INFO-F105 – Langages de programmation 1
Pointeurs & Tableaux

1 Adressage

Chaque donnée manipulée par un programme est stockée à un endroit dans la mémoire, accessible à partir d'une *adresse* dont la taille dépend de l'architecture du processeur.

1.1 Stack

Le *stack* (= *pile*) est une zone mémoire où les valeurs sont *empilées*.

C'est là que sont enregistrées les variables locales définies dans les fonctions.

```
void foo() {
  int a = 5;
  int b = 2;
  int c = a + b;
}
```

Les valeurs de `a`, `b` et `c` dans la fonction `foo` sont allouées une par une sur le stack.

Le *Stack Pointer* (SP) est un registre qui indique l'emplacement de la dernière valeur ajoutée sur le stack et est ramené à son état initial une fois la fonction terminée. Les valeurs de `a`, `b` et `c` sont toujours sur le stack mais seront écrasées la prochaine fois qu'on voudra l'utiliser.

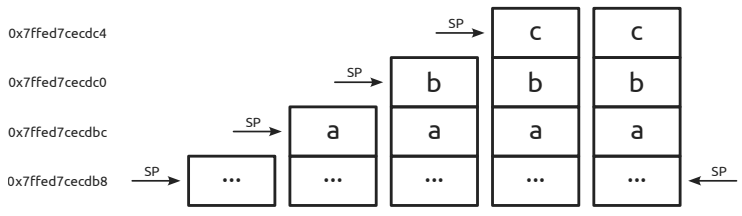


Figure 1: Allocation de `a`, `b` et `c` sur le stack

Le stack peut se voir comme un tableau (≈ liste) où chaque "case" a une *adresse* (≈ indice).

Les adresses sont de très grands nombres qu'on représente en hexadécimal.

1.2 Heap

Le *heap* (= *tas*) est une zone mémoire où les valeurs sont stockées de façon non contiguë.

Il est principalement utile lorsqu'on souhaite qu'une donnée allouée dans une fonction reste accessible une fois la fonction terminée. Ce n'est pas possible avec le stack car le *stack pointer* est réinitialisé en quittant la fonction.

```
void foo(int** a, int** b) {
  *a = new int(5);
  *b = new int(2);
}
```

La syntaxe de ce code est expliquée plus bas. Ce qui nous intéresse pour le moment est de voir de quelle façon les données sont enregistrées dans la mémoire.

Figure 2: Allocation des entiers `5` et `2` dans le heap

Le heap étant désordonné, il est nécessaire de connaitre les adresses pour accéder aux données.

1.3 Tailles de types

Il est important de connaitre la taille de chaque donnée pour manipuler leurs adresses.

Cette taille dépend du type de la variable et peut être connue grâce à l'opérateur `sizeof`.

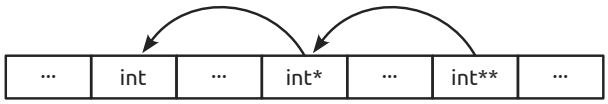
```
std::cout << sizeof(int) << std::endl;
```

Par exemple, le type `int` fait 4 bytes et couvre donc 4 adresses de 1 byte dans la mémoire.

2 Pointeurs

Les *pointeurs* sont des types représentant des adresses vers d'autres types.

```
int* p;
int** p2;
```



→ `p` est un pointeur vers un `int` et contient une adresse à laquelle se trouve un `int`.

→ `p2` est un pointeur vers un `int*` et contient une adresse à laquelle se trouve un `int*`.

2.1 Opérateurs

Il existe deux opérateurs pour manipuler des adresses.

Opération	Description
`&var`	Récupère l'adresse de la variable `var`
`*ptr`	Récupère la valeur à l'adresse `ptr`

```
int a = 5;
int* p = &a;
int b = *p;  // b == a
```

Il existe également deux opérateurs pour manipuler le heap.

Opération	Description
`new`	Allocation
`delete`	Désallocation

```
int* ptr = new int(5);
delete ptr;
```

Attention : il est essentiel de `delete` toutes les adresses allouées avec `new`, sinon la mémoire ne sera pas libérée et cela provoquera un *memory leak*.

2.2 Conversions

Il peut arriver de vouloir convertir un pointeur vers un type en un pointeur vers un aute type.

```
short a;
short* ptr1 = &a;
char* ptr2 = (char*) ptr1;  // Conversion
```

Il faut cependant faire attention car cette conversion ne change rien aux données en mémoire. Dans cet exemple, `*ptr2` ne couvrira que la moitié des bits de `*ptr1` car un `char` est deux fois plus petit qu'un `short`.

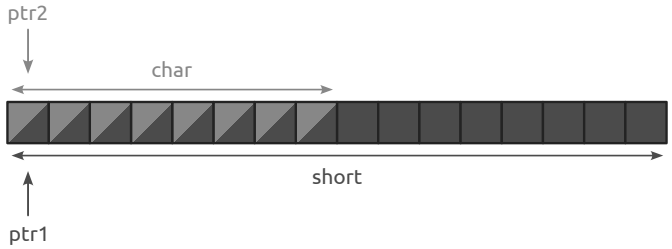


Figure 3: Conversion `short*` → `char*`

2.3 Constantes

Il faut distinguer pointeur vers une constante et pointeur constant.

```
const int* ptr1;
int* const ptr2;
```

→ `ptr1` est un pointeur vers un `const int` : la valeur `*ptr1` ne peut pas changer.

→ `ptr2` est un pointeur constant vers un `int` : la valeur `ptr2` ne peut pas changer.

2.4 Arithmétique

Les adresses étant des nombres, il est possible de faire des opérations avec.

```
int* ptr1;
int* ptr2 = ptr1 + 1;
int* ptr3 = ptr1 & 0x111111111100;
```

Une subtilité importante à prendre en compte est que le `+1` à la ligne 2 n'ajoute pas 1 à l'adresse, mais l'augmente de `sizeof(int)` pour arriver à l'élément `int` suivant.

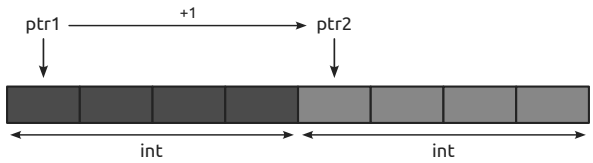


Figure 4: Incrémentation d'un pointeur

Par convention, on utilise le type `ptrdiff_t` défini dans `<cstddef>` pour les opérations avec des pointeurs, car sa taille correspond à celle des adresses sur l'architecture visée.

```
for (ptrdiff_t i = 0; i < 100; ++i) {
  *(ptr + i) = i;
}
```

3 Tableaux

Un *tableau* (*array*) est comparable à une liste Python avec des différences importantes :

- Tous les éléments sont de même type.
- La longueur du tableau est fixe.

```
int arr[5];
```

→ `arr` est un tableau contenant 5 éléments de type `int`.

```
int arr[] = { 48, 7, 51, 38, 12 };
```

→ Il n'est pas nécessaire de préciser la taille de `arr` lorsqu'il y a une assignation explicite.

3.1 Représentation en mémoire

Un tableau est alloué sur le stack et ses valeurs sont placées les unes après les autres.

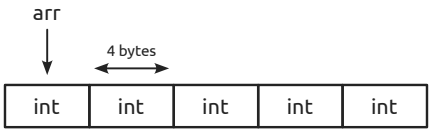


Figure 5: Tableau de `int` (4 bytes)

Un tableau est en fait un pointeur avec une syntaxe plus intuitive.

```
int arr[5];
int* ptr = arr;  // arr[0] == *(ptr + 0)
                 // arr[1] == *(ptr + 1)
                 // ...
```

La syntaxe `arr[n]` ne fait en réalité qu'une addition de `arr` et `n`, et est équivalente à `n[arr]`.

3.2 Itération

```
int arr[10];
for (size_t i = 0; i < 10; ++i) {
  std::cout << arr[i] << std::endl;
}
```

Le type `size_t` est défini dans `<cstddef>` et est utilisé pour les indices de tableaux.

Le type `ssize_t` est défini dans `<unistd.h>` et peut aussi contenir `-1` (indice inexistant).

3.3 Paramètre

Lorsqu'on passe un tableau comme paramètre de fonction, il devient un simple pointeur vers le premier élément du tableau.

```
void foo(int* parr, size_t length) {
  for (size_t i = 0; i < length; ++i) {
    parr[i] = i;
  }
}

int main() {
  int arr[10];
  foo(arr, 10);
}
```

On passe généralement aussi la taille du tableau dans un autre paramètre.

Remarquez comme il est possible d'utiliser l'opérateur `[]` sur le pointeur `parr`.

3.4 Dimensions

Un tableau à 2 dimensions (matrice) est en fait un tableau à une seule dimension où les lignes du tableau 2D ont été mises les unes à la suite des autres.

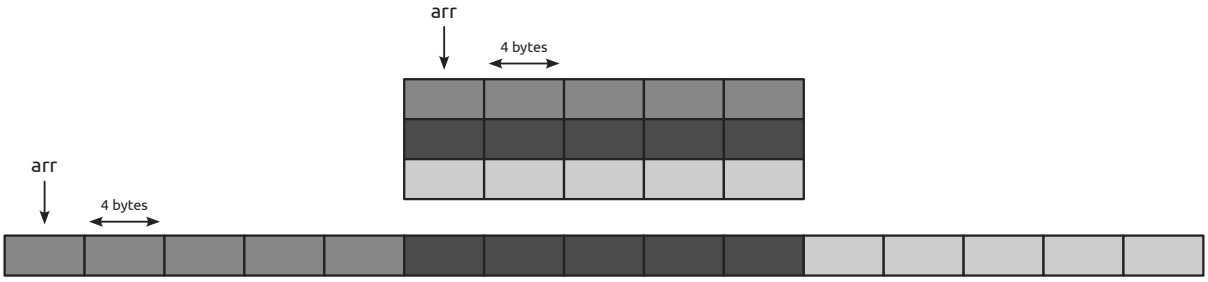


Figure 6: Tableau 3×5 de `int`

```
int arr[3][5];
int* ptr = arr;  // arr[0][0] == arr[0 * 5 + 0]
                 // arr[0][1] == arr[0 * 5 + 1]
                 // ...                  ^-> taille du "sous-tableau"
                 // arr[1][0] == arr[1 * 5 + 0]
                 // arr[1][1] == arr[1 * 5 + 1]
                 // ...
```

3.5 VLA

Un *VLA* (*Variable Length Array*) est un tableau dont la taille n'est pas connue à la compilation.

```
void foo(size_t n) {
  int arr[n];
  // ...
}
```

Cette syntaxe n'est pas standard en C++ et est donc déconseillée.

On préférera utiliser les opérateurs `new[]` et `delete[]`.

```
void foo(size_t n) {
  int* arr = new int[n];
  // ...
  delete[] arr;
}
```

4 Strings

En C++, les chaines de caractères sont des tableaux de `char`. On en distingue deux types :

- *C strings*
- `std::string`

4.1 C strings

```
char* str = "Hello!";  // str[6] == '\0'
```

Le type `char*` étant un simple pointeur (une adresse), il ne contient pas la taille de la chaine. Au lieu de ça, la chaine se termine toujours par le caractère `\0` (`chr(0)`) pour indiquer la fin.

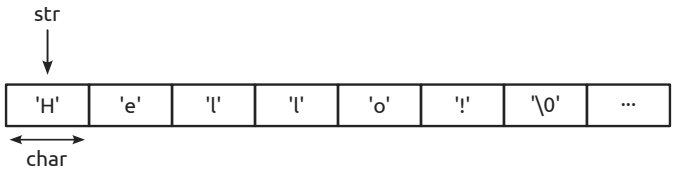


Figure 7: C string

```
size_t i = 0;
while (str[i] != '\0') {
  std::cout << str[i++];
}
std::cout << std::endl;
```

4.2 `std::string`

Le type `std::string` est défini dans l'en-tête `<iostream>`.

```
std::string str = "Hello!";
```

Ce type est un *wrapper* autour d'un `char*` et offre donc d'autres fonctionnalités[1].

```
for (char c: str) {
  std::cout << c;
}
std::cout << std::endl;
```

```
for (size_t i = 0; i < str.length(); ++i) {
  std::cout << str[i];
}
std::cout << std::endl;
```

Un `std::string` est alloué sur le stack, mais le tableau de `char` à l'intérieur est dans le heap.

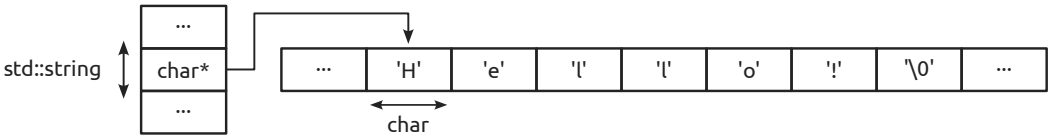


Figure 8: Allocation en mémoire d'une `std::string`

5 Erreurs courantes

5.1 Segmentation fault

L'erreur `segmentation fault` indique que vous essayez d'accéder à une adresse invalide.

```
int arr[10];
arr[100000] = 5;
```

Dans cet exemple, l'indice `[100000]` dépasse la taille du tableau.

Il est parfois difficile de trouver d'où vient une telle erreur. Utilisez le flag `-fsanitize=address` au moment de la compilation puis relancez le programme pour afficher des informations supplémentaires lorsque cela se produit.

5.2 Memory leak

Un *memory leak* (ou *fuite de mémoire*) survient lorsqu'on oublie de libérer le heap.

```
new int(5);
```

Dans cet exemple, il n'y a pas de `delete` pour libérer la mémoire allouée avec le `new`.

5.3 Dangling pointer

Un *dangling pointer* (ou *pointeur fou*) est un pointeur vers une zone de mémoire déjà libérée.

```
int* p = new int(5);
delete p;
std::cout << *p << std::endl;
```

La valeur affichée par ce code ne sera pas forcément celle attendue car la zone mémoire a peut-être été réallouée pour une autre donnée depuis.

5.4 Double free

Un *double free* survient lorsqu'on essaie de libérer une zone mémoire déjà libre.

```
int* p = new int(5);
delete p;
delete p;
```

Références

[1] "Std::basic_string - cppreference.com." Available: https://en.cppreference.com/w/cpp/string/basic_string