# Programmation « orientée système »
# LANGAGE C – POINTEURS (5/5)

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

# Objectifs du cours d'aujourd'hui

- Arithmétique des pointeurs
- Complément (et mise en garde) sur `sizeof`
- Retour sur les « *flexible array member* »
- Débordement de tampons

# Pointeurs et tableaux

On a vu dans les cours et exercices précédents qu'on pouvait par exemple allouer un pointeur sur une zone de 3 `double` :

```
double* ptr;
ptr = calloc(3, sizeof(double));
```

Pourtant `ptr` en tant que tel ne pointe que sur **un** `double` !
(regardez son type : `double*`)

Que vaut `*ptr` ?
☞ la valeur du **premier** `double` stocké dans cette zone.

Comment accéder aux 2 autres ?

☞ avec une syntaxe identique aux tableaux : `ptr[1]` et `ptr[2]`


Jean-Cédric Chappelier

# Pointeurs et tableaux (2)

En C, un tableau est en fait très similaire à un **pointeur** (on l'a déjà vu lors du passage d'argument à une fonction) **constant** sur une zone allouée **statiquement** (lors de la déclaration du tableau).

Ainsi `int[]` est pratiquement identique à « `int* const` »
et `*p` est strictement équivalent à `p[0]`

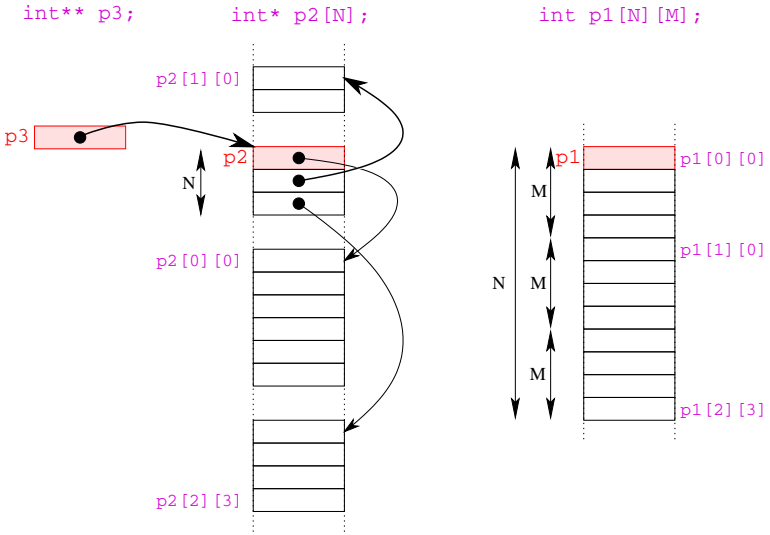
**MAIS**
`int**` ou `int*[]` sont **très différents** de `int[][]` (qui d'ailleurs n'existe pas en tant que tel ! `int[][M]`, oui)

`int**` :

- n'est pas continu en mémoire ;
- n'est pas alloué au départ ;
- les lignes n'ont pas forcément le même nombre d'éléments.


Jean-Cédric Chappelier

# Pointeurs et tableaux (2)



# Pointeurs et tableaux (3)

Qu'affiche la portion de code suivant ?

```
#define N 2
#define M 14
// ...
  double   p1[N][M];
  double*  p2[N];
  double** p3;

  p3 = calloc(N, sizeof(double*));

  for (size_t i = 0; i < N; ++i) {
    p2[i] = calloc(M, sizeof(double));
    p3[i] = calloc(M, sizeof(double));
  }

  printf("&(p1[1][2]) - p1 = %u doubles\n",
      (unsigned int) &(p1[1][2]) - (unsigned int) p1) / sizeof(double));

  printf("&(p2[1][2]) - p2 = %u doubles\n",
     ((unsigned int) &(p2[1][2]) - (unsigned int) p2) / sizeof(double));

  printf("&(p3[1][2]) - p3 = %u doubles\n",
     ((unsigned int) &(p3[1][2]) - (unsigned int) p3) / sizeof(double));

// ... // en particulier les free() !!
```


Jean-Cédric Chappelier

# Pointeurs et tableaux (4)

Réponse (sur ma machine à un moment donné) :

```
&(p1[1][2]) - p1 = 16 doubles
&(p2[1][2]) - p2 = 151032928 doubles
&(p3[1][2]) - p3 = 49 doubles
```

# Arithmétique des pointeurs

On peut facilement déplacer un pointeur en mémoire à l'aide des opérateurs + et - (et bien sûr leurs cousins ++, +=, etc.)

« Ajouter » 1 à un pointeur revient à le déplacer « en avant » dans la mémoire, d'un emplacement égal à *une* place mémoire **de la taille de l'objet pointé**.

Exemples (très pratiques) :

```
int tab[N];
...
const int* const end = tab + N;
for (int* p = tab; p < end; ++p) { ... *p ... }
```

```
char* s;  char* p;  char lu;
...
p = s;
while (lu = *p++) { ... lu ... }
```


Jean-Cédric Chappelier

# Explication de l'exemple précédent

```
char* s;  char* p;  char lu;
...
p = s;
while (lu = *p++) { ... lu ... }
```

- Que veut dire `*p++` ?
  Est-ce `*(p++)` ou `(*p)++` ?
- Que fait l'autre `(*p)++` ?
- Est-ce que `*p++` est pareil que `*++p` ?
- Pourquoi une variable `lu` plutôt que `*p` directement dans le corps de la boucle ?
  Par exemple : `while(*p++) { ... *p ... }`
- Erreur dans la condition d'arrêt de la boucle ? (`==` au lieu de `=`) ?

# Attention !

**Attention !** Le résultat de « `p = p + 1` » **dépend du type de** `p` !
(Et c'est souvent là une source d'erreur !)

Le plus simple (avant ce qui va suivre) est de comprendre « `p = p + 1` » comme « *passe à l'objet (pointé) suivant* ».

En clair :

> Toutes les opérations avec les pointeurs tiennent compte automatiquement du type et de la grandeur des objets pointés.

Il faut éviter de penser aux vraies valeurs (adresses, en tant que nombres entiers), mais si l'on y tient vraiment, on aura donc :

```
(int) (p+1) == (int) p + sizeof(Type)
```

pour `p` un pointeur de type « `Type*` »

**Note :** on ne peut donc pas faire d'arithmétique des pointeurs sur des `void*` !


Jean-Cédric Chappelier

# Soustraction de pointeurs

On a vu qu'il existait les opérateurs `ptr + int` et `ptr - int` (chacun de type `ptr`).

Il existe aussi `ptr - ptr`

« `p2 - p1` » retourne le nombre d'objets stockés entre `p1` et `p2` (de même type).

**Attention !** Le type de cet opérateur (soustraction de pointeurs) est `ptrdiff_t` (défini dans `stddef`).
**CE N'EST PAS** `int` **!** (ceci est une grave erreur !)

```
ptrdiff_t dp = p2 - p1;
```

# Pointeurs et tableaux (synthèse)

(Pour `int* p;` `int t[N];` et `int i;`)

`t[i]` est en fait exactement `*(t+i)`

À noter que c'est symétrique... (...et on peut en effet écrire `3[t]` ! !)

`t` est en fait exactement `&t[0]` (et est un `int* const`)

`int t2[N][M]` n'a **rien** à voir avec un `int**` (et est plus proche d'un `int* const`)

`void f(int t[N])` (ou `void f(int t[])`) sont en fait exactement `void f(int* t)` :

- attention à la sémantique de `t` (et en particulier à sa taille) dans le corps de `f` ;
- nécessité absolue de toujours passer la taille de `t` comme argument supplémentaire.


Jean-Cédric Chappelier

# Complément (et mise en garde) sur `sizeof`

L'opérateur `sizeof` accepte comme argument soit un type, soit une expression C (laquelle *n'est pas* évaluée)
(et retourne la taille mémoire nécessaire à stocker le type de l'expression en question)

Exemples :

```
int i;
int tab[N];
... sizeof(double) ...
... sizeof(i) ...                    // = sizeof(int)
... sizeof(tab)/sizeof(tab[0]) ... // donne N, mais ATTENTION !!
```

Mais il faut faire attention à ne pas mal l'employer :

```
int tab[1000];
int* t = tab;
... sizeof(t) ... /* combien ca vaut ? */
```

# Complément (et mise en garde) sur `sizeof`

**Attention !** PIRE !

```
#define N 1000

void f(int t[N]) {
  ... sizeof(t)/sizeof(int) ... /* combien ca vaut ? */
}
```

☞ Je répète qu'un tableau passé en argument de fonction n'a **AUCUNE** connaissance de sa taille ! !

---

Autre (**mauvais**) exemple, plus subtil : où est le bug ? :

```
int tab[1000];
...
const int* const end = tab + sizeof(tab);
for (int* t = tab; t < end; ++t) {
    utiliser(*t, ...);
}
```

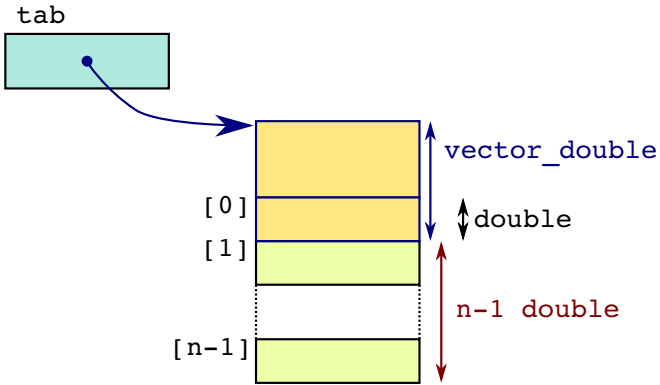
# Rappel : flexible array member

```c
struct vector_double {
  size_t size; // nombre d'éléments
  double data[1];
};
```

Ceci permet d'avoir des tableaux dynamiques en C, via l'allocation dynamique :

```c
const size_t N_MAX = (SIZE_MAX - sizeof(struct vector_double)) / sizeof(double) + 1;
if (nb <= N_MAX) {
  struct vector_double* tab = malloc(sizeof(struct vector_double)
                                     + (nb-1)*sizeof(double) );
  if (tab != NULL) {
    tab->size = nb;
  }
}
```

# Flexible array member




Utilisation :

```
tab->data[i]
```

# Autre exemple : listes chaînées

Rappel :

Une **liste chaînée** est un ensemble homogène d'éléments *successifs* (pas d'accès direct)

Interface :

- accès au premier élément (sélecteur)
- accès à l'élément suivant d'un élément (sélecteur)
- modifier l'élément courant (modificateur)
- insérer/supprimer un élément après(/avant) l'élément courant (modificateur)
- tester si la liste est vide (sélecteur)
- parcourir la liste (itérateur)

☞ faire **absolument** l'exercice 3 de la série 8 (semaine passée) !


Jean-Cédric Chappelier

# Débordement de tampons

Qu'est-ce qui ne va pas dans ce code :

```c
#include <stdio.h>

int main(void) {
  char nom[44];
  printf("Quel est votre prénom ?\n");
  gets(nom); // Note : gets est deprecated in C99, removed from C11
  printf("Bonjour %s\n", nom);
  return 0;
}
```

# Débordement de tampons

Exemple d'exécution :

```
monShell>./hello
Quel est votre nom ?
Jean-Pierre André Charles-Édouard Émile-Gustave Pierre-Adrien
Bonjour Jean-Pierre André Charles-Édouard Émile-Gustave Pierr...
Segmentation fault
```

Notez que ça plante **après** le `printf` (c.-à-d. sur le `return`, en fait).


Jean-Cédric Chappelier

# Débordement de tampons

On pourrait très bien se dire :

> « *Oui bon, ce n'est pas un programme très robuste, il risque parfois d'écrire un peu plus loin...*
> *Et alors ? Il s'exécute quand même ! Peu importe s'il finit par un `SEGV` ou pas, non ?* »

Le problème est que ce genre d'erreur (débordement de tampon) est une des **principales sources d'insécurité des systèmes informatiques** !

Comment est-ce possible ?
Comment peut-on exploiter une faute aussi banale pour « casser » un système informatique ?

Le but n'est pas ici de vous donner la réponse complète (qui nécessite d'autres connaissances), mais de vous sensibiliser au problème afin que vous **prêtiez une attention très particulière à l'écriture de vos codes**, surtout lorsqu'ils manipulent des pointeurs (soit directement, soit sous forme de chaînes de caractères ou de tableaux).


Jean-Cédric Chappelier

# Débordement de tampons

## ① Quel est le problème ?

☞ lors de la saisie d'un prénom par l'utilisateur, le « tampon » (c.-à-d. ici : « zone mémoire continue ») `nom` peut « déborder » : dès que l'utilisateur saisit un prénom de plus de **43** caractères.

Le problème est que ce débordement se fait sur des zones mémoires utilisées par ailleurs par le programme (« la pile »).

et donc, en fonction de comment se produit ce débordement, cela peut même sérieusement perturber le déroulement du programme...

...même au point de pouvoir lui faire faire toutes sortes de choses indiquées par l'utilisateur (et non voulues par le programmeur !), comme par exemple prendre la main sur la machine.


EPFL

# Débordement de tampons

Exemple (pour un système linux 2.4 sur Intel x86) :

```
monShell>./hello
Quel est votre prénom ?
Jean-Cédric
Bonjour Jean-Cédric
monShell>./hello
Quel est votre prénom ?
ë^_^v^H1ÀF^GF^L°^KóN^HV^LÍ1Û0@ÍèÜÿÿÿ/bin/shwtever 15 chars@ôÿ¿
Bonjour ë^1ÀFF
                °
                 óV
                   Í1Û0@ÍèÜÿÿÿ/bin/shwtever 15 chars@ôÿ¿

sh-2.05b$
```

(Le « prénom » en question contient 64 caractères bien choisis :

```
eb 1f 5e 89 76 08 31 c0 88 46 07 89 46 0c b0 0b 89 f3 8d 4e 08 8d
56 0c cd 80 31 db 89 d8 40 cd 80 e8 dc ff ff ff 2f 62 69 6e 2f 73
68 77 74 65 76 65 72 20 31 35 20 63 68 61 72 73 40 f4 ff bf
```

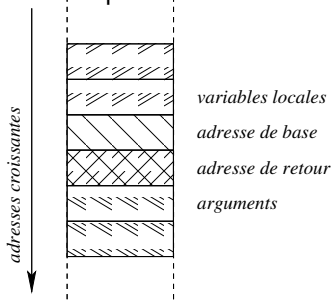
lequel nous **ouvre un « shell »** (avec les mêmes droits que le programme `./hello` !))

# Débordement de tampons

② Explication du problème (pour un système linux 2.4 sur Intel x86) :

La plupart des architectures d'ordinateurs modernes utilisent la même zone mémoire (la « pile ») pour stocker (entre autres) les arguments des fonctions, la valeur de retour, l'adresse de retour et les variables locales.

Pour une architecture linux 2.4 sur un processeur Intel x86, on a le schéma suivant :




Et si donc une variable locale vient à « déborder », elle peut **écraser l'adresse de retour** !

# Débordement de tampons

Exercice : qu'affiche le programme suivant (sur Intel x86) ?

```c
#include <stdio.h>

void f(void) {
  int tab[] = { 1, 2 };
  tab[4] += 7;
}

int main(void) {
  int x = 33;
  f();
  x = 1;
  printf("x=%d\n", x);
  return 0;
}
```

Indication : l'instruction `x=1;` prend « 7 places » en mémoire une fois compilée en langage machine :

```
c7 45 fc 01 00 00 00 movl $0x1,-0x4(%rbp)
```

Note : sur certaines machines, peut nécessister de supprimer quelques protections telles que :

- supprimer les protection du compilateur : `-fno-stack-protector`
  (voir aussi `-fstack-protector-all`, `-fstack-protector-strong` et `-Wstack-protector`)
- supprimer l'*address space layout randomization* (ASLR) :

```
echo 0 > /proc/sys/kernel/randomize_va_space
```


Jean-Cédric Chappelier

# Ce que j'ai appris aujourd'hui

- des compléments au sujet des pointeurs :
  arithmétique des pointeurs, `sizeof`, prédéclaration ;
- à faire attention à mon code pour ne pas permettre de « buffer overflow ».


Jean-Cédric Chappelier
EPFL