démarrage d’un programme :

* stdin : entrée standard, c’est le flux depuis lequel on peut récupérer les informations fournies par l’utilisateur (par défaut le clavier).
* stdout : sortie standard, c’est le flux grâce auquel on peut transmettre des informations à l’utilisateur (par défaut l’écran).
* stderr : sortie d’erreur standard, c’est le flux à privilégier lorsqu'on transmet des messages d’erreurs ou des avertissements à l’attention de l’utilisateur (par défaut l’écran).

# Opérations sur les fichiers - mode binaire

Maintenant que vous savez ce qu’est un fichier, un enregistrement et un flux, il est temps de rentrer dans le vif du sujet. Dans cet atelier, vous allez apprendre à utiliser les fonctions d’entrées/sorties en utilisant le mode « binaire ». C’est en effet souvent de cette façon que des enregistrements structurés sont mémorisés dans un fichier.

## Ouverture

Il s’agit d’**ouvrir un flux** entre le fichier physique, celui qui sera réellement présent sur le support physique, avec un chemin d’accès et un nom, et le fichier logique, l’élément du programme qui fait référence au fichier physique.

La fonction utilisée a le prototype suivant :

errno\_t fopen\_s(FILE \*\* pFile, char \* filename, char \* mode);

Cette fonction ouvre le fichier filename dans le mode d’ouverture mode et mémorise l’adresse du flux, s’il est ouvert, dans pFile.

Il faut remplacer

* pFile par l’adresse d’une variable de type FILE \*.
* filename par le nom, et éventuellement le chemin d’accès, du fichier auquel vous voulez avoir accès. Il s’agit d’une chaîne de caractères.
* mode par le mode d’ouverture. Il s’agit d'une des chaînes de caractères suivantes :
  + "r"  Ouvre le fichier en lecture. Le pointeur est placé au **début** du fichier.
  + "r+" Ouvre le fichier en lecture et écriture. Le pointeur est placé au **début** du fichier.
  + "w" Ouvre le fichier en écriture. Le pointeur est placé au **début** du fichier.  
     Le fichier est créé s'il n'existait pas. S'il existait déjà, il est tronqué.
  + "w+" Ouvre le fichier en lecture et écriture. Le pointeur est placé au **début** du fichier.  
     Le fichier est créé s'il n'existait pas. S'il existait déjà, il est tronqué.
  + "a" Ouvre le fichier en écriture. Le pointeur est placé à la **fin** du fichier.  
     Le fichier est créé s'il n'existait pas.
  + "a+" Ouvre le fichier en lecture et écriture. Le pointeur est placé à la **fin** du fichier.  
     Le fichier est créé s'il n'existait pas.

Pour passer en mode binaire, il faut ajouter la lettre 'b' à chacune de ces chaînes pour donner par exemple "rb", "rb+"…

FILE

La définition de la structure FILE dépend du système d'exploitation. Cependant, on sait qu’elle doit contenir entre autres l’adresse du début du tampon, ainsi que l’adresse de la position à laquelle vous êtes dans le fichier, appelée « curseur ».

Il est dangereux de travailler directement sur les champs de cette structure puisqu’elle dépend de l’OS, d’où le fait d’en récupérer l’adresse et de n’utiliser que des fonctions déjà définies dans les bibliothèques standard pour y accéder.

En cas d'erreur, l’adresse NULL est mémorisée dans pFile.

Un code d’erreur de type errno\_t est renvoyé par la fonction afin de donner d’éventuelles informations sur la raison de l’échec de la fonction.

Code d’erreur

errno\_t est le type prévu par la norme ISO/IEC[[3]](#footnote-3) pour renvoyer un code d’erreur prédéfini dans la bibliothèque standard errno.h. Il s’agit en fait d’une simple redéfinition du type int fait dans la bibliothèque errno.h.

Cette bibliothèque standard est incluse parce que le fait d’inclure stdio.h amène à l’inclusion de diverses bibliothèques dont errno.h.

Reprenez l’exercice sur les cartes (faites-en une copie afin de garder la version sans fichier, au cas où vous ne seriez pas certain de vous, le code attendu à la fin de l’atelier sur les structures se trouve à la fin de ce document).

Vous allez ajouter deux fonctions, l’une pour sauver le contenu du tableau dans un fichier et l’autre pour le lire et garnir le tableau avec les informations récupérées.

Quels vont être les prototypes de chacune de ces fonctions ?

La fonction de sauvegarde n’a besoin que des données à sauver. Écrivez ce prototype. Pensez au fait que le tableau n’est pas nécessairement rempli jusqu’à la fin…

La fonction de lecture est supposée renvoyer le tableau garni avec les informations lues dans le fichier. Écrivez ce prototype.

Ces deux prototypes devraient correspondre à ceci :

void sauverCartes(Carte cartes[], int nbCartes);

int lireCartes(Carte cartes[]);

Dans chacune de ces fonctions, il va falloir associer un flux au fichier physique.

Commencez par la **fonction de sauvegarde**, car si aucune donnée n’a été mémorisée dans le fichier, la fonction de lecture est inutile.

Avant tout, il vous faut déclarer la variable qui permet de mémoriser l’adresse de la structure FILE. Écrivez cette déclaration. Ensuite vous pouvez utiliser cette variable pour ouvrir le flux en respectant la syntaxe proposée ci-dessus.

La déclaration est semblable à la suivante :

FILE \* fiCartes;

L’instruction d’ouverture doit ressembler à la suivante :

fopen\_s(&fiCartes, "cartes.dat", "wb");

L’extension « dat » pour « data » signifie que le fichier contient des données.

Après l'exécution de cette instruction, la question se pose de savoir si le flux est en effet accessible via l’adresse mémorisée dans fiCarte.   
Pour le vérifier, il est important, après chaque instruction d’ouverture de tester la valeur de cette variable. Si elle vaut NULL, c’est que l’ouverture n’a pas eu lieu correctement.

Les instructions pour faire ce test sont les suivantes :

if (fiCartes == NULL)

printf("Erreur lors de l'ouverture du fichier.\n");

Dans ce cas, comme il s’agit d’une ouverture en écriture, si le fichier n’existe pas, il est créé. Cela ne correspond donc pas réellement à une erreur.

Pour les plus téméraires, vous pouvez essayer de différencier ces deux cas au moyen de errno.h.

Procédez de la même façon pour la fonction de lecture, mais n’oubliez pas d’adapter le mode d’ouverture.

## Fermeture

Tout fichier qui a été ouvert doit être fermé. Dans le cas contraire, la prochaine ouverture risque de poser un problème !

Il est donc nécessaire d’utiliser la fonction suivante pour fermer le flux associé au fichier physique.

int fclose(FILE \* stream);

Il faut remplacer stream par l’adresse du flux.

Dans le cas de la gestion des cartes, l’instruction devient la suivante :

fclose(fiCartes);

La fonction fclose ferme le flux via son adresse. Si l’adresse est NULL, cela peut amener le programme à se terminer avec une erreur… Cette fonction ne doit donc être appelée que si l’ouverture du fichier s’est bien produite et que la valeur de stream est autre que NULL.

Ajoutez cette instruction au bon endroit dans les fonctions de sauvegarde et de lecture.

## Écriture d’un enregistrement

Pour écrire un enregistrement dans un fichier, vous allez utiliser la fonction suivante :

size\_t fwrite(const void \* buffer, size\_t size, size\_t count, FILE \* stream);

Il faut remplacer

* buffer, par l’adresse du premier enregistrement à écrire, quel que soit son type.
* size, par la taille d’un enregistrement, en octets.
* count, par le nombre maximum d’enregistrements à écrire.
* stream, par l’adresse de la structure de type FILE.

La fonction renvoie le nombre d’enregistrements réellement écrits dans le fichier, qui peut être inférieur à count si une erreur s’est produite.

Plus d’informations sont disponibles dans la documentation : <https://docs.microsoft.com/fr-fr/cpp/c-runtime-library/reference/fwrite?view=vs-2019>.

Vous pouvez maintenant compléter la fonction de sauvegarde des cartes en faisant en sorte de sauver chacune des cartes mémorisées dans le tableau de cartes.

Avant tout, vous devez parcourir le tableau et faire appel à la fonction fwrite dans la boucle jusqu’à ce que toutes les cartes soient sauvées. Dans ce cas, l’enregistrement à écrire est le contenu d’une cellule du tableau. Pour déterminer la taille de l’enregistrement, utilisez l’opérateur sizeof. Enfin, vous allez écrire un enregistrement à la fois.

La fonction devrait ressembler à ce qui suit :

#define FICHIER\_CARTE "cartes.dat"

void sauverCartes(Carte cartes[], int nbCartes) {

FILE \* fiCartes;

fopen\_s(&fiCartes, FICHIER\_CARTES, "wb");

if (fiCartes == NULL)

printf("Erreur lors de l'ouverture du fichier.\n");

else {

for (int iCarte = 0; iCarte < nbCartes; iCarte++) {

fwrite(&cartes[iCarte], sizeof(Carte), 1, fiCartes);

}

fclose(fiCartes);

}

}

Gestion des cas d’erreur

Vous remarquez que dans cette solution, on traite le cas d’erreur **au plus tôt**. C’est une pratique assez courante. Elle permet notamment de limiter le niveau d’imbrication des alternatives gérant ces cas d’erreur. De plus, il n’est pas besoin de lire tout le code de la fonction pour s’assurer que la gestion des cas d’erreur est correcte.

Il est donc fortement recommandé de procéder ainsi à partir d’ici…

## Lecture d’un enregistrement

Pour lire un enregistrement du fichier, vous allez utiliser la fonction suivante :

size\_t fread\_s (const void \* buffer, size\_t buffersize, size\_t elementsize, size\_t count, FILE \* stream);

Il faut remplacer

* buffer, par l’adresse à laquelle les enregistrements lus sont mémorisés.
* buffersize, la taille de l’espace mémoire disponible dans le buffer, en octets.
* elementsize, la taille d’un enregistrement, en octets.
* count, le nombre maximum d’enregistrements à lire.
* stream, l’adresse de la structure de type FILE.

La fonction renvoie le nombre d’enregistrements réellement lus dans le fichier, qui peut être inférieur à count si une erreur s’est produite.

Plus d’informations sont disponibles dans la documentation :

<https://docs.microsoft.com/fr-fr/cpp/c-runtime-library/reference/fread-s?view=vs-2019>.

Pour garnir le tableau de cartes sur base des données mémorisées dans le fichier, vous devez parcourir le fichier et faire appel à la fonction fread comme s’il s’agissait d’une obtention.

Vous devriez avoir besoin d’une fonction supplémentaire qui permet de savoir si lors de la dernière lecture, vous avez atteint la fin du fichier.

Il s’agit de la fonction suivante :

int feof(FILE \*stream);

Cette fonction renvoie une valeur différente de zéro si l’opération de lecture a tenté de lire au-delà de la fin du fichier ; sinon, elle renvoie 0. Bien qu’il ne s’agisse pas de valeur booléenne, vous êtes autorisés à ne pas écrire l’opération de comparaison à 0. Le code en devient en effet plus lisible.

Chaque enregistrement lu doit être mémorisé dans la cellule du tableau qui suit la précédente. Pour déterminer la taille de l’enregistrement, utilisez à nouveau l’opérateur sizeof. Enfin, vous allez lire un enregistrement à la fois.

Vous pouvez enfin compléter la fonction de lecture des cartes en faisant en sorte de mémoriser chacune des cartes lues dans le tableau de cartes. Réfléchissez bien à l’algorithmique !

La fonction devrait ressembler à ce qui suit :

int lireCartes(Carte cartes[]) {

FILE \* fiCartes;

Carte carte;

int iCarte = 0;

fopen\_s(&fiCartes, FICHIER\_CARTES, "rb");

if (fiCartes == NULL)

printf("Erreur lors de l'ouverture du fichier.\n");

else {

fread\_s(&carte, sizeof(Carte), sizeof(Carte), 1, fiCartes);

while (!feof(fiCartes)) {

cartes[iCarte] = carte;

iCarte++;

fread\_s(&carte, sizeof(Carte), sizeof(Carte), 1, fiCartes);

}

fclose(fiCartes);

}

return iCarte;

}

Vous devriez maintenant avoir un projet permettant l’obtention de cartes, leur mémorisation dans un fichier et leur chargement lorsque vous relancez le programme.

Si vous n’avez pas encore atteint ce résultat, réfléchissez au meilleur moment de faire appel à ces deux nouvelles fonctions.   
Placez ces appels judicieusement dans la fonction principale et le tour est joué !

# Code de *cartes.c*

#include <stdio.h>

#include <string.h>

#include <stdbool.h>

#define T\_NOM 100

#define NB\_MOTS\_CLES 3

#define T\_MOT\_CLE 25

#define NB\_CARTES 50

typedef struct carte Carte;

struct carte {

char nom[T\_NOM];

int nbCasesHauteur;

int nbCasesLargeur;

bool officielle;

char motsCles[NB\_MOTS\_CLES][T\_MOT\_CLE];

};

Carte carteObtenue(void);

void afficheCarte(Carte carte);

void main (void) {

Carte cartes[NB\_CARTES];

int nbCartes = 0;

Carte carte;

bool encorePlace = nbCartes < NB\_CARTES;

carte = carteObtenue();

while (strcmp(carte.nom, "") != 0 && encorePlace) {

cartes[nbCartes] = carte;

nbCartes++;

encorePlace = nbCartes < NB\_CARTES;

if(encorePlace) carte = carteObtenue();

} // ou do… while

if (nbCartes == 0) {

printf("Aucune carte mémorisée !");

} else {

for(int iCarte = 0; iCarte < nbCartes; iCarte++) {

afficheCarte(cartes[iCarte]);

}

}

}

Carte carteObtenue(void) {

Carte carte;

char reponse;

printf("Nom : ");

gets\_s(carte.nom, T\_NOM);

if (strcmp(carte.nom, "") != 0) {

printf("Hauteur (en cases) : ");

scanf\_s("%d", &carte.nbCasesHauteur);

printf("Largeur (en cases) : ");

scanf\_s("%d", &carte.nbCasesLargeur);

getchar();

printf("Officielle (O/N) : ");

scanf\_s("%c", &reponse, 1);

getchar();

carte.officielle = reponse == 'O';

for (int i = 0; i < NB\_MOTS\_CLES; i++) {

printf("Mot cle %d : ", i + 1);

scanf\_s("%s", carte.motsCles[i], T\_MOT\_CLE);

}

}

return carte;

}

void afficheCarte(Carte carte) {

printf("%s (%d x %d - %sofficielle)\n",

carte.nom,

carte.nbCasesHauteur,

carte.nbCasesLargeur,

carte.officielle ? "" : "pas ");

for (int i = 0; i < NB\_MOTS\_CLES; i++) {

puts(carte.motsCles[i]);

}

}

1. Certains éléments de cet atelier sont inspirés du chapitre sur les fichiers du cours de Christophe Darmangeat (<http://pise.info/algo/fichiers.htm>), du cours sur les fichiers de Christiane Glime (notes manuscrites prises lors de l’année académique 1995-96) et de l’article de Guilhem Wailly sur les entrées/sorties (<http://gdw.free.fr/linux-magazine/fr/c-17.html>). [↑](#footnote-ref-1)
2. Inspiré et extrait de <https://zestedesavoir.com/tutoriels/755/le-langage-c-1/1043_aggregats-memoire-et-fichiers/4911_les-fichiers/> [↑](#footnote-ref-2)
3. <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1146.pdf> [↑](#footnote-ref-3)