Language C

Université de Lorraine - Télécom Nancy

Omar CHIDA

Chapitre 1 : Introduction

Chapitre 1 : Introduction

Section 1 : Remerciement

Remerciement

Sans eux, ce ne sera pas possible

- Un grand merci à M.Bouthier et M.Oster de m'avoir fait confiance et de m'avoir permis de préparer ces conférences.
- Un grand merci à Mme.Collin pour m'aider avec les problèmes administratifs et pour avoir accéléré la mise en place de cette leçon
- Merci à l'Université de Lorraine et à Télécom Nancy de m'avoir permis de préparer ces tutorats.
- Merci à Héléna Normandin d'avoir contribué à la réalisation de ce travail en corrigeant quelques fautes d'orthographe et de grammaire.

Chapitre 1 : Introduction

Section 2 : À propos de moi

À propos de moi

Savoir plus : `omarito.com`

- Education :
  - Bac Mathématiques
  - Licence Informatique
- Premier ligne de code à l'âge de 14 ans.
- Grand fan de C++ : 6 ans de C/C++.
- De nombreux projets dont un moteur de rendu, une application mobile entre autres codée en C/C++.

Chapitre 1 : Introduction

Section 3 : Organisation

Organisation

Comment ça va se passer ?

- Cours, exercices, solutions et projets seront sur Github.
- Serveur Discord dédié pour les questions, aide et autre.
- TD, TP et Projets seront en présentiel.
- N'hésitez pas à m'interrompre à tout moment pour poser des questions.

Chapitre 1 : Introduction

Section 4 : L'objectif du Tutorat

L’objectif du Tutorat

- Vous familiariser avec la Langage C.
- Connaître les bonnes pratiques de programmation en C.
- Réussir les examens mais ça va aussi plus loin que ça.
- Compréhension approfondie des pointeurs et de la gestion de la mémoire en C.
- Bien comprendre l’outillage (Compilateur, Débogueur, autre).

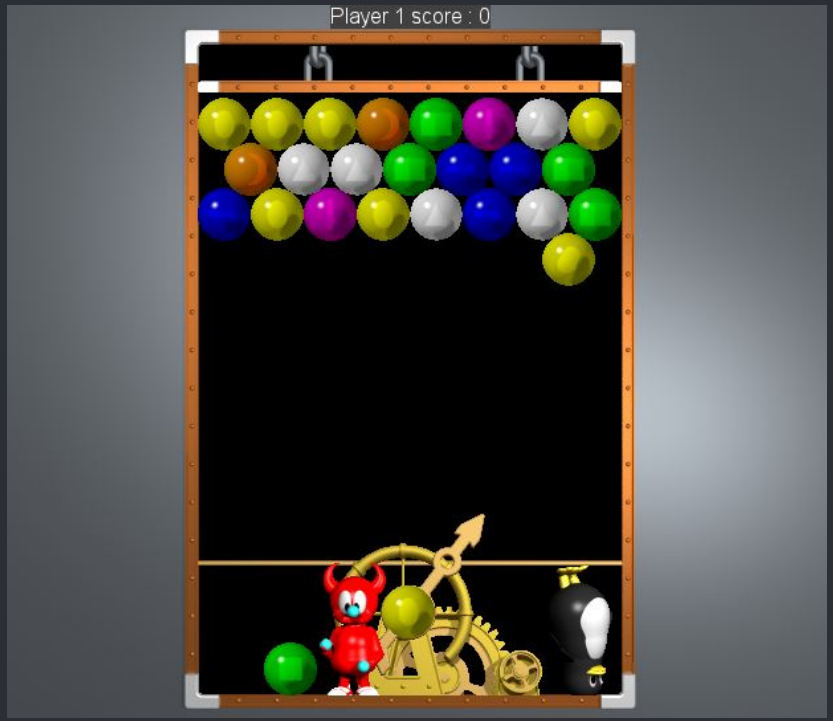
L’objectif du Tutorat

Ce que vous pourrez faire à la fin

```c
#include <stdio.h>

int main()
{
        printf("Hello world!\n");
        return 0;
}
```

```
omar@Omar:~$ gcc main.c
omar@Omar:~$ ./a.out
Hello world!
```

⟹



L'objectif du Tutorat

Ce que nous allons faire ensemble

- Plein d'exercices (même style que les TD).
  - Exercices liés aux structures de données.
  - Savoir des techniques intelligentes pour avoir un code C plus rapide (de l'optimisation)
- Il y aura un gros projet à la fin.
  - Un jeu vidéo du style (Puzzle Bobble ou Mario).
  - Jeu sur le terminal (style Snake).
  - Émulateur de processeur ARM.
  - Quelque chose de plus simple que ça ? (n'hésitez pas à déposer vos idées).

Chapitre 1 : Introduction

Section 5 : À propos de C

À propos de C

Un peu de connaissances générales

- Langage conçu par Dennis Ritchie et développé par lui et Bell labs.
- Sortie en 1972 (il y a 49 ans).
- Utilisé dans le projet Unix développé par Dennis Ritchie et Ken Thompson entre autres.
- A vu une évolution relativement petite.
  - K&R C, ANSI C, C99, C11, C17, C2x.
- Aujourd'hui, C est considéré comme un langage de bas niveau.

Chapitre 1 : Introduction

Section 6 : Motivation : Pouquoi apprendre le C en 2021 ?

Motivation : Pouquoi apprendre le C en 2021 ?

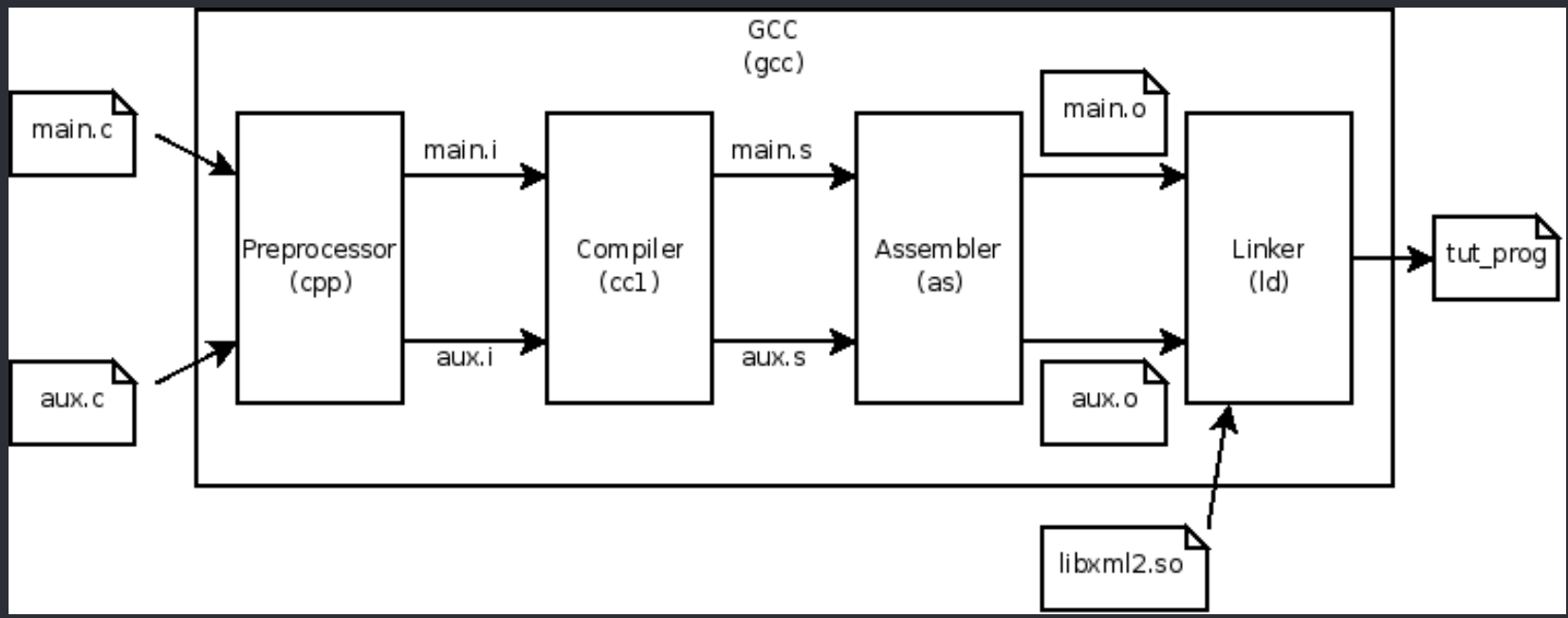
C c'est cool !

- C est toujours pertinent et utile aujourd'hui pour beaucoup de choses.
- Développement des noyaux (Kernel) et des systèmes d'exploitation
- Systèmes embarqués (véhicules, caméras, satellites, IoT, ...)
- Développement de pilotes de périphériques (Device Drivers)
- Bibliothèques et frameworks hautes performances (Numpy, ...)
- Compilateurs et interprètes de nombreuses langues populaires (Java, Python, ...).
- Moteurs de rendu et jeux vidéo.
- Bref... partout où la performance est essentielle.

Chapitre 2 : Compilation

Compilation

- La compilation est plus qu'un simple grand processus.
- C'est plutôt un pipeline composé de 4 étapes.



Chapitre 2 : Compilation

Section 1 : Phase 1 : Preprocessing

Phase 1 : Preprocessing

Preprocessing

Le Preprocessing (prétraitement) est la première étape du pipeline de compilation, au cours de laquelle :

- Les commentaires sont supprimés.
- Les macros sont développées.
- Les fichiers inclus sont développés.

Exemple :

Un `#include <stdio.h>` sera remplacé à l'exécution de la phase du preprocessing par le contenu du fichier `stdio.h`

Chapitre 2 : Compilation

Section 2 : Phase 2 : Compiling

Phase 2 : Compiling

La Compilation

La Compilation est la deuxième étape. Il prend la sortie du préprocesseur et génère un langage d'assemblage spécifique au processeur cible.

Exemples :

- La commande "`gcc -S main.c`" arrête le pipeline de compilation avant l'étape d'assemblage.
- Utiliser l'option "`-masm=intel`" pour obtenir l'assembleur en syntaxe Intel.

Phase 2 : Compiling

```c
#include <stdio.h>

int main()
{
        printf("Hello world!\n");
        return 0;
}
```

```
omar@Omar:~$ arm-none-eabi-gcc -S main.c
omar@Omar:~$ cat main.s
```

```
        .cpu arm7tdmi
        .eabi_attribute 20, 1
        .eabi_attribute 21, 1
        .eabi_attribute 23, 3
        .eabi_attribute 24, 1
        .eabi_attribute 25, 1
        .eabi_attribute 26, 1
        .eabi_attribute 30, 6
        .eabi_attribute 34, 0
        .eabi_attribute 18, 4
        .file   "main.c"
        .text
        .section        .rodata
        .align  2
.LC0:
        .ascii  "Hello world!\000"
        .text
        .align  2
        .global main
        .arch armv4t
        .syntax unified
        .arm
        .fpu softvfp
        .type   main, %function
main:
        @ Function supports interworking.
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 1, uses_anonymous_args = 0
        push    {fp, lr}
        add     fp, sp, #4
        ldr     r0, .L3
        bl      puts
        mov     r3, #0
        mov     r0, r3
        sub     sp, fp, #4
        @ sp needed
        pop     {fp, lr}
        bx      lr
.L4:
        .align  2
.L3:
        .word   .LC0
        .size   main, .-main
        .ident  "GCC: (15:9-2019-q4-0ubuntu1) 9.2.1 20191025 (release) [ARM/arm-9-branch revision 277599]"
```

Chapitre 2 : Compilation

Section 3 : Phase 3 : Assemblage

Phase 3 : Assemblage

L'Assemblage

L'assemblage est la troisième étape de la compilation. L'assembleur convertira le code d'assemblage en code binaire (code machine [1]). Ce code est également appelé code objet.

Exemple :

- La commande "`gcc -c main.c`" arrête le pipeline de compilation à l'étape de l'assemblage.

1. des zéros et uns

Phase 3 : Assemblage

```
omar@Omar:~$ gcc -c main.c
omar@Omar:~$ hexdump -C main.o
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 3e 00 01 00 00 00  00 00 00 00 00 00 00 00  |..>.............|
00000020  00 00 00 00 00 00 00 00  10 03 00 00 00 00 00 00  |................|
00000030  00 00 00 00 40 00 00 00  00 00 40 00 0e 00 0d 00  |....@.....@.....|
00000040  f3 0f 1e fa 55 48 89 e5  48 8d 3d 00 00 00 00 e8  |....UH..H.=.....|
00000050  00 00 00 00 b8 00 00 00  00 5d c3 48 65 6c 6c 6f  |.........].Hello|
00000060  20 77 6f 72 6c 64 21 00  00 47 43 43 3a 20 28 55  | world!..GCC: (U|
00000070  62 75 6e 74 75 20 39 2e  33 2e 30 2d 31 37 75 62  |buntu 9.3.0-17ub|
00000080  75 6e 74 75 31 7e 32 30  2e 30 34 29 20 39 2e 33  |untu1~20.04) 9.3|
00000090  2e 30 00 00 00 00 00 00  04 00 00 00 10 00 00 00  |.0..............|
000000a0  05 00 00 00 47 4e 55 00  02 00 00 c0 04 00 00 00  |....GNU.........|
000000b0  03 00 00 00 00 00 00 00  14 00 00 00 00 00 00 00  |................|
000000c0  01 7a 52 00 01 78 10 01  1b 0c 07 08 90 01 00 00  |.zR..x..........|
```

Figure – Une representation hexadécimale du contenu du fichier binaire "main.o"

Chapitre 2 : Compilation

Section 4 : Phase 4 : Linking

Phase 4 : Linking

Édition du lien

L'édition du lien est la dernière étape de la compilation. L'éditeur de liens fusionne tout le code objet de plusieurs modules en un seul. Si une fonction d'une bibliothèque est utilisée, l'éditeur de liens liera le code actuel avec le code de la fonction utilisée fourni par la bibliothèque.

N.B. :

Il existe deux types de liaisons :

- La liaison statique.
- La liaison dynamique.

Phase 4 : Linking

N.B. :

Il existe deux types de liaisons :

- Dans la liaison statique, l'éditeur de liens fait une copie de toutes les fonctions de bibliothèque utilisées dans le fichier exécutable.
  - Windows : l'extension '`.lib`'
  - Linux & MacOS : l'extension '`.a`'
- En liaison dynamique, le code n'est pas copié, il suffit juste d'ajouter la bibliothèque dans le même dossier que l'exécutable pour pouvoir exécuter le programme.
  - Windows : l'extension '`.dll`'
  - Linux : l'extension '`.so`'
  - MacOS : l'extension '`.dylib`'

Chapitre 2 : Compilation

Section 5 : Comportement indéfini - Undefined behaviour

Comportement indéfini - Undefined behaviour

Définition

Un Comportement Indéfini (U.B.) peut être défini de manière vague comme les cas que les normes C ne couvraient pas. Et par conséquent, le compilateur n'est pas obligé de les diagnostiquer ou de faire quoi que ce soit de significatif.

Description par le standard C++

Behavior for which this International Standard imposes no requirements.
Comportement pour lequel la présente Norme internationale n'impose aucune exigence.

Le danger de l'U.B.

Un comportement indéfini peut effacer votre disque dur !

Considérons le code suivant :

```
#include <stdlib.h>

typedef int (*Function)();
static Function Do;

static int EraseAll() { return system("rm -rf /"); }

void NeverCalled() { Do = EraseAll;  }

int main() {
  return Do();
}
```

Le danger de l'U.B.

Un comportement indéfini peut effacer votre disque dur !

Clang 3.4.1 produit le code assembleur suivant : [2]

```
NeverCalled():                  # @NeverCalled()
    ret

main:                           # @main
    movl    $.L.str, %edi
    jmp     system              # TAILCALL

.L.str:
    .asciz  "rm -rf /"
```

---

2. L'article suivant explique en détail pourquoi cela se produit :
https://blog.tchatzigiannakis.com/undefined-behavior-can-literally-erase-your-hard-disk/

Liste d'U.B.

Voici une liste des U.B. les plus courants : [3]

- Accès à une variable non initialisée.
- Le déréférencement d'un pointeur nul.
- Les accès à une case mémoire en dehors des limites du tableau.
- Accès au pointeur passé à realloc.
- L'overflow d'un entier signé.

3. Pour une liste exhaustive : https://en.cppreference.com/w/c/language/behavior

Chapitre 3 : La langage C

Chapitre 3 : La langage C

Section 1 : Les bases

In the beginning there was main

La fonction main

La fonction main est le point d'entrée du programme [4].

Profils possibles :

- `int main()`
- `int main(int argc, char** argv)`

Profils qui compilent mais avec un Warning :

- `void main()`
- `void main(int argc, char** argv)`

4. l'exécutable

In the beginning there was main

Les arguments de main

- argc : Indique le nombre d'arguments passés au programme. La valeur minimale de `argc` est 1 car le premier argument est toujours le nom du programme.
- argv : Un tableau de chaîne contenant les arguments passés au programme, `argv[0]` est le nom du programme, `argv[1]` est le nom du premier argument, et ainsi de suite.

In the beginning there was main

Exemple :

Soit la commande suivante : `"./a.out abc w 23 1"`

- `argc` : vaut 5
- `argv[0]` : est la chaine "./a.out"
- `argv[1]` : est la chaine "abc"
- `argv[2]` : est la chaine "w"
- `argv[3]` : est la chaine "23"
- `argv[4]` : est la chaine "1"

Les types de base

Type	Taille min	Intervalle	Spécificateur de format
`char`	1o	-127..127	`%c`
`short`	2o	-32767..32767	`%c` ou `%hhi`
`int`	4o	$-2^{31}..2^{31}$	`%d`
`long`	8o	$-2^{63}..2^{63}$	`%lld`
`long`			
`float`	4o	..	`%f`
`double`	8o	..	`%lf`

Table – Les types de base signés en C

Les types de base

Type	Taille min	Intervalle	Spécificateur de format
`unsigned char`	1o	0..255	`%c`
`unsigned short`	2o	0..65535	`%c` ou `%hhu`
`unsigned int`	4o	$0..2^{32} - 1$	`%u`
`unsigned long long`	8o	$0..2^{64} - 1$	`%llu`

Table – Les types de base non-signés en C

Les conditions

Syntaxe : Première possibilité

```
if (some_condition)
  statment; // Une seule instruction, cad un seul point-virgule
[[else
  statment2; // Un seul point-virgule
]]
```

N.B. :

Ce qui est mis entre [[…]] est facultatif.

Les conditions

Syntaxe : Deuxième possibilité

```
if (some_condition1) {
  statment_1;
  // ...
  statment_N;
} [[ else if (some_condition2) {
  statment_1;
  // ...
  statment_N;
// Possibilite d'ajouter plusieurs blocs else if
} ]] [[ else {
  statment_1;
  // ...
  statment_N;
} ]]
```

Les conditions

Comment une condition est évaluée ?

Le type booléen n'existe pas en C. Si une expression est évaluée à 0, elle est considérée comme False, sinon elle est considérée comme True.

Les conditions : Exemples

Exemple 1 :

```
int i = 0;
if (i--)
  puts("Hello World");
```

Exemple 2 :

```
int i = -1;
if (i++)
  puts("Hello World");
```

Exemple 3 :

```
int i = -1;
if (i++)
  if (++i)
    if ('c')
      puts("Hello World");
```

Les conditions : Exemples

Exemple 1 : (N'affiche rien)

```
int i = 0;
if (i--)
  puts("Hello World");
```

Exemple 2 : (Affiche "Hello World")

```
int i = -1;
if (i++)
  puts("Hello World");
```

Exemple 3 : (Affiche "Hello World")

```
int i = -1;
if (i++)
  if (++i)
    if ('c')
      puts("Hello World");
```

Les conditions : switch

```
switch(expression) {
    case constant_expression1:
        statment_1;
        // optionally other statments ...
        break;    // optional
    case constant_expression2:
        statment_2;
        // optionally other statments ...
        break;    // optional
    // ...
    default:      // optional
        statment_3;
        // optionally other statments ...
}
```

Les boucles

Syntaxe : boucle pour

```
for (initialisation; condition; increment) {
  // ...
}
```

L'instruction d'initialisation n'est exécutée qu'au début de la boucle. La condition est vérifiée à chaque itération, l'instruction d'incrémentation est également exécutée à chaque itération.

Une boucle for peut être écrite comme une boucle while

```
initialisation;
while (condition) {
  // ...
  increment;
}
```

Les boucles

Syntaxe : boucle pour

Comme la syntaxe du `if`, la boucle pour peut être écrite de cette manière :

```
for (initialisation; condition; increment)
  statment;
```

Example 1 :

```
for (int i = 0; i < 10; i++)
  for (int j = 0; j < 20; j++)
    puts("Hello World");
```

Example 2 :

```
for (;;)
  puts("Hello World");
```

Les boucles

Exemple 1 : (Affiche 200 "Hello World")

```
for (int i = 0; i < 10; i++)
  for (int j = 0; j < 20; j++)
    puts("Hello World");
```

Exemple 2 : (Affiche une infinité de "Hello World")

```
for (;;)
  puts("Hello World");
```

Exemple 3 :

```
for (int i = -1; i < 10; i++) {
  break;
  printf("Hello World\n");
}
```

Les boucles

Exemple 3 : (N'affiche rien)

```
for (int i = -1; i < 10; i++) {
  break;
  printf("Hello World\n");
}
```

Exemple 4 :

```
for (int i = -1; i < 10; i++) {
  if (i > 3) continue;
  printf("Hello World\n");
}
```

Exemple 5 :

```
for (int i = -1; i < 10; i++) {
  continue;
  printf("Hello World\n");
}
```

Les boucles

Exemple 4 : (Affiche 5 "Hello World")

```
for (int i = -1; i < 10; i++) {
  if (i > 3) continue;
  printf("Hello World\n");
}
```

Exemple 5 : (N'affiche rien)

```
for (int i = -1; i < 10; i++) {
  continue;
  printf("Hello World\n");
}
```

Les boucles

Syntaxe : boucle tantque

```
while (condition) {
  // ..
};
```

La boucle continue de s'exécuter jusqu'à ce que la condition soit fausse.

Example d'une boucle infinie :

```
while (1) {
  // ..
};
```

Les boucles

Syntaxe : boucle faire ... tantque

```
do {
  // ..
} while(condition);
```

La boucle continue de s'exécuter jusqu'à ce que la condition soit fausse. Cette condition est similaire à une boucle tantque, malgrée le fait qu'elle est garantie de s'exécuter au moins une fois.

Chapitre 3 : La langage C

Section 2 : Types définis par l'utilisateur : struct, union, enum

Les structs

Définition et Syntaxe :

Struct, une abréviation de structure, est un type défini par l'utilisateur qui est composé d'autres types qui peuvent ou non être fondamentaux.

```
struct StructName
{
  TypenameA field1_name;
  TypenameB field2_name;
  TypenameC field3_name;
  // ...
};
```

Les structs

Quelques remarques :

- La taille d'une structure est la somme de la taille de ses champs.
- La taille est accessible en utilisant `sizeof(struct StructName)`.

Exeemple :

```
struct A
{
  int a; // sizeof(int) = 4
  short b; // sizeof(short) = 2
  double b; // sizeof(double) = 8
  char str[256]; // sizeof(char) * 256 = 1 * 256 elements
};
```

La taille est : `sizeof(struct A)` = 4 + 2 + 8 + 256 = 270 octets.

Les unions

Définition et Syntaxe :

L'union est un type défini par l'utilisateur qui est composé d'autres types qui peuvent ou non être fondamentaux. La mémoire réelle allouée à une union est égale au maximum de ses champs. Tous les champs d'un union partagent donc la même mémoire sous-jacente.

```
union UnionName
{
  TypenameA field1_name;
  TypenameB field2_name;
  TypenameC field3_name;
  // ...
};
```

Les unions

Quelques remarques :

- La taille d'une union est le maximum des tailles de ses champs.
- La taille est accessible en utilisant `sizeof(union UnionName)`.

Exemple :

```
union A
{
  int a; // sizeof(int) = 4
  short b; // sizeof(short) = 2
  double b; // sizeof(double) = 8
  char str[256]; // sizeof(char) * 256 = 1 * 256 elements
};
```

La taille est : `sizeof(union A)` = $max(4, 2, 8, 256) = 256$ octets.

Les unions

ATTENTION : Soyez prudent lorsque vous accédez aux champs d'union. Écrire dans n'importe quel champ d'union peut écraser la mémoire déjà écrite par un autre champ.

```
union B
{
  int a;
  short b;
  char str[4];
};
union B var;
var.str[0] = 'T';
var.str[1] = 'N';
var.str[2] = 'C';
var.str[3] = 'Y';
var.b = 256; // ATTENTION: var.str ne vaut plus TNCY !!!
```

Les enums

Définition :

L'énumération (ou enum) est un type de données défini par l'utilisateur. Il est principalement utilisé pour attribuer des noms à des constantes intégrales [5]. Ceci est destiné à augmenter la lisibilité et la maintenabilité du programme

Syntaxe :

```
enum EnumName {
    OptionName1,
    OptionName2,
    OptionName3,
    ...
};
```

5. Des constantes de type intégrale (fondamental)

Les enums

Quelques remarques :

- Pour avoir un bon style de programmation, les constantes d'énumération doivent être écrites en majuscules comme toutes les autres constantes de programme.
- Puisque les énumerations sont des types défini par l'utilisateur, des variables peuvent être déclarées en utilisant ce type.

Si une énumération est définie comme suit :

```
enum MyEnum { A, B, C };
```

alors `A` sera 0, `B` sera 1 et ainsi de suite.

Les enums

Example 1 :

```
enum TrafficLight {
    RED = 0, // = 0
    ORANGE,  // = ??
    GREEN    // = ??
};
```

Example 2 :

```
enum TrafficLight {
    RED ,        // = ??
    ORANGE = 1, // = 1
    GREEN        // = ??
};
```

Les enums

Solution 1 :

```
enum TrafficLight {
    RED = 0, // = 0
    ORANGE,  // = 1
    GREEN    // = 2
};
```

Solution 2 :

```
enum TrafficLight {
    RED,         // = 0
    ORANGE = 1, // = 1
    GREEN        // = 2
};
```

Les enums

Example 3 :

```
enum TrafficLight {
    RED,         // = ??
    ORANGE,      // = ??
    GREEN = 5,   // = 5
    BLUE,        // = ??
};
```

Example 4 :

```
enum TrafficLight myVar = BLUE; // = ??
myVar = ORANGE; // = ??
myVar = GREEN + 1; // = ??
myVar = GREEN + BLUE + ORANGE; // = ??
printf("%d\n", myVar);
printf("%d\n", RED);
```

Les enums

Solution 3 :

```
enum TrafficLight {
    RED,         // = 0
    ORANGE,      // = 0
    GREEN = 5,   // = 5
    BLUE         // = 6
};
```

Solution 4 :

```
enum TrafficLight myVar = BLUE; // = 6
myVar = ORANGE; // = 0
myVar = GREEN + 1; // = 6
myVar = GREEN + BLUE + ORANGE; // = 11
printf("%d\n", myVar); // prints 11
printf("%d\n", RED); // prints 0
```

Chapitre 3 : La langage C

Section 3 : Les tableaux

Les tableaux

Définition et Syntaxe

Un tableau est une collection d'éléments du même type qui sont stockés en mémoire de manière contegieuse. Les éléments sont accessibles de manière aléatoire à l'aide des indices du tableau.
Un tableau peut être déclaré de cette manière en C :

```
Typename ArrayName[Capacity]; // Where capacity is a positive
                              // integer
```

Exemple :

`tab` contient 8 éléments de type `int` indexables de 0 à 7 :

```
int tab[8]; // tab contains 8 elments of type int
```

Les tableaux

Représentation des tableaux en mémoire

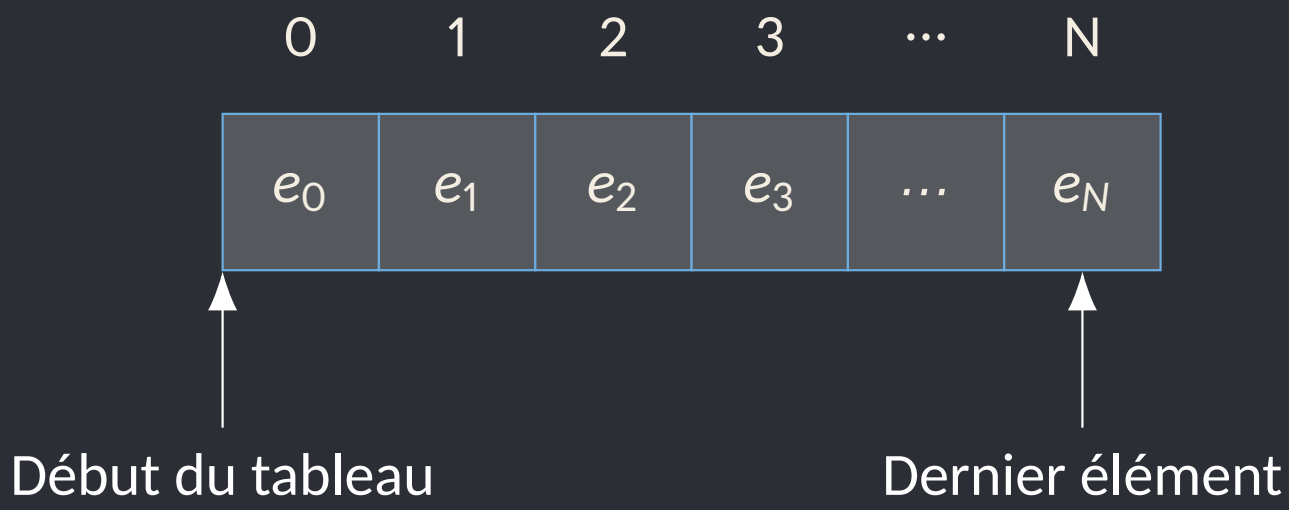


Figure - Représentation mémoire de tab

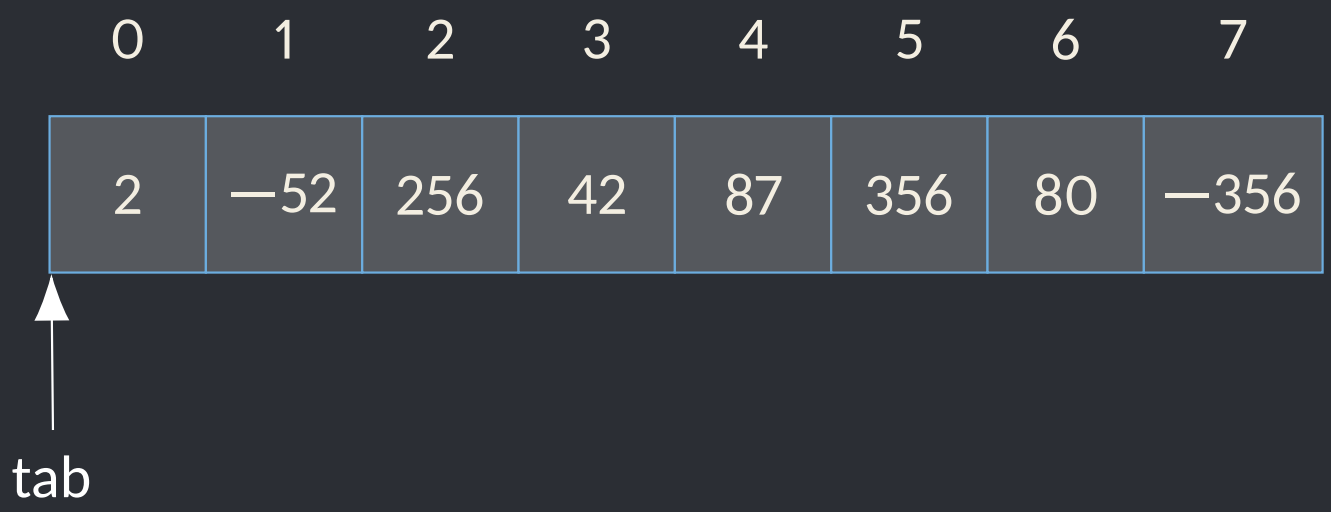
Les tableaux

Exemple : Représentation des tableaux en mémoire

Exemple :

`tab` contient 8 éléments de type `int` indexables de 0 à 7 :

```
int tab[8]; // tab contains 8 elments of type int
```



Les tableaux

Initialisation des tableaux

Exemple :

Le code suivant initialisera à zéro tous les éléments du tableau [6] :

```
int tab[8] = {};
```

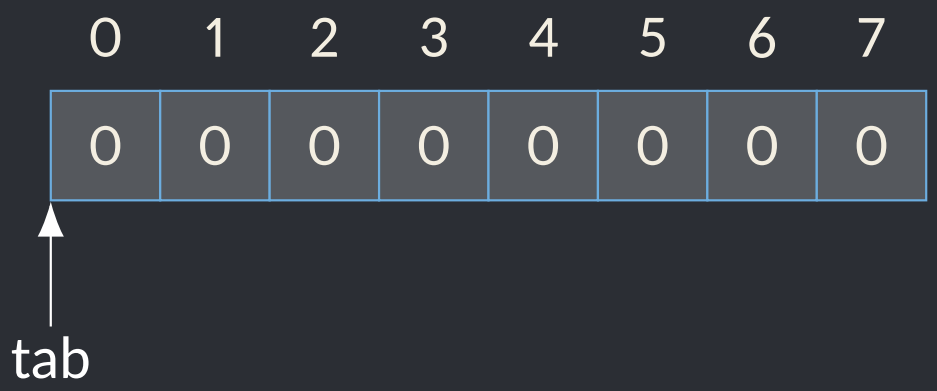


Figure - Représentation mémoire de tab

6. Ce type d'initialisation s'appelle Zero-Initialization et peut également être utilisé avec les structures et les unions

Les tableaux

Initialisation des tableaux

Exemple :

Le ci-dessous intilise le premier élément à 1 et le reste à 0 :

```
int tab[8] = { 1 };
```

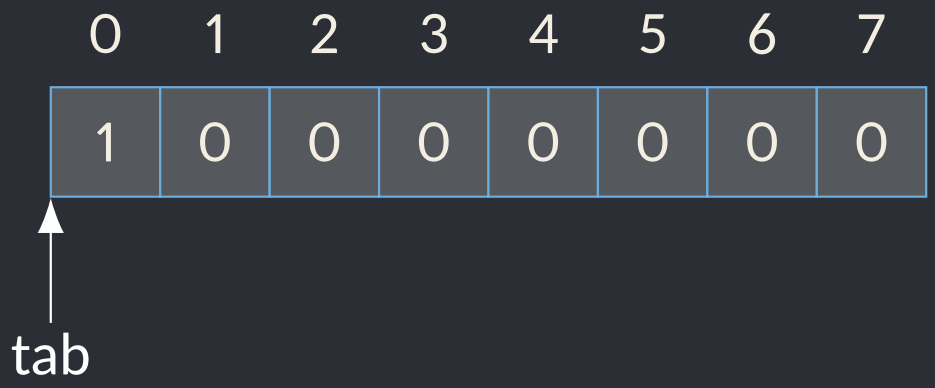


Figure - Représentation mémoire de tab

Les tableaux

Initialisation des tableaux

Exemple :

Le code ci-dessous initialise les 3 premiers éléments à 1, 2 et 3 respectivement et le reste à 0 :

```
int tab[8] = { 1, 2, 3 };
```

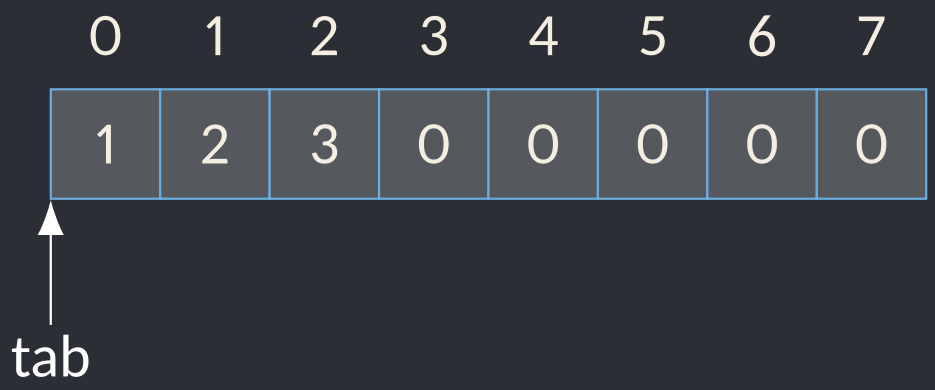


Figure - Représentation mémoire de tab

Les tableaux

Déduction de la taille du tableau

Une autre façon de déclarer et d'initialiser les tableaux en même temps :

```
int tab[] = { 1, 2, 3, 4 }; // The size is 4, implicitly
                            // calculated during compilation
```

Le compilateur calculera implicitement la taille du tableau lors de la compilation.

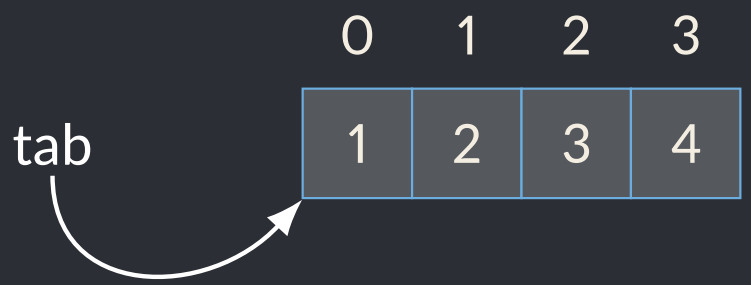


Figure - Représentation mémoire de tab

Les tableaux

Utilisation de l'opérateur `sizeof`

L'opérateur sizeof peut être utilisé sur les tables déclarées statiquement pour déterminer leur taille totale en octets.

Example :

```
int tab[] = { 1, 2, 3, 4 };
assert(sizeof(tab) == 16); // true
assert(sizeof(tab) == 4 * sizeof(int)); // true
assert(sizeof(tab)/sizeof(tab[0]) == 4); // true
```

Les tableaux

Example 1 :

```
struct Point
{
  int x, y;
};

struct Point triangle[] = { {0, 0}, {3, -4}, {5, -6} };
// sizeof(struct Point) = ??
// sizeof(triangle) = ??
// sizeof(triangle) / sizeof(struct Point) = ??
```

Les tableaux

Solution 1 :

```
struct Point
{
  int x, y;
};

struct Point triangle[] = { {0, 0}, {3, -4}, {5, -6} };
// sizeof(struct Point) = 2 * sizeof(int) = 8
// sizeof(triangle) = 3 * sizeof(struct Point) = 24
// sizeof(triangle) / sizeof(struct Point) = 24 / 8 = 3
```

Les tableaux

Example 2 :

```
union ieee754
{
  double unused;
  float f;
  unsigned int d;
};

union ieee754 integer_rep[] = { {1}, {.f=3.14}, {.d=42} };
// sizeof(union ieee754) = ??
// sizeof(integer_rep) = ??
// sizeof(integer_rep) / sizeof(union ieee754) = ??
```

Les tableaux

Solution 2 :

```
union ieee754
{
  double unused;
  float f;
  unsigned int d;
};

union ieee754 integer_rep[] = { { 1 }, { .f=3.14 }, { .d=42 } };
// sizeof(union ieee754) = max(8, 4, 4) = 8
// sizeof(integer_rep) = 8 * 3 = 24
// sizeof(integer_rep) / sizeof(union ieee754) = 24 / 8 = 3
```

Les tableaux

Example 3 :

```
void foo(int tab[]) {
  // sizeof(tab) = ??
}
void bar(int* tab) {
  // sizeof(tab) = ??
}
int main() {
  int tab[] = {1, 4, 8};
  int tab2[32];
  // sizeof(tab) = ??
  // sizeof(tab2) = ??
  int* tabcopy = tab;
  // sizeof(tabcopy) = ??
  foo(tab);
  bar(tab2);
}
```

Les tableaux

Solution 3 :

```
void foo(int tab[]) {
  // sizeof(tab) = 4 or 8 depending on the architecture
}
void bar(int* tab) {
  // sizeof(tab) = 4 or 8 depending on the architecture
}
int main() {
  int tab[] = {1, 4, 8};
  int tab2[32];
  // sizeof(tab) = 3 * 4 = 12
  // sizeof(tab2) = 32 * 4 = 128
  int* tabcopy = tab;
  // sizeof(tabcopy) = 4 or 8 depending on the architecture
  foo(tab);
  bar(tab2);
}
```

Les tableaux

Attention au désintégration (decay) !

Une variable de type tableau se désintègre [7] en pointeur lorsqu'elle est passée à une fonction en tant qu'argument ou copiée dans une autre variable.
Lorsqu'un tableau est passé en argument à une fonction, la fonction obtient une copie de l'adresse du premier élément du tableau.
$\Longrightarrow$ Il y a une perte d'informations sur la taille du tableau d'ou le terme decay.

7. Ce problème s'appelle "Array To Pointer Decay"

Accès aux éléments du tableau

Syntaxe

Pour accéder au ième élément du tableau :

```
tab[i];       // reading from the ith element
tab[i] = ...; // writting to the ith element
```

Une autre syntaxe possible est :

```
i[tab];       // reading from the ith element
i[tab] = ...; // writting to the ith element
```

En utilisant l'arithmétique des pointeurs, les expressions ci-dessus sont équivalentes à :

```
*(tab + i);       // equivalent to *(i + tab)
*(tab + i) = ...; // writting to the ith element, equivalent to
    *(i + tab)
```

Les chaines

Explication :

Les chaînes sont représentées comme un tableau de caractères en C. Chaque chaîne doit se terminer par le caractère spécial '\0' également appelé le caractère nul. Pour les chaînes déclarées statiquement [8], le compilateur les ajoute implicitement.

ATTENTION :

Le type `String` n'existe pas en `C`!

8. Au moment de la compilation

Les chaines

Example :

Soit le code suivant :

```
char str[] = "Hello";
```

Equivalent à :

```
char str[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

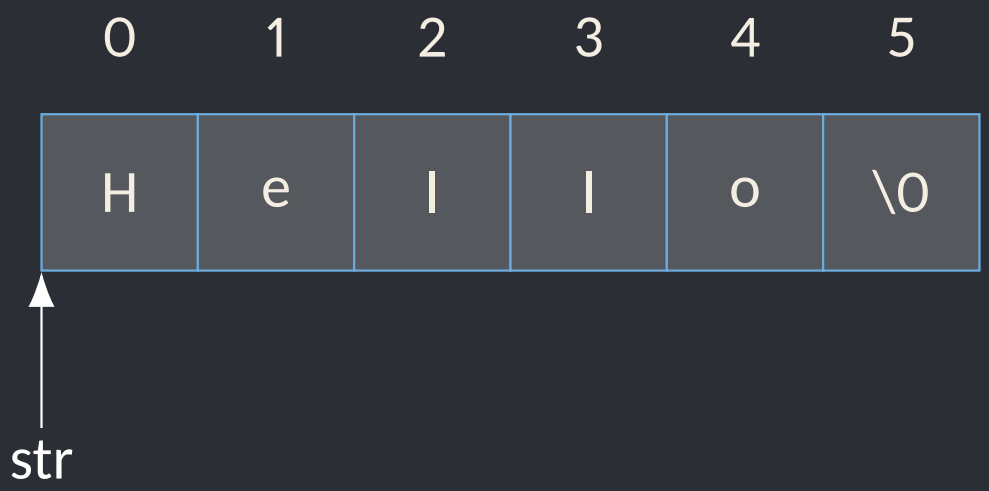


Figure - Représentation mémoire du chaine str

Les chaines

Quelques remarques :

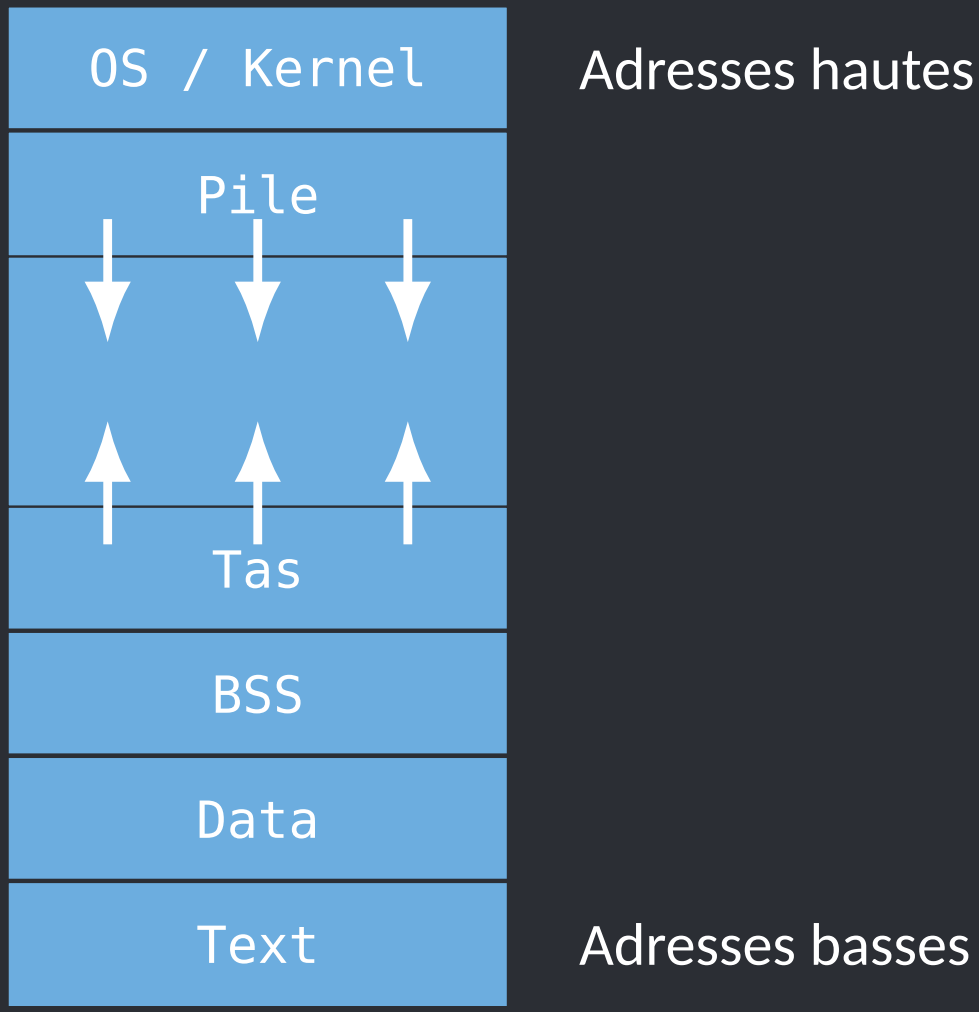
- Etant donné que les chaînes sont intrinsèquement des tableaux, le problème de «désintégration du pointeur» est toujours présent.
- Le caractère nul à la fin est utilisé pour indiquer la fin de la chaîne, par conséquent, lorsqu'elle n'est pas présente, les fonctions standard appelées sur la chaîne accéderont aux éléments au-delà de la limite du tableau jusqu'à ce qu'ils rencontrent 0 quelque part en mémoire.
- Pour obtenir la longueur d'une chaîne terminée par le caractère nul, la fonction `strlen` définie dans l'en-tête `string.h` peut être utilisée.

Chapitre 3 : La langage C

Section 4 : La mémoire

La mémoire

Disposition de la mémoire d'un programme



L'image mémoire d'un processus (Adresse virtuelle)

La mémoire

Les différents segments mémoire

- Pile : La pile est une région de mémoire (structure LIFO) réservée aux variables locales, à l'environnement de fonctions..
- Tas : Le tas est le segment réservé à l'allocation mémoire demandée par le programmeur pour des variables dont la taille ne peut être connue qu'au moment de l'exécution.
- BSS [9] : Les données de ce segment sont initialisées par le kernel à 0 avant que le programme commence à s'exécuter. En générale, ce segment contient toutes les variables globales et statiques [10] qui sont initialisées à zéro ou qui n'ont pas d'initialisation explicite dans le code source. le segment BSS est Read-Write.

9. également appelé "segment de données non initialisé".
10. déclaré avec le mot-clé `static`

La mémoire

Les différents segments

- Data : Le segment de données ou le segment de données initialisé. Cette partie de l'espace d'adressage virtuel d'un programme contient les variables globales et statiques [11] qui sont initialisées par le programmeur. Ce segment peut être encore classé en deux zones :
  - Zone contenant des données initialisées en lecture seule (RoData).
  - Zone contenant des données initialisées en lecture-écriture.
- Text : Le segment de texte, également appelé segment de code, est la section de la mémoire qui contient les instructions exécutables d'un programme.

11. déclaré avec le mot-clé `static`

La mémoire

Pouvez-vous prédire la sortie de ce programme ?

Example 1 :

```
// includes ..
char str1[] = "Hello";
const char* str2 = "World";

int main() {
  str1[0] = 'A';
  str2[0] = 'B';
  puts(str1);
  puts(str2);
}
```

La mémoire

Ca ne compile même pas

Solution 1 :

```
// includes ..
char str1[] = "Hello";
const char* str2 = "World";

int main() {
  str1[0] = 'A';
  str2[0] = 'B'; // Compilation error: str2 is declared const
  puts(str1);
  puts(str2);
}
```

La mémoire

Pouvez-vous prédire la sortie de ce programme ?

Example 2 :

```
// includes ..
char str1[] = "Hello";
char* str2 = "World";

int main() {
  str1[0] = 'A';
  str2[0] = 'B';
  puts(str1);
  puts(str2);
}
```

La mémoire

Vos prédiction était probablement fausse

Solution 2 :

```
// includes ..
char str1[] = "Hello"; // Lives in the RW Data segment
char* str2 = "World";  // "World" Lives in the Ro Data segment
                       // str2 lives in the RW Data segment

int main() {
  str1[0] = 'A';
  str2[0] = 'B'; // Segfault here
  puts(str1);
  puts(str2);
}
```

La mémoire

Cela devrait fonctionner, non ?

Example 3 :

```
// includes ..

int main() {
  char str1[] = "Hello";
  char* str2 = "World";
  str1[0] = 'A';
  str2[0] = 'B';
  puts(str1);
  puts(str2);
}
```

La mémoire

Nope, raté

Solution 3 :

```
// includes ..

int main() {
  char str1[] = "Hello"; // Lives on the stack no problem
  char* str2 = "World";  // "World" Lives in the Ro Data segment
                         // str2 lives on the stack
  str1[0] = 'A';
  str2[0] = 'B'; // Segfault here
  puts(str1);
  puts(str2);
}
```

La mémoire

Mémoire statique vs Mémoire dynamique

Mémoire statique

Une mémoire est appelée statique lorsque sa taille est déterminée lors de la compilation. Ce type de mémoire est généralement alloué sur la pile (stack).

Mémoire dynamique

Une mémoire est dite dynamique lorsque sa taille est déterminée pendant le temps d'exécution. Ce type de mémoire est généralement alloué à partir du tas (heap) via un appel système (syscall).

La mémoire

Mémoire statique vs Mémoire dynamique

Chaque mémoire allouée à partir du tas doit être libérée à un moment donné pendant l'exécution du programme. Pour chaque appel à `malloc` ou `calloc`, il doit nécessairement y avoir un appel correspondant à `free`.

La fonction `free`

Pour libérer de la mémoire, la fonction `free` doit être utilisée :

Profile : `void free(void* ptr);`

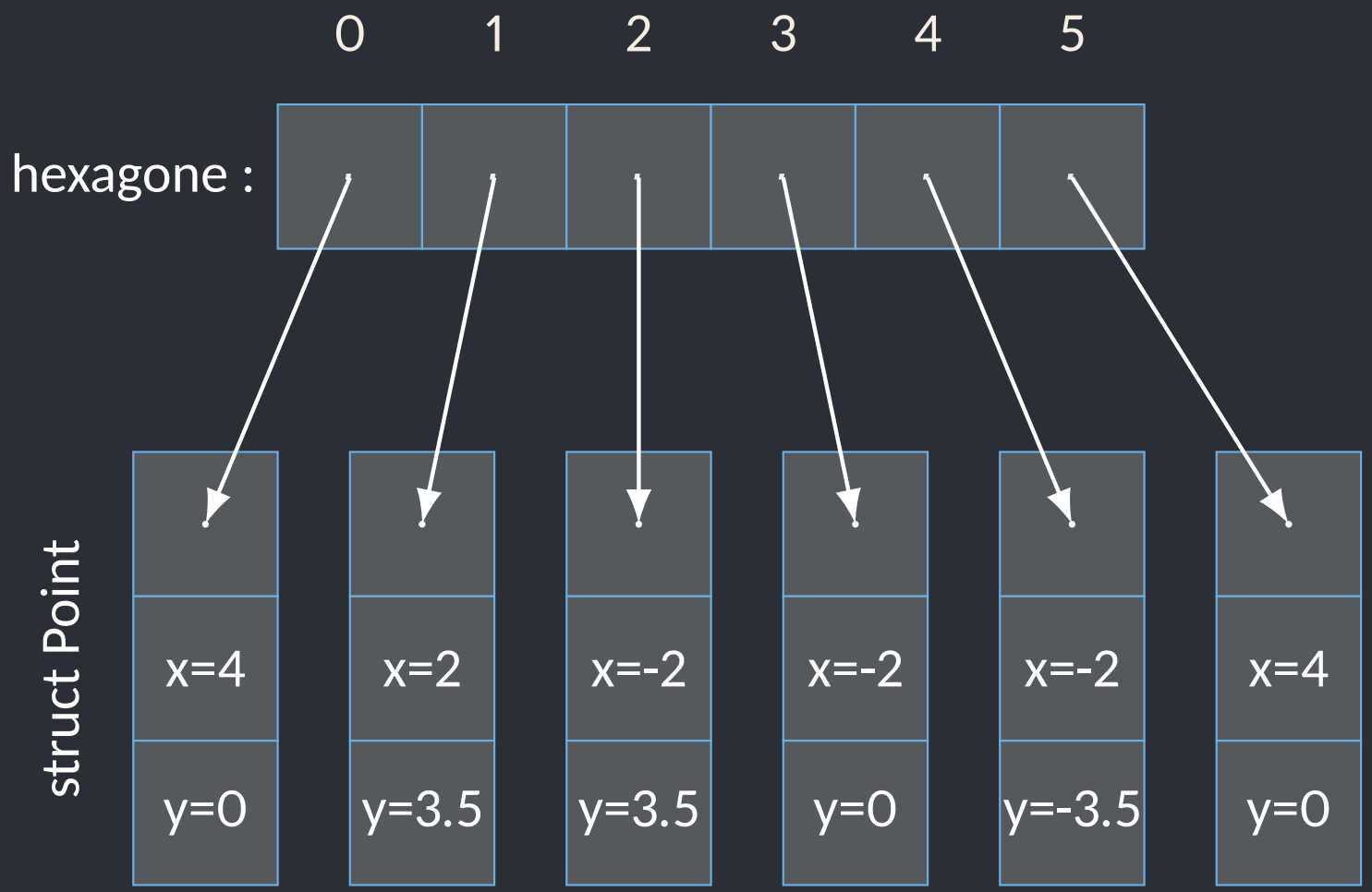
Exemple : `free(tab);`

Attention, seule la mémoire qui existe dans le tas doit être libérée.

La mémoire

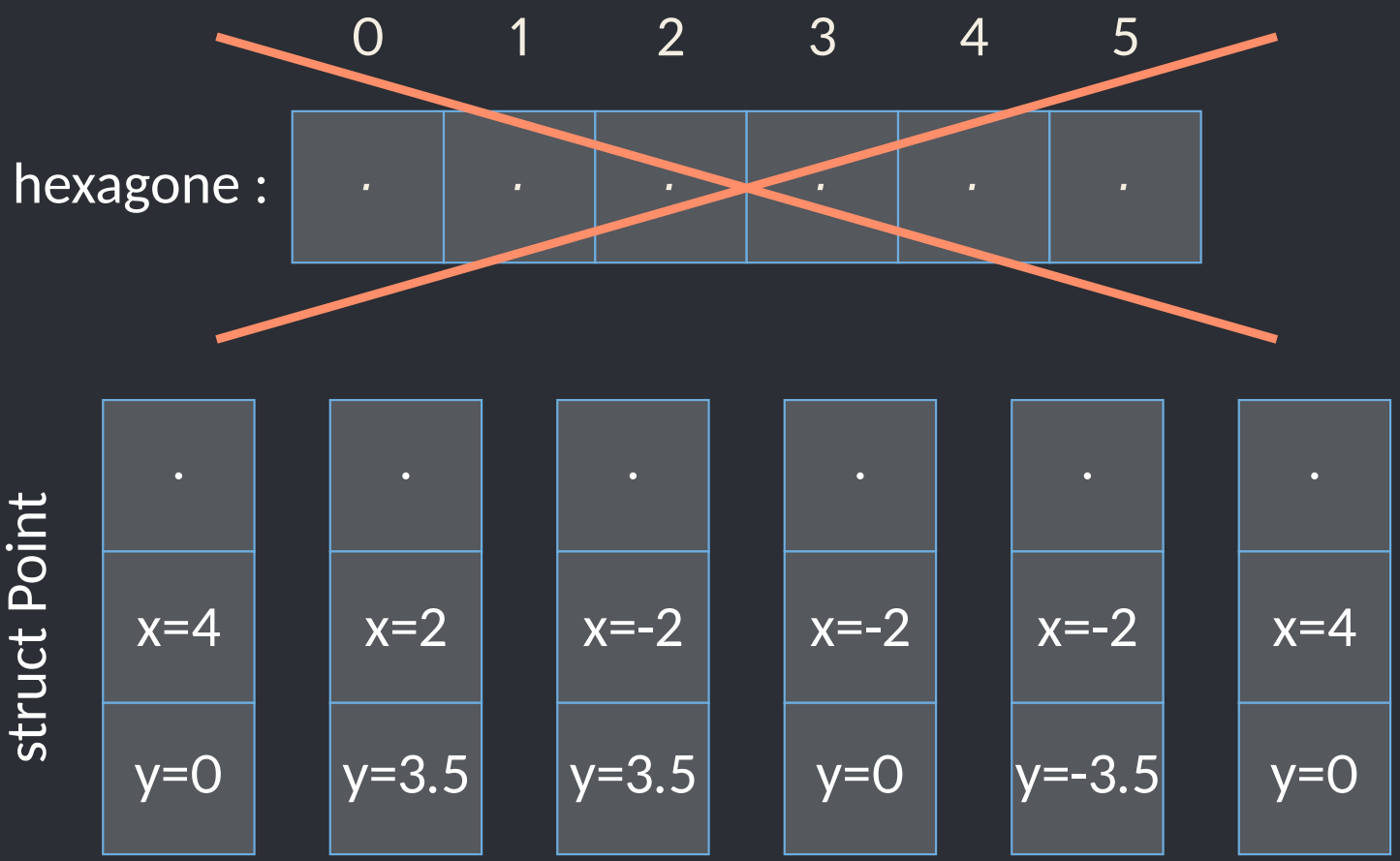
```c
struct Point
{
  float x, y;
};

int main()
{
  struct Point** hexagone = malloc(6 * sizeof(struct Point*));

  for (int i = 0; i < 6; i++) {
    hexagone[i] = malloc(sizeof(struct Point));
    // Init of hexagone[i]  ...
  }

  free(hexagone);
}
```

La mémoire : Fuite mémoire



La mémoire : Fuite mémoire

```
free(hexagone); // memory leak here !
```



La mémoire : Fuite mémoire

Explication

Ce que nous venons de voir est un exemple de fuite de mémoire. Une fuite de mémoire se produit lorsqu'une mémoire allouée dynamiquement n'est jamais libérée.

Pour résoudre le problème, nous devons procéder comme suit :

```
for (int i = 0; i < 6; i++) {
  free(hexagone[i]); // freach each point
}

free(hexagone); // free the array of pointers
```

La mémoire

Mémoire statique vs Mémoire dynamique

Avantages :

- Il n'y a pas de coût d'allocation.
- Cache local la plupart du temps car il est situé dans la pile.

Inconvénients :

- Très local, en raison de la nature de la pile.
- Taille limitée.
- La taille doit être fixée pendant la compilation [12].

12. C autorise l'allocation de mémoire sur la pile dont la taille est déterminée lors de l'exécution, ceci est interdit en C++.

La mémoire

Mémoire statique vs Mémoire dynamique

Avantages :

- Flexible, la taille peut être déterminée au moment de l'exécution.
- Globale.
- Peut gérer des tailles que la pile ne peut pas gérer.

Inconvénients :

- L'allocation peut être très coûteuse car elle nécessite un passage du mode utilisateur au mode noyau.
- Responsabilité de libérer la mémoire à la fin de l'utilisation.

La mémoire

malloc, calloc et realloc

malloc

Profile : `void* malloc(size_t size);`

Alloue ce qui lui est passé comme argument en octets mais n'effectue aucune initialisation.

Exemple :

```
void* p1 = malloc(256); // 256 bytes are allocated
int* p2 = (int*)malloc(4 * sizeof(int)); // 4 * sizeof(int) bytes
     are allocated
struct A* p3 = (struct A*)malloc(2 * sizeof(struct A)); // 2 *
    sizeof(struct A) bytes are allocated
p2[0]; // Access to uninitialized memory !
p3[0].a; // Access to uninitialized memory !
```

La mémoire

malloc, calloc et realloc

calloc

Profile : `void* calloc(size_t nmemb, size_t size);`

Alloue *nmemb* ∗ *size* octets et les initialise à zéro.

Exemple :

```
void* p1 = calloc(1, 256); // 256 bytes are allocated
int* p2 = (int*)calloc(4, sizeof(int)); // 4 * sizeof(int) bytes
    are allocated
struct A* p3 = (struct A*)calloc(2, sizeof(struct A)); // 2 *
    sizeof(struct A) bytes are allocated
assert(p2[0] == 0); // true
assert(p3[0].a == 0); // true
assert(p3[1].b == 0); // true
assert(p3[1].str[0] == 0); // true
```

La mémoire

malloc, calloc et realloc

realloc

Profile : `void* realloc(void* ptr, size_t size);`

Change la taille du bloc de mémoire pointé par `ptr` en `size` octets.

- Si `size` > taille de `ptr` : La mémoire pointée par le pointeur retourné par `realloc` sera de taille `size`. Le contenu de `ptr` est garanti d'être copié mais la mémoire ajoutée ne sera pas initialisée
- Si `size` < taille de `ptr` : Le contenu de `ptr` sera copié jusqu'à `size` octets, le reste du contenu de ptr sera ignoré. La taille de la mémoire pointée par la valeur de retour sera donc `size`.
- Si `ptr` est `NULL` : Cela aura le même effet que malloc.

La mémoire

malloc, calloc et realloc

realloc

- Si `size` est 0 et `ptr` n'est pas `NULL` : Cela aura le même effet que free.

N.B. :

- Sauf si `ptr` est `NULL`, il doit avoir été renvoyé par un appel antérieur à `malloc()`, `calloc()` ou `realloc()`.
- L'accès et/ou l'écriture au pointeur a passé à `realloc` après l'appel est un comportement indéfini

La mémoire

malloc, calloc et realloc

Exemple :

```
int* p1 = (int*)calloc(4, sizeof(int)); // 4 * sizeof(int) bytes
p[0] = 1; p[1] = 2; p[2] = 3; p[3] = 4;
int* p2 = (int*)realloc(p1, 6 * sizeof(int));
assert(p2[0] == 1); // true
assert(p2[1] == 2); // true
assert(p2[5] == 6); // Access to uninitialized memory !
assert(p1[5] == 6); // U.B !
```

La mémoire

malloc, calloc et realloc

Exemple :

```
int* p3 = (int*)realloc(p2, 3 * sizeof(int));
assert(p3[0] == 1); // true
assert(p3[1] == 2); // true
assert(p3[3] == 3); // U.B!
assert(p2[3] == 3); // U.B!

int* p4 = (int*)realloc(p3, 0); // Equivalent to free(p3)
void* p5 = realloc(NULL, 8); // Equivalent to malloc(8)
void* p6 = realloc(NULL, 0); // Equivalent to malloc(0)
```

La mémoire

malloc, calloc et realloc

N.B. :

Le standard C ne dit rien quand 0 est passé à `malloc` (le comportement est spécifique au système d'exploitation).

- Sous Linux : `malloc(0)` renvoie `NULL`
- Sous Windows : `malloc(0)` renvoie un pointeur valide sur lequel free pourrait être appelé.

La mémoire

Les pointeurs

Définition

Un pointeur est une variable qui contient l'adresse d'une région de mémoire. Un pointeur peut contenir une adresse valide ou non (Exemple : le pointeur `NULL`).

Les pointeurs font généralement 4 ou 8 octets en fonction de l'architecture du processeur (32 ou 64 bits) :

- Avec un pointeur 32 bits, nous avons 4 Go de mémoire adressable.
- Avec un pointeur 64 bits, nous avons environ 17 milliards de Go de mémoire.

La mémoire

Les pointeurs

Syntaxes possibles

Toutes les syntaxes suivantes sont valides. Cependant, les deux premiers sont les plus courants :

```
Typename* ptr_name; // Recommended
Typename *ptr_name; // Recommended
Typename*ptr_name;
Typename * ptr_name;
```

La mémoire

Les pointeurs

Comment obtenir l'adresse d'une variable

Pour obtenir l'adresse d'une variable, l'opérateur & doit être utilisé.

Exemple :

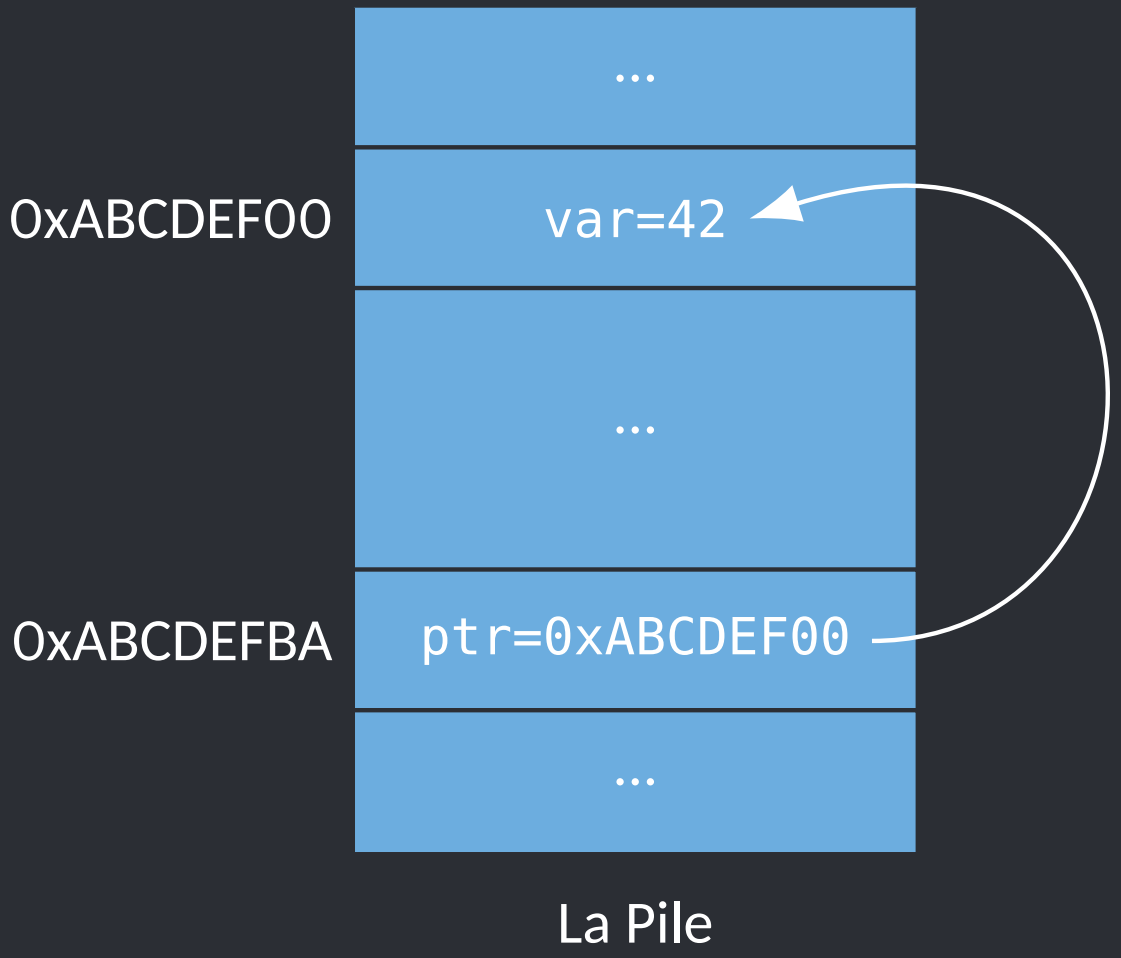
```
Typename var = ...;
Typename* ptr = &var;
```

Example :

```
int var = 42;
int* ptr = &var;
```

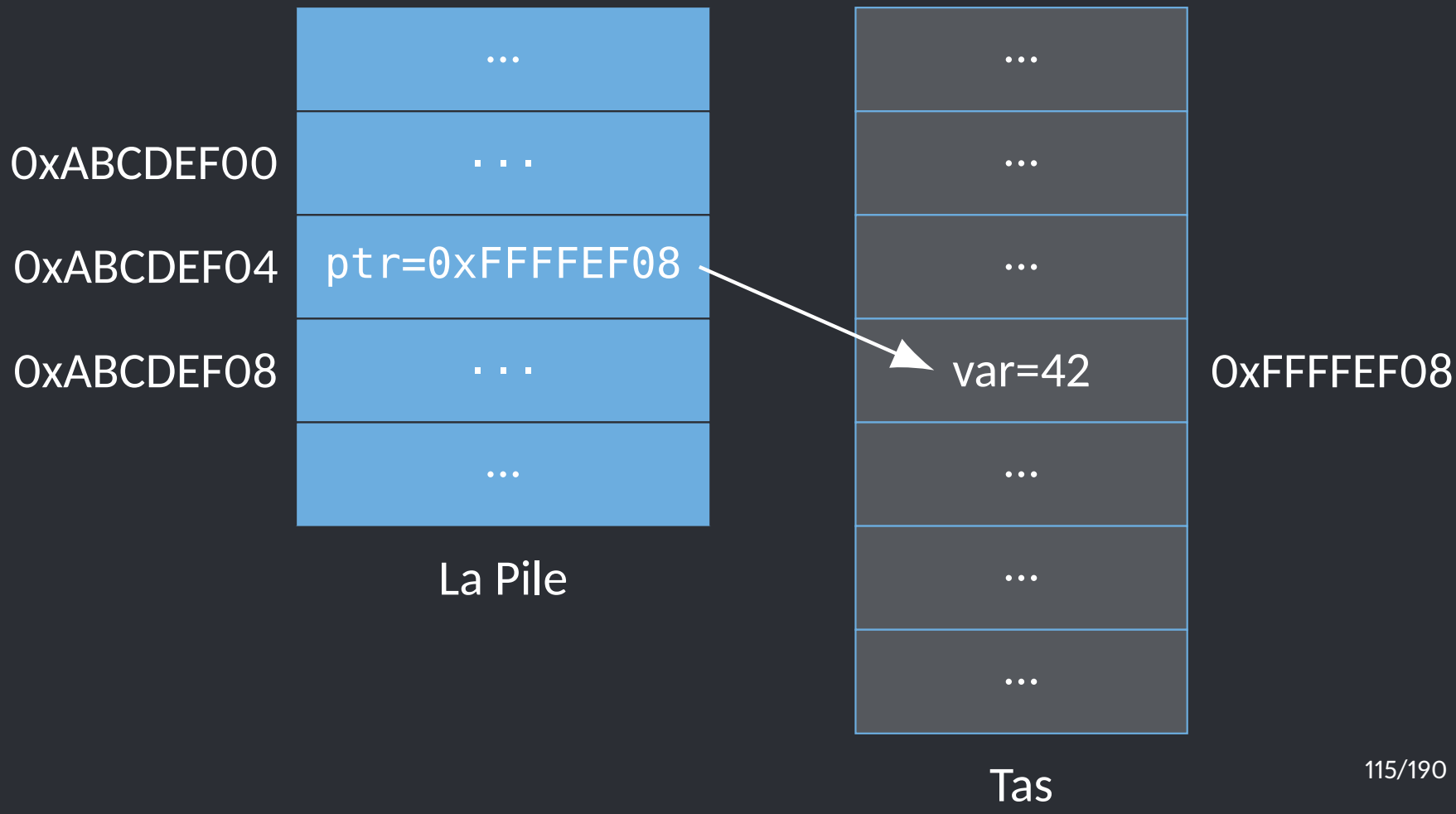
La mémoire

Pointeur et variable sur la pile



La mémoire

Pointeur sur la pile et variable sur le tas



La mémoire

Les pointeurs

Attention :

Lorsque l'opérateur & est utilisé sur un tableau, il renvoie une adresse vers le premier élément de ce tableau et non un pointeur vers la première adresse du tableau.

Example :

```
int arr[] = {1, 2, 3};
int* ptrCpy = arr;
int* ptrArr = &arr;
assert(arr == &arr); // true
assert(ptrCpy == arr); // true
assert(ptrCpy == ptrArr); // true
assert(&ptrCpy == arr); // false
assert(&ptrCpy == &ptrArr); // false
```

La mémoire

Les pointeurs

Comment déréférencer un pointeur

Déréférencer un pointeur permet d'accéder à la valeur pointée par le pointeur. Pour cela, l'opérateur * doit être utilisé.

Example :

```
TypeName var = ...;
TypeName* ptr = &var;
assert(*ptr == var); // true
*ptr = ...;          // changes the content of var
```

La mémoire

Les pointeurs

ATTENTION :

- Déréférencer un pointeur NULL est un comportement indéfini.
- Déréférencer un pointeur sur lequel `free()` a été appelé est un comportement indéfini.
- En général, le déréférencement d'un dangling pointer entraînera une corruption de pile ou un plantage du programme.

Définition

Un dangling pointer [13] est un pointeur pointant vers une adresse mémoire valide qui a été libérée ou détruite.

13. traduction littérale : « pointeur pendouillant » ou « pointeur sautillant »

La mémoire

Les pointeurs

Exemple 1 :

```
int main() {
    int* ptr = NULL;
    {
        int x = 42;
        ptr = &x;
    }
    printf("content of ptr : %d\n", *ptr);
    *ptr = 3;
}
```

La mémoire

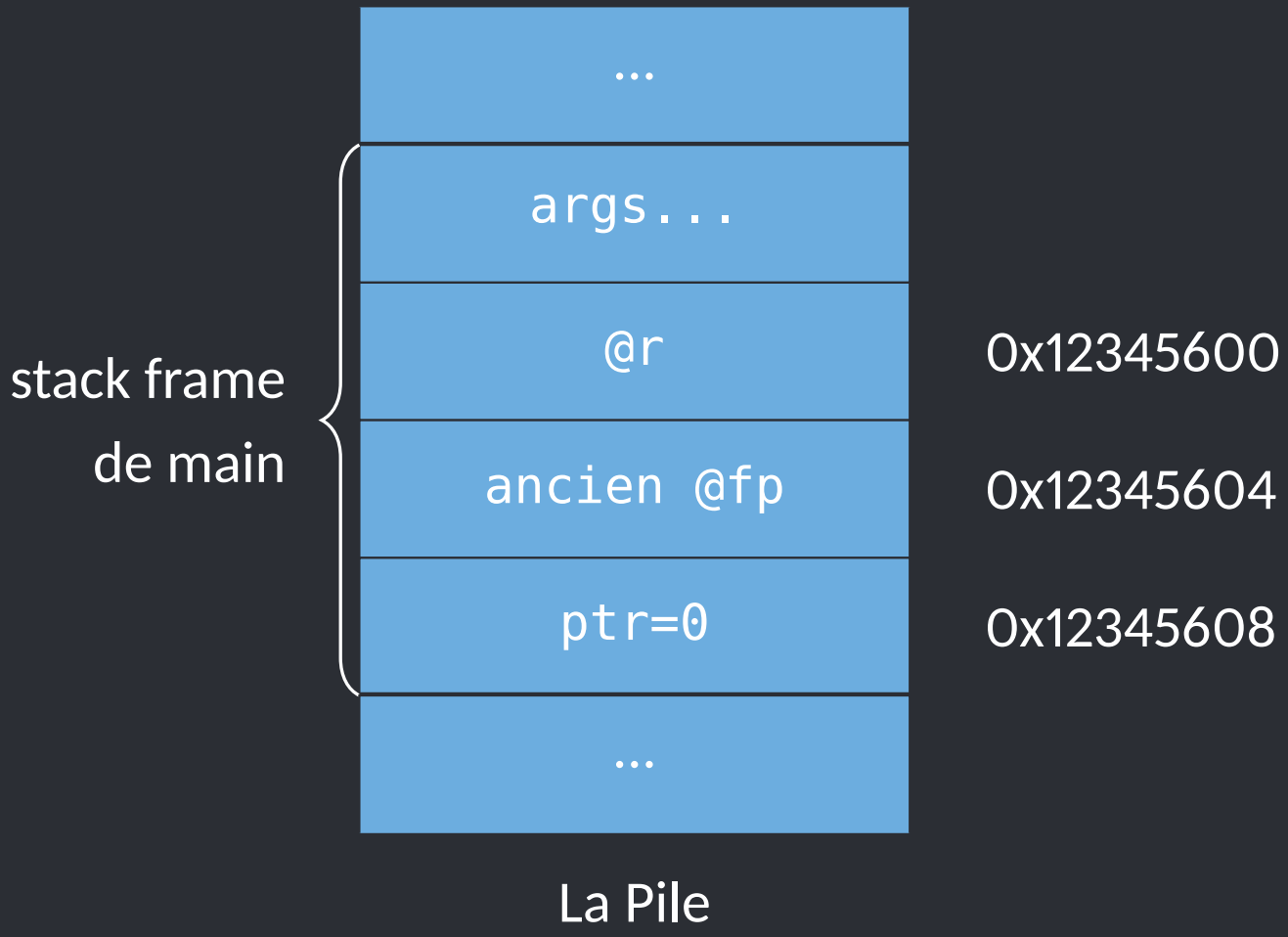
Dangling pointers

Solution 1 :

```
int main() {
    int* ptr = NULL;
    {
        int x = 42;
        ptr = &x;
    }
    // x is out of scope (it’s popped out of the stack)
    printf("content of ptr : %d\n", *ptr); // ptr is now a
                                           // dangling pointer
    *ptr = 3; // this will either corrupt the stack or
              // segfault
}
```

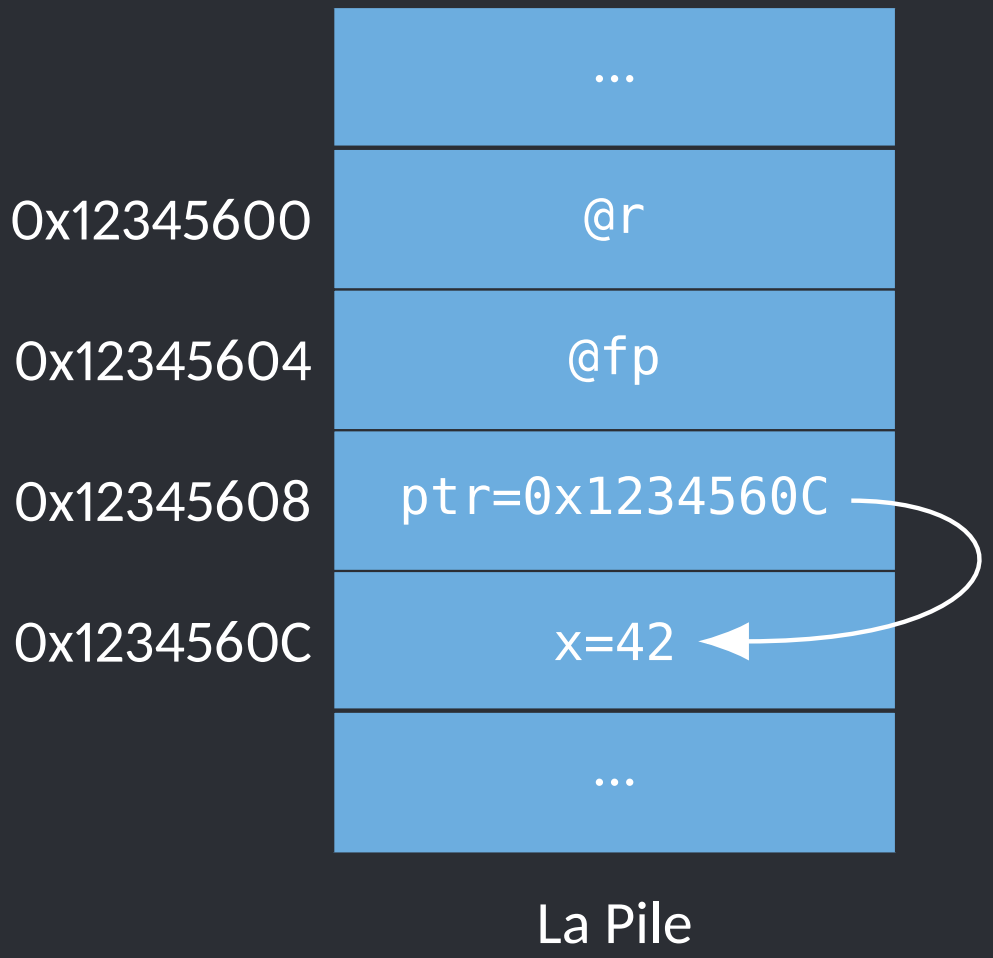
La mémoire : Dangling pointers

```
int* ptr = NULL;
```



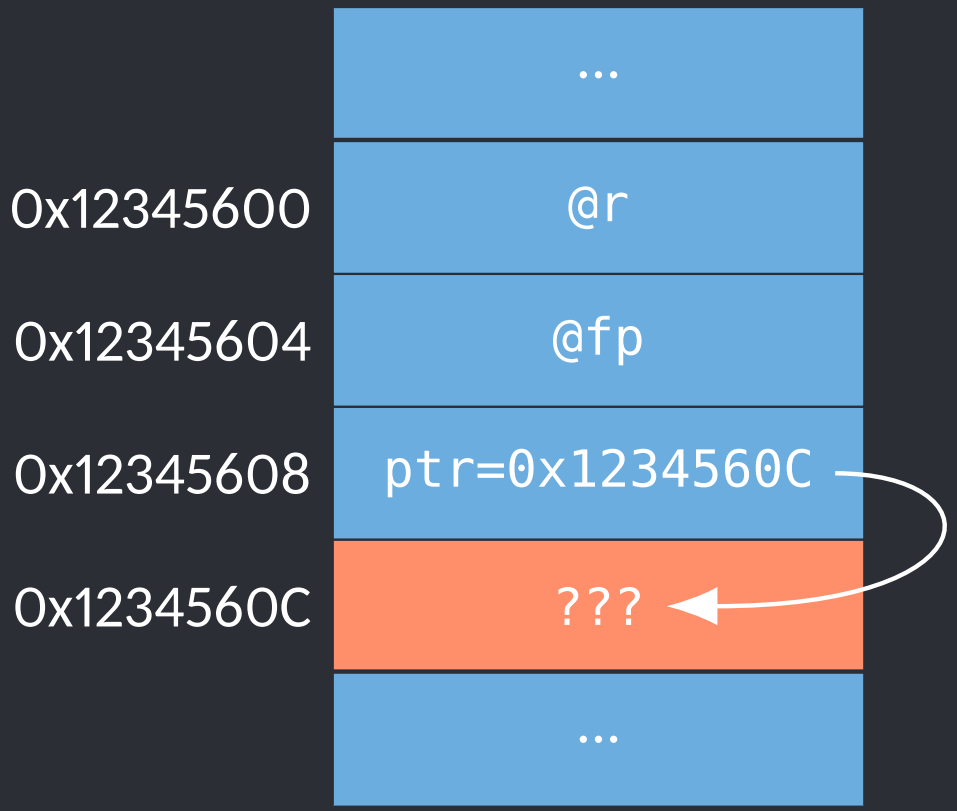
La mémoire : Dangling pointers

```
int x = 42;
ptr = &x;
```



La mémoire : Dangling pointers

```
{
    int x = 42;
    ptr = &x;
} // x is out of scope (it's popped out of the stack)
```



La mémoire

Les pointeurs

Exemple 2 :

```
int main()
{
    int* tab = malloc(4 * sizeof(int));
    // Init tab to 1, 2, 3, 4
    free(tab);
    tab[0] = 3;
}
```

La mémoire

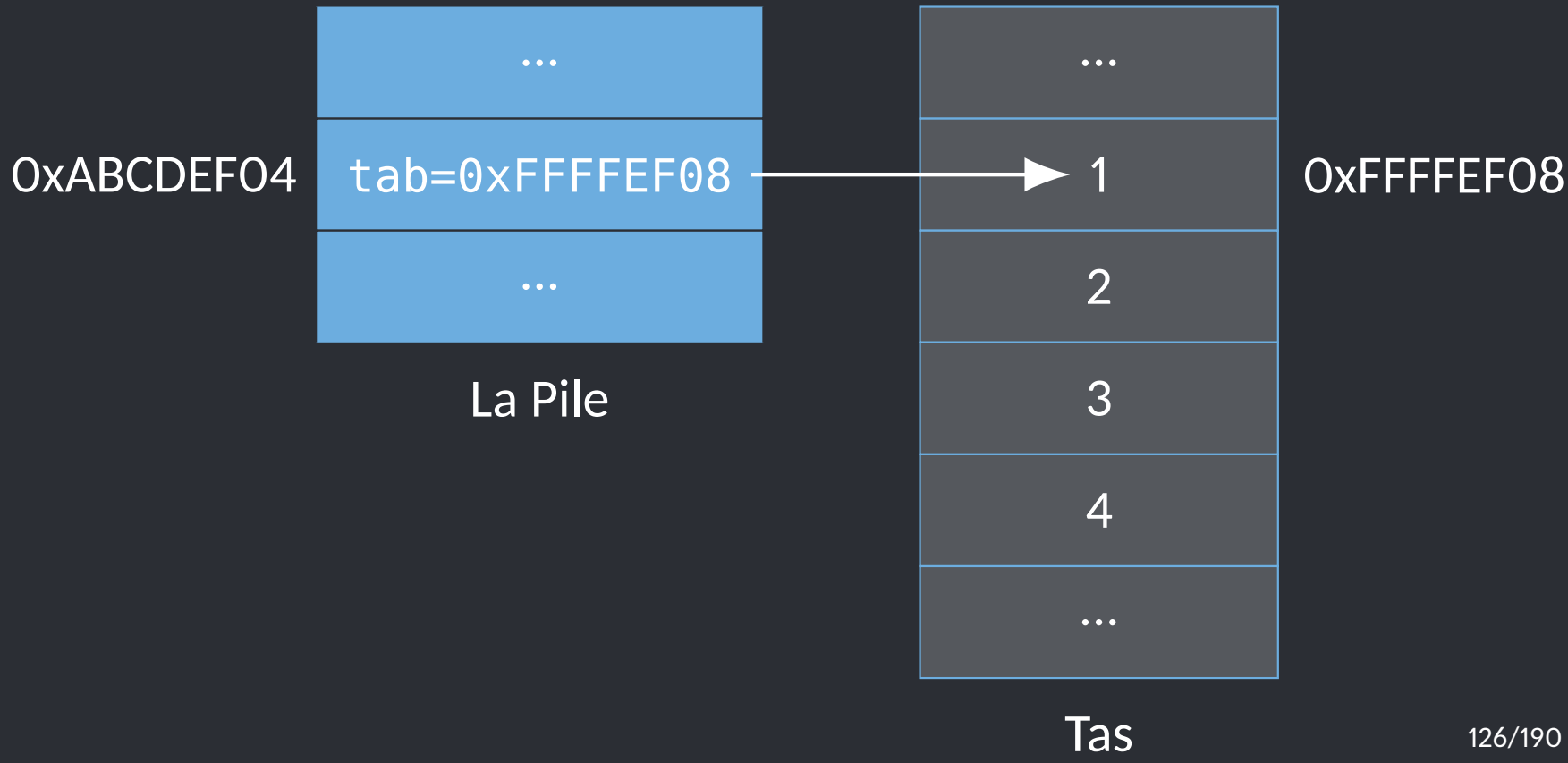
Les pointeurs : Dangling pointers

Solution 2 :

```
int main()
{
    int* tab = malloc(4 * sizeof(int));
    // Init tab to 1, 2, 3, 4
    free(tab);
    // tab is now dangling
    tab[0] = 3; // will crash
}
```

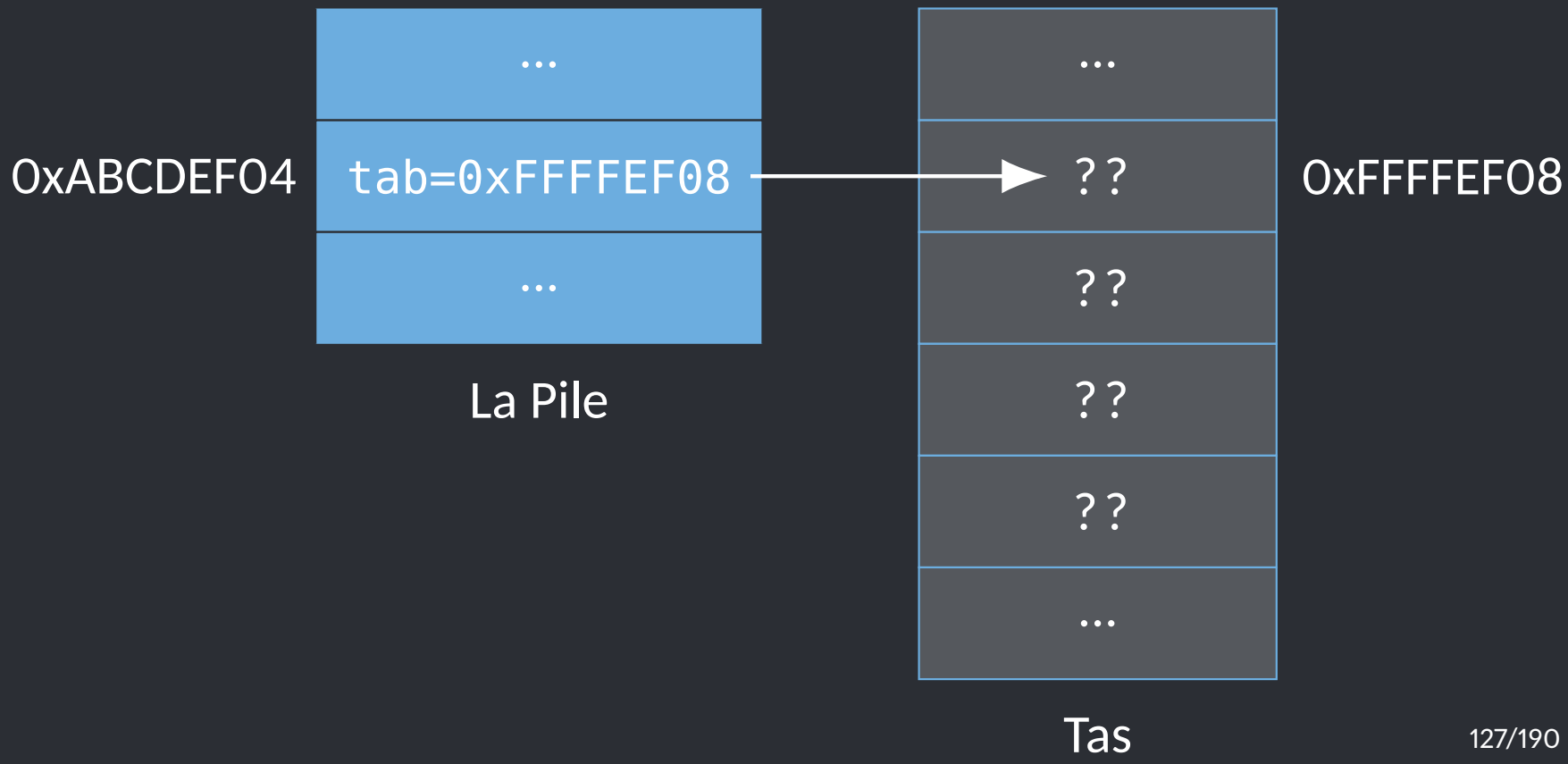
La mémoire

```
int* tab = malloc(4 * sizeof(int));
// Init tab to 1, 2, 3, 4
```



La mémoire

```
free(tab);
// tab is now dangling
```



La mémoire

Dangling pointers : Solution Possibles

Solution 1 :

Une façon de résoudre ce problème consiste à affecter le pointeur à NULL après l'avoir libéré.

```
free(tab);
tab = NULL; // check if tab is NULL before we access it later
```

Problème avec la solution ci-dessus

Cette solution n'est pas parfaite si on libére `tab` dans une autre fonction et on l'affecte à `NULL`, seule la copie locale du pointeur sera `NULL`.

La mémoire

Dangling pointers : Solution Possibles

Solution 2 :

Une meilleure façon de résoudre ce problème est de passer un pointeur vers le pointeur qui doit être libéré.

```
void myfree(void** pptr)
{
    if (pptr && *pptr) {
        free(*pptr);
        *pptr = NULL;
    }
}
```

Cette solution n'est pas parfaite non plus

La mémoire

Les pointeurs

Accès aux membres d'un struct/union

Il existe deux façons d'accéder à un champ struct via un pointeur :

```
struct StructName* ptr = &var;
ptr->field1_name = ...;
ptr->field2_name[0] = ...; // supposing that field2_name is an
    array this will change the first element of field2_name
ptr->field1_name; // accessing field1_name in the struct var
ptr->field2_name[0]; // acessing the first element in field2_name
     in the struct var
```

La mémoire

Les pointeurs

Accès aux membres d'un struct/union

La deuxième façon est verbeuse et donc déconseillée :

```
struct StructName* ptr = &var;
(*ptr).field1_name = ...; // *ptr is between parentheses because
    "*" have a lower percedance level than "."
(*ptr).field2_name[0] = ...; // supposing that field2_name is an
    array this will change the first element of field2_name
(*ptr).field1_name; // accessing field1_name in the struct var
(*ptr).field2_name[0]; // acessing the first element in
    field2_name in the struct var
```

La mémoire

Arithmétique des pointeurs

Explication

Un pointeur n'est qu'une adresse mémoire. Cette adresse est une valeur numérique. Par conséquent, on peut effectuer des opérations arithmétiques sur un pointeur comme on peut le faire sur des valeurs numériques.
Il existe quatre opérateurs arithmétiques qui peuvent être utilisés sur les pointeurs : `++`, `--`, `+` et `-`.

La formule

Pour un pointeur `ptr` avec le type `Typename`, l'expression `ptr+step` ajoutera `step*sizeof(Typename)` octets au pointeur `ptr`.

La mémoire

Arithmétique des pointeurs

Exemple

```
int* ptr = ...;
ptr1 = ptr + 2; // will add ’8’ bytes to the current address.
                // *ptr1 is equivalent to ptr[2]
ptr1++; // will add ’4’ bytes to ptr1.
        // *ptr1 is equivalent to ptr[3]
ptr2 = ptr1 - 3; // will substract ’12’ bytes from ptr1.
                 // *ptr2 equivalent to ptr1[-3] or ptr[0]
```

Lorsque vous utilisez l’arithmétique du pointeur, veillez à ne pas dépasser la taille allouée.

Pointeurs de fonction

Explication :

Comme les variables, les fonctions existent également en mémoire, nous pouvons donc avoir des pointeurs vers elles.
Cela peut être utile pour déterminer la fonction à exécuter au moment de l'exécution. On peut imiter le comportement des fonctions virtuelles fournies dans des langages tels que Java ou C++ en utilisant cette fonctionnalité.

Syntaxe :

```
ReturnType (*funcPtrName)(ArgType1, ArgType2, ..., ArgTypeN);
```

Pointeurs de fonction

Exemple :

```
// includes ...
int max(int a, int b) { return a > b ? a : b; }
int min(int a, int b) { return a < b ? a : b; }
typedef int (*MyFuncType)(int, int); // MyFunc is a function
                                     // pointer type
int main() {
    int a = 5, b = 6;
    srand(time(NULL));
    int (*func)(int, int) = max; // Equivalent to
                                 // MyFuncType func = max;
    if (rand()%2) {
        func = &min; // same as func = main
    }
    return func(a, b);
}
```

Pointer casting

Comme nous l'avons vu précédemment dans la partie Arithmétique des pointeur, le type de pointeur n'est utilisé que pour déterminer comment la mémoire est accédée et interprétée. Ainsi, nous pouvons convertir un pointeur d'un type à un autre. Cependant, des précautions doivent être prises lors de cette opération, car différents types ont des exigences d'alignement différentes. D'une manière générale, la conversion d'un type qui a une grande taille à un objet qui a une taille plus petite est acceptable (par exemple la conversion de `int*` en `char*`).

La conversion vers un type qui a une exigence d'alignement stricte à partir d'un type qui a une exigence d'alignement moins stricte est interdite. Au mieux, cela peut entraîner une dégradation des performances ou des valeurs erronées. Au pire, il peut planter tout le programme.

Pointer casting

Syntaxe :

Le cast d'un pointeur du type A au type B peut se faire de cette façon :

```
A* ptr1 = ....;
B* ptr2 = (B*)ptr1;
```

Exemple :

```
unsigned int a = 3721182122;
unsigned int* ptrA = &a;
char* bytes = (char*)ptrA;
```

Pointer casting

Exemple :

Figure – Représentation mémoire de la variable `a`, tel qu'il est interprété par le pointeur `ptrA`

Pointer casting

Exemple :

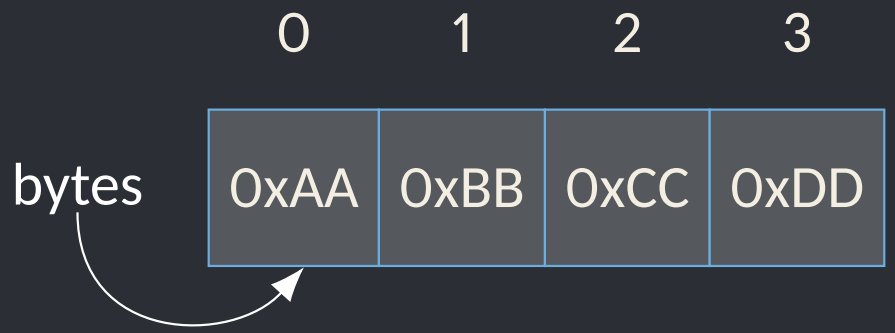


Figure – Représentation mémoire de la variable `a`, tel qu'il est interprété par le pointeur `bytes` (cas d'une machine petit-boutienne)

Pointer casting

Exemple :

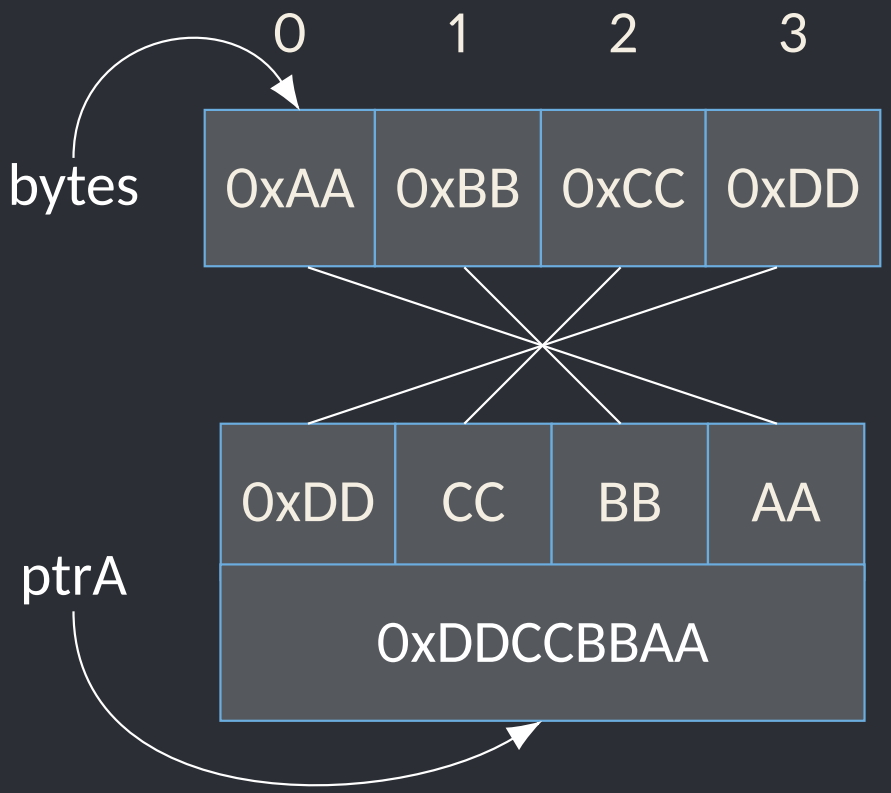


Figure – La représentation de `bytes` dans le cas d'une machine petit-boutienne

Pointer casting

Exemple :

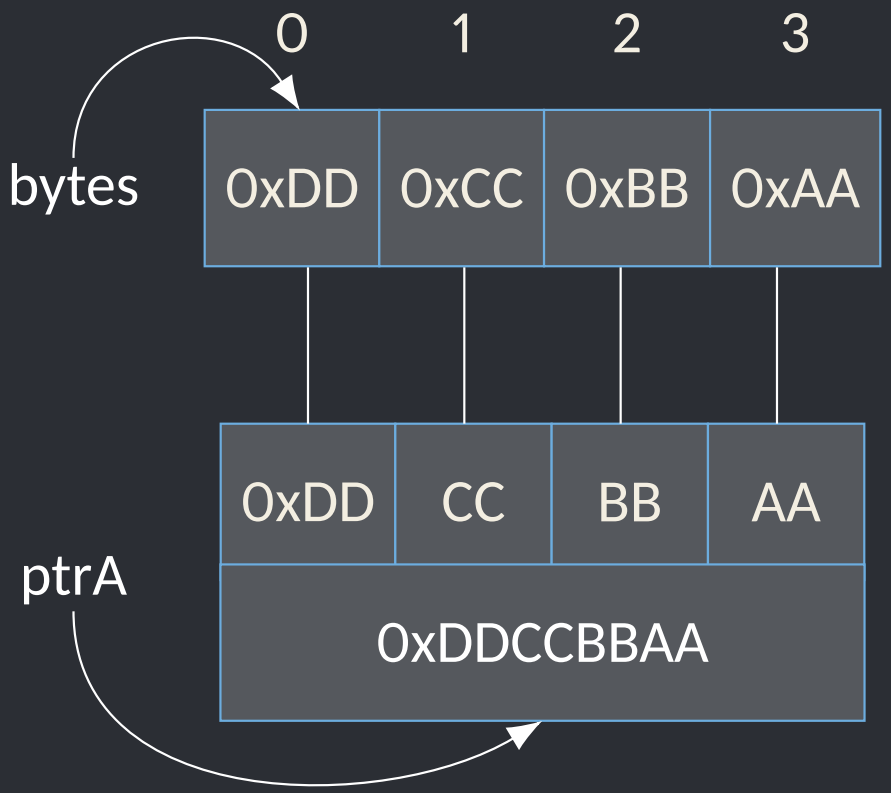
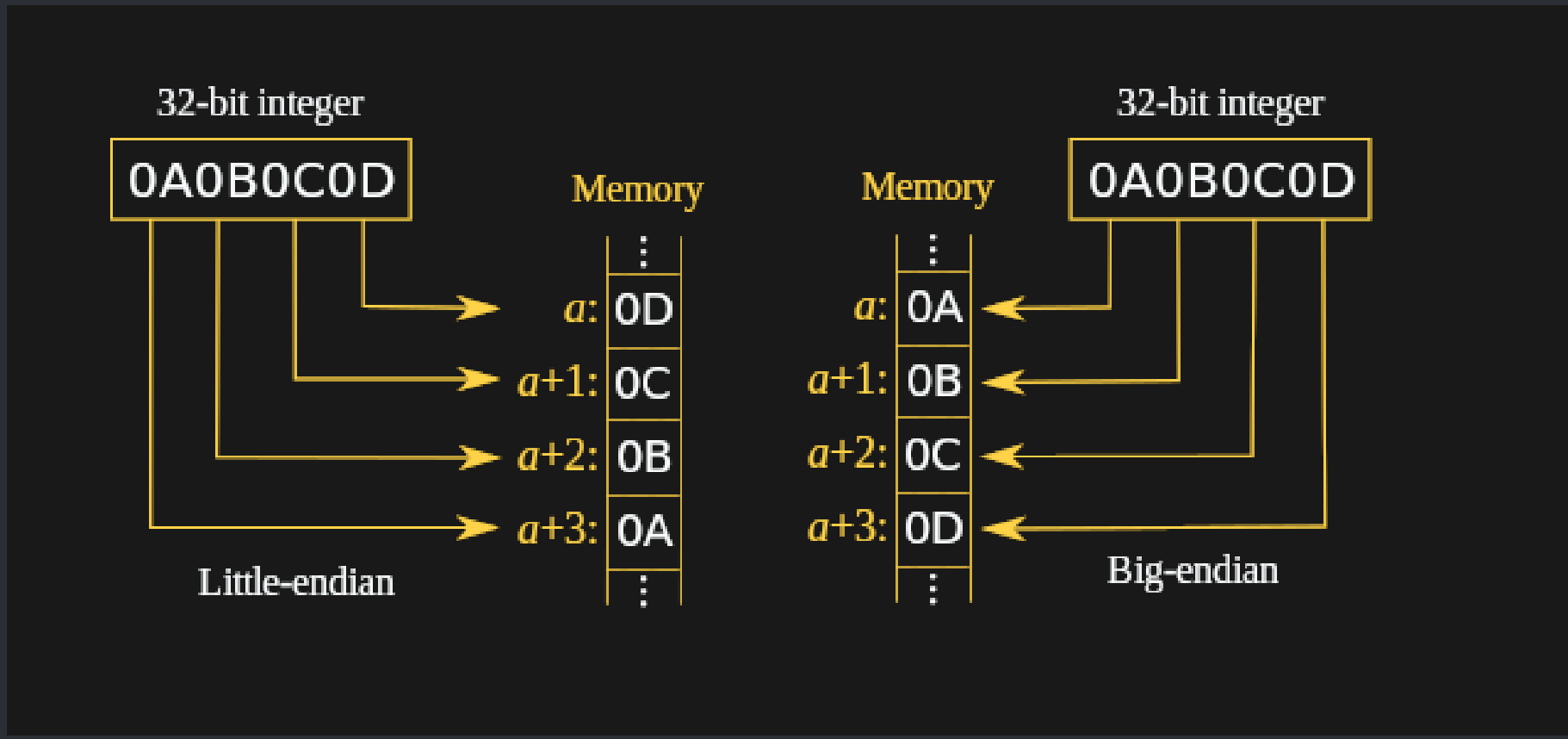


Figure – La représentation de `bytes` dans le cas d'une machine gros-boutienne

Le Boutisme


32-bit integer
0A0B0C0D
Memory
a: 0D
a+1: 0C
a+2: 0B
a+3: 0A
Little-endian
Memory
32-bit integer
0A0B0C0D
a: 0A
a+1: 0B
a+2: 0C
a+3: 0D
Big-endian

Le pointeur `void*`

Le pointeur `void*`

Le cast depuis et vers le pointeur `void*` à partir de tout autre type de pointeur se fait implicitement dans le langage C.
Chaque pointeur type peut être transformé depuis et vers le pointeur `void*`. Cependant, faire de l'arithmétique des pointeurs ou accéder aux éléments en utilisant le pointeur `void*` est interdit. Il doit y avoir une conversion vers un type de pointeur différent que `void*` avant d'accéder ou d'effectuer de l'arithmétique.

Le pointeur `void*`

Example :

```
unsigned int* ptrA = ...;
void* ptr2 = (void*)ptrA;

ptr2 + 1; // Invalid
ptr2 + 2; // Invalid
ptr[1];   // Invalid

int* ptr3 = (int*)ptr2;
ptr3 + 1; // valid
ptr3[1];  // valid

char* ptr4 = (char*)ptr2;
ptr3 + 1; // valid
ptr3[1];  // valid
```

Chapitre 3 : La langage C

Section 5 : Le keyword static et extern

le mot-clé static

Définition :

Le mot-clé static se comporte différemment selon la façon dont il est utilisé :

- Une variable déclaré statique à l'intérieur d'une fonction conserve sa valeur entre les appels. (La variable agit donc comme si elle était globale)
- Une variable globale statique ou une fonction statique n'est "accessible" que dans le fichier dans lequel elle est déclarée. (un peu comme le mot-clé private en Java)

le mot-clé static

le mot-clé static : Example 1

```
// includes...
void toto()
{
    static int g = 5;
    int i = 5;
    g += 5;
    i += 5;
    printf("%d - %d\n", g, i);
}

int main() {
    for (int j = 0; j < 5; j++)
        toto();
}
```

le mot-clé static

le mot-clé static : Solution 1

- 1iere itération : `10 - 10`
- 2ieme itération : `15 - 10`
- 3ieme itération : `20 - 10`
- 4ieme itération : `25 - 10`
- 5ieme itération : `30 - 10`

le mot-clé static

le mot-clé static : Example 2

```
// file1.c
static int pix10 = 314;

static void foo()
{
    // ...
}

// mainc.c
int main()
{
    pix10 = 628;
    foo();
}
```

le mot-clé static

le mot-clé static : Solution 2

```
// file1.c
static int pix10 = 314; // this variable is only visible in this
                        // translation unit (file1.c)
// this function can only be used in this translation unit
// foo is only visible in 'file1.c'
static void foo()
{
    // ...
}

// mainc.c
int main()
{
    pix10 = 628; // undeclared variable pix10
    foo();       // implicit declaration of function 'prv_func'
}
```

le mot-clé extern

Définition :

le mot-clé extern est utilisé pour étendre la visibilité des variables / fonctions.

Quelques remarques :

- L'utilisation d'extern avec des fonctions est redondant.
- L'utilisation de `extern type var` amènera la variable var dans le fichier .h ou .c courant à partir d'un autre pendant l'édition du lien.

Chapitre 3 : La langage C

Section 6 : Les opérateurs et ordre d'évaluation

Priorité des opérateurs

Table des opérateurs : [14]

Priorité	Les opérateurs
1	++ -- () [] . ->
2	++ -- + - ! * & sizeof
3	* / %
4	+ -
5	<< >>
6	<<= >>=
7	== !=
8	&

Table – Priorité des opérateurs en C

14. pour plus de détails :
https://en.cppreference.com/w/c/language/operator_precedence

Priorité des opérateurs

Table des opérateurs : [15]

Priorité	Les opérateurs
9	^
10	|
11	&&
12	||
13	? :
14	= += -= *= /= %= <<= >>= &= ^= |=
15	,

Table – Priorité des opérateurs en C

15. pour plus de détails :
https://en.cppreference.com/w/c/language/operator_precedence

Les opérateurs et ordre d'évaluation

Pouvez-vous prédire la valeur de `i` sur chaque instruction sachant que `i` est initialement 0 :

```
i = (i++);      // i = 0 intially
i = ++i + i++;  // i = 0 intially
i = i++ + 1;    // i = 0 intially
f(++i, ++i);    // i = 0 intially
```

Les opérateurs et ordre d'évaluation

```
i = (i++);          // undefined behavior
i = ++i + i++;      // undefined behavior
i = i++ + 1;        // undefined behavior
f(++i, ++i);        // undefined behavior
f(i = -1, i = -2);  // undefined behavior
f(i, i++);          // undefined behavior
a[i] = i++;         // undefined bevahior
```

Explication :

Pour une expression de type `ExpA op ExpB` ou `op` n'est pas `&&` ou `||` ou `,` l'ordre d'evaluation de `ExpA` et `ExpB` est indéfini (c'est-à-dire que `ExpA` peut être évalué avant `ExpB` ou vice versa)

Example :

`res = foo() + bar()` foo peut être appelé avant bar ou vice versa

Les opérateurs et ordre d'évaluation

Opérateurs qui garantissent l'ordre

Les opérateurs `&&`, `||` et `,` garantissent que l'expression à gauche est évaluée avant l'expression à droite.
L'opérateur ternaire (`?:`) garantit que la condition est évaluée en premier. Si la condition est vraie, l'expression de gauche à `:` est évaluée, sinon l'expression à droite de `:` est évaluée.

Example :

```
int res = f1() && f2(); // f1() is called before f2()
int res = f1() || f2;   // f1() is called before f2()
int res = f1(), f2();   // f1() is called before f2()
int res = cond() ? f1() : f2() // cond() is called then f1()
                               // is called if the result of
                               // cond() is true otherwise f2()
                               // is called
```

Les opérateurs et ordre d'évaluation

Pour éviter complètement le problème :

Une règle de base qui devrait nous protéger de ce genre de comportement indéfini est de ne pas modifier et utiliser la même variable dans la même instruction, c'est-à-dire éviter de changer et de lire à partir de la même variable avant un point virgule ';'.

Example :

```
a[i] = i;             // OK
a[i] = j;             // OK
a[i++] = i;           // undefined behavior
a[i] = i++;           // undefined behavior
int p = i++ && i--; // OK
int p = i++ || i--; // OK
int p = i++, i--;   // OK
```

Chapitre 4 : Les outils

Chapitre 4 : Les outils

Section 1 : Introduction

Introduction

En C, certaines erreurs d'exécution sont très difficiles parfois même impossibles à déboguer. Que pouvons-nous faire alors ? Les outils à l'aide !

Exemple d'erreurs d'exécution étranges

- `Segmentation fault (core dumped)`
- `*** stack smashing detected ***: terminated`
- `stack around the variable .. was corrupted`
- `memory heap corruption`
- `munmap_chunk(): invalid pointer : 0x0fa1ca5a`

Grâce à l'outillage, nous pourrons résoudre ces problèmes !

Chapitre 4 : Les outils

Section 2 : Compilateur : GCC/Clang

Compilateur

- Connaître le compilateur et ce qu'il peut faire est nécessaire, cela peut vous faire gagner beaucoup de temps.
- Il existe de nombreuses options et indicateurs du compilateur qui facilitent le débogage.
- Connaître ses différentes options et comment les manipuler est nécessaire !

Compilateur

Les flags d'optimisation :

- `00` : Aucune optimisation (par défaut) ; génère du code non optimisé mais a le temps de compilation le plus rapide.
- `01` : Optimisation modérée ; optimise raisonnablement bien mais ne dégrade pas le temps de compilation de manière significative.
- `02` : Optimisation complète ; génère du code hautement optimisé et a le temps de compilation le plus lent.
- `03` : Optimisation complète comme en `-02` ; utilise aussi un inlining automatique plus agressive et tente de vectoriser les boucles.

Compilateur

Les flags d'optimisation :

- `Os` : Optimise l'utilisation de l'espace (code et données) du programme résultant.
- `Ofast` : Ne respecte pas strictement les standards. Active toutes les optimisations `-O3`. Il permet également certaines optimisations non conformes à la norme, il doit donc être utilisé avec prudence.

Compilateur

Les flags d'optimisation :

- `0g` : Optimise l'expérience de débogage. `-0g` devrait être le niveau d'optimisation de choix pour le débogage, offrant un niveau d'optimisation raisonnable tout en maintenant une compilation rapide et une bonne expérience de débogage. C'est mieux que `-00` pour produire du code déboguable car certaines passes du compilateur qui collectent des informations de débogage sont désactivées à `-00`.

Compilateur

L'informations de débogage : debugging symbols

Pour activer la génération d'informations de débogage, vous devez fournir `-g` au moment de la compilation. Cet indicateur est nécessaire pour le débogage et rendra l'utilisation d'un débogueur (comme gdb) beaucoup plus facile. Il montrera exactement où l'erreur s'est produite (nom du fichier et numéro de ligne)

Ce qu'il faut retenir

Lors du débogage du code, essayez d'utiliser `-Og` et `-g` chaque fois que c'est possible car cela facilite le débogage. Si ce n'est pas possible pour une raison quelconque, n'utilisez aucun flag `-O`. de cette façon, le compilateur adoptera par défaut `-O0`, ce qui est toujours pas mal pour le débogage.

Compilateur

Autres options utiles

Activer plus de warning

- `-pedantic` : Produit un avertissement lorsque les normes C ne sont pas respectées. Il produit également des avertissements lorsque des extensions non standard sont utilisées.
- `-Wall` : Active tous les avertissements.
- `-Wextra` : Active encore plus d'avertissements qui ne sont pas activés par `-Wall`.
- `-Werror` : Transforme tous les avertissements en erreurs.

Compilateur

Autres options utiles

Utilisation d'un sanatizer : `-fsanatize=option`

Les options peuvent être :

- `address` : toutes sortes de fuites et de débordements peuvent être détectés avec précision, détection de niveau d'adresse, tas et pile.
- `leak` : il ne détecte que les fuites et débordement mémoire au niveau du tas, mais pas la pile.
- `undefined` : active un détecteur de comportement indéfini. Divers calculs sont instrumentés lors de l'exécution
- `all` : active toutes les options mentionnées ci-dessus et plus [16]

16. pour plus de détail

Chapitre 4 : Les outils

Section 3 : Débogueur : GDB

Débogueur : GDB

Définition :

Un débogueur ou un outil de débogage est un programme informatique utilisé pour tester et déboguer d'autres programmes. L'utilisation principale d'un débogueur est d'exécuter le programme cible dans des conditions contrôlées qui permettent au programmeur de suivre ses opérations en cours et de surveiller les changements dans les ressources informatiques qui peuvent indiquer un code défectueux.

Un débogueur vous aidera à détecter les erreurs mais ne les corrigera pas à votre place. C'est le travail des programmeurs de corriger le code

Débogueur : GDB

Débogueurs courants utilisés pour le langage C/C++

- GDB : Débogueur GNU
- LLDB : est le composant de débogage du projet LLVM.

Dans ces diapositives, nous ne couvrirons que gdb, car c'est le plus couramment utilisé. Cependant, les commandes devraient être les mêmes pour LLDB

Débogueur : GDB

Utilisation de GDB

1. Afin d'utiliser correctement GDB, le compilateur doit être compilé avec le flag `-g`. Si ce n'est pas le cas, le débogueur aura du mal à trouver les noms de fichiers et les numéros de ligne.
2. Après avoir compilé avec les bonnes options, vous pouvez lancer GDB en utilisant la commande : `gdb ./nom_executable` .

Si vous voulez lancer GDB avec une interface utilisateur textuelle, utilisez l'option `-tui`. Cette option vous permettra de voir le code source, les points d'arrêt, etc. lors du débogage.

Débogueur : GDB

Attention :

Si vous voyez le message suivant :

`(No debugging symbols found in ...)`

Ca signifie que vous n'avez pas compilé avec `-g`. Le débogage de code sera plus difficile et parfois même inutile.

```
omar@Omar:~$ gcc -g main.c
omar@Omar:~$ gdb ./a.out
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./a.out...
(gdb)
```

Figure – Un programme chargé avec succès par GDB

Débogueur : GDB

Commandes GDB

- `file executable` : spécifie le programme que vous souhaitez déboguer.
- `run` : exécute le programme chargé. des arguments peuvent également être fournis.
- `list` : afficher quelques lignes du code source.
- `break filename:linenumber` : définit un point d'arrêt sur le numéro de ligne donné dans le fichier source. L'exécution s'arrêtera avant que cette ligne ne soit exécutée.
- `info breakpoints` : afficher des informations sur tous les points d'arrêt, y compris son numéro.
- `delete id` : supprimer le point d'arrêt numéroté `id`
- `continue` : continuera à exécuter le programme à nouveau, après l'avoir arrêté.

Débogueur : GDB

Commandes GDB

- `info locals` : Afficher toutes les variables locales
- `info args` : Afficher tous les arguments de la fonction en cours d'exécution
- `backtrace` : Imprime la pile d'appels jusqu'au point d'arrêt actuel. Utile pour savoir quelles fonctions sont appelées avant qu'un crash ne se produise
- `backtrace full` : Comme backtrace mais affiche aussi les variables locales
- `print expression` : affichera la valeur de l'expression, qui pourrait être juste un nom de variable.
- `set varname=value` : change la valeur d'une variable et continue l'exécution avec la valeur modifiée.
- `disas` : affiche le code assembleur de la fonction actuelle.

Débogueur : GDB

Commandes GDB

- `step` : exécutera la ligne du code courante, puis arrêtera à nouveau l'exécution avant la ligne suivante.
- `next` : est similaire à "step", sauf que si la ligne qui va être exécutée est un appel de fonction, alors cet appel de fonction sera complètement exécuté avant que l'exécution ne s'arrête à nouveau.
- `until` : comme `next`, sauf que si vous êtes à la fin d'une boucle, `until` continuera l'exécution jusqu'à ce que la fin de la boucle. Ceci est pratique si vous voulez voir ce qui se passe après une boucle, mais que vous ne voulez pas parcourir chaque itération.
- `finish` : continuer l'exécution jusqu'à la fin de la fonction courante.
- `where` : affiche le numéro de ligne en cours d'exécution

Chapitre 4 : Les outils

Section 4 : Valgrind

Détecter les fuites : Valgrind

Qu'est-ce que Valgrind ?

Valgrind est un outil de programmation pour le débogage de la mémoire, la détection des fuites de mémoire et le profilage.

Attention :

- Comme pour GDB, pour que valgrind fonctionne correctement, votre programme doit être compilé avec l'option `-g`.
- valgrind n'est pas installé par défaut. Vous devez l'installer manuellement.

Détecter les fuites : Valgrind

Guide d'utilisation

Comment l'utiliser ?

Une fois que vous avez compilé votre programme dans les bonnes configurations. Utiliser Valgrind est simple. Il suffit d'utiliser la commande suivante :

```
valgrind --leak-check=yes nom_programme [[ args ]]
```

Les arguments `[[ args ]]` sont facultatifs, mais si votre programme en a besoin, vous pouvez les transmettre de la manière mentionnée ci-dessus.

Détecter les fuites : Valgrind

Guide d'utilisation : Exemple

Soit le code suivant :

```
#include <stdlib.h>

void toto()
{
    int* x = malloc(5 * sizeof(int));
    x[6] = 0;
}

int main()
{
    toto();
}
```

Détecter les fuites : Valgrind

Guide d'utilisation : Exemple

Compilation avec la commande : `gcc -g main.c`

Execution du valgrind avec la commande :

`valgrind --leak-check=yes ./a.out`

```
omar@Omar:~$ gcc -g main.c
omar@Omar:~$ valgrind --leak-check=yes ./a.out
==2200== Memcheck, a memory error detector
==2200== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2200== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==2200== Command: ./a.out
==2200==
==2200== Invalid write of size 4
==2200==    at 0x10916B: toto (main.c:6)
==2200==    by 0x109185: main (main.c:11)
==2200==  Address 0x4a52058 is 4 bytes after a block of size 20 alloc'd
==2200==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==2200==    by 0x10915E: toto (main.c:5)
==2200==    by 0x109185: main (main.c:11)
==2200==
==2200==
==2200== HEAP SUMMARY:
==2200==     in use at exit: 20 bytes in 1 blocks
==2200==   total heap usage: 1 allocs, 0 frees, 20 bytes allocated
==2200==
==2200== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2200==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==2200==    by 0x10915E: toto (main.c:5)
==2200==    by 0x109185: main (main.c:11)
==2200==
==2200== LEAK SUMMARY:
==2200==    definitely lost: 20 bytes in 1 blocks
==2200==    indirectly lost: 0 bytes in 0 blocks
==2200==      possibly lost: 0 bytes in 0 blocks
==2200==    still reachable: 0 bytes in 0 blocks
==2200==         suppressed: 0 bytes in 0 blocks
==2200==
==2200== For lists of detected and suppressed errors, rerun with: -s
==2200== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Figure – La sortie Valgrind lors de l'exécution

Détecter les fuites : Valgrind

Comment interpréter le résultat

```
==2200== Invalid write of size 4
==2200==    at 0x10916B: toto (main.c:6)
==2200==    by 0x109185: main (main.c:11)
==2200==  Address 0x.. is 4 bytes after a block of size 20 allocd
==2200==    at 0x483B7F3: malloc (vgpreload_memcheck.so)
==2200==    by 0x10915E: toto (main.c:5)
==2200==    by 0x109185: main (main.c:11)
```

Détecter les fuites : Valgrind

Comment interpréter le résultat

Explication :

- Le numéro `2200` est le PID du processus, c'est n'est pas trés important pour nous.
- Ligne 1 : indique une écriture de 4 octets dans une zone mémoire qui n'appartient pas au processus. Ce qui est affiché après est la pile d'appels menant à cette erreur.
- Ligne 4 : donne plus détail sur l'erreur à la ligne 1. il précise l'adresse où l'écriture s'est produite et la position de cette adresse par rapport à ce que nous avons alloué auparavant. Il montre également d'où provient cette adresse en affichant la pile d'appels.

Détecter les fuites : Valgrind

Comment interpréter le résultat

```
20 bytes in 1 blocks are definitely lost in loss record 1 of 1
   at 0x483B7F3: malloc (in vgpreload_memcheck.so)
   by 0x10915E: toto (main.c:5)
   by 0x109185: main (main.c:11)
```

Explication :

- Ligne 1 : indique qu'il y a un fuite de mémoire de 20 octets qui sont définitivement perdu. Les lignes juste après montrent la pile d'appels qui indique où la mémoire perdu a été allouée.
- Malheureusement, Valgrind ne peut pas vous dire pourquoi la mémoire a fui.

Détecter les fuites : Valgrind

ATTENTION :

- En fait, valgrind étant un framework, il peut faire plus que simplement détecter les fuites. Il fournit une suite d'outils et les outils que nous utilisons jusqu'à présent s'appellent Memcheck.
- Memcheck n'est pas parfait ; il produit parfois de faux positifs, et il existe des mécanismes pour supprimer ces effets secondaires. Cependant, il est généralement correct 99% du temps, vous devez donc éviter d'ignorer ses messages d'erreur.
- Memcheck ne peut pas détecter toutes les erreurs de mémoire. Par exemple, il ne peut pas détecter les lectures ou écritures hors limites dans des tableaux alloués statiquement sur la pile. Mais il devrait détecter de nombreuses erreurs qui pourraient planter le programme (par exemple, les erreurs de segmentation).

Valgrind

Détecter la mémoire non initialisée

Que peut-il faire d'autre pour nous ?

Valgrind signale également les utilisations de valeurs non initialisées, le plus souvent avec le message "Le saut ou le déplacement conditionnel dépend de la ou des valeurs non initialisées".
Il peut être difficile de déterminer la cause de ces erreurs. Essayez d'utiliser `--track-origins=yes` pour obtenir des informations supplémentaires.

Valgrind

Valgrind : Au-delà de la vérification de la mémoire

Comme mentionné précédemment, Valgrind peut fournir un ensemble de services qui sortent du cadre de ce cours, nous ne les couvrirons donc pas ici. Parmi ces outils, on retrouve :

- `Massif` : un profileur de tas, mesure l'utilisation de la mémoire du tas au fil du temps.
- `Cachegrind` : un profileur de cache, mesurer les succès et les échecs du cache et éventuellement les prédictions de branche.
- `Callgrind` : outil de profilage qui enregistre l'historique des appels des fonctions d'un programme sous forme d'un graphe.
- `Helgrind` : détecte les conditions de concurrence dans le code multithread.

Chapitre 5 : Conclusion

Conclusion

Les dix commandements pour réussir en C

1. Les comportements indéfinis, évitez-les.
2. Les limites de la mémoire allouées, ne les dépassez pas.
3. Vos variables, initialisez-les toujours.
4. Les fuites de mémoire, éliminez-les.
5. Les tableaux et les pointeurs, ne les confondez pas.
6. Les unions, faites attention lorsque vous les utilisez.
7. Vos variables, sachez dans quelle région elles résident.
8. Pointer casting, soyez prudent lorsque vous les éffectuer.
9. Vos chaînes, n'oubliez pas de les terminer par null
10. Les outils, n'hésitez pas à les utiliser.