Haute École Léonard de Vinci
SCIENCES ET TECHNIQUES

I2011 Langage C : bases

Chap. 4 Pointeurs

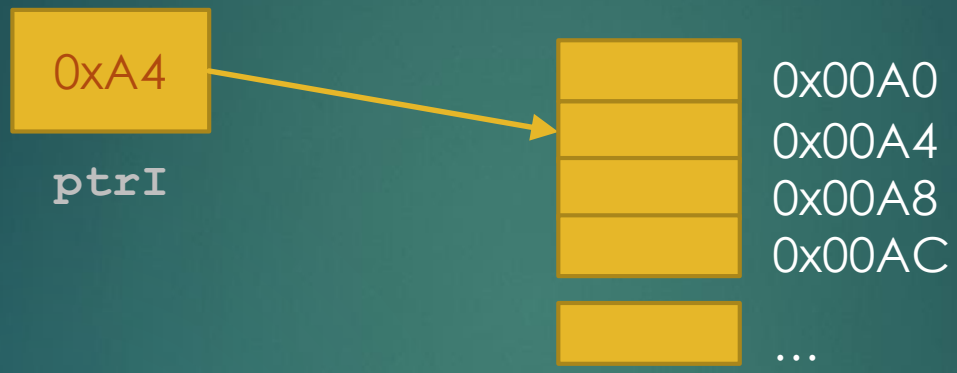
Anthony Legrand
Jérôme Plumat

Les tableaux statiques

- **Avantage** : simplicité d'utilisation
- **Inconvénient** : ils sont statiques !
  - → Leur taille doit être connue dès la compilation
  - → Le programmeur est obligé de prévoir la taille maximale du tableau dont il aura besoin.
    - ⇒ limitation du domaine de validité du programme (borne maximale fixée)
    - ⇒ gaspillage de mémoire (espace mémoire alloué mais inutilisé si taille réelle < taille maximale)

Les pointeurs

- Les pointeurs contiennent des adresses mémoires



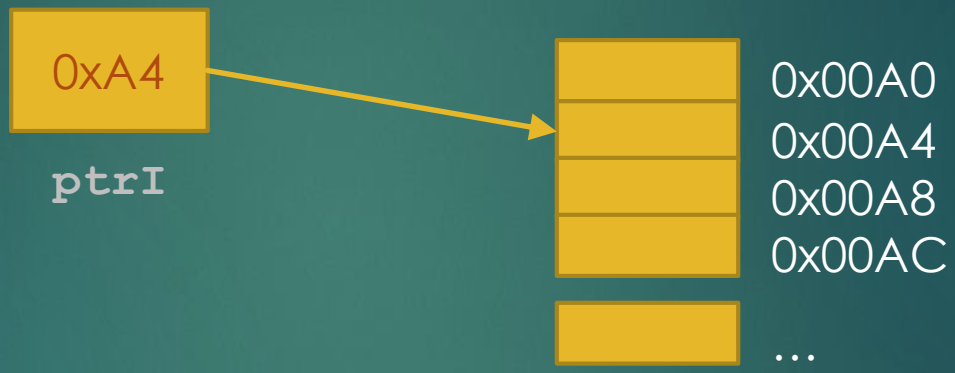
- Les pointeurs sont typés

```
int* ptrI;   // pointeur vers un entier
double* ptrI; // pointeur vers un double
Noeud* ptrN;    // pointeur vers un Noeud
```

- Un peu comme une référence Java mais…

Ça sert à quoi?

- Accéder une adresse précise (driver, mémoire vidéo…)



- Allouer dynamiquement de la mémoire
- Créer des structures de données chaînées
- Passer des paramètres par référence

Initialisation (1)

- Pas initialisé par défaut ⇒ Danger!
- Prendre l'adresse d'une variable

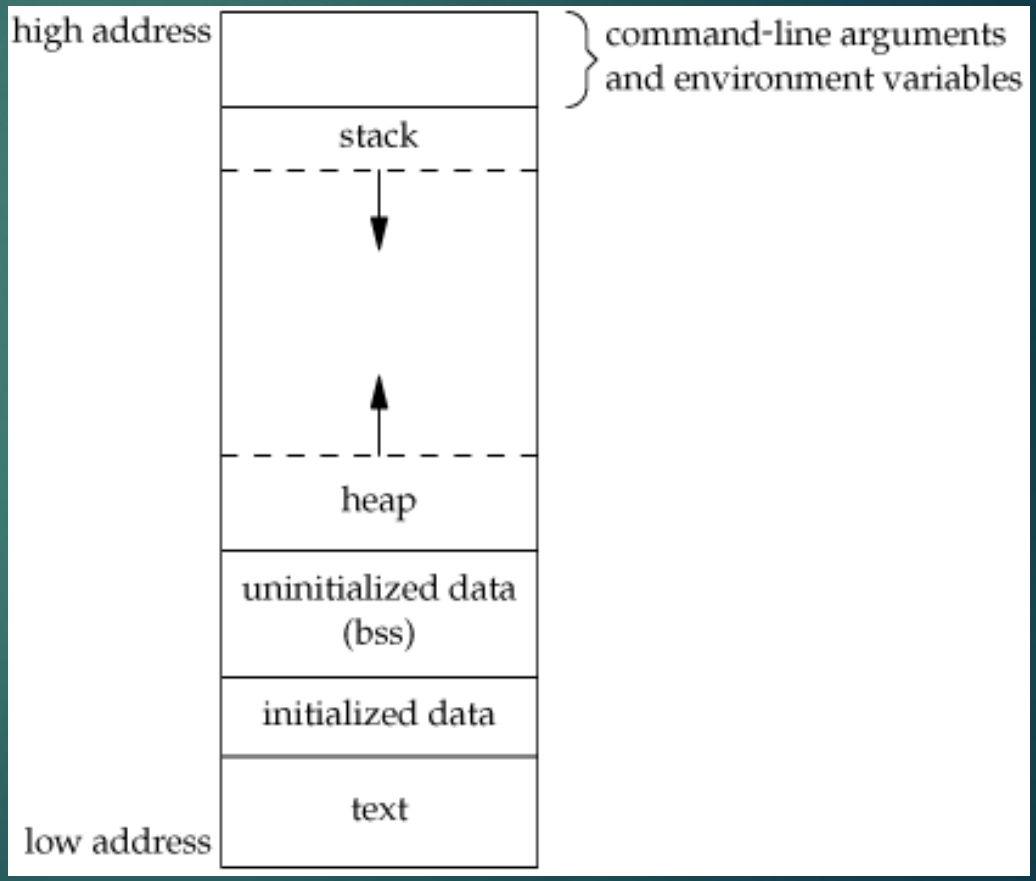
```
int a = 38;

int* ptrI = &a;  // ptrI reçoit l'adresse de a
```

- Vous avez déjà utilisé l'opérateur '&'.
  Où ça?

Initialisation (2)


```
int truc() {
    int a = 38;
    int* ptrI = &a;
    ...
}
```

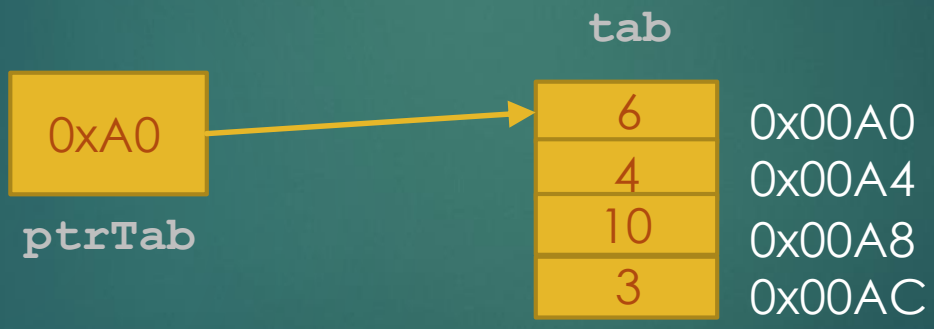
- *a* est sur le stack



Initialisation (3)

- Prendre l'adresse d'un tableau

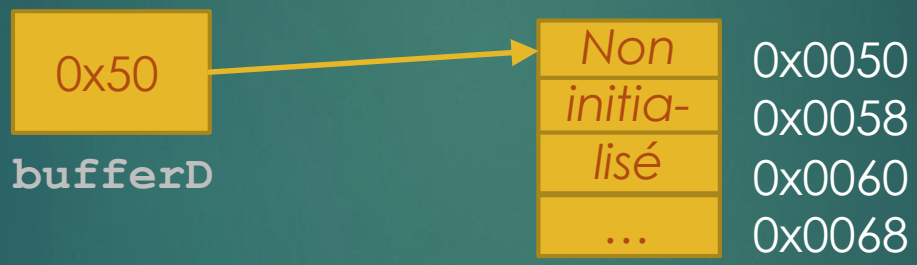
```
int tab[4] = {6, 4, 10, 3};

int* ptrTab = tab;  // adresse de l'élément 0
```



Mémoire dynamique (1)

- Allouer une zone mémoire de taille quelconque

```
double* bufferD =
    (double*) malloc(4*sizeof(double));
```



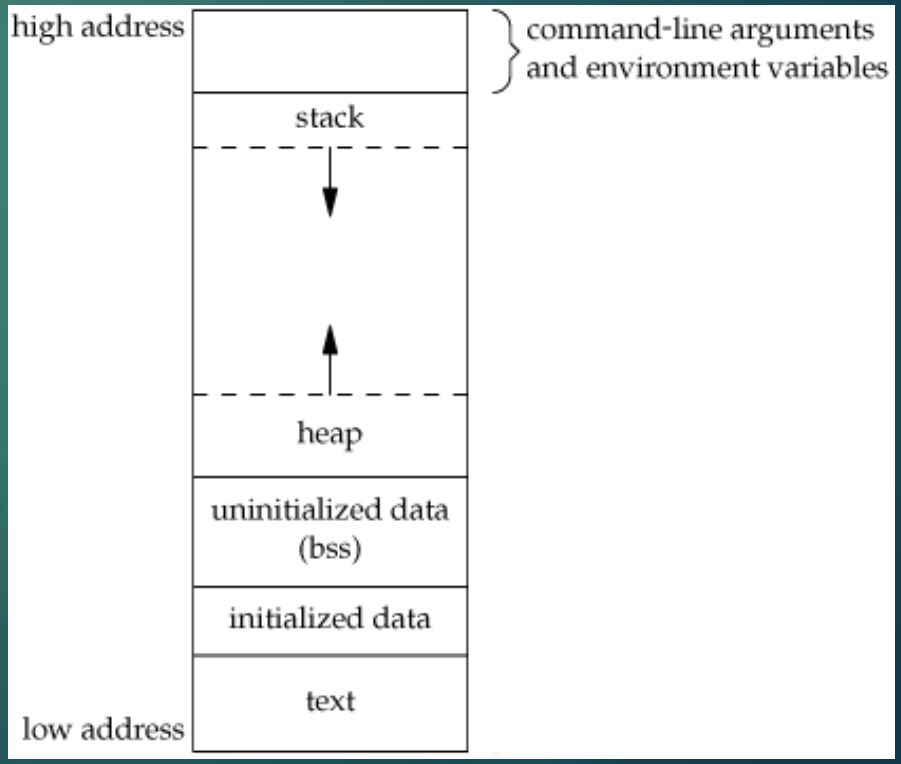
- Remarquez les +8 dans les adresses. Pourquoi?
- Cf. **`man malloc`**

Mémoire dynamique (2)

- Allouer une zone mémoire de taille quelconque

```
double* bufferD =
    (double*) malloc(4*sizeof(double));
```

- *malloc* travaille sur le heap



Mémoire dynamique (3)

- Si à court de mémoire

```
double* bufferD =
    (double*) malloc(4*sizeof(double));

if (bufferD == NULL) ...
```

Mémoire dynamique (4)

- Pas de garbage collector
- Libérez la mémoire!

```
double* bufferD =
     (double*)malloc(4*sizeof(double));

...

free(bufferD);
```

Déréférencement (1)

- Accéder au contenu de l'élément pointé via l'opérateur *

```
int a = 38;

int* ptrI = &a;  // reçoit l'adresse de a

int b = *ptrI;  // déréférencement: prend le
                // contenu de l'élément pointé
```

Déréférencement (2)

- Modifier le contenu de l'élément pointé via l'opérateur *

```
int a = 38;

int* ptrI = &a;  // reçoit l'adresse de a

*ptrI = 27;  // modifie le contenu de a!
```

Déréférencement (3)

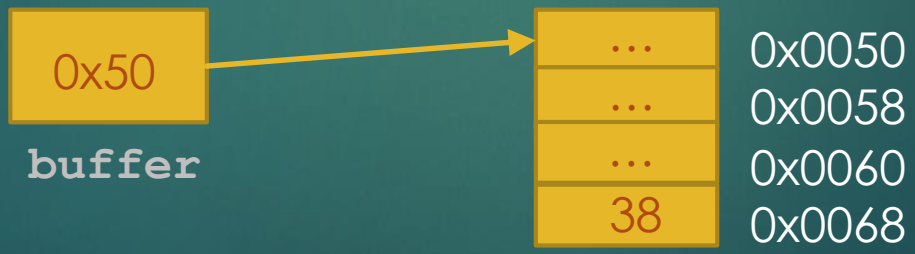
- Accéder au contenu de l'élément pointé via un indice

```
double* buffer =
     (double*) malloc(4*sizeof(double));
...
double db = buffer[1]; // déréférencement: accès
                       // au contenu de l'élément 1
```

Déréférencement (4)

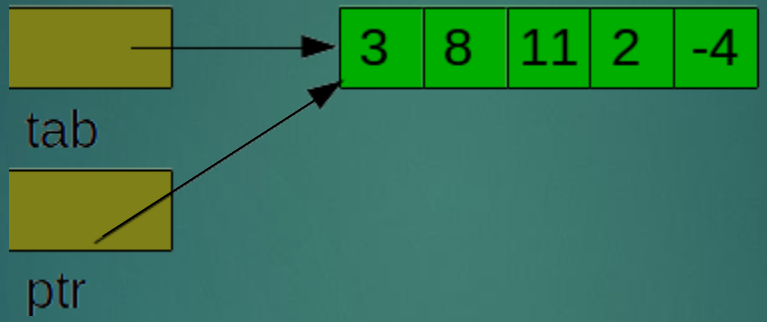
- Modifier le contenu de l'élément pointé via un indice

```
double* buffer =
    (double*) malloc(4*sizeof(double));
...
buffer[3] = 38;  // modifie l'élément 3
```

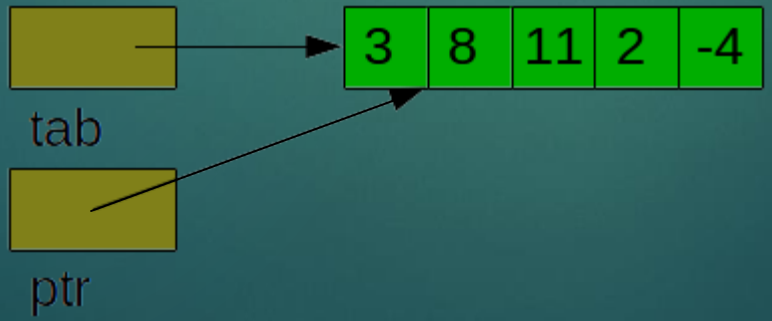


Arithmétique des pointeurs (1)


```
int tab[5] = {3, 8, 11, 2, -4};

int *ptr = tab;
```



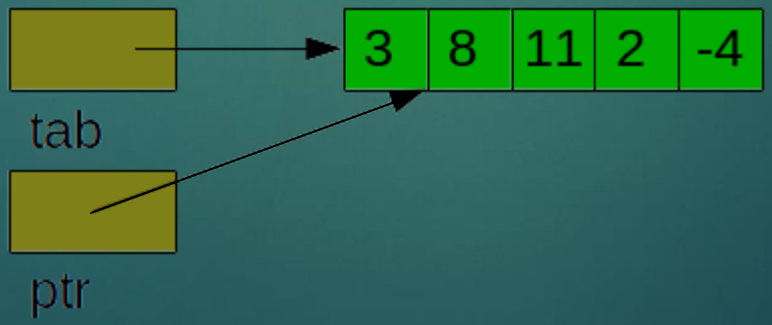

```
ptr++;  // <==> adresse ptr + sizeof(int) bytes
```



Arithmétique des pointeurs (2)

- Parcours (performant) d'un tableau avec un baladeur de type pointeur

```
int tab[5] = {3, 8, 11, 2, -4};
for (int *ptr=tab; ptr-tab < 5; ptr++) {
    // traitement de l'entier *ptr
}
```



Equivalence des notations

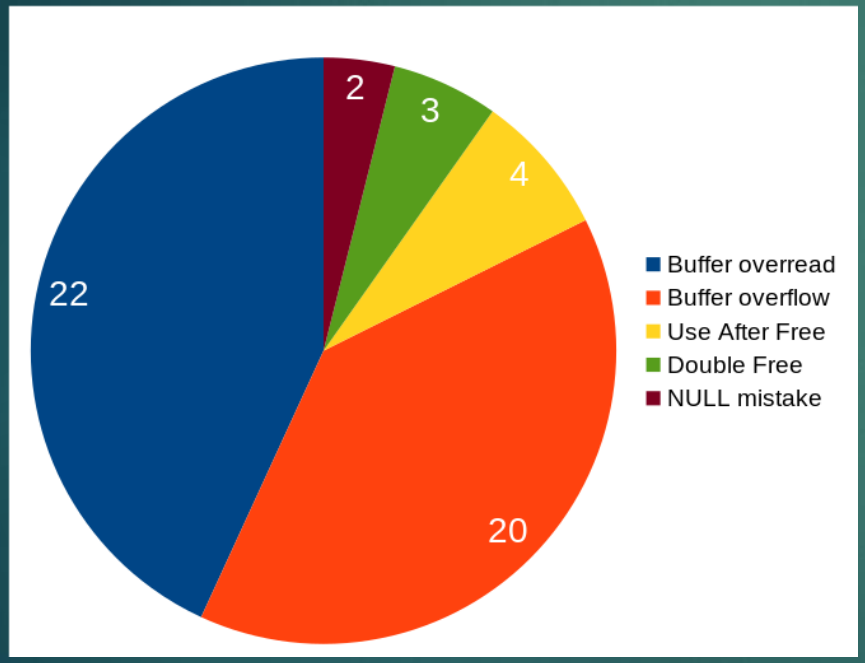
- L'arithmétique des pointeurs permet de montrer l'équivalence de notation entre les déréférencements par indice et par l'opérateur *

```
double* buffer =
     (double*) malloc(4*sizeof(double));
```

```
double* ptr = buffer      ⇔   double* ptr = &buffer[0]
double db = *buffer       ⇔   double db = buffer[0]
double* ptr = buffer+i    ⇔   double* ptr = &buffer[i]
double db = *(buffer+i)   ⇔   double db = buffer[i]
```

Erreurs communes en C

- La plupart des erreurs de programmation en C sont liées à la gestion de la mémoire



- **Buffer overread** – reading outside the buffer size/boundary.
- **Buffer overflow** – code wrote more data into a buffer than it was allocated to hold.
- **Use after free** – code used a memory area that had already been freed.
- **Double free –** freeing a memory pointer that had already been freed.
- **NULL mistakes** – NULL pointer dereference.

Source: https://daniel.haxx.se/blog/2021/03/09/half-of-curls-vulnerabilities-are-c-mistakes/

Débuggage

- Les fautes classiques liées à la manipulation des pointeurs provoquent des erreurs d'exécution
- Généralement l'erreur « **segmentation fault** »

  = tentative d'accès à un emplacement mémoire qui n'est pas alloué au programme
- Difficile d'identifier la cause d'une ***segfault***

  ⇒ Utilisation d'un débugger :
  - **gdb** (*GNU Debugger*) pour UNIX / Linux
  - **lldb** (*Low Level Debugger*) pour macOS


Wow, a different error message... Finally some progress!
ENERGY
ENERGY
AGENT-X COMICS
WWW.AGENT-X.COM.AU