Langage *C* : Modèle mémoire

Sébastien Jean

IUT de Valence
Département Informatique

v1.0, 20 novembre 2025

IUT Valence
Université Grenoble Alpes

Interlude : Création/clonage d'un projet Gitlab

- **Créer un projet** `ModeleMemoireC` sur Gitlab (avec un `README`)
- **Cloner** le projet depuis *VsCode* et ouvrir le dépôt
- **Modifier** le fichier `README.md` pour indiquer à quoi sert ce projet
- N.B. : pour chacun des exemples/exercices `ExX` suivant :
  - **Ecrire le programme** dans `ExX/src/main.c` et le compiler dans `ExX/build/ExX`
  - Rédiger (si nécessaire) un jeu d'essai dans un fichier `ExerciceX/Essai`

IUT Valence Université Grenoble Alpes

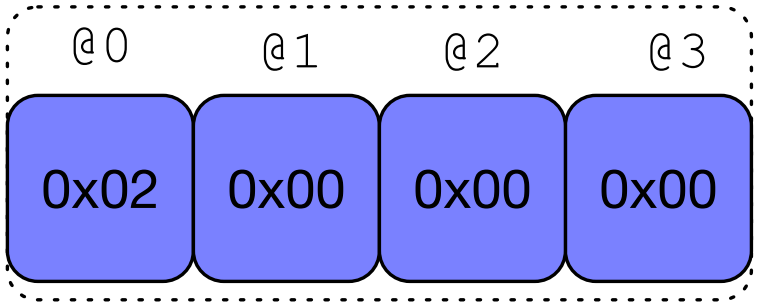
Représentation mémoire

- Variable Vs mémoire (exemple avec `int`)

Déclaration/affectation `int i = 2;`

Représentation logique

Représentation mémoire



Sérialisation *Little-Endian*

- Endianness ← convention de sérialisation des octets/bits
  - *Little-Endian* : octet de poids faible en premier
  - *Big-Endian* : octet de poids fort en permier

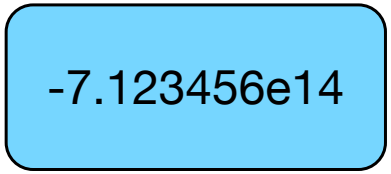
IUT Valence Université Grenoble Alpes

Représentation mémoire

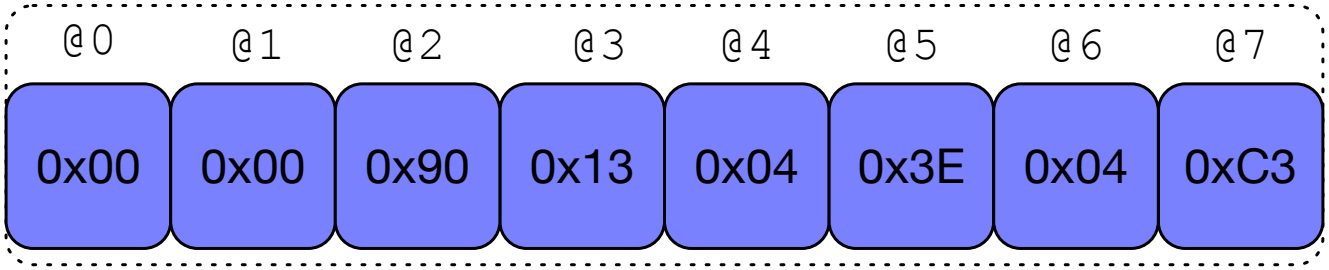
- Variable Vs mémoire (exemple avec `double`)

Déclaration/affectation `double d = -7.123456e14;`

Représentation logique



Représentation mémoire

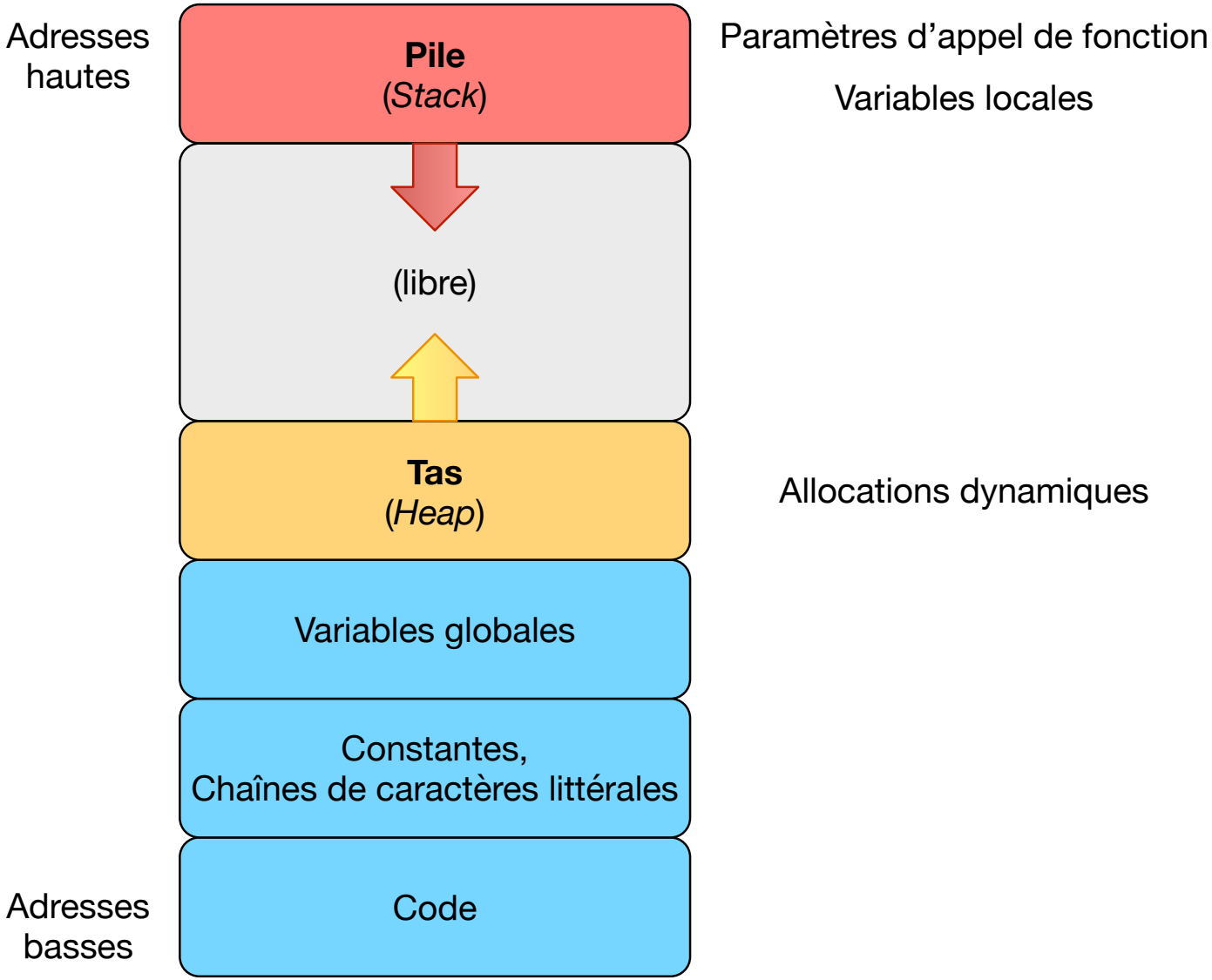


Sérialisation *Little-Endian*

- Représentation des flottants à la norme IEEE754

Modèle mémoire

- Espace mémoire réservé à l'**exécution d'une application**

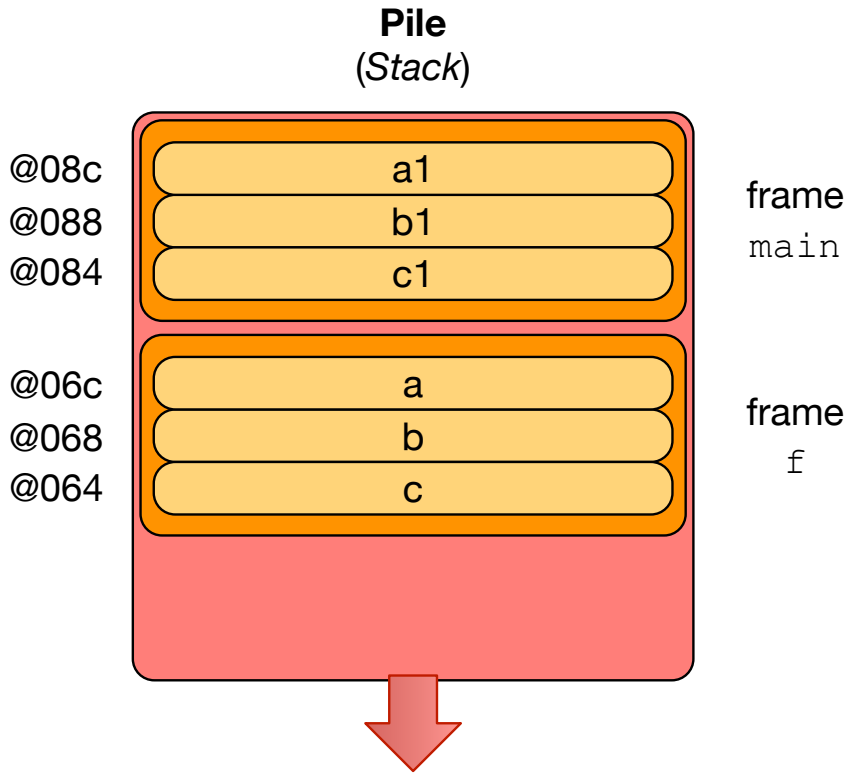


IUT Valence Université Grenoble Alpes

Modèle mémoire

- La **pile** est une succession de *frames* (contextes d'appels de fonction)
  - Lors de l'appel d'une fonction, une **nouvelle *frame*** est créée, les paramètres d'appels y sont copiés, les variables locales y sont allouées et cette *frame* devient le **contexte courant**
  - Lorsque l'appel d'une fonction se termine, sa *frame* est **détruite** et la *frame appelante* redevient le **contexte courant**

```c
#include <stdio.h>
#include <stdlib.h>

int f(int a, int b) {

    int c = a + b;
    return c;
}

int main() {
    int a1 = 3;
    int b1 = 2;

    int c1 = f(a1,b1);

    printf("%d\n", c1);
}
```



Récursivité et pile

- Récursivité → imbrication d'appels de fonction
- Imbrication d'appels → **risque de dépassement de pile**

- **Faire déborder la pile par une imbrication infinie d'appels d'une même fonction**
  - Observer les avertissements à la compilation, et l'erreur à l'exécution
  - Afficher le nombre d'appels (pris en paramètre) dans la fonction

IUT Valence Université Grenoble Alpes

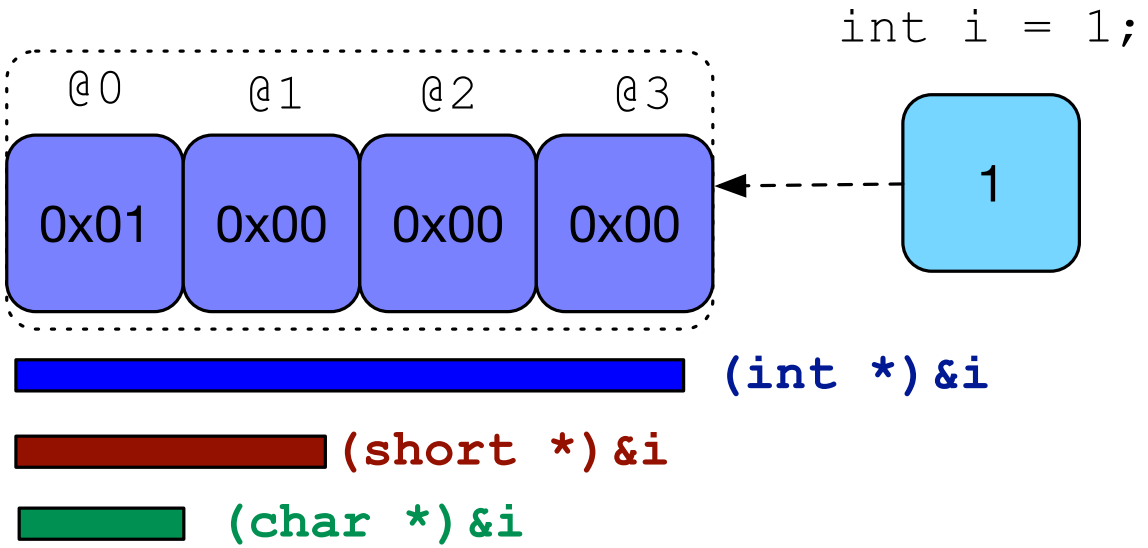
Récursivité et pile

Code C

```c
#include <stdio.h>

int f(unsigned long n) {
    printf("%llu\n", n);
    return f(n+1);
}

int main() {

    f(0);

    return 0;
}
```

Cast de pointeurs

- Un pointeur désigne une adresse mémoire et permet de manipuler une valeur typée située à partir de cette adresse, mais finalement la zone mémoire n'est pas typée
- Le type de la valeur d'un pointeur, une adresse, **peut être forcé dans tous les types pointeur vers `T`**



Arithmétique de pointeurs

- En C, l'**arithmétique de pointeurs** est possible

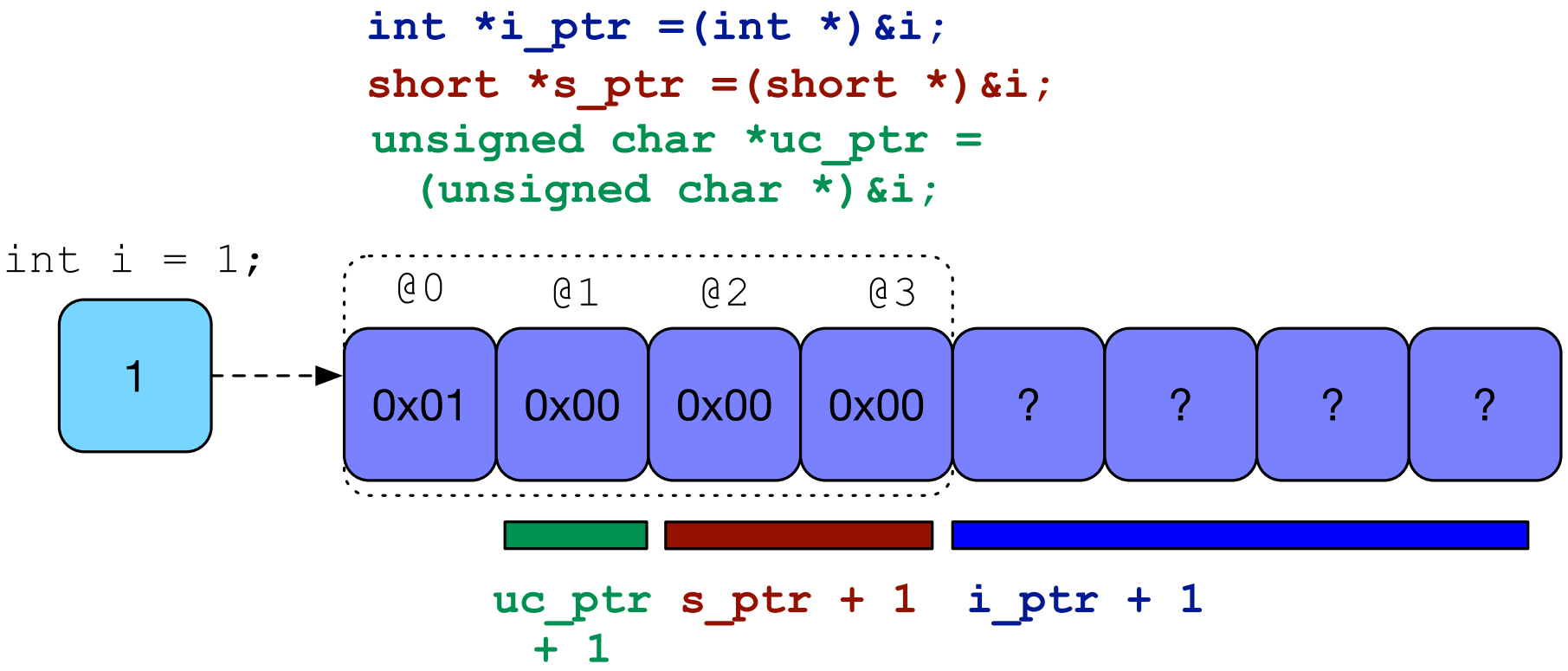
Code C

```c
int i = 3;

int *i_ptr = &i;
*i = 0;

int *i2_ptr = i_ptr + 1;
*i2 = 0;    // equivalent à *(i_ptr+1)

// risqué car potentiellement occupé par une autre
   variable !
```

IUT Valence Université Grenoble Alpes

Arithmétique de pointeurs (suite)

- Attention : ajouter 1 à la valeur d'un pointeur vers `T`, c'est en réalité **décaler l'adresse de** `sizeof(T)`



Cast et arithmétique de pointeurs

- A tester (dans une fonction `main`) !

Code C

```c
int i = 0;
printf("%d\n", i);

int *i_ptr = &i;
printf("%d\n", *i_ptr);

unsigned char *c_ptr = (unsigned char *) i_ptr;
*(c_ptr) = 1;
printf("%d\n", *i_ptr);
*(c_ptr+1) = 2;
printf("%d\n", *i_ptr);
*(c_ptr+2) = 3;
printf("%d\n", *i_ptr);
*(c_ptr+3) = 4;
printf("%d\n", *i_ptr);
```

Exercice

- Ecrire une application qui **affiche le contenu de la zone mémoire associée à** :
  - Une variable de type `int` initialisée à `1`
  - Une variable de type `double` initialisée à `-7.123456e14`
- L'affichage sera de la forme `adresse hexa : contenu hexa`
  - N.B. : utiliser `%p` pour les adresses et `%02x` pour le contenu

IUT Valence
Université Grenoble Alpes

Exercice

Code C

```c
#include <stdio.h>
#include <stdlib.h>

int main() {

    int i = 1;
    char * c_ptr = (char *)&i;

    for (int j=0; j<(sizeof(i)); j++) {
        printf(" %p : %02x\n", c_ptr+j,
                ((unsigned char) *(c_ptr+j)));
    }
    printf("\n");

(à suivre ...)
}
```

Exercice

Code C

```c
#include <stdio.h>
#include <stdlib.h>

int main() {

(...suite )

    double d = -7.123456e14;
    char * d_ptr = (char *)&d;

    for (int j=0; j<(sizeof(d)); j++) {
        printf(" %p : %02x\n", d_ptr+j,
            ((unsigned char) *(d_ptr+j)));
    }

    return 0;
}
```

Exercice

- **Isoler** l'affichage hexadécimal d'une zone mémoire (début/taille) dans une fonction
- Déplacer cette fonction dans un **module** `mem_utils`

IUT Valence
Université Grenoble Alpes

Fin !