Language C

Université de Lorraine - Télécom Nancy

Omar CHIDA

Chapitre 1

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.

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++.

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.

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

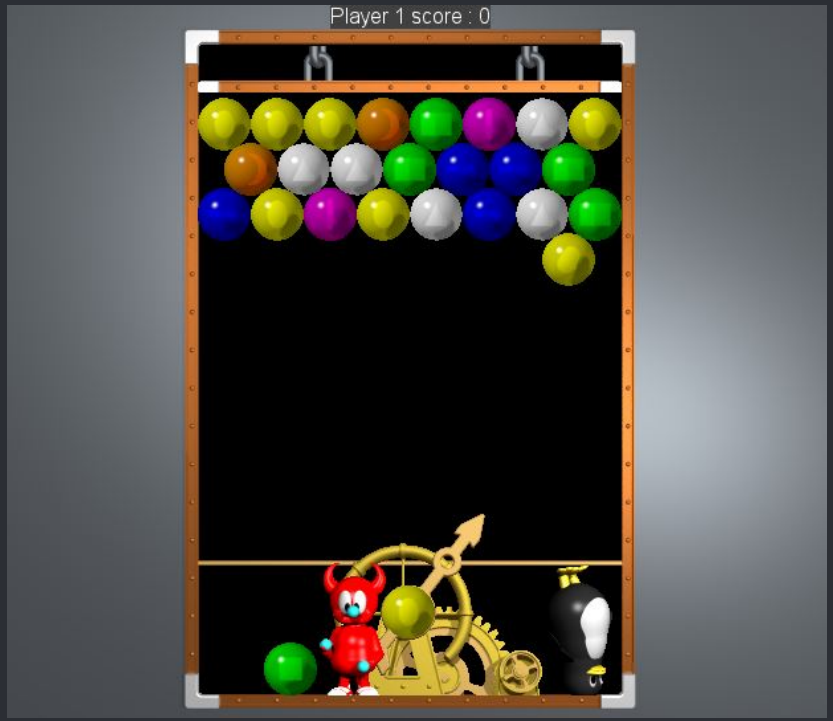
Ce que vous pourrez faire à la fin

```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).

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.

Motivation : Pouquoi apprendre le C en 2021 ?

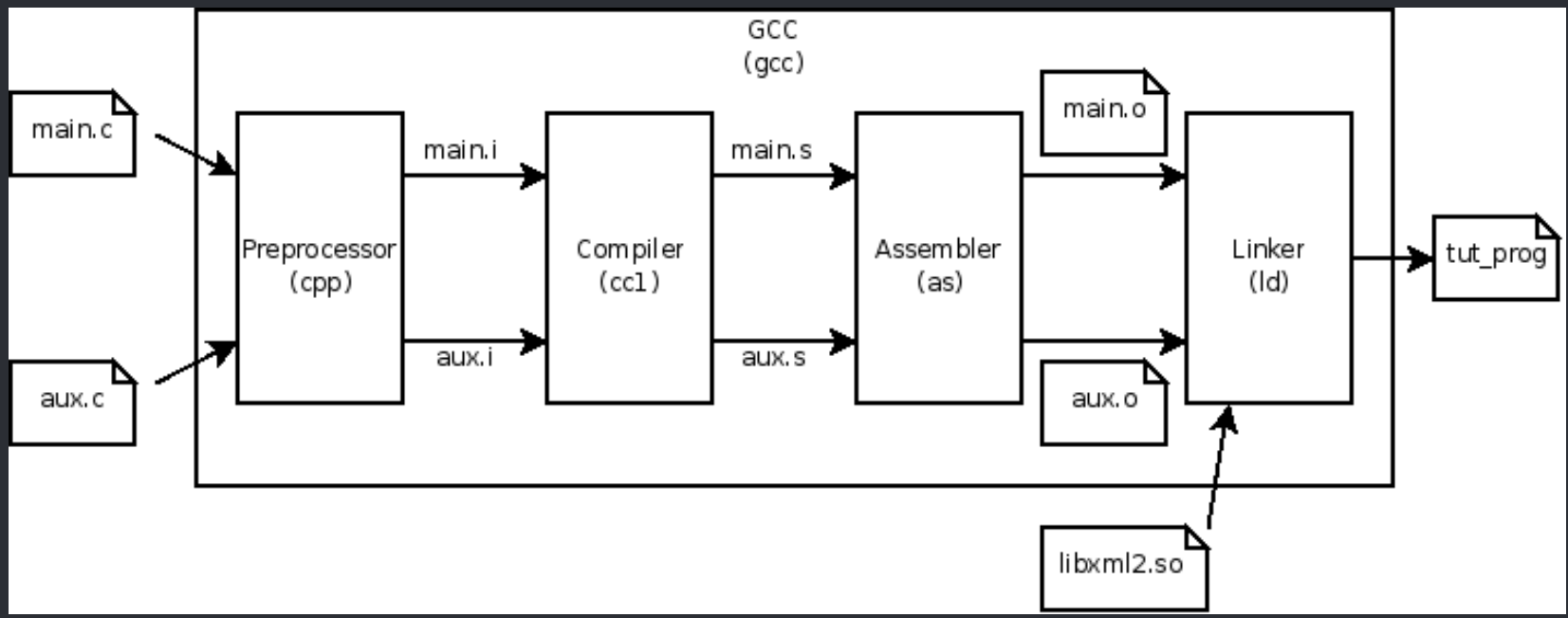
C c'est cool !

- 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

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



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`

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]"
```

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"

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`'

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

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 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.

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
{
  TypeA field1_name;
  TypeB field2_name;
  TypeC 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
{
  TypeA field1_name;
  TypeB field2_name;
  TypeC 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.

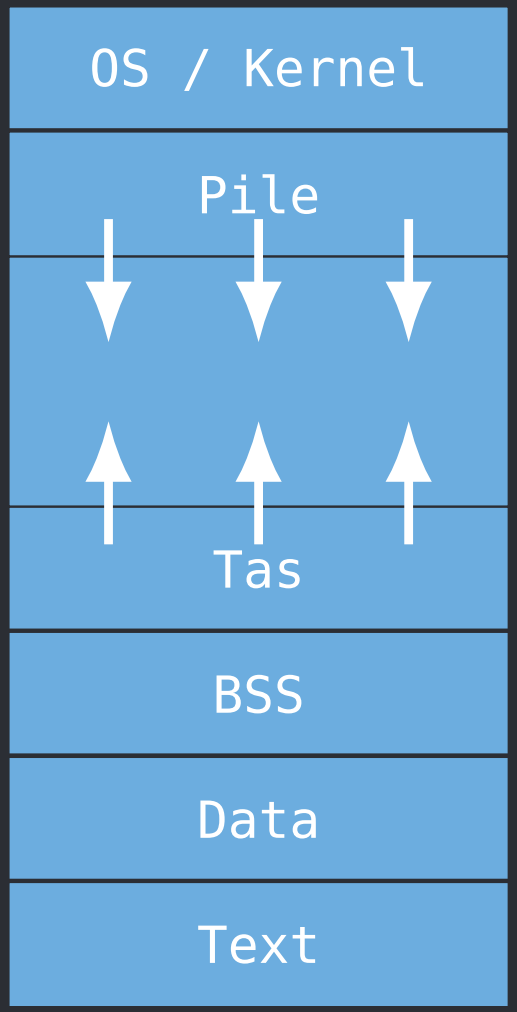
```
union B
{
  int a;
  short b;
  char str[4];
};
union B var;
var.str[0] = 'T';
var.str[1] = 'N';
var.str[2] = 'C';
var.str[3] = 'Y';
var.b = 256; // ATTENTION: var.str ne vaut plus TNCY !!!
```

Les tableaux

Les chaines

La mémoire

Disposition de la mémoire d'un programme



Haute adresse

Adresses basses

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

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 [5] : 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 qui sont initialisées à zéro ou qui n'ont pas d'initialisation explicite dans le code source. le segment BSS est Read-Write. Le segment BSS est également appelé "segment de données non initialisé".

5. Block Started by Symbol

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 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.

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 erreur: 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";
char* str2 = "World";

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

La mémoire

Cela devrait fonctionner

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";
  char* str2 = "World";
  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 `*alloc` [6], il doit nécessairement y avoir un appel correspondant à `free`.

6. malloc et calloc

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 [7].

7. 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

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

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

Exemple :

```
void* p1 = calloc(1, 256); // 512 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

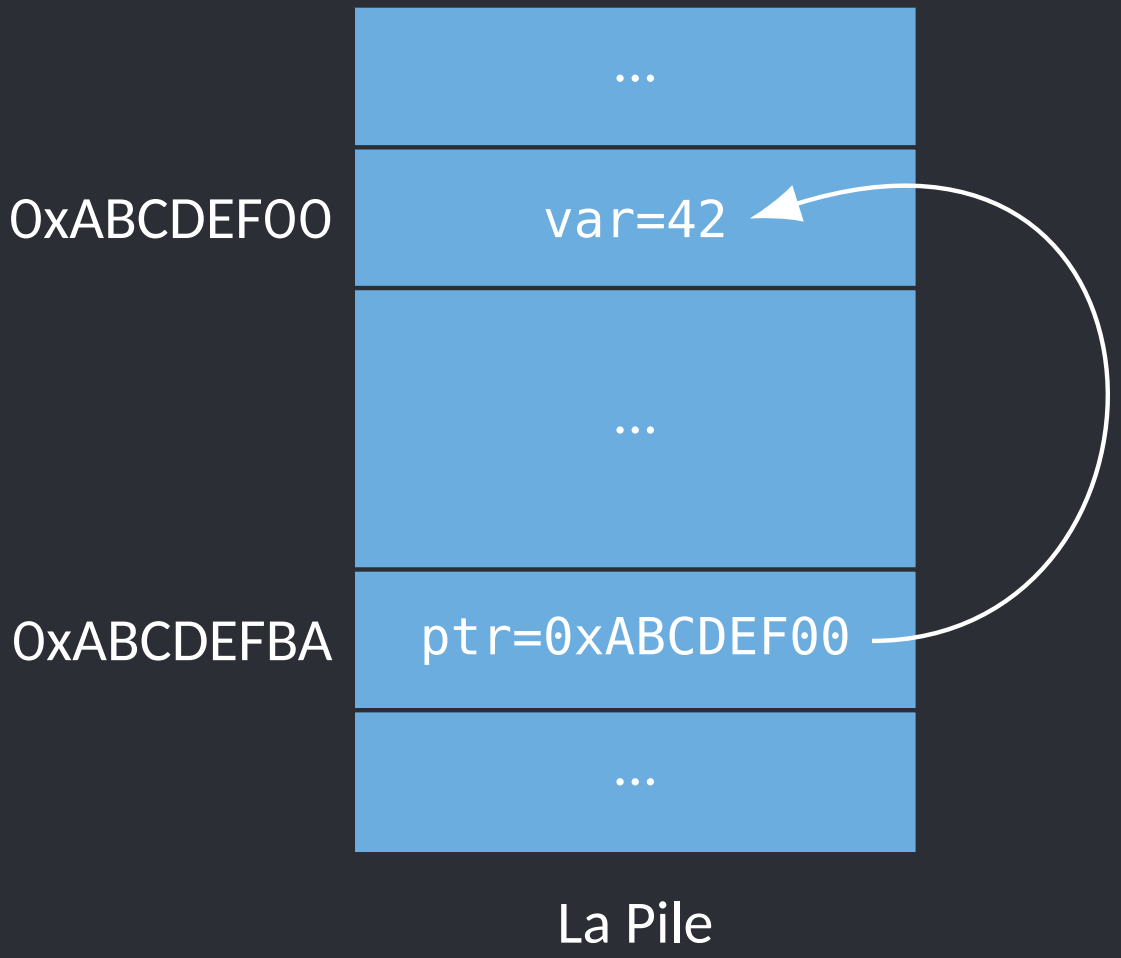
Pour obtenir l'adresse d'une variable, l'opérateur & doit être utilisé.

Exemple :

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

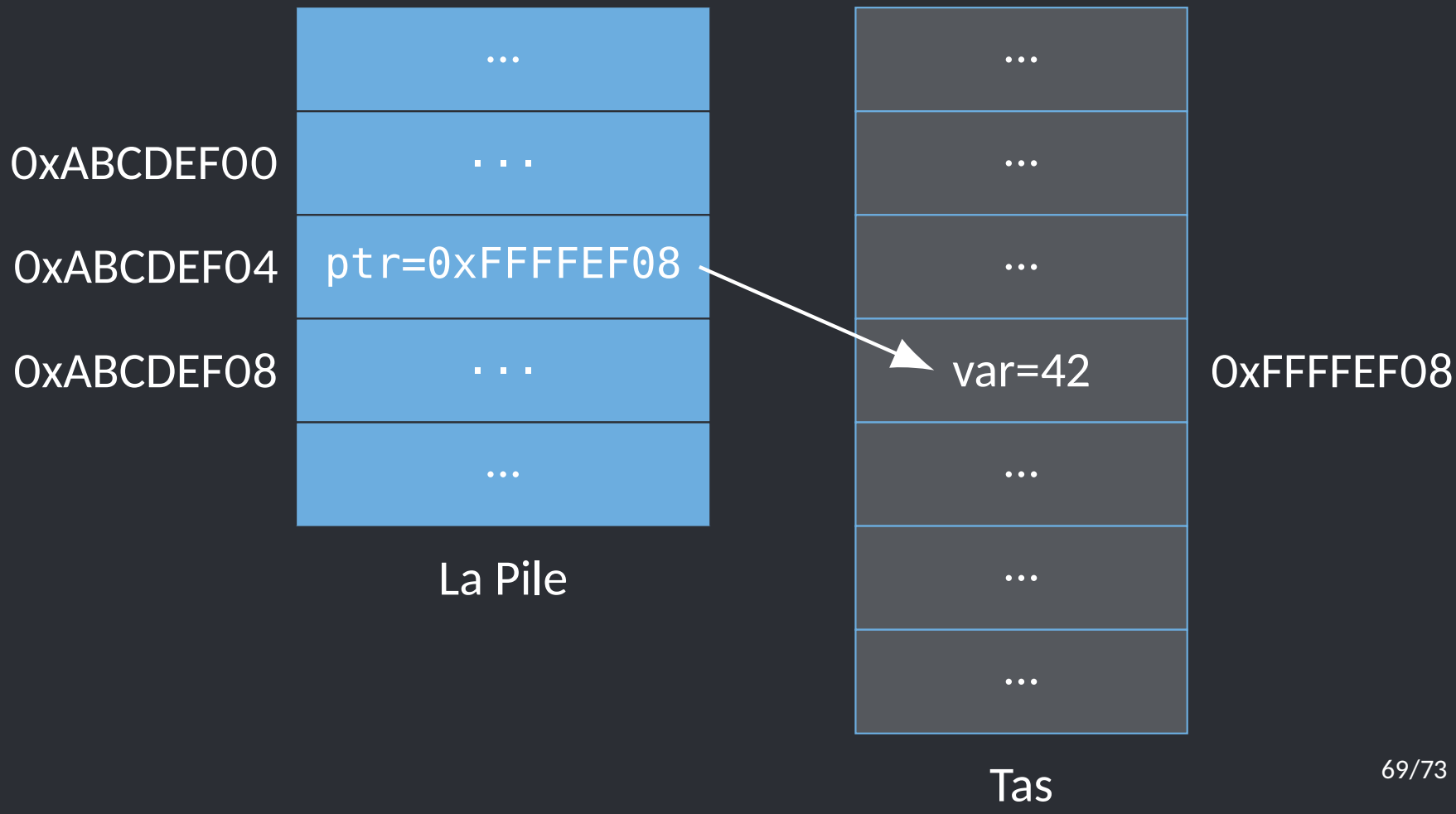
La mémoire

Pointeur et variable sur la pile



La mémoire

Pointeur sur la pile et variable sur le tas



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 = ...; // changed the value 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.

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

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 TYPE, l'expression `ptr+step` ajoutera `step*sizeof(TYPE)` 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.

Pointer casting

content...

Chapitre 4