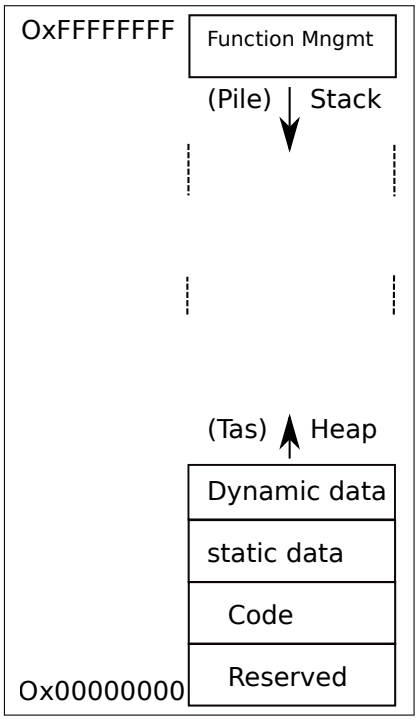
CRO TP1 :

Pointeurs
(1 séance –4h– sur machine), 6 octobre 2025

1 Introduction des pointeurs

Le langage C, inventé dans les années 70, permet de manipuler explicitement la mémoire de l'ordinateur, c'est à dire *chaque case mémoire individuellement et indépendamment de ce qu'elle contient*. Cette possibilité offre une grande souplesse au langage, particulièrement adapté pour les couches logicielles bas niveau, la programmation embarquée ou les systèmes d'exploitation.
Les langages plus "évolués" (Java, python, matlab, etc) cachent eux la gestion de la mémoire au programmeur (d'où les fameux *garbage collector* qui récupèrent la mémoire inutilisée de temps en temps). Mais bien sûr ces langages utilisent de manière sous-jacente la notion de pointeur que nous allons voir en C aujourd'hui (qui n'a pas vu le message "NullPointeurException" en Java ?).
Programmer en C nécessite de bien comprendre la notion de pointeur, ce qui nécessite d'avoir une représentation claire des données en mémoire.

Commençons par un rappel de l'organisation de la mémoire. Nous l'avons représentée partant de l'adresse 0 (soit 0x00000000 en 32 bits) à la plus haute adresse possible (soit 0XFFFFFFFF en 32 bits).
La séparation de la mémoire en plusieurs zones est purement logique, rappelons que nous sommes dans le cadre d'une machine de type Von Neuman, et que donc les données et le code sont dans la même mémoire, et que le code est lui-même une forme de donnée (par exemple, il existe des programmes auto-modifiant). La structuration montrée ci-contre est utilisé, par convention, par la grande majorité des processeurs.
On trouve au début de la mémoire une zone réservée qui sert à différents usages pour le système (adresse des périphériques dans le cas d'un système embarqué comme nous le verrons plus loin dans le cours). Puis une zone où se trouve le code, suivie d'une zone ou se trouvent les *données statiques* (i.e. données qui vont être utilisées sur toute la durée du programme et que l'on connaît au moment de la compilation). Ensuite une plage mémoire est utilisée pour les *données dynamiques* , c'est-à-dire les données dont on va découvrir le besoin au cours de l'exécution du programme : il s'agit du **tas** (*heap* en anglais). Enfin, du coté des grandes adresses, on trouve la pile que l'on a déjà rencontré en cours d'architecture.



Lvalue

Toute variable manipulée dans un programme est stockée quelque part en mémoire. Pour retrouver une variable, il faut connaître l'adresse de l'octet où elle est stockée. C'est le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire. Le langage C permet aussi de manipuler directement les cases mémoires à partir de leur adresse, pour cela il introduit la notion de *Lvalue*.
On appelle **Lvalue** (modifiable left value) tout objet pouvant être placé à gauche d'un opérateur d'affectation. Une Lvalue est caractérisée par :

— son adresse, c'est-à-dire l'adresse mémoire à partir de laquelle l'objet est stocké ;

— sa valeur, c'est-à-dire ce qui est stocké à cette adresse.

Par exemple, une variable entière est une Lvalue (ex : `i=1`), un tableau ou une chaîne de caractère ne sont pas des Lvalue (on accède uniquement aux *ases* du tableau).
Une Lvalue est rangée à une adresse, **on accède à cette adresse par l'opérateur adresse** `&`, l'adresse est en général rangée dans un entier (affiché en hexadécimal).
L'adresse d'une Lvalue n'est **pas** une Lvalue, c'est une constante (on ne peut pas la modifier).

QUESTION 1 ▶ Si je déclare un tableau d'entier : `int Tab[5]`, Pourquoi est ce que ce n'est pas une Lvalue, il est pourtant stocké en mémoire ?

Pointeur

Un pointeur est un objet (Lvalue) dont la valeur est égale à l'adresse d'un autre objet.
On déclare un pointeur par l'instruction :
`type *nom-du-pointeur;`
où `type` est le type de l'objet pointé. Cette déclaration déclare un identificateur, `nom-du-pointeur`, associé à un objet dont la valeur est l'adresse d'un autre objet de type `type`.
L'identificateur nom-du-pointeur est donc en quelque sorte un *identificateur d'adresse*. Comme pour n'importe quelle Lvalue, sa valeur est modifiable.

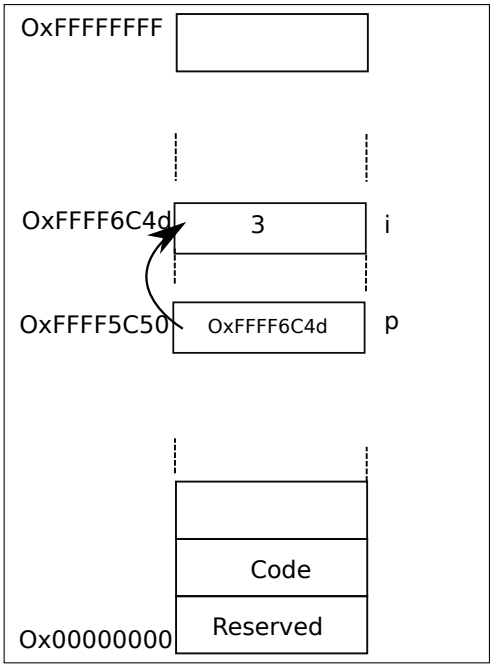
Par exemple Si on écrit :

```
int i; //variable i de type int
int *p; //variable p de type
        // pointeur vers int
i=3;
p=&i
```

`p` contient l'adresse de `i` (qui est par exemple la valeur `OxFFFF6C4d`).
Si on regarde a l'adresse `OxFFFF6C4d`, on trouve l'entier 3 rangé sur 4 octets
on y accède par opérateur unaire d'indirection `*` :

`*p` **désigne l'objet pointé par** `p`

c'est à dire la case mémoire à l'adresse `OxFFFF6C4d`. La valeur de `*p` est 3.



QUESTION 2 ▶ Télechargez l'archive "pointeurs". Commencez par étudier le `Makefile` fourni. Il fait usage de *règles implicites* qui permettent de gagner beaucoup de temps lorsque l'on a beaucoup de fichiers à compiler. Vérifiez que vous comprenez bien le `Makefile`.

QUESTION 3 ▶ Regardez le code du programme `pointeur1.c`, exécutez-le, vérifiez que le résultat correspond à celui que vous attentiez. Quelle est l'adresse à laquelle est rangée la variable i ? Utilisez

le format `%p` pour afficher la valeur d'un pointeur.

QUESTION 4 ▶ Pour les programmes `pointeur2` et `pointeur3`, quelles sont les valeurs de i et j en fin de programme ? Compléter le programme en affichant les différentes variables pour remplir les tableau suivants :

pointeur2

objet	adresse	valeur
i		
j		
p1		
*p1		
p2		
*p2		

pointeur3

objet	adresse	valeur
i		
j		
p1		
*p1		
p2		
*p2		

QUESTION 5 ▶ Est-ce-que ces valeurs sont les mêmes pour deux exécutions successives du programme ? Déduisez-en que la question précédente est vraiment inutile... pourquoi l'avons nous posée ?

QUESTION 6 ▶ Continuons avec les questions inutiles. Remarquez dans ces différentes exécutions de l'*écart* entre les adresses des différentes variables ne change pas (affichez la différence par exemple). Pouvez vous en déduire la seul chose qui change d'une exécution à l'autre ?

2 Arithmétique de pointeurs et parcours de tableau

Arithmétique de pointeurs

La valeur d'un pointeur étant un entier (`unsigned long int` précisément), on peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques. Les seules opérations arithmétiques valides sur les pointeurs sont :

- l'addition d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
- la soustraction d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
- la différence de deux pointeurs pointant tous deux vers des objets de même type. Le résultat est un entier.

D'autre part ces opérations ne produisent pas le même résultat suivant le type de pointeur. Si `i` est un entier et `p` est un pointeur sur un objet de type `type`, l'expression `p + i` désigne un pointeur sur un objet de type `type` dont la valeur est égale à la valeur de `p` incrémentée de `i * sizeof(type)`. Si `p` pointe vers une zone mémoire, `*(p+i)` est équivalent à `p[i]`.

QUESTION 7 ▶ Éditez le programme `produitScalaireSol.c` qui contient la solution du produit scalaire du dernier TP et modifier le parcours des tableaux A et B dans la fonction en utilisant des pointeurs que l'on incrémentera à chaque tour de boucle.
On utilise **énormément** en C, les pointeurs pour parcourir les tableaux.

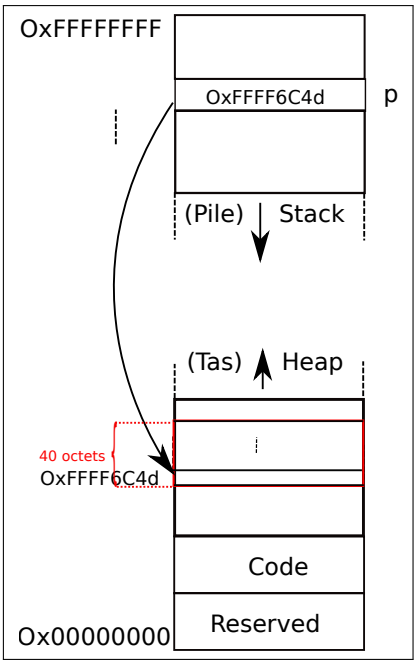
3 Allocation dynamique

Allocation dynamique

Le fait de réserver un espace mémoire pour stocker un objet qui n'était pas prévu à la compilation s'appelle *allocation dynamique*. Beaucoup de langages l'implémentent avec le mot-clé `new`.
En C, cela se fait par la fonction `malloc` de la librairie standard stdlib.h :
`char *malloc(int nombre-octets);`
Cette fonction retourne un pointeur de type `char *` pointant vers un objet de taille `nombre-octets` octets.
Pour initialiser des pointeurs vers des objets qui ne sont pas de type `char`, il faut convertir le type de la sortie de la fonction malloc à l'aide d'un *cast* (conversion de type). L'argument `nombre-octets` est souvent donné à l'aide de la fonction `sizeof()` qui renvoie le nombre d'octets utilisés pour stocker un objet.

Exemple : allocation de 10 entiers. le schéma de droite montre les différents emplacement des variables : la variables `p` est (probablement) une variable locale, donc allouée dans la pile, alors que l'objet pointé par `p` est alloué dans le tas.

```
#include <stdlib.h>
[...]
int *p;
p=(int *)malloc(10*sizeof(int));
if (p==0)
  {
    fprintf(stderr,"Erreur d'allocation\n");
    exit(-1);
  }
for (i=0;i<10;i++)
   *(p+i)=0; //initialisation
[...]
free(p);
[...]
```



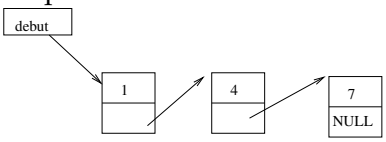
Pensez à *tester le résultat* de l'exécution de `malloc`, comme tous les *appels système*, les erreurs sont simplement remontées dans le code du résultat, si ces appels échouent il faut absolument le prendre en compte dans le programme sous réserve de comportement incompréhensible. Rappelez vous aussi que *allocation* ne signifie pas *initialisation à 0*.
Enfin, lorsque la zone allouée n'est plus utile, on libère la mémoire avec la fonction `free`.

QUESTION 8 ▶ Modifiez le `main()` du programme `produitScalaireSol.c` pour que les vecteurs `A` et `B` soient alloués dynamiquement (attention : ne modifiez pas la fonction `produit_scalaire`). libérez la mémoire à la fin du programme.

4 Pointeur et structure

Structures auto-référencées

On a souvent besoin en C de modèles de structure dont un des membres est un *pointeur vers une structure de même modèle*. Cette représentation permet en particulier de construire des listes chaînées. Un élément de la liste chaînée est une structure qui contient la valeur de l'élément et un pointeur sur l'élément suivant. Le dernier élément pointe sur la liste vide `NULL`.



C'est la façon dont on implémente le type `list` en C. Une liste est une collection d'élément de même type, mais contrairement aux tableaux, cette structure est dynamique, on peut rajouter ou enlever des élément au cours de l'exécution du programme. La contrepartie est qu'on ne peut pas accéder directement à chaque élément de la liste. En général on accède au premier élément, à partir duquel on peut *parcourir* la liste pour trouver l'élément recherché. Ce mécanisme est implémenté par les liste chaînées constituées de l'association de structure et de pointeurs.

QUESTION 9 ► On considère le type `ELEMLIST` suivant qui définit un élément d'une liste chaînée :

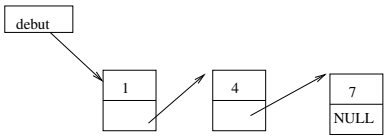
```
struct model_elem
{
  int val;
  struct model_elem *suivant;
};
typedef struct model_elem ELEMLIST;
typedef ELEMLIST *LISTE;
```

C'est l'occasion de revenir sur le constructeur `typedef` qui permet de *définir* des nouveau types. Etes vous sûr de bien comprendre la définition ci-dessus ? Demandez à l'enseignant le cas échéant.

Création de structure

Si on considère une structure de ce type pour gérer des listes :

```
struct model_elem
{
  int val;
  struct model_elem *suivant;
};
typedef struct model_elem ELEMLIST;
typedef ELEMLIST *LISTE;
```



La création d'un élément de liste va se faire avec la fonction malloc :

```
[...]
LIST new;
new = (LIST)malloc(sizeof(EMELIST);
[...]
```

l'accès aux champs de l'élément pointés par `new` se fait par : `(*new).elem` et `(*new).suivant` (souvenez vous que new est un pointeur).
Attention aux parenthèses : si vous écrivez `*new.suivant`, cela équivaut à `*(new.suivant)` (opérateur "." plus prioritaire que "*"), ce qui est faux, vu que `new` n'a pas de champs `suivant`. Donc les parenthèses sont indispensables.
Pour éviter le travail pénible d'écrire `(*new).val` à chaque accès, la syntaxe C propose l'opérateur "`->`" :
`(*new).val;` ⇔ `new -> val;`
`(*new).suivant;` ⇔ `new -> suivant;`

QUESTION 10 ► proposez une fonction :
`ELEMLIST *new_list(int val);`
qui crée un nouvelle élément de liste contenant la value `val` et retourne un pointeur vers cet élément. Cette fonction devra utiliser un `malloc` pour allouer le nouvel élément et utilisera une variable locale qui est bien *un pointeur* sur un `ELEMLIST`. On placera nos fonctions dans un fichier `list.c` (associé au fichier `list.h` correspondant). On utilisera aussi un fichier **different** `main.c` pour tester ces fonctions ainsi qu'un Makefile pour compiler.

QUESTION 11 ► On va maintenant décrire un type `LISTE` comme étant un pointeur vers un `ELEMLIST` :

Écrire le code C d'une fonction :
`LISTE ajouterListeTete (LISTE liste1, int val)`
lle insère un nouvel entier dans la liste **en tête de liste** et retourne la nouvelle liste.

QUESTION 12 ►
Écrire maintenant le code C d'une fonction **récursive** :
`LISTE ajouterListeEnFin (LISTE liste1, int val)`
qui insère un nouvel entier dans la liste **en fin de liste** (on doit donc parcourir la liste à partir de la tête de liste avant d'insérer).

QUESTION 13 ► On va proposer des fonctions pour manipulé des listes d'entier *triées* (i.e. plus petit élément en premier). On utilisera encore le type `LISTE` présenté ci-dessus.
Écrire le code C d'une fonction **récursive** qui insère un nouvel entier dans la liste triée. On pourra utiliser le prototype suivante :

`LISTE ajouterListeTrie (LISTE liste1, int val)`

5 Liste pour stocker des informations

Voici une structure `struct student` qui contient trois champs d'information ainsi qu'un champ `suivant`. On définit le type `ETUDIANT` qui est un raccourçi pour `struct student`. On définit aussi le type `PROMOTION` comme une liste d'étudiant.

```
struct student        //declaration de la structure
{
  char nom[100];
  char prénom[100];
  int numero_etudiant;
  struct student *suivant;
};
typdef struct student ETUDIANT;   // déclaration du type ETUDIANT
typdef ETUDIANT *PROMOTION;   // déclaration du type PROMOTION
```

QUESTION 14 ▶ écrire une fonction
`PROMOTION ajouterEtudiant(PROMOTION TC, char *nom, char *prenom, int numero)`
qui ajoute un nouvel étudiant à la promotion et renvoie la promotion modifiée. On fera en sorte d'insérer le nouvel étudiant de manière à avoir une liste triée par numéro d'étudiants.

QUESTION 15 ▶ écrire une fonction `void affichePromotion(PROMOTIION)` qui affiche une promotion, faite un Makefile, une fonction `main` et tester votre fonction.