**Langage de programmation avancé**

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

**DA-IA – bloc 1**

Atelier 3 – Pointeurs

Objectifs

* Continuer à utiliser les éléments vus dans les modules précédents
* Expliquer ce qu’est un pointeur
* Démontrer l’utilité des pointeurs
* Utiliser les pointeurs

[Introduction 2](#_Toc159941770)

[A. Pourquoi des pointeurs ? 3](#_Toc159941771)

[B. Déclarer et utiliser des pointeurs 4](#_Toc159941772)

[Déclaration 4](#_Toc159941773)

[Affectation 5](#_Toc159941774)

[Utilisation 6](#_Toc159941775)

[C. Arithmétique des pointeurs 9](#_Toc159941776)

[Pointeurs et tableaux 9](#_Toc159941777)

[Pourquoi éviter ce genre de notations ? 13](#_Toc159941778)

[D. Pointeur générique 15](#_Toc159941779)

[E. Allocation dynamique de la mémoire 17](#_Toc159941780)

[Première fonction d’allocation : malloc 17](#_Toc159941781)

[Application : tableau dynamique 17](#_Toc159941782)

[Libération de la mémoire allouée 19](#_Toc159941783)

[Qu’est-ce que la pile d’appels ? 19](#_Toc159941784)

[Qu’est-ce que le tas ? 19](#_Toc159941785)

[F. Manipulation de structures de données complexes 22](#_Toc159941786)

[Pointeurs et structures 22](#_Toc159941787)

[Application : liste chaînée 25](#_Toc159941788)

[G. Passage par adresse/référence 27](#_Toc159941789)

# Introduction

Durant les précédents cours de programmation, vous avez écrit des programmes qui permettaient de mémoriser les informations sous la forme de variables de type scalaire. Vous avez également appris à regrouper plusieurs informations sous la forme d’un tableau (plusieurs informations de même type) ou sous la forme d’une structure (plusieurs informations qui peuvent être de types différents). Il s’agit des différentes façons de mémoriser des informations de façon **statique**.

Dans cet atelier, vous allez apprendre à gérer la mémoire de façon **dynamique**.

En effet, comme vous allez le voir dans la suite de cet atelier, il est possible de travailler en allouant la mémoire **au fur et à mesure de l’exécution du programme**, au lieu de tout allouer lors de la compilation. C’est ce qui est appelé l’allocation **dynamique** de la mémoire. Pour ce faire, on a besoin d’utiliser des variables particulières, couramment appelées **pointeurs**.

La théorie sur les pointeurs est souvent considérée comme complexe. En fait, c’est loin d’être la matière la plus complexe en informatique, mais il s’agit d’être rigoureux dans les notations et clair dans le nommage des variables… comme toujours !

Vous allez donc d’abord appréhender les raisons pour lesquelles les pointeurs sont nécessaires en programmation.

Ensuite, les éléments syntaxiques associées aux pointeurs sont présentés en commençant pas les notations de base pour aller vers des notations plus complexes.

Ces notations comprises, vous apprendrez à manipuler une première fonction de la bibliothèque standard pour allouer de la mémoire.

Enfin, deux autres fonctions permettant d’allouer la mémoire de façon dynamique sont présentées.

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 Visual Studio. Donnez-lui un nom vous permettant de vous y retrouver par la suite, par exemple, Module3Atelier.

Ajoutez lui un fichier source appelé pointeurs.c qui contient la fonction main.

N’hésitez pas à réaliser cet atelier en plusieurs fois !

Et n’oubliez pas que s’il est long, c’est parce que les solutions en font partie …

# Pourquoi des pointeurs ?

Avant de savoir à quoi ils servent, voici une courte définition de ce qu’est un pointeur. Il s’agit d’une variable qui **contient l’adresse d’une autre variable**. C’est tout !

Pourquoi les programmeurs ont-ils eu besoin de créer ce type de variable ?

Pour vous expliquer cela, rien de tel que de vous montrer quelques lacunes du C en l’absence du concept de pointeur.

Dans le cadre du programme permettant de gérer les cartes (voir ateliers précédents), plusieurs choses pourraient être améliorées :

1. Taille du tableau de cartes : vous ne connaissez pas le nombre de cartes qui seront mémorisées par les utilisateurs du site. Comme elles sont mémorisées dans un tableau, dont la mémoire est allouée statiquement, il est nécessaire de **déterminer arbitrairement** le nombre maximum de cartes qu’il peut contenir. Il se peut donc que trop ou trop peu de place mémoire soit allouée.

Dans le cadre des exercices de ce cours, cela semble anodin, mais imaginez que l’application tourne sur un serveur et qu’elle soit mise à disposition de tous les abonnés au site… Comment déterminer ce nombre maximum ?

De plus, ce nombre **ne peut pas être modifié** lorsque l’application est en cours d’exécution. En effet, le nombre maximum de cartes est défini au moment de l’écriture du code source. Cette valeur est donc fixée dans le programme exécutable et ne pourra plus être changé sans repasser par le code source, la phase de compilation…

Amélioration : avoir un moyen pour mettre à jour la taille du tableau en fonction du nombre de cartes réellement mémorisées. Ce nombre pourrait alors évoluer bien que l’application soit en cours d’exécution.

1. Taille des chaînes de caractères : un autre cas est celui des chaînes de caractères, comme le nom de la carte ou les mots clés. À nouveau, avec une gestion statique de la mémoire, il faut **déterminer arbitrairement** la taille des chaînes… Il faut donc que le programmeur soit prévoyant et se mette à la place des utilisateurs pour s’assurer qu’aucune erreur ne survienne lors de la mémorisation de ces chaînes.

Amélioration : avoir un moyen pour allouer la place mémoire nécessaire pour chaque chaîne de caractères de façon à être indépendant de celle-ci lorsque l’application est en cours d’exécution.

1. Mémorisation sous la forme d’un tableau : le fait d’utiliser un tableau pour mémoriser des informations est justifié dans certain cas, mais peut amener à des lourdeurs algorithmiques dans d’autres. Dans le cas des cartes, si le programme les mémorise en ordre alphabétique sur le nom de celles-ci et qu’il permet d’en ajouter et d’en supprimer, un tableau n’est pas la solution idéale.

Amélioration : utiliser d’autres techniques que les tableaux pour mémoriser un grand nombre de données : les listes chaînées, les arbres…

1. Obtention d’une « fiche » et du booléen ficheExiste : lors de l’obtention d’une carte (qui est donc une fiche selon les termes utilisés en Compléments de programmation), la fonction ne peut pas renvoyer plusieurs valeurs. Or, en Compléments de programmation, il est demandé de renvoyer la carte ainsi qu’un booléen attestant de son existence. Cette façon de faire n’est pas envisageable en C.

Amélioration : trouver une astuce pour permettre à la fonction **appelante** de récupérer plusieurs résultats suite à l’exécution de la fonction **appelée**.

Trois améliorations sont donc nécessaires :

* l’allocation dynamique de la mémoire (tableaux dynamiques) 🡺 il faut pouvoir allouer un espace mémoire, dont la taille est déterminée lors de l’exécution du programme, et en récupérer l’**adresse**…
* la manipulation de structures de données complexes : listes chaînées, arbres, graphes… 🡺 il faut allouer/libérer de la mémoire au fur et à mesure de la gestion des données, et donc avoir accès aux **adresses** des espaces mémoire alloués.
* le passage par adresse/référence (par opposition au passage par valeur/copie) 🡺 il faut passer, à la fonction, l’**adresse** d’une variable (comme pour les tableaux) afin que ce soit cette variable qui soit directement modifiée.

Ces améliorations nécessitent toutes les trois de pouvoir mémoriser des adresses mémoire. Elles sont mémorisées dans des variables appelées « pointeurs » !

Il est à noter que la partie sur les structures complexes est vue dans les exercices en parallèle avec le cours d’algorithmique.

# Déclarer et utiliser des pointeurs

Comme toute variable en C, un pointeur doit être déclaré avant son utilisation.

## Déclaration

Pour rappel, une variable correspond à une zone mémoire dans laquelle on stocke une valeur, une information, une donnée…

Quel que soit le langage, elle est caractérisée par son **nom**, son **type**, sa **valeur**, son **adresse** et sa **portée**.

Dans le cas du pointeur, c’est le **type de la variable** dont il contient l’adresse qu’il faut utiliser dans sa déclaration. Pour préciser qu’il s’agit d’un pointeur, on ajoute un \* entre le type cible et le nom de la variable/du pointeur.

Pour déclarer un pointeur, il faut donc respecter la syntaxe suivante :

<target\_element\_type>\* <var\_name>;

Il faut y remplacer

* <target\_element\_type> par le type de la variable dont il contient l’adresse ou vers laquelle il pointe,
* <var\_name> par le nom de la variable/du pointeur.

L’instruction ci-dessous permet de déclarer une variable de type « pointeur sur un entier ».

int \* pNbTorchesSac;

## Affectation

Lorsqu'on déclare un pointeur, on réserve uniquement de la place pour stocker une adresse. L'objectif est généralement d'y stocker ensuite l'adresse-mémoire où se trouve une valeur, mais la déclaration du pointeur ne réserve pas automatiquement de la place pour cette valeur. Cela fonctionne un peu comme l'emplacement d'une carte postale où on est censé écrire l'adresse du destinataire : il y a la place pour écrire l'adresse, mais cela ne construit pas automatiquement une maison à l'adresse en question.

Il faut donc utiliser une seconde opération pour réserver l'espace-mémoire destiné au stockage de la valeur. Cela peut se faire de plusieurs manières, par exemple en utilisant une fonction d'allocation de mémoire (voir plus loin). Dans un premier temps, on utilisera une méthode plus simple : récupérer l'adresse d'une autre variable déclarée indépendamment du pointeur.

L’opérateur qui permet de récupérer l’adresse d’une variable est le &. Vous l’avez déjà utilisé dans le cadre de la fonction scanf\_s.

Pour mémoriser l’adresse de la variable nbTorchesSac dans la pNbTorchesSac, procédez comme ci-dessous.

int nbTorchesSac;

int \* pNbTorchesSac = &nbTorchesSac;

Pour observer ce que contient chacune des deux variables, il est intéressant de les afficher. Utiliser %p comme format d’affichage du pointeur. Il s’agit d’un format d’affichage prévu pour les adresses.

Écrivez ces deux instructions dans votre fichier source, dans la fonction principale. Ajoutez les instructions qui permettent d’afficher la valeur de chaque variable. Compilez le programme puis exécutez-le. N’oubliez pas de donner une valeur à la première variable, par exemple 20. Si vous ne le faites pas, vous allez afficher ce qui est déjà en mémoire…

Le fichier source devrait ressembler à ce qui est ci-dessous, à la différence que dans ce code, on affiche également les adresses de chacune des variables entre parenthèses, pour permettre une meilleure visualisation de ce dont il est question.

void main(void) {

int nbTorchesSac = 20;

int \* pNbTorchesSac = &nbTorchesSac;

printf("Valeur de nbTorchesSac %d (%p)\n", nbTorchesSac, &nbTorchesSac);

printf("Valeur de pNbTorchesSac %p (%p)\n", pNbTorchesSac, &pNbTorchesSac);

}

L’exécution de ce programme doit afficher quelque chose comme ce qui suit. Les adresses ne sont probablement pas les mêmes puisque c’est le système d’exploitation qui gère la mémoire.

Valeur de nbTorchesSac 20 (003DF820)

Valeur de pNbTorchesSac 003DF820 (003DF81C)

Vous remarquez que la valeur la variable pNbTorchesSac est bien l’adresse de la variable nbTorchesSac.

Voici un aperçu de ce qui est en mémoire. La mémoire est un ensemble d’octets (b***y***tes) consécutifs, représentés par les carrés, chaque octet ayant une adresse.

octet

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | valeur |  |  |  | entier |  |  |  |  | 4 octets (32 bits) |  |  |  |  |  |  |  |  |  |
|  | 3DF820 | 00 | 00 | 00 | 14 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 00 | 3D | F8 | 20 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Lors de la déclaration de nbTorchesSac (abrégée nTS dans le dessin), 4 octets sont alloués pour mémoriser une valeur de type entier. Cette variable se situe à l’adresse 3DF820 et elle contient la valeur 2010 = 1416 (ici exprimée sur 4 octets, donc 00000014).

Lors de la déclaration de pNbTorchesSac (abrégée pNTS dans le schéma), 4 octets sont alloués pour mémoriser une adresse de 32 bits. La taille de l’adresse dépend du système d’exploitation et du matériel… On lui affecte l’adresse de la variable nbTorchesSac, c’est-à-dire 3DF820.

Une dernière notion importante, relative aux pointeurs : il est possible de mettre une valeur dans une variable de type pointeur permettant de préciser que celui-ci ne pointe nulle part. Cette valeur particulière est le zéro du type et est notée NULL.

int \* pNbTorchesSac = NULL;

Attention cependant à ne pas faire d’initialisation inutile !

## Utilisation

Vous savez déclarer une variable de type pointeur, lui affecter une adresse (celle d’une variable déjà déclarée), mais comment accéder à la valeur de la variable vers laquelle il pointe ?

En effet, dans l’exemple ci-dessus, vous disposez à la fois de la variable et d’un pointeur vers celle-ci. Pour afficher la valeur de la variable, il est donc plus facile de passer par la variable en elle-même. Cependant, si vous ne disposiez pas de la variable pointée, par exemple parce que vous êtes dans une fonction qui a reçu l’adresse de celle-ci, vous ne pouvez en connaître la valeur que via le pointeur.

Pour atteindre la valeur via le pointeur, un nouvel opérateur est nécessaire, appelé **opérateur d’indirection** (ou **opérateur de déréférencement**). En C, cet opérateur se note par une étoile \*. La syntaxe est la suivante :

\*<var\_name>

Il faut y remplacer <var\_name> par le nom de la variable/du pointeur.

Notations

Vous avez remarqué que l’étoile \* est utilisée pour exprimer deux choses différentes : la multiplication et l’indirection.

Avant de continuer, il est intéressant de s’arrêter sur une notion importante de la définition d’un langage : « Les notations **infixée** (ou **infixe**), **préfixée** (ou **préfixe**) et **postfixée** (ou **postfixe**) sont des formes d'écritures d'expressions algébriques qui se distinguent par la position relative qu'y prennent les opérateurs et leurs opérandes. Un opérateur est écrit

* **avant** ses opérandes en notation **préfixée** (opérateur opérande),
* **entre** ses opérandes en notation **infixée** (opérande1 opérateur opérande2) et
* **après** ses opérandes en notation **postfixée**  (opérande opérateur).[[1]](#footnote-2) »

La syntaxe, grammaire du langage, est définie de façon à éviter que le compilateur ne confonde ces deux opérateurs : l'opérateur de multiplication est **infixé** et l'opérateur d'indirection est **préfixé**.

De plus, il faut distinguer l’utilisation de l’étoile \* lors de la déclaration du pointeur et lors de l’accès à la valeur de la variable pointée par celui-ci :

* lors de la déclaration du pointeur, bien que la norme demande que l’étoile soit collée au nom de la variable, vous pouvez (automatique dans les options de base de Visual Studio) la mettre du côté du type afin de visualiser qu’il s’agit du type de la variable : int\* pEntier;
* lors de l’accès à la valeur de la variable pointée, vous devez mettre l’étoile contre le nom de la variable : \*pEntier = 5;

Ainsi, pour afficher la valeur de la variable dont l’adresse est dans pNbTorchesSac, autrement dit la variable pointée par pNbTorchesSac, il faut utiliser l’instruction suivante :

printf("Valeur de \*pNbTorchesSac %d", \*pNbTorchesSac);

Écrivez cette nouvelle instruction, au bon endroit, dans votre fichier source. Compilez-le puis exécutez-le.

L’exécution de ce programme doit afficher quelque chose comme ce qui suit.

Valeur de nbTorchesSac 20 (0054FC2C)

Valeur de pNbTorchesSac 0054FC2C (0054FC28)

Valeur de \*pNbTorchesSac 20

La valeur affichée via le pointeur est bien la valeur de la variable nbTorchesSac. De plus, vous pouvez observer que d’une exécution à l’autre, la mémoire allouée n’est pas la même puisque les adresses ont changé.

Ajoutez, dans votre fichier source, l’instruction qui permet d’augmenter de 3 la valeur de la variable pointée via le pointeur. Recopiez ensuite l’instruction d’affichage de la valeur de la variable pointée et affichez la valeur de la variable nbTorchesSac et exécutez-le.

Attention, c’est bien la valeur de la variable pointée que vous devez modifier, pas la valeur du pointeur. Si vous changez la valeur du pointeur, vous modifiez alors l’adresse qu’il contient. Vous pouvez tester ce phénomène en modifiant la valeur du pointeur, c’est-à-dire l’adresse qu’il contient…

Le fichier source doit ressembler à ce qui est ci-dessous.

void main(void) {

int nbTorchesSac = 20;

int \* pNbTorchesSac = &nbTorchesSac;

printf("Valeur de nbTorchesSac %d (%p)\n", nbTorchesSac, &nbTorchesSac);

printf("Valeur de pNbTorchesSac %p (%p)\n", pNbTorchesSac, &pNbTorchesSac);

printf("Valeur de \*pNbTorchesSac %d\n\n", \*pNbTorchesSac);

printf("== Modification de la valeur pointee ==\n");

\*pNbTorchesSac += 3;

printf("Valeur de \*pNbTorchesSac %d\n", \*pNbTorchesSac);

printf("--> Valeur de nbTorchesSac %d\n\n", nbTorchesSac);

}

L’exécution de ce programme doit afficher quelque chose comme ce qui suit.

Valeur de nbTorchesSac 20 (007BFBFC)

Valeur de pNbTorchesSac 007BFBFC (007BFC00)

Valeur de \*pNbTorchesSac 20

== Modification de la valeur pointee ==

Valeur de \*pNbTorchesSac 23

--> Valeur de nbTorchesSac 23

Assurez-vous d’avoir bien compris !

Si vous n’avez pas mis l’opérateur d’indirection, vous devriez obtenir un code correspondant à celui-ci.

printf("== Modification du pointeur ==\n");

printf("Valeur de pNbTorchesSac %p avant modification.\n", pNbTorchesSac);

pNbTorchesSac += 3; // expliqué en détail plus loin...

printf("Valeur de pNbTorchesSac %p apres modification.\n", pNbTorchesSac);

printf("Valeur de \*pNbTorchesSac %d", \*pNbTorchesSac);

L’affichage devrait similaire à ce qui suit.

== Modification du pointeur ==

Valeur de pNbTorchesSac 007BFBFC avant modification.

Valeur de pNbTorchesSac 007BFC08 apres modification.

Valeur de \*pNbTorchesSac 11898856

Essayez de comprendre pourquoi la valeur du pointeur, donc l’adresse qu’il contient est augmentée de 1210 = C16. Si vous n’y arrivez pas, pas de stress, c’est le sujet du prochain point.

Voici le tableau des priorités des opérateurs, mise à jour avec ces derniers opérateurs :

|  |  |
| --- | --- |
| .(structure) | → |
| -(unaire) ++ -- (*type*) **\**(indirection)*** | ← |
| \* / % | → |
| + -(binaire) | → |
| < <= > >= | → |
| == != | → |
| ?: | ← |
| = += -= \*= /= %= | ← |

# Arithmétique des pointeurs

Un pointeur contenant une adresse, il semble logique que seules l’addition et la soustraction puissent être utilisées avec ce type de variable. Il ne serait en effet pas utile, voire complètement aberrant, de multiplier une adresse ou même de la diviser.

Les opérateurs qui peuvent être utilisés sur une variable de type pointeur sont les opérateurs binaires + et -, et les opérateurs unaires ++ et --. La suite de ce point permet de comprendre comment les utiliser et pourquoi limiter leur utilisation.

## Pointeurs et tableaux

L’arithmétique des pointeurs permet donc de modifier la valeur d’une variable de type pointeur. Mais à quoi cela peut-il bien servir ?

En fin de point précédent, vous avez exécuté votre code (ou le code qui vous a été fourni) pour essayer de comprendre ce qui arrive si vous ajoutez 3 à l’adresse contenue dans la variable pointeur. Le code à analyser est le suivant…

printf("Valeur de pNbTorchesSac %p avant modification.\n", pNbTorchesSac);

pNbTorchesSac += 3; // expliqué en détail plus loin...

printf("Valeur de pNbTorchesSac %p apres modification.\n", pNbTorchesSac);

printf("Valeur de \*pNbTorchesSac %d", \*pNbTorchesSac);

Vous avez observé quelque chose comme ce qui est proposé ci-dessous.

Valeur de pNbTorchesSac 007BFBFC avant modification.

Valeur de pNbTorchesSac 007BFC08 apres modification.

Valeur de \*pNbTorchesSac 11898856

Voici quelques mots d’explications sur l’effet de l’instruction suivante.

pNbTorchesSac += 3;

Lorsque vous ajoutez 3 à la valeur contenue dans pNbTorchesSac, vous modifiez l’adresse contenue dans cette variable. Le pointeur ne pointe donc plus vers la variable nbTorchesSac, mais 3 entiers plus loin dans la mémoire.

Ainsi, si nbTorchesSac est à l’adresse 7BFBFC, pNbTorchesSac contient cette adresse. Si vous ajoutez 3 à cette valeur (cette adresse), pNbTorchesSac contiendra 7BFC08 (7BFBFC + la taille de 3 entiers, donc + 3 fois 4 octets, donc + 12 octets).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7BFBFC  7BFC00  7BFC04  7BFC08  entier  4 octets (32 bits)  valeur |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 00 | 00 | 00 | 14 |  |  |  |  |  |  |  |  | 11 | 89 | 88 | 56 |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 00 | 7B | FC | 08 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

D’où, quand vous affichez la valeur pointée, après avoir modifié le pointeur, vous avez la valeur en mémoire à cette nouvelle adresse. Dans l’exemple, la valeur est une suite de 32 bits dont la valeur en hexadécimal est 11898856.

octet

Vous comprenez ? Si oui, parfait ! Si non, relisez et faites des dessins, surlignez en fluo, prenez des notes… Faites en sorte de comprendre parfaitement ce qu’il se passe à chaque étape !

Maintenant que vous avez compris le principe, vous devriez vous rendre compte qu’il n’est donc absolument pas de bon goût de modifier l’adresse contenue dans une variable de type pointeur… sauf dans le cas des tableaux…

Et oui, un tableau est composé de plusieurs valeurs de même type, ces valeurs étant placées les unes à la suite des autres dans la mémoire, en C. L’arithmétique des pointeurs prend alors tout son sens.

Dans le même projet (fichier source) que précédemment, mettez les instructions précédentes en commentaire et déclarez un tableau caPersos qui contient la classe d’armure (capacité à résister à une attaque) de chacun des trois personnages que vous avez, ici appelés Elminster, Artémis et Bruenor[[2]](#footnote-3). *Les noms ne sont pas mémorisés, mais ils permettent de décrire la situation en mémoire avec quelque chose de plus parlant que des indices.*

Le tableau est initialisé avec les valeurs correspondant à la classe d’armure des personnages en question. Elminster a une CA de 17, Artémis de 16 et Bruenor de 22.

L’instruction de déclaration et initialisation est donc la suivante :

int caPersos[3] = {17, 16, 22};

Affichez ensuite l’adresse à laquelle le tableau se trouve. Pensez au fait que le nom du tableau correspond à l’adresse de la première cellule du tableau…

L’instruction qui suit permet de voir à quelle adresse est le tableau en utilisant les deux notations déjà vues.

printf("caPersos est a l'adresse %p - %p.\n", caPersos, &caPersos[0]);

Cette instruction donne quelque chose qui ressemble à ceci :

caPersos est a l'adresse 00DDFADC - 00DDFADC.

Cela suppose que le tableau se trouve en mémoire à l’adresse DDFADC comme sur le dessin ci-dessous. La première cellule, celle à l’indice 0, se trouve à cette même adresse. La deuxième cellule se trouve 4 octets plus loin puisqu’il s’agit d’entiers (32 bits 🡺 4 octets). Et la dernière cellule est à nouveau 4 octets plus loin que la précédente…

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7BFBFC  7BFC00  7BFC04  7BFC08  entier  4 octets (32 bits)  valeur |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 00 | 00 | 00 | 11 | 00 | 00 | 00 | 10 | 00 | 00 | 00 | 16 | DDFAE8 |  |  |  |  |  |
| DDFADC |  |  |  |  | DDFAE0 |  |  |  | DDFAE4 |  |  |  |  |  |  |  |  |  |  |

Affichez la valeur et l’adresse de chaque cellule en passant d’abord (1) par un indice (comme d’habitude), pour avoir une idée du résultat attendu.

En y ajoutant un peu de forme, les instructions à ajouter au programme sont les suivantes.

for(int i = 0; i < 3; i++) {

printf("caPersos[%d] vaut %d et est a l'adresse %p.\n",   
 i, caPersos[i], &caPersos[i]);

}

Ajoutez ensuite les instructions qui permettent d’afficher la valeur et l’adresse de chaque cellule en passant (2) par un pointeur. Commencez par déclarer et initialiser le pointeur !

En y ajoutant un peu de forme, le résultat devrait ressembler au programme ci-dessous.

int \* pCaPerso = caPersos;

for(int i = 0; i < 3; i++) {

printf("caPersos[%d] vaut %d et est a l'adresse %p.\n",   
 i, \*pCaPerso, pCaPerso);

pCaPerso++;

}

(1) Dans la première boucle, vous parcourez le tableau au moyen d’un **indice** i.

Vous déclarez donc cet indice et accédez à la valeur grâce à la notation « tableau » caPersos[i]. Comme vous l’avez vu dans l’atelier sur les tableaux, cette notation est traduite par le compilateur en caPersos + i \* sizeof(int), caPersos correspondant à l’adresse de la première cellule du tableau, comme montré plus haut.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7BFBFC  7BFC00  7BFC04  7BFC08  entier  4 octets (32 bits)  valeur |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 00 | 00 | 00 | 11 | 00 | 00 | 00 | 10 | 00 | 00 | 00 | 16 | DDFAE8 | 00 | 00 | 00 | 01 |  |
| DDFADC |  |  |  |  | DDFAE0 |  |  |  | DDFAE4 |  |  |  |  |  |  |  |  |  |  |

Pour avoir la CA d’Artémis, il suffit d’écrire caPersos[i] avec i qui vaut 1. L’adresse de cette cellule est celle de la première cellule DDFADC à laquelle s’ajoute un déplacement de « i \* la taille d’une cellule », c’est-à-dire   
DDFADC + 1 \* sizeof(int) 🡺 DDFADC + 1 \* 4 🡺 DDFAE0

Comme vous pouvez l’observer, le résultat de l’application de ce déplacement par rapport à l’adresse de la première cellule correspond bien à l’adresse de la cellule d’indice 1, c’est-à-dire DDFAE0, qui contient la CA d’Artémis. Et ainsi de suite…

(2) Dans la seconde boucle, vous parcourez le tableau au moyen d’un **pointeur** pCaPerso.

La variable contenant l’indice n’est plus nécessaire pour le parcours du tableau (mais bien pour compter le nombre d’itérations…). Le but est de récupérer la valeur d’une cellule au moyen de son adresse.

Il faut donc déclarer une variable de type « pointeur sur un entier » afin d’y mémoriser l’adresse d’une cellule du tableau. Elle doit être initialisée avec l’adresse de la première cellule du tableau DDFADC, qui comme vous le savez maintenant correspond au nom du tableau. L’instruction qui correspond à ces deux étapes est la suivante.

int \* pCaPerso = caPersos;

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7BFBFC  7BFC00  7BFC04  7BFC08  entier  4 octets (32 bits)  valeur |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 00 | 00 | 00 | 11 | 00 | 00 | 00 | 10 | 00 | 00 | 00 | 16 | DDFAE8 | 00 | 00 | 00 | 01 |  |
| DDFADC |  |  |  |  | DDFAE0 |  |  |  | DDFAE4 |  |  |  |  |  |  |  |  |  |  |
|  |  | 00 | DD | FA | DC |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Ensuite, pour afficher la valeur qui est à cette adresse, vous devez utiliser l’**opérateur d’indirection** sur cette variable, comme ci-dessous.

printf("caPersos[%d] vaut %d et est a l'adresse %p.\n",   
 i, \*pCaPerso, pCaPerso);

Enfin, pour avancer dans le tableau, il faut incrémenter le pointeur de 1. Cela provoque en effet un déplacement de la taille de ce sur quoi il pointe, ici un entier, donc quatre octets. L’instruction qui permet de faire cela est la suivante.

pCaPerso++;

Cette incrémentation a pour effet de faire passer la valeur du pointeur de DDFADC à **DDFAE0**, et donc d’avancer exactement d’une cellule, comme le montre la partie en bleu clair (et gras) ci-dessous. La valeur de la cellule pointée par le pointeur après incrémentation est donc bien la CA d’Artémis.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7BFBFC  7BFC00  7BFC04  7BFC08  entier  4 octets (32 bits)  valeur |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 00 | 00 | 00 | 11 | 00 | 00 | 00 | 10 | 00 | 00 | 00 | 16 | DDFAE8 | 00 | 00 | 00 | 01 |  |
| DDFADC |  |  |  |  | **DDFAE0** |  |  |  | DDFAE4 |  |  |  |  |  |  |  |  |  |  |
|  |  | 00**00** | DD**DD** | FA **FA** | DC **E0** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

C’est à cela que servent les opérateurs arithmétiques sur les pointeurs. De façon plus théorique, lorsque vous modifiez la valeur d’un pointeur au moyen d’un opérateur arithmétique, vous effectuez un déplacement en mémoire… Des exercices à ce sujet seront abordés plus en détail dans les séries d’exercices qui suivent le cours de théorie.

Vous comprenez ? Si oui, parfait ! Si non, relisez et faites des dessins, surlignez en fluo, prenez des notes… Faites en sorte de comprendre parfaitement ce qu’il se passe à chaque étape !

Pour vous assurer d’avoir tout compris, tentez d’expliquer pourquoi le code proposé ci-dessous utilise une notation qui n’est pas valide… Si vous n’y arrivez pas, testez le bout de code et lisez le message d’erreur.

#define NB\_PERSOS 3

void main(void) {

int caPersos[NB\_PERSOS] = {17, 16, 22};

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

printf("caPersos[%d] vaut %d et est a l'adresse %p.\n",    
 i, \*caPersos, caPersos);

caPersos++;

}

}

Ce bout de code n’est pas valide car il utilise l’opérateur d’indirection sur un élément qui n’est pas un pointeur mais simplement un tableau

Et maintenant, pourquoi la façon de rédiger proposée ci-dessous ne pose-t-elle plus problème, malgré que ce ne soit toujours pas « joli ».

void afficheCA(int \* caPersos) {

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

printf("caPersos[%d] vaut %d et est a l'adresse %p.\n",    
 i, \*caPersos, caPersos);

caPersos++;

}

}

void main(void) {

int caPersos[NB\_PERSOS] = {17, 16, 22};

afficheCA(caPersos) ;

}   
  
  
Ici le code ne pose pas de problèmes car le code de la fonction travaille bien sur un pointeur mais cepandant le nom du pointeur est le même que celui du tableau ce qui pose problème.

ATTENTION, ceci n’est pas une bonne façon de travailler en termes de nom de variables !

Si vous ne savez pas répondre, discutez-en avec les autres étudiants sur Discord !

## Pourquoi éviter ce genre de notations ?

Le problème de l’utilisation abusive des notations offertes par l’arithmétique des pointeurs est que le code produit devient très vite illisible et incompréhensible.

Essayez de trouver à quoi sert cette fonction.

void nomADeviner (char \* d, char \* s) {

while( \*d++ = \*s++ );

}

Avez-vous trouvé ?

Pas simple à déchiffrer… Rien que les noms de variables sont critiquables !

Il s’agit d’une fonction permettant de copier une chaîne de caractères (source) dans une autre (destination)… Mais comment celle-ci peut-elle bien faire cela ?

Au lieu de décomposer chaque étape afin de rendre le programme facile à comprendre pour n’importe quel programmeur, même celui qui apprend le C, le code est basé sur des notations à effets de bord. En effet, la « condition » de la boucle regroupe, à elle seule, 3 instructions et 3 expressions.

En respectant la priorité des opérateurs,

1. ce sont les opérateurs d’indirection qui prennent la main en récupérant les valeurs pointées par chacun des pointeurs : \*d et \*s (deux expressions),
2. ensuite viennent les opérateurs d’incrémentation qui permettent de, déjà, passer à l’adresse suivante : d++ et s++ (deux instructions),
3. suit l’opérateur d’affectation qui permet de mettre la valeur récupérée au point 1 par l’expression \*s à l’adresse pointée par d : \*d = \*s (une instruction),
4. et enfin, l’évaluation de la condition (une expression), particulièrement incompréhensible, dont la valeur résulte de l’évaluation de l’expression de droite de l’opérateur d’affectation, ici le caractère pointé par s. Dans le cas où le caractère pointé est autre que '\0', sa valeur est *truthy* et l’évaluation de la condition donne true… la boucle continue. Dans le cas où ce caractère est celui de fin de chaine de caractères '\0', la valeur est *fasly* et l’évaluation de la condition donne false… la boucle s’arrête !

Truthy, falsy

Une valeur est *falsy* si, considérée dans un contexte booléen, elle équivaut à false. En C, toute les valeurs équivalentes à false sont, selon leur type : 0, 0.0, '\0', NULL…

Une valeur est *truthy* si, considérée dans un contexte booléen, elle équivaut à true. En C, les valeurs équivalentes à true sont celles qui ne sont pas *falsy*.

Pour une explication plus détaillée de ce bout de code, allez lire le site suivant : <https://medium.com/@larissafeng/understanding-while-s-t-abb2cc518f96>.

Cet exemple, parmi tant d’autres, permet de vous montrer qu’il est possible de se faire passer pour « un expert des pointeurs » grâce à l’une ou l’autre instruction illisible, mais cela ne fait pas de vous un bon programmeur, que du contraire !

Vous pouvez également aller lire la prose de Geoff Wozniak, « just one of those persons on the Internet », sur le livre intitulé « *Mastering C Pointers: Tools for Programming Power »* de Robert J. Traister. Il y discute de la façon dont l’auteur présente ces morceaux de code comme étant la bonne façon de faire alors que ce sont tout simplement des exemples de « Ce qu’il ne faut pas faire… ».

Geoff Wozniak s’est intéressé à ce livre après avoir entendu Brian Kernighan, l’un des deux premiers auteurs sur le langage, montrer ce même exemple tout en disant qu’il s’agissait d’un livre qui est probablement « the worst C programming textbook ever written. ».

Voici, pour les plus curieux d’entre vous, le lien vers la conférence de Brian Kernighan : <https://www.youtube.com/watch?v=8SUkrR7ZfTA>.

Il vous est donc recommandé la plus grande prudence face à ce genre de notations, aussi bien lors de vos rédactions de code que lors de vos recherches et lectures. L’utilisation abusive de ces notations est souvent la preuve que le programmeur qui se cache derrière n’est pas la meilleure référence en la matière…

# Pointeur générique

Un pointeur générique est un pointeur qui peut contenir l'adresse de n'importe quel type de variable. Le but est de pouvoir prévoir un espace mémoire pour une adresse sans connaître le type de la variable pointée.

Jusqu’ici, le mot réservé void était associé à « rien », dorénavant, suivi de l’étoile \*, il sert aussi à dire « pointeur vers je ne sais pas encore quoi ».

Pour déclarer un pointeur générique, il suffit donc de lui associer le type void \*, comme dans l’exemple ci-dessous.

void \* pNombre;

La variable pNombre est donc prévue pour contenir l’adresse d’une autre variable, mais aucune précision quant au type de celle-ci n’est fournie.

Dans le même projet (fichier source) que précédemment, assurez-vous d’avoir déclaré la variable nbTorchesSac. Ajoutez la déclaration de pNombre comme montrée ci-dessus et affectez-lui l’adresse de nbTorchesSac. Essayez maintenant de donner la valeur 42 à la variable nbTorchesSac en passant par le pointeur.

Les instructions doivent ressembler à ce qui est proposé ci-dessous.

pNombre = &nbTorchesSac;

\*pNombre = 42;

Lors de la compilation de ce programme, un *warning* est affiché avec le message suivant.

warning C4047: '=' : les x² d'indirection de 'void \*' et de 'int' sont différents

Quelle est la raison de ce *warning* ?

Le problème vient du fait que le pointeur étant générique, la syntaxe n’autorise pas l’utilisation directe de l’opérateur d’indirection.

C’est pourquoi, le compilateur signale que le fait que le pointeur soit de type void \* alors que la valeur affectée est de type int lui pose problème. En effet, il ne sait pas de quelle place il dispose pour mémoriser cette valeur, puisqu’il ne connaît pas le type de la variable pointée…

Vous comprenez ? Si oui, parfait ! Si non, relisez et faites des dessins, surlignez en fluo, prenez des notes…

En effet, la place mémoire occupée par la variable pointée par ce pointeur dépend du type de cette variable :

* S’il s’agit de la variable nbTorchesSac qui contient un entier int, elle occupe 4 octets (32 bits).
* Par contre la variable prix qui contient une valeur réelle double, occupe 8 octets (64 bits).

Le dessin ci-dessous illustre cette différence (nbTorchesSac vaut 20 et prix vaut 0,99).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | valeur |  |  |  | entier |  |  | 4 octets (32 bits) |  |  |  |  |  |  |  |  |  |  |  |
|  | FFFE2C | 00 | 00 | 00 | 14 |  |  |  |  |  |  | 00 | FF | FE | 2C |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 07 | EF | 5C | 28 | FE | B8 | 51 | EB |  |  |  |  |  |  |  |  |  |  |
|  | FFFE24 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Le compilateur n’est pas capable de déduire cette taille, il faut donc lui préciser lors de l’utilisation du pointeur en question.

Pour pouvoir accéder à la valeur de la variable pointée par pNombre (après lui avoir affecté une adresse), il faut donc préciser la taille que cette valeur occupe dans la mémoire. Pour ce faire, on utilise l’opérateur de transtypage/*casting*, comme montré ci-dessous, dans le cas d’un entier…

\*(int \*)pNombre

Modifiez votre programme en conséquence et essayez à nouveau de le compiler. Normalement, le message du *warning* a disparu.

Le programme doit donc contenir les instructions suivantes.

int nbTorchesSac;

pNombre = &nbTorchesSac;

\*(int \*)pNombre = 20;

Ajoutez maintenant la variable réelle prix et affectez son adresse à pNombre. En respectant les mêmes règles syntaxiques que ci-avant, affectez, via le pointeur, la valeur 0,99 à la variable prix.

Le programme final, en y ajoutant deux instructions d’affichage afin de visualiser ce qu’il se passe, est le suivant.

void main(void) {

int nbTorchesSac;

double prix;

void \* pNombre;

pNombre = &nbTorchesSac;

\*(int \*)pNombre = 20;

printf("nbTorchesSac : %d (%p)\n", nbTorchesSac, &nbTorchesSac);

pNombre = &prix;

\*(double \*)pNombre = 0.99;

printf("prix : %f (%p)\n", prix, &prix);

}

Cette notion est à nouveau abordée dans la suite de cet atelier.

# Allocation dynamique de la mémoire

Les notions de base des pointeurs étant comprises, vous allez maintenant gérer la taille de vos tableaux de façon dynamique.

Pour allouer de la mémoire de façon dynamique, il faut utiliser une des fonctions prévues à cet effet dans la bibliothèque standard. Seule la première est abordée dans un premier temps ; deux autres fonctions d’allocation de mémoire seront présentées au cours de théorie.

Ces fonctions sont fournies par la bibliothèque standard stdlib.h.

## Première fonction d’allocation : malloc

Le prototype de la fonction malloc est le suivant.

void \* malloc(size\_t size);

La fonction malloc tente de réserver un bloc de mémoire de size octets sans l'initialiser. Elle renvoie

* soit l'adresse de l'espace mémoire ainsi alloué, s’il reste de la mémoire disponible,
* soit NULL, s’il n’y a plus de mémoire disponible.

Il faut remplacer size par le nombre d’octets à allouer.

Le type size\_t est un synonyme proposée dans la bibliothèque standard pour préciser qu’il s’agit d’une taille, et donc d’un entier représentant un nombre d’octets.

Le type de retour est un pointeur générique car la fonction est générique et susceptible de renvoyer l’adresse d’un bloc de mémoire correspondant soit à un des types primitifs du langage, soit un type structuré…

Il est donc nécessaire d'effectuer un transtypage/*casting* avant d'affecter l'adresse renvoyée à un pointeur de type défini.

… = (int \*)malloc(…);

… = (Carte \*)malloc(…);

## Application : tableau dynamique

Dans le cadre du programme permettant de mémoriser les CA de vos personnages, il faut réserver un tableau d’entiers (les CA sont des entiers) dont la taille est le nombre de personnages que vous avez.

Jusqu’ici, vous deviez décider à l’avance (lors de la rédaction du code) de la taille du tableau, et celle-ci n’était pas adaptable lors de l’exécution du programme. Il s’agissait d’**allocation statique** de mémoire.

Maintenant que vous connaissez la notion de pointeur ainsi que la fonction malloc, vous pouvez demander au système de vous renvoyer l’adresse d’un espace mémoire dont la taille est exactement celle dont vous avez besoin. Il s’agit d’**allocation dynamique** de mémoire.

Pour reprendre l’exemple des CA de personnages, si vous en avez 3, vous demandez alors de la place mémoire pour 3 entiers, grâce à l’opérateur sizeof.

… = (int \*)malloc(3 \* sizeof(int));

Cette instruction permet d’allouer un espace mémoire constitué de 3 entiers. Ces entiers sont consécutifs dans la mémoire, comme dans le cas de l’allocation statique. La seule différence est que vous pouvez décider de la taille lors de l’exécution… selon vos besoins. En effet le but ici est bien de pouvoir gérer cet espace mémoire comme étant un tableau d’entiers.

Vérifiez que vous avez compris…

Déclarez une variable permettant de mémoriser l’adresse de la première cellule d’un tableau d’entiers, en utilisant la notation « pointeur » (donc pas la notation « tableau »). En effet vous ne connaissez pas la taille du tableau !

Ensuite, écrivez les instructions qui permettent de demander le nombre de personnages que vous avez. Vous n’avez pas besoin de pointeur, ici…

Allouez ce nombre de cellules, en sachant que les CA sont mémorisées sous la forme d’entiers, et affectez l’adresse reçue à la variable déclarée ci-avant.

Vous avez donc déclaré une variable de type pointeur sur un entier, à laquelle vous avez affecté le résultat de l’appel à la fonction malloc avec la taille exprimée en octets, c’est-à-dire un espace mémoire permettant de mémoriser le nombre demandé d’entiers consécutifs.

Enfin, en utilisant les notations « pointeur », garnissez ces cellules avec des valeurs obtenues… et affichez-les pour visualiser ce que vous venez de faire ! Avant d’afficher les CA, n’oubliez pas de vous « repositionner » en début de tableau.

Avez-vous compris pourquoi il ne faut pas toucher à la variable qui contient l’adresse du premier élément du tableau ? (Voir fin du point « Pointeurs et tableaux ».)

Le programme qui résulte de ces différentes étapes est le suivant.

void main(void) {

int nbPersos;

int \* caPersos;

printf ("Nombre de personnages : ");

scanf\_s("%d", &nbPersos); // idéalement à valider

caPersos = (int \*)malloc(sizeof(int) \* nbPersos);

int \* pCaPerso = caPersos;

for(int i = 0; i < nbPersos; i++) {

printf("CA du personnage %d : ", i + 1);

scanf\_s("%d", pCaPerso); // Il ne faut pas mettre le signe & car il s’agit déjà d’une adresse mémoire.

pCaPerso++;

}

puts("------------------------");

pCaPerso = caPersos; // Repositionnement au début du tableau   
 // en donnant l’adresse de la première cellule

for(int i = 0; i < nbPersos; i++) {

printf("CA du personnage %d : %d\n", i + 1, \*pCaPerso);

pCaPerso++;

}

}

## Libération de la mémoire allouée

Deux notions supplémentaires sont nécessaires à la compréhension de ce point :

* la pile d’appels, et
* le tas.

La pile et le tas sont deux mécanismes de gestion de la mémoire dont le fonctionnement et l’utilisation diffèrent.

Les explications qui suivent sont très « vulgarisées » car ce sont des notions qui sont en lien direct avec le système d’exploitation et la gestion de mémoire.

### Qu’est-ce que la pile d’appels ? [[3]](#footnote-4)

La pile d’appels est une partie de la mémoire allouée par le système d'exploitation pour l'exécution des fonctions appelées par un processus (version exécutée d’un programme).

La source est le code que vous avez rédigé (.c), le programme est le résultat de la compilation de ce code (.exe), le processus est ce qui est généré lors de l’exécution du programme.

Elle repose sur les mêmes principes que la structure de données du même nom, c’est-à-dire en LIFO (*last in, first out*) : le dernier élément qui y a été posé est le premier à pouvoir en être retiré.

Alloué quand ? Ainsi, comme montré dans le module sur les fonctions, lors de l’exécution d’un programme, chaque appel de fonction provoque la création d’un nouveau bloc au sommet de la pile, contenant les informations liées à la fonction en question, telles que les variables locales, les paramètres…

Taille ? La taille du bloc est déterminée par le système d'exploitation au moment de l’appel de la fonction et ne peut pas changer.

Supprimé quand ? Ce bloc est supprimé à la fin de l'exécution de la fonction. Ainsi les variables locales et autres éléments faisant partie de ce bloc sont supprimés automatiquement.

Le système ne doit pas mémoriser l'emplacement de retour après l’exécution de la fonction et toute la mémoire allouée à la fonction est libérée d’un coup. Cela rend la pile beaucoup plus facile et plus rapide à utiliser.

### Qu’est-ce que le tas ?[[4]](#footnote-5)

Le tas est un espace mémoire utilisé pour les allocations dynamiques. C’est un espace mémoire partagé par toutes les fonctions d’un même processus.

Alloué quand ? Le tas est créé au démarrage d'un processus et est lié à celui-ci. L’accès au tas dépend du langage utilisé. En C, le tas est mis à contribution lors de l’appel aux fonctions d’allocation telles que malloc.

Taille ? La taille du tas varie et peut augmenter si le programme a besoin de plus de mémoire.

Supprimé quand ? L’espace alloué au tas n’est pas automatiquement libéré lorsqu’une fonction est terminée, et heureusement puisqu’il est partagé par toutes les fonctions du processus. Il est nécessaire de libérer la mémoire occupée lorsque vous n'en avez plus besoin grâce à la fonction free.

Ce système rend la gestion du tas plus complexe et moins rapide que la pile. En effet, il est nécessaire de connaitre en permanence quels blocs sont alloués et de libérer la mémoire quand celle-ci n’est plus utilisée.

Malgré ce que certains pourraient lire sur le web, nos systèmes d’exploitation sont suffisamment évolués pour éviter toute perte de mémoire. En effet, la mémoire allouée par un processus est libérée dès qu’il est fini. Donc pas de panique inutile !

Comme vous venez de le lire, vous avez besoin d’une fonction qui permet de libérer la mémoire lorsque celle-ci n’est plus utilisée.

Le prototype de cette fonction est le suivant.

void free(void \* ptr);

Cette fonction libère l’espace mémoire dont l’adresse est dans ptr.

Il faut remplacer ptr par l’adresse du bloc que vous voulez libérer.

Le paramètre de la fonction est un pointeur générique. En effet, la fonction est prévue pour libérer n’importe quel emplacement mémoire. Lorsque vous lui fournissez une adresse vers un entier, elle va libérer exactement l’espace mémoire associé à cet entier.

Dans le programme précédent, ajoutez, juste après avoir garni le tableau, l’instruction qui permet de libérer l’espace mémoire alloué pour le tableau caPersos. Compilez et exécutez le programme. Que se passe-t-il ?

Sous Visual Studio, il vous envoie sur les roses à sa façon puisque lorsque vous affichez le contenu du tableau après la libération de la mémoire, il affiche quelque chose ressemblant à ce qui suit.

Nombre de personnages : 3

CA du personnage 1 : 17

CA du personnage 2 : 16

CA du personnage 3 : 22

------------------------

CA du personnage 1 : -572662307

CA du personnage 2 : -572662307

CA du personnage 3 : -572662307

Lorsque vous compilez et exécutez avec la ligne de commande cl, le résultat devient le suivant.

Nombre de personnages : 3

CA du personnage 1 : 17

CA du personnage 2 : 16

CA du personnage 3 : 22

------------------------

CA du personnage 1 : 17976560

CA du personnage 2 : 17929584

CA du personnage 3 : 22

N’hésitez pas à le tester vous-même…

Il est donc dangereux d’utiliser une adresse après sa libération. En effet, l’utilisation du pointeur après libération de la zone mémoire allouée n’est pas prévue par la définition du langage. Cette utilisation provoque donc des effets imprévisibles. Dans ce cas-ci, malgré la libération de la mémoire, lorsque vous tentez d’accéder au premier élément du tableau dans la boucle d’affichage, les valeurs en mémoire n’ayant pas été écrasées, le programme se comporte comme vous l’espérez. Mais vous n’avez aucune assurance que ce soit toujours le cas…

Sous Linux, l’exécution du même programme (un peu adapté) donne l’affichage suivant.

Nombre de personnages : 3

CA du personnage 1 : 17

CA du personnage 2 : 16

CA du personnage 3 : 22

------------------------

CA du personnage 1 : 0

CA du personnage 2 : 16

CA du personnage 3 : 22

Comme vous ne pouvez pas présumer de ce que va provoquer l’utilisation d’un pointeur après sa libération, il est, de façon très générale, conseillé :

* d'attribuer la valeur NULL au pointeur juste après la libération de la zone pointée, et à tout autre pointeur faisant référence à la même adresse,
* de tester la valeur nulle avant l’utilisation d'un pointeur qui pourrait avoir été libéré.

Ajoutez, juste après la libération de caPersos, toujours avant la boucle d’affichage, l’instruction qui met cette variable à NULL et retestez le programme.

Sous Visual Studio, il affiche quelque chose comme ce qui suit.

Nombre de personnages : 3

CA du personnage 1 : 17

CA du personnage 2 : 16

CA du personnage 3 : 22

------------------------

Vous devriez également avoir une fenêtre contextuelle avec un message d’erreur tel que…

Exception levée : violation d'accès en lecture.

pCaPerso a été nullptr.

Sous la ligne de commande, il affiche la même chose, mais vous n’avez aucun message d’erreur.

Sous Linux, il se comporte quasi comme sous Visual Studio en affichant la première partie suivie d’un message d’erreur se limitant à…

Erreur de segmentation

Enfin, donner en argument à la fonction free l'adresse d'un bloc qui n'a pas été alloué au moyen d’une des fonctions d'allocation cause à nouveau un comportement indéfini.

Après ces quelques expérimentations, placez la libération de mémoire au bon endroit. Comme le pointeur n’est plus utilisé par la suite dans le programme, mettre sa valeur à NULL est inutile…

# Manipulation de structures de données complexes

Une autre façon de mémoriser de nombreuses informations est l’utilisation de listes chainées, d'arbres…

Lorsque vous connaissez le nombre d’informations à mémoriser, qu’elles ne sont pas triées et que peu de suppressions sont nécessaires, le choix du tableau statique voire dynamique semble judicieux.

Mais lorsque vous devez ajouter régulièrement de nouveaux éléments tout en gardant le tableau trié ou lorsque vous devez supprimer régulièrement des éléments de celui-ci, le choix de la liste chainée, voire de l’arbre ou du graphe, s’avère plus approprié.

En effet, à chaque ajout ou suppression dans un tableau trié, vous devez procéder à de multiples décalages. Si le tableau est grand, la charge de travail est conséquente et le programme devient lent.

Remarque

La théorie sur les listes chainées n’est pas approfondie dans cet atelier puisqu’elle est vue en algorithmique.

Vous allez donc apprendre à utiliser l’allocation dynamique de la mémoire pour des structures de données un peu plus complexes et plus particulièrement les listes chainées.

## Pointeurs et structures

Reprenez l’exercice sur les cartes (faites-en une copie afin de garder la version précédente).

Pour rappel, une carte est décrite par un nom, le nombre de cases en hauteur (entier), le nombre de cases en largeur (entier), le fait qu’elle soit officielle (tirée d’un livre officiel) ou pas (booléen), et une liste de 3 mots clés (un tableau de 3 chaînes de caractères).

Tout comme les types primitifs, les structures peuvent faire l’objet d’une allocation dynamique au moyen de la fonction malloc.

Déclarez un pointeur vers une grotte pGrotte. Allouez ensuite l’espace mémoire nécessaire pour y mémoriser les informations associées.

Ces deux étapes peuvent se résumer à la ligne suivante.

Carte \* pGrotte = (Carte \*)malloc(sizeof(Carte));

Comme dans les diagrammes d’actions, vous devez utiliser la flèche -> pour atteindre un champ d’une structure pointée par un pointeur.

Écrivez les instructions qui permettent d’affecter les valeurs proposées ci-dessous aux membres de la structure pointée par pGrotte.

* Nom : "Grottes pleines de rats",
* Hauteur : 15,
* Largeur : 25,
* Non officielle,
* Mots clés :   
  "souterrain", "rats", "sombre"

N’oubliez pas que copier une chaine de caractères nécessite l’utilisation de la fonction strcpy. Elle est fournie par la bibliothèque standard string.h.

Le résultat se rapproche normalement de ce qui suit.

strcpy\_s(pGrotte->nom, T\_NOM, "Grottes pleines de rats");

pGrotte->nbCasesHauteur = 15;

pGrotte->nbCasesLargeur = 25;

pGrotte->officielle = false;

strcpy\_s(pGrotte->motsCles[0], T\_MOT\_CLE, "souterrain");

strcpy\_s(pGrotte->motsCles[1], T\_MOT\_CLE, "rats");

strcpy\_s(pGrotte->motsCles[2], T\_MOT\_CLE, "sombre");

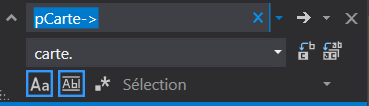
Pour pouvoir voir ce que vous avez mémorisé dans la structure pointée par pGrotte, modifiez la signature de la fonction afficheCarte pour qu’elle reçoive l’adresse de la structure dont vous voulez afficher les champs. N’oubliez pas de modifier le prototype de la fonction également, si vous en avez utilisé un.

Changez ensuite chaque carte. par la notation appropriée aux champs d’une structure pointée… N’oubliez pas l’utilisation du fameux CTRL+H pour remplacer un bout de texte par un autre. Si vous ne savez pas l’utiliser, lisez le cadre qui suit !

Étapes du CTRL + H

Dans Visual Studio, comme dans la plupart des logiciels d’édition (Word, Notepad++, sublime text…), un raccourci souvent associé à celui de la recherche dans un document (CTRL + F) est le rechercher/remplacer.

1. Sélectionnez le bout de code cible, celui qui contient le bout de texte à modifier.
2. Appuyez sur la touche CTRL et la touche H en même temps. La fenêtre contextuelle proposée à la *Figure 1* s’ouvre au-dessus à gauche de votre fenêtre d’édition.



*Figure 1 - CTRL + H VS2019*

1. Sélectionnez les options  qui activent la sensibilité à la casse ainsi que le fait que vous ciblez un mot complet.
2. Si vous avez sélectionné du code, le mot  est affiché.   
   Vous pouvez également cibler le document actif, un projet voire toute une solution…
3. Écrivez le texte à remplacer et le texte de remplacement.
4. Cliquez sur le bouton  pour remplacer toutes les occurrences du premier mot par le second.
5. Admirez le résultat et n’oubliez pas que si vous avez fait une bêtise, c’est au tour du CTRL + Z de vous donner un coup de main…

**À n’utiliser que si/quand vous savez ce que vous faites !**

Le code de la fonction doit maintenant ressembler à ce qui suit.

void afficheCarte(Carte \* pCarte) {

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

pCarte->nom,

pCarte->nbCasesHauteur,

pCarte->nbCasesLargeur,

pCarte->officielle ? "" : "pas ");

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

puts(pCarte->motsCles[i]);

}

}

De la même façon, modifiez la fonction qui permet d’obtenir les informations sur une carte…

Avez-vous compris qu’en passant le pointeur, vous passez l’adresse de ce qui est à garnir et qu’il n’est donc pas utile de renvoyer ce pointeur ? Si non, faites un dessin !

Le code de la fonction doit maintenant ressembler à ce qui suit.

void carteObtenue(Carte \* pCarte) {

char reponse;

printf("Nom : ");

gets\_s(pCarte->nom, T\_NOM);

if (strcmp(pCarte->nom, "") != 0) {

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

scanf\_s("%d", &pCarte->nbCasesHauteur);

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

scanf\_s("%d", &pCarte->nbCasesLargeur);

getchar();

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

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

pCarte->officielle = reponse == 'O';

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

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

scanf\_s("%s", pCarte->motsCles[i], T\_MOT\_CLE);

}

getchar();

}

}

Si vous avez bien travaillé, la fonction principale se limite à quelques instructions, pour l’instant.

void main(void) {

Carte \* pGrotte = (Carte \*)malloc(sizeof(Carte));

carteObtenue(pGrotte);

afficheCarte(pGrotte);

}

Voici le tableau des priorités des opérateurs, mis à jour avec ces derniers opérateurs :

|  |  |
| --- | --- |
| .(structure) **->(pointeur)** | → |
| -(unaire) ++ -- (*type*) **\**(indirection)*** | ← |
| \* / % | → |
| + -(binaire) | → |
| < <= > >= | → |
| == != | → |
| ?: | ← |
| = += -= \*= /= %= | ← |

## Application : liste chaînée

Maintenant que vous savez utiliser les pointeurs avec les structures, vous allez enfin transformer ce tableau de cartes en liste chainée de cartes.

Écrivez la définition de la structure représentant une carte comme étant un chaînon d’une liste chainée.

Deux solutions sont envisageables :

1. ajouter un champ pour en faire une structure autoréférentielle, c’est-à-dire qui a un champ qui fait référence à elle-même.
2. créer une nouvelle structure contenant deux champs, une carte et le pointeur vers la carte suivante.

Si vous allez vers la première solution (1), vous devez modifier plusieurs éléments. En effet, les types changent, les accès à un champ de la structure Carte ne s’écrivent pas de la même façon, les signatures des fonctions doivent être adaptées… tout cela change.

C’est ce qui fait du C un langage qui induit un couplage fort entre les données et les traitements, à la différence d’un langage orienté objet qui applique le concept d’encapsulation…

Si vous allez vers la seconde solution (2), vous allez à nouveau devoir changer certains éléments du programme existant, mais les fonctions qui permettent l’obtention et l’affichage d’une carte ne devront pas être modifiées.

Afin de faire la démarche dans son ensemble, c’est la première solution (1) qui est proposée dans la suite. Il ne tient qu’à vous de tester l’autre solution…

La définition de la structure devient alors celle proposée ci-dessous.

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 \* pSuiv;

};

Comprenez-vous maintenant l’intérêt de définir le synonyme avant la définition de la structure ?

Au lieu du pointeur sur pGrotte, déclarez maintenant 2 pointeurs :

* un pointeur pCartes qui pointe sur la première carte de la liste chainée, le pointeur de début,
* un pointeur pCarte qui pointe vers la carte en cours de traitement, le chainon courant.

Initialisez le pointeur de début de liste à NULL.

Vous devez avoir les déclarations suivantes.

Carte \* pCartes = NULL;

Carte \* pCarte;

Avant de demander à l’utilisateur de fournir les informations sur la carte qu’il veut ajouter, il faut savoir s’il y a encore de la place mémoire pour mémoriser cette nouvelle carte.

Tentez d’allouer la mémoire nécessaire pour une carte et mémoriser la réponse dans la variable pCarte.

Si la fonction malloc a pu allouer la mémoire nécessaire à la nouvelle carte, pCarte contient une adresse autre que NULL.

Testez si la mémoire a bien été allouée et si c’est le cas, demandez les informations sur la carte via la fonction carteObtenue.

Le code se voit donc ajouter les instructions suivantes.

pCarte = (Carte \*)malloc(sizeof(Carte));

if (pCarte != NULL) {

carteObtenue(pCarte);

}

Écrivez maintenant la boucle qui suit cette première obtention.

Faites attention à l’ordre des conditions !

Le code de la fonction principale doit maintenant ressembler à ce qui suit.

Carte \* pCartes = NULL;

Carte \* pCarte;

bool fin, placeDisponible;

do {

pCarte = (Carte \*)malloc(sizeof(Carte));

placeDisponible = pCarte != NULL;

if (placeDisponible) {

carteObtenue(pCarte);

fin = strcmp(pCarte->nom, "") == 0;

}

} while (placeDisponible && !fin);

Avant de passer à la fonction d’ajout d’une carte à la liste chainée, prévoyez une boucle dont le but est d’afficher chaque chainon de la liste chainée.

Pensez à faire appel à la fonction d’affichage d’une carte…

Les instructions à ajouter en fin de fonction principale sont les suivantes.

pCarte = pCartes;

while (pCarte != NULL) {

afficheCarte(pCarte);

pCarte = pCarte->pSuiv;

}

Vous pouvez maintenant réfléchir à la fonction à laquelle vous allez faire appel pour ajouter un chainon à la liste chainée. L’ajout doit se faire en début de liste.

Quel est le prototype de la fonction ?

Cette fonction doit donc recevoir le pointeur de début de liste afin de le mettre à jour à chaque ajout, puisque ces derniers se font en début de liste. De plus, le pointeur vers le nouveau chainon est nécessaire afin de compléter le chainage en l’ajoutant à la liste.

Le prototype de la fonction est proposé ci-dessous. Faites à nouveau attention aux singuliers/pluriels des noms de variable.

Carte \* ajouteCarte(Carte \* pCartes, Carte \* pCarte);

Essayez de trouver le corps de cette fonction… et ajoutez l’appel au bon endroit dans la fonction principale.

La définition de la fonction d’ajout d’une carte doit ressembler à ce qui est proposé ci-dessous.

Carte \* ajouteCarte(Carte \* pCartes, Carte \* pCarte) {

// ajout en début de liste

pCarte->pSuiv = pCartes;

pCartes = pCarte;

return pCartes;

}

L’appel à cette fonction doit se faire à la fin de la boucle, mais seulement s’il y a de la place et s’il ne s’agit pas de la fin. L’instruction d’appel est la suivante.

pCartes = ajouteCarte(pCartes, pCarte);

Vous avez maintenant tout ce qu’il vous faut pour gérer une liste chainée.

# Passage par adresse/référence

Jusqu’à présent, il ne vous a jamais été demandé, en C, de renvoyer deux valeurs. Cependant, c’est un cas qui est couramment rencontré en programmation. Vous verrez différentes façons de procéder dans le cours de « Technologies web » du bloc 2. En attendant, vous allez devoir faire avec ce qui est à votre disposition. Et vu que vous êtes dans l’atelier sur les pointeurs, il y a des chances que la solution proposée à ce problème y soit liée…

L’exemple qui a été pris en début d’atelier pour introduire ce problème est celui-ci : lors de l’obtention d’une carte, il est demandé de renvoyer la carte ainsi qu’un booléen attestant de son existence. Cette façon de faire n’est pas envisageable en C sans utiliser un pointeur et sans mettre en œuvre le concept de passage par adresse ou par référence.

La fonction qui permet d’obtenir une carte, sans passer par un pointeur, est définie comme suit.

Carte carteObtenue(void) {

Carte carte;

char reponse;

printf("Nom (ENTER pour terminer) : ");

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);

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);

}

getchar();

}

return carte;

}

Vous devez donc déclarer une variable carte locale à la fonction, garnir les champs de celle-ci et ensuite la renvoyer. Il n’est donc pas possible de renvoyer en même temps le fait qu’elle contient bien des informations.

Dans la dernière version de la fonction, comme vous avez travaillé via un pointeur. Vous avez simplement passé ce pointeur, et donc l’adresse de l’espace en mémoire qui doit être garni par les données obtenues… Le code est ainsi devenu le suivant.

void carteObtenue(Carte \* pCarte) {

char reponse;

printf("Nom : ");

gets\_s(pCarte->nom, T\_NOM);

if (strcmp(pCarte->nom, "") != 0) {

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

scanf\_s("%d", &pCarte->nbCasesHauteur);

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

scanf\_s("%d", &pCarte->nbCasesLargeur);

getchar();

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

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

pCarte->officielle = reponse == 'O';

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

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

scanf\_s("%s", pCarte->motsCles[i], T\_MOT\_CLE);

}

getchar();

}

}

Vous avez donc passé la carte par adresse ou par référence ! Ainsi vous avez permis à la fonction de renvoyer une autre valeur à la fonction appelante.

Faites-en sorte que la fonction renvoie un booléen correspondant à carteExiste.

Le code correspondant devrait ressembler à ce qui est proposé ci-dessous.

bool carteObtenue(Carte \* pCarte) {

char reponse;

bool carteExiste;

printf("Nom : ");

gets\_s(pCarte->nom, T\_NOM);

carteExiste = strcmp(pCarte->nom, "") != 0;

if (carteExiste) {

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

scanf\_s("%d", &pCarte->nbCasesHauteur);

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

scanf\_s("%d", &pCarte->nbCasesLargeur);

getchar();

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

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

pCarte->officielle = reponse == 'O';

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

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

scanf\_s("%s", pCarte->motsCles[i], T\_MOT\_CLE);

}

getchar();

}

return carteExiste;

}

Dans la suite du cours, vous utiliserez cette technique dans les fonctions de recherche, d’ajout et autres, afin de permettre aux fonctions de renvoyer un code d’erreur éventuel.

1. Extrait de <https://fr.wikipedia.org/wiki/Notations_infix%C3%A9e,_pr%C3%A9fix%C3%A9e,_polonaise_et_postfix%C3%A9e> [↑](#footnote-ref-2)
2. Personnages des Royaumes oubliés <https://fr.wikipedia.org/wiki/Royaumes_oubli%C3%A9s> [↑](#footnote-ref-3)
3. Inspiré de <https://medium.com/@c.lacroixblum/tout-ce-que-vous-devez-savoir-ou-presque-sur-la-pile-et-le-tas-650334b61021> [↑](#footnote-ref-4)
4. Inspiré de <https://waytolearnx.com/2019/03/difference-entre-tas-et-pile.html> [↑](#footnote-ref-5)